TreeNode

1.5.2

Generated by Doxygen 1.9.5

Chapter 1

TreeNode C# library

This is the documentation website for the **TreeNode C# library**.

TreeNode is a library for reading, writing and manipulating phylogenetic trees in C# and R. It can open and create files in the most common phylogenetic formats (i.e. Newick/New Hampshire and NEXUS) and adds support for two new formats, the `Newick-with-Attributes` and `Binary format`. The C# library also supports the `NCBI ASN.1` format (text and binary).

The **TreeNode C# library**, in addition to providing methods to read and write phylogenetic tree files, also includes methods to manipulate the resulting trees (e.g. to reroot the tree, compute a consensus tree, find the last common ancestor of a group, etc.).

TreeNode is released under the `GPLv3` licence.

1.1 Getting started

The TreeNode C# library targets .NET Standard 2.1, thus it can be used in projects that target .NET Standard 2.1+ and .NET Core 3.0+, as well as Mono and Xamarin.

To use the library in your project, you should install the `TreeNode NuGet package`.

1.2 Usage

The `Examples` project in the `TreeNode GitHub repository` contains an example C# .NET Core console program showing some of the capabilities of the library.

The `PhyloTree` namespace contains the `TreeNode` class, which is used to represent nodes in a tree. `TreeNode` does not distinguish between internal nodes, tips or even whole trees (except when looking at some specific properties - e.g. a tip will not have any `Children`, and the root node of the tree will not have any `Parent`). This makes it possible to navigate the tree in an intuitive manner: for example, the ancestor of a `TreeNode` can be accessed using its `Parent` property (which is itself a `TreeNode`) and the descendants of a node can be found as the node's `Children` (which is a `List<TreeNode>`).

A full list of the information that can be extracted and the manipulations that can be performed on `TreeNode` objects can be obtained by looking at the methods and properties of the `TreeNode` class in this website.

In addition to this, the PhyloTree namespace contains the Phylotree.Formats namespace. The three classes in this namespace (NWKA, NEXUS and BinaryTree) contain methods that can be used to read and write TreeNode objects to files in the respective format.

Each of these classes offers (at least) the following methods (with additional optional arguments):

```
//Methods to read trees
IEnumerable<TreeNode> ParseTrees(string inputFile);
IEnumerable<TreeNode> ParseTrees(Stream inputStream);
List<TreeNode> ParseAllTrees(string inputFile);
List<TreeNode> ParseAllTrees(Stream inputStream);
//Methods to write trees
void WriteTree(TreeNode tree, string outputFile);
void WriteTree(TreeNode tree, Stream outputStream);
void WriteAllTrees(IEnumerable<TreeNode> trees, string outputFile);
void WriteAllTrees(IEnumerable<TreeNode> trees, Stream outputStream);
void WriteAllTrees(List<TreeNode> trees, string outputFile);
void WriteAllTrees(List<TreeNode> trees, Stream outputStream);
```

The ParseTrees methods can be used to read trees off a file or a Stream, without having to load them completely into memory. This can be useful if each tree only needs to be processed briefly. The ParseAllTrees methods instead load all the trees from the file into memory.

The WriteTree methods are used to write a single tree to a file or a stream, while the WriteAllTrees methods write a collection of trees.

In addition to this, the library also provides the TreeCollection class, which represents a collection of trees, much like a List<TreeNode>. However, a TreeCollection can also be created by passing a stream of trees in binary format to it: in this case, the TreeCollection will only parse trees from the stream when necessary, thus reducing the amount of memory that is necessary to store them.

The key feature of TreeCollection is that this is done *transparently*: accessing an element of the collection, e.g. by using treeCollection[i], will automatically perform all the reading and parsing operations from the stream to produce the TreeNode that is returned. This makes it possible to have an "agnostic" interface that behaves in the same way whether the trees in the collection have been completely loaded into memory or not.

Chapter 2

Namespace Index

2.1 Package List

Here are the packages with brief descriptions (if available):

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 PhyloTree Namespace Reference

Contains classes and methods to read, write and manipulate phylogenetic trees.

Namespaces

- namespace Extensions

  Contains useful extension methods.
- namespace Formats

  Contains classes and methods to read and write phylogenetic trees in multiple formats
- namespace SequenceSimulation

  Contains classes and methods that can be used to simulate sequence evolution.

Classes

- class AttributeDictionary

  Represents the attributes of a node. Attributes Name, Length and Support are always included. See the respective properties for default values.
- class TreeCollection

  Represents a collection of TreeNode objects. If the full representations of the TreeNode objects reside in memory, this offers the best performance at the expense of memory usage. Alternatively, the trees may be read on demand from a stream in binary format. In this case, accessing any of the trees will require the tree to be parsed. This reduces memory usage, but worsens performance. The internal storage model of the collection is transparent to consumers (except for the difference in performance/memory usage).
- class TreeNode

  Represents a node in a tree (or a whole tree).

6.1.1 Detailed Description

Contains classes and methods to read, write and manipulate phylogenetic trees.

6.2 PhyloTree.Extensions Namespace Reference

Contains useful extension methods.

Classes

- class TypeExtensions

  Useful extension methods

6.2.1 Detailed Description

Contains useful extension methods.

6.3 PhyloTree.Formats Namespace Reference

Contains classes and methods to read and write phylogenetic trees in multiple formats

Classes

- struct Attribute

  Describes an attribute of a node.
- class BinaryTree

  Contains methods to read and write tree files in binary format.
- class BinaryTreeMetadata

  Holds metadata information about a file containing trees in binary format.
- class NcbiAsnBer

  Contains methods to read and write trees in the NCBI ASN.1 binary format.
  ***Note**: this is a hackish reverse-engineering of the NCBI binary ASN format. A lot of this is derived by assumptions and observations.*
- class NcbiAsnText

  Contains methods to read and write trees in the NCBI ASN.1 text format.
- class NEXUS

  Contains methods to read and write trees in NEXUS format.
- class NWKA

  Contains methods to read and write trees in Newick and Newick-with-Attributes (NWKA) format.

6.3.1 Detailed Description

Contains classes and methods to read and write phylogenetic trees in multiple formats

6.4 PhyloTree.SequenceScores Namespace Reference

Classes

- class LikelihoodScores

  Contains methods to compute likelihood scores on a tree.
- class MissingDataException

  Exception that is thrown when not enough data has been supplied.
- class ParsimonyScore

  Contains methods to compute parsimony scores for a tree.

6.5 PhyloTree.SequenceSimulation Namespace Reference

Contains classes and methods that can be used to simulate sequence evolution.

Classes

- class ImmutableRateMatrix

  Represents a rate matrix whose values cannot be changed after initialisation.
- class IMutableRateMatrix

  Represents a rate matrix whose values can be changed after initialisation.
- class IndelModel

  Represents a model for sequence insertion/deletion.
- struct Insertion

  Represents an insertion event.
- class MutableRateMatrix

  Represents a rate matrix whose values can be changed after initialisation.
- class RateMatrix

  Represents a rate matrix for a continuous-type Markov chain model. This type cannot be instantiated directly, please use MutableRateMatrix or ImmutableRateMatrix instead, or access the static members for some pre-baked common rate matrices for DNA and protein evolution.
- class Sequence

  Represents a sequence of characters.
- class SequenceSimulation

  Contains methods to simulate sequence evolution.

6.5.1 Detailed Description

Contains classes and methods that can be used to simulate sequence evolution.

6.6 PhyloTree.TreeBuilding Namespace Reference

Classes

- class BirthDeathTree

  Contains methods to simulate birth-death trees.
- class CoalescentTree

  Contains methods to simulate coalescent trees.
- class DistanceMatrix

  Contains methods to compute distance matrices.
- class NeighborJoining

  Contains methods to compute neighbour-joining trees.
- class RandomTree

  Contains methods to generate random trees.
- class ThreadSafeRandom

  Represents a thread-safe random number generator.
- class UPGMA

  Contains methods to compute UPGMA trees.

Enumerations

- enum EvolutionModel

  The sequence evolution model used to compute the distance matrix from the alignment.
- enum AlignmentType

  The kind of sequences in the alignment.

6.6.1 Enumeration Type Documentation

6.6.1.1 AlignmentType

`enum PhyloTree.TreeBuilding.AlignmentType`

The kind of sequences in the alignment.

Definition at line 45 of file DistanceMatrix.cs.

6.6.1.2 EvolutionModel

`enum PhyloTree.TreeBuilding.EvolutionModel`

The sequence evolution model used to compute the distance matrix from the alignment.

Definition at line 11 of file DistanceMatrix.cs.

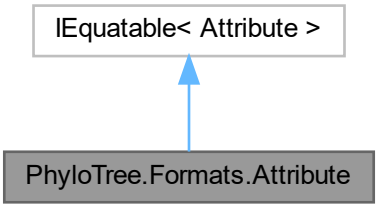
Chapter 7

Class Documentation

7.1 PhyloTree.Formats.Attribute Struct Reference

Describes an attribute of a node.

Inheritance diagram for PhyloTree.Formats.Attribute:



Public Member Functions

- Attribute (string attributeName, bool isNumeric)

  Constructs a new Attribute.
- override bool Equals (object obj)

  Compares an Attribute and another object.
- override int GetHashCode ()

  Returns the hash code for this Attribute.
- bool Equals (Attribute other)

  Compares two Attributes.

Static Public Member Functions

- static bool operator== (Attribute left, Attribute right)

  Compares two Attributes.
- static bool operator!= (Attribute left, Attribute right)

  Compares two Attributes (negated).

Properties

- string AttributeName [get]

  The name of the attribute.
- bool IsNumeric [get]

  Whether the attribute is represented by a numeric value or a string.

7.1.1 Detailed Description

Describes an attribute of a node.

Definition at line 692 of file Binary.cs.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 Attribute()

```
PhyloTree.Formats.Attribute.Attribute (
            string attributeName,
            bool isNumeric )
```

Constructs a new Attribute.

Parameters

attributeName	The name of the attribute.
isNumeric	Whether the attribute is represented by a numeric value or a string.

Definition at line 709 of file Binary.cs.

7.1.3 Member Function Documentation

7.1.3.1 Equals() [1/2]

```
bool PhyloTree.Formats.Attribute.Equals (
            Attribute other )
```

Compares two Attributes.

Parameters

other	The Attribute to compare to.

Returns

`true` if *other* has the same AttributeName (case insensitive) and value for IsNumeric as the current instance. `false` otherwise.

Definition at line 768 of file Binary.cs.

7.1.3.2 Equals() [2/2]

```
override bool PhyloTree.Formats.Attribute.Equals (
            object obj )
```

Compares an Attribute and another object.

Parameters

obj	The object to compare to.

Returns

`true` if *obj* is an Attribute and it has the same AttributeName (case insensitive) and value for IsNumeric as the current instance. `false` otherwise.

Definition at line 720 of file Binary.cs.

7.1.3.3 GetHashCode()

```
override int PhyloTree.Formats.Attribute.GetHashCode ( )
```

Returns the hash code for this Attribute.

Returns

The hash code for this Attribute.

Definition at line 736 of file Binary.cs.

7.1.3.4 operator"!=()

```
static bool PhyloTree.Formats.Attribute.operator!= (
            Attribute left,
            Attribute right )  [static]
```

Compares two Attributes (negated).

Parameters

left	The first Attribute to compare.
right	The second Attribute to compare.

Returns

`false` if both Attributes have the same AttributeName (case insensitive) and value for IsNumeric. `true` otherwise.

Definition at line 758 of file Binary.cs.

7.1.3.5 operator==()

```
static bool PhyloTree.Formats.Attribute.operator== (
            Attribute left,
            Attribute right )  [static]
```

Compares two Attributes.

Parameters

left	The first Attribute to compare.
right	The second Attribute to compare.

Returns

`true` if both Attributes have the same AttributeName (case insensitive) and value for IsNumeric. `false` otherwise.

Definition at line 747 of file Binary.cs.

7.1.4 Property Documentation

7.1.4.1 AttributeName

```
string PhyloTree.Formats.Attribute.AttributeName  [get]
```

The name of the attribute.

Definition at line 697 of file Binary.cs.

7.1.4.2 IsNumeric

```
bool PhyloTree.Formats.Attribute.IsNumeric  [get]
```

Whether the attribute is represented by a numeric value or a string.

Definition at line 702 of file Binary.cs.

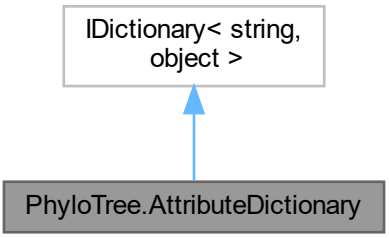
The documentation for this struct was generated from the following file:

- Binary.cs

7.2 PhyloTree.AttributeDictionary Class Reference

Represents the attributes of a node. Attributes Name, Length and Support are always included. See the respective properties for default values.

Inheritance diagram for PhyloTree.AttributeDictionary:



Public Member Functions

- void Add (string name, object value)

  Adds an attribute with the specified name and value to the AttributeDictionary. Throws an exception if the AttributeDictionary already contains an attribute with the same name .
- void Add (KeyValuePair< string, object > item)

  Adds an attribute with the specified name and value to the AttributeDictionary. Throws an exception if the AttributeDictionary already contains an attribute with the same name.
- void Clear ()

  Removes all attributes from the dictionary, except the "Name", "Length" and "Support" attributes.
- bool Contains (KeyValuePair< string, object > item)

  Determines whether the AttributeDictionary contains the specified item .
- bool ContainsKey (string name)

  Determines whether the AttributeDictionary contains an attribute with the specified name name .
- void CopyTo (KeyValuePair< string, object >[] array, int arrayIndex)

  Copies the elements of the AttributeDictionary to an array, starting at a specific array index.
- IEnumerator< KeyValuePair< string, object > > GetEnumerator ()

Returns an enumerator that iterates through the AttributeDictionary.

- bool Remove (string name)

  Removes the attribute with the specified name from the AttributeDictionary. Attributes "Name", "Length" and "Support" cannot be removed.
- bool Remove (KeyValuePair< string, object > item)

  Removes the attribute with the specified name from the AttributeDictionary. Attributes "Name", "Length" and "Support" cannot be removed.
- bool TryGetValue (string name, out object value)

  Gets the value of the attribute with the specified name . Getting the value of attributes "Name", "Length" and "Support" does not require a dictionary lookup.
- AttributeDictionary ()

  Constructs an AttributeDictionary containing only the "Name", "Length" and "Support" attributes.

Properties

- string Name [get, set]

  The name of this node (e.g. the species name for leaf nodes). Default is "". Getting the value of this property does not require a dictionary lookup.
- double Length [get, set]

  The length of the branch leading to this node. This is double.NaN for branches whose length is not specified (e.g. the root node). Getting the value of this property does not require a dictionary lookup.
- double Support [get, set]

  The support value of this node. This is double.NaN for branches whose support is not specified. The interpretation of the support value depends on how the tree was built. Getting the value of this property does not require a dictionary lookup.
- object this[string name] [get, set]

  Gets or sets the value of the attribute with the specified name . Getting the value of attributes "Name", "Length" and "Support" does not require a dictionary lookup.
- ICollection< string > Keys [get]

  Gets a collection containing the names of the attributes in the AttributeDictionary.
- ICollection< object > Values [get]

  Gets a collection containing the values of the attributes in the AttributeDictionary.
- int Count [get]

  Gets the number of attributes contained in the AttributeDictionary.
- bool IsReadOnly [get]

  Determine whether the AttributeDictionary is read-only. This is always false in the current implementation.

7.2.1 Detailed Description

Represents the attributes of a node. Attributes Name, Length and Support are always included. See the respective properties for default values.

Definition at line 13 of file AttributeDictionary.cs.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 AttributeDictionary()

```
PhyloTree.AttributeDictionary.AttributeDictionary ( )
```

Constructs an AttributeDictionary containing only the "Name", "Length" and "Support" attributes.

Definition at line 294 of file AttributeDictionary.cs.

7.2.3 Member Function Documentation

7.2.3.1 Add() [1/2]

```
void PhyloTree.AttributeDictionary.Add (
            KeyValuePair< string, object > item )
```

Adds an attribute with the specified name and value to the AttributeDictionary. Throws an exception if the AttributeDictionary already contains an attribute with the same name.

Parameters

item	The item to be added to the dictionary.

Definition at line 153 of file AttributeDictionary.cs.

7.2.3.2 Add() [2/2]

```
void PhyloTree.AttributeDictionary.Add (
            string name,
            object value )
```

Adds an attribute with the specified *name* and *value* to the AttributeDictionary. Throws an exception if the AttributeDictionary already contains an attribute with the same *name* .

Parameters

name	The name of the attribute.
value	The value of the attribute.

Definition at line 144 of file AttributeDictionary.cs.

7.2.3.3 Clear()

```
void PhyloTree.AttributeDictionary.Clear ( )
```

Removes all attributes from the dictionary, except the "Name", "Length" and "Support" attributes.

Definition at line 161 of file AttributeDictionary.cs.

7.2.3.4 Contains()

```
bool PhyloTree.AttributeDictionary.Contains (
            KeyValuePair< string, object > item )
```

Determines whether the AttributeDictionary contains the specified *item* .

Parameters

item	The item to locate in the AttributeDictionary

Returns

`true` if the AttributeDictionary contains the specified *item* , `false` otherwise.

Definition at line 179 of file AttributeDictionary.cs.

7.2.3.5 ContainsKey()

```
bool PhyloTree.AttributeDictionary.ContainsKey (
            string name )
```

Determines whether the AttributeDictionary contains an attribute with the specified name *name* .

Parameters

name	The name of the attribute to locate.

Returns

`true` if the AttributeDictionary contains an attribute with the specified *name* , `false` otherwise.

Definition at line 189 of file AttributeDictionary.cs.

7.2.3.6 CopyTo()

```
void PhyloTree.AttributeDictionary.CopyTo (
            KeyValuePair< string, object >[] array,
            int arrayIndex )
```

Copies the elements of the AttributeDictionary to an array, starting at a specific array index.

Parameters

array	The array to which the elements will be copied.
arrayIndex	The index at which to start copying.

Definition at line 199 of file AttributeDictionary.cs.

7.2.3.7 GetEnumerator()

```
IEnumerator< KeyValuePair< string, object > > PhyloTree.AttributeDictionary.GetEnumerator ( )
```

Returns an enumerator that iterates through the AttributeDictionary.

Returns

An enumerator that iterates through the AttributeDictionary.

Definition at line 214 of file AttributeDictionary.cs.

7.2.3.8 Remove() [1/2]

```
bool PhyloTree.AttributeDictionary.Remove (
            KeyValuePair< string, object > item )
```

Removes the attribute with the specified name from the AttributeDictionary. Attributes `"Name"`, `"Length"` and `"Support"` cannot be removed.

Parameters

item	The attribute to remove (only the name will be used).

Returns

A `bool` indicating whether the attribute was succesfully removed.

Definition at line 241 of file AttributeDictionary.cs.

7.2.3.9 Remove() [2/2]

```
bool PhyloTree.AttributeDictionary.Remove (
            string name )
```

Removes the attribute with the specified name from the AttributeDictionary. Attributes `"Name"`, `"Length"` and `"Support"` cannot be removed.

Parameters

name	The name of the attribute to remove.

Returns

A `bool` indicating whether the attribute was succesfully removed.

Definition at line 224 of file AttributeDictionary.cs.

7.2.3.10 TryGetValue()

```
bool PhyloTree.AttributeDictionary.TryGetValue (
            string name,
            out object value )
```

Gets the value of the attribute with the specified *name* . Getting the value of attributes `"Name"`, `"Length"` and `"Support"` does not require a dictionary lookup.

Parameters

name	The name of the attribute to get.
value	When this method returns, contains the value of the attribute with the specified *name* , if this is found in the AttributeDictionary, or `null` otherwise.

Returns

A `bool` indicating whether an attribute with the specified *name* was found in the AttributeDictionary.

Definition at line 259 of file AttributeDictionary.cs.

7.2.4 Property Documentation

7.2.4.1 Count

```
int PhyloTree.AttributeDictionary.Count  [get]
```

Gets the number of attributes contained in the AttributeDictionary.

Definition at line 132 of file AttributeDictionary.cs.

7.2.4.2 IsReadOnly

```
bool PhyloTree.AttributeDictionary.IsReadOnly [get]
```

Determine whether the AttributeDictionary is read-only. This is always `false` in the current implementation.

Definition at line 137 of file AttributeDictionary.cs.

7.2.4.3 Keys

```
ICollection<string> PhyloTree.AttributeDictionary.Keys [get]
```

Gets a collection containing the names of the attributes in the AttributeDictionary.

Definition at line 122 of file AttributeDictionary.cs.

7.2.4.4 Length

```
double PhyloTree.AttributeDictionary.Length [get], [set]
```

The length of the branch leading to this node. This is `double.NaN` for branches whose length is not specified (e.g. the root node). Getting the value of this property does not require a dictionary lookup.

Definition at line 40 of file AttributeDictionary.cs.

7.2.4.5 Name

```
string PhyloTree.AttributeDictionary.Name [get], [set]
```

The name of this node (e.g. the species name for leaf nodes). Default is `""`. Getting the value of this property does not require a dictionary lookup.

Definition at line 22 of file AttributeDictionary.cs.

7.2.4.6 Support

```
double PhyloTree.AttributeDictionary.Support [get], [set]
```

The support value of this node. This is `double.NaN` for branches whose support is not specified. The interpretation of the support value depends on how the tree was built. Getting the value of this property does not require a dictionary lookup.

Definition at line 58 of file AttributeDictionary.cs.

7.2.4.7 this[string name]

```
object PhyloTree.AttributeDictionary.this[string name] [get], [set]
```

Gets or sets the value of the attribute with the specified *name* . Getting the value of attributes `"Name"`, `"Length"` and `"Support"` does not require a dictionary lookup.

Parameters

name	The name of the attribute to get/set.

Returns

The value of the attribute, boxed into an `object`.

Definition at line 76 of file AttributeDictionary.cs.

7.2.4.8 Values

```
ICollection<object> PhyloTree.AttributeDictionary.Values  [get]
```

Gets a collection containing the values of the attributes in the AttributeDictionary.

Definition at line 127 of file AttributeDictionary.cs.

The documentation for this class was generated from the following file:

- AttributeDictionary.cs

7.3 PhyloTree.Formats.BinaryTree Class Reference

Contains methods to read and write tree files in binary format.

Static Public Member Functions

- static bool HasValidTrailer (Stream inputStream, bool keepOpen=false)

  Determines whether the tree file stream has a valid trailer.
- static bool IsValidStream (Stream inputStream, bool keepOpen=false)

  Determines whether the tree file stream is valid (i.e. it has a valid header).
- static BinaryTreeMetadata ParseMetadata (Stream inputStream, bool keepOpen=false, BinaryReader reader=null, Action< double > progressAction=null)

  Reads the metadata from a file containing trees in binary format.
- static IEnumerable< TreeNode > ParseTrees (Stream inputStream, bool keepOpen=false, Action< double > progressAction=null)

  Lazily parses trees from a file in binary format. Each tree in the file is not read and parsed until it is requested.
- static List< TreeNode > ParseAllTrees (Stream inputStream, bool keepOpen=false, Action< double > progressAction=null)

  Parses trees from a file in binary format and completely loads them in memory.
- static IEnumerable< TreeNode > ParseTrees (string inputFile, Action< double > progressAction=null)

  Lazily parses trees from a file in binary format. Each tree in the file is not read and parsed until it is requested.
- static List< TreeNode > ParseAllTrees (string inputFile, Action< double > progressAction=null)

  Parses trees from a file in binary format and completely loads them in memory.

- static void WriteTree (TreeNode tree, Stream outputStream, bool keepOpen=false, Stream additionalData↩ ToCopy=null)

  Writes a single tree in Binary format.
- static void WriteTree (TreeNode tree, string outputFile, bool append=false, Stream additionalDataTo↩ Copy=null)

  Writes a single tree in Binary format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, string outputFile, bool append=false, Action< int > progressAction=null, Stream additionalDataToCopy=null)

  Writes trees in binary format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, Stream outputStream, bool keepOpen=false, Action< int > progressAction=null, Stream additionalDataToCopy=null)

  Writes trees in binary format.
- static void WriteAllTrees (IList< TreeNode > trees, string outputFile, bool append=false, Action< double > progressAction=null, Stream additionalDataToCopy=null)

  Writes trees in binary format.
- static void WriteAllTrees (IList< TreeNode > trees, Stream outputStream, bool keepOpen=false, Action< double > progressAction=null, Stream additionalDataToCopy=null)

  Writes trees in binary format.

7.3.1 Detailed Description

Contains methods to read and write tree files in binary format.

Definition at line 16 of file Binary.cs.

7.3.2 Member Function Documentation

7.3.2.1 HasValidTrailer()

```
static bool PhyloTree.Formats.BinaryTree.HasValidTrailer (
            Stream inputStream,
            bool keepOpen = false )  [static]
```

Determines whether the tree file stream has a valid trailer.

Parameters

inputStream	The Stream from which the file should be read. Its Stream.CanSeek must be `true`. It does not have to be a FileStream.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.

Returns

`true` if the *inputStream* has a valid trailer, `false` otherwise.

Definition at line 24 of file Binary.cs.

7.3.2.2 IsValidStream()

```
static bool PhyloTree.Formats.BinaryTree.IsValidStream (
            Stream inputStream,
            bool keepOpen = false )  [static]
```

Determines whether the tree file stream is valid (i.e. it has a valid header).

Parameters

inputStream	The Stream from which the file should be read. Its Stream.CanSeek must be `true`. It does not have to be a FileStream.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.

Returns

`true` if the *inputStream* has a valid header, `false` otherwise.

Definition at line 61 of file Binary.cs.

7.3.2.3 ParseAllTrees() [1/2]

```
static List< TreeNode > PhyloTree.Formats.BinaryTree.ParseAllTrees (
            Stream inputStream,
            bool keepOpen = false,
            Action< double > progressAction = null )  [static]
```

Parses trees from a file in binary format and completely loads them in memory.

Parameters

inputStream	The Stream from which the file should be read. Its Stream.CanSeek must be `true`. It does not have to be a FileStream.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A List<T> containing the trees defined in the file.

Definition at line 384 of file Binary.cs.

7.3.2.4 ParseAllTrees() [2/2]

```
static List< TreeNode > PhyloTree.Formats.BinaryTree.ParseAllTrees (
            string inputFile,
            Action< double > progressAction = null )  [static]
```

Parses trees from a file in binary format and completely loads them in memory.

Parameters

inputFile	The path to the input file.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A List<T> containing the trees defined in the file.

Definition at line 407 of file Binary.cs.

7.3.2.5 ParseMetadata()

```
static BinaryTreeMetadata PhyloTree.Formats.BinaryTree.ParseMetadata (
            Stream inputStream,
            bool keepOpen = false,
            BinaryReader reader = null,
            Action< double > progressAction = null )  [static]
```

Reads the metadata from a file containing trees in binary format.

Parameters

inputStream	The Stream from which the file should be read. Its Stream.CanSeek must be `true`. It does not have to be a FileStream.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
reader	A BinaryReader to read from the *inputStream* . If this is `null`, a new BinaryReader will be initialised and disposed within this method.
progressAction	An Action that may be invoked while parsing the tree file, with an argument ranging from 0 to 1 describing the progress made in reading the file (determined by the position in the stream).

Returns

A BinaryTreeMetadata object containing metadata information about the tree file.

Definition at line 108 of file Binary.cs.

7.3.2.6 ParseTrees() [1/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.BinaryTree.ParseTrees (
            Stream inputStream,
            bool keepOpen = false,
            Action< double > progressAction = null )  [static]
```

Lazily parses trees from a file in binary format. Each tree in the file is not read and parsed until it is requested.

Parameters

inputStream	The Stream from which the file should be read. Its Stream.CanSeek must be `true`. It does not have to be a FileStream.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A lazy IEnumerable<T> containing the trees defined in the file.

Definition at line 252 of file Binary.cs.

7.3.2.7 ParseTrees() [2/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.BinaryTree.ParseTrees (
            string inputFile,
            Action< double > progressAction = null )  [static]
```

Lazily parses trees from a file in binary format. Each tree in the file is not read and parsed until it is requested.

Parameters

inputFile	The path to the input file.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A lazy IEnumerable<T> containing the trees defined in the file.

Definition at line 395 of file Binary.cs.

7.3.2.8 WriteAllTrees() [1/4]

```
static void PhyloTree.Formats.BinaryTree.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            Action< int > progressAction = null,
            Stream additionalDataToCopy = null )  [static]
```

Writes trees in binary format.

Parameters

trees	An IEnumerable<T> containing the trees to be written. It will ony be enumerated once.
outputStream	The Stream on which the trees should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
progressAction	An Action that will be invoked after each tree is written, with the number of trees written so far.
additionalDataToCopy	A stream containing additional data that will be copied into the binary file.

Definition at line 460 of file Binary.cs.

7.3.2.9 WriteAllTrees() [2/4]

```
static void PhyloTree.Formats.BinaryTree.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            string outputFile,
            bool append = false,
            Action< int > progressAction = null,
            Stream additionalDataToCopy = null )  [static]
```

Writes trees in binary format.

Parameters

trees	An IEnumerable<T> containing the trees to be written. It will ony be enumerated once.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
progressAction	An Action that will be invoked after each tree is written, with the number of trees written so far.
additionalDataToCopy	A stream containing additional data that will be copied into the binary file.

Definition at line 446 of file Binary.cs.

7.3.2.10 WriteAllTrees() [3/4]

```
static void PhyloTree.Formats.BinaryTree.WriteAllTrees (
            IList< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            Action< double > progressAction = null,
            Stream additionalDataToCopy = null )  [static]
```

Writes trees in binary format.

Parameters

trees	A collection of trees to be written. Each tree will be accessed twice.
outputStream	The Stream on which the trees should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
progressAction	An Action that will be invoked after each tree is written, with a value between 0 and 1 depending on how many trees have been written so far.
additionalDataToCopy	A stream containing additional data that will be copied into the binary file.

Definition at line 524 of file Binary.cs.

7.3.2.11 WriteAllTrees() [4/4]

```
static void PhyloTree.Formats.BinaryTree.WriteAllTrees (
            IList< TreeNode > trees,
            string outputFile,
            bool append = false,
            Action< double > progressAction = null,
            Stream additionalDataToCopy = null )  [static]
```

Writes trees in binary format.

Parameters

trees	A collection of trees to be written. Each tree will be accessed twice.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
progressAction	An Action that will be invoked after each tree is written, with a value between 0 and 1 depending on how many trees have been written so far.
additionalDataToCopy	A stream containing additional data that will be copied into the binary file.

Definition at line 509 of file Binary.cs.

7.3.2.12 WriteTree() [1/2]

```
static void PhyloTree.Formats.BinaryTree.WriteTree (
            TreeNode tree,
            Stream outputStream,
            bool keepOpen = false,
            Stream additionalDataToCopy = null )  [static]
```

Writes a single tree in Binary format.

Parameters

tree	The tree to be written.

Parameters

outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
additionalDataToCopy	A stream containing additional data that will be copied into the binary file.

Definition at line 420 of file Binary.cs.

7.3.2.13 WriteTree() [2/2]

```
static void PhyloTree.Formats.BinaryTree.WriteTree (
            TreeNode tree,
            string outputFile,
            bool append = false,
            Stream additionalDataToCopy = null )  [static]
```

Writes a single tree in Binary format.

Parameters

tree	The tree to be written.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
additionalDataToCopy	A stream containing additional data that will be copied into the binary file.

Definition at line 432 of file Binary.cs.

The documentation for this class was generated from the following file:

- Binary.cs

7.4 PhyloTree.Formats.BinaryTreeMetadata Class Reference

Holds metadata information about a file containing trees in binary format.

Properties

- IEnumerable< long > TreeAddresses [get, set]

  The addresses of the trees (i.e. byte offsets from the start of the file).
- bool GlobalNames [get, set]

  Determines whether there are any global names stored in the file's header that are used when parsing the trees.
- IReadOnlyList< string > Names [get, set]

  Contains any global names stored in the file's header that are used when parsing the trees.
- IReadOnlyList< Attribute > AllAttributes [get, set]

  Contains any global attributes stored in the file's header that are used when parsing the trees.

7.4.1 Detailed Description

Holds metadata information about a file containing trees in binary format.

Definition at line 666 of file Binary.cs.

7.4.2 Property Documentation

7.4.2.1 AllAttributes

```
IReadOnlyList<Attribute> PhyloTree.Formats.BinaryTreeMetadata.AllAttributes  [get], [set]
```

Contains any global attributes stored in the file's header that are used when parsing the trees.

Definition at line 686 of file Binary.cs.

7.4.2.2 GlobalNames

```
bool PhyloTree.Formats.BinaryTreeMetadata.GlobalNames  [get], [set]
```

Determines whether there are any global names stored in the file's header that are used when parsing the trees.

Definition at line 676 of file Binary.cs.

7.4.2.3 Names

```
IReadOnlyList<string> PhyloTree.Formats.BinaryTreeMetadata.Names  [get], [set]
```

Contains any global names stored in the file's header that are used when parsing the trees.

Definition at line 681 of file Binary.cs.

7.4.2.4 TreeAddresses

```
IEnumerable<long> PhyloTree.Formats.BinaryTreeMetadata.TreeAddresses  [get], [set]
```

The addresses of the trees (i.e. byte offsets from the start of the file).

Definition at line 671 of file Binary.cs.

The documentation for this class was generated from the following file:

- Binary.cs

7.5 PhyloTree.TreeBuilding.BirthDeathTree Class Reference

Contains methods to simulate birth-death trees.

Static Public Member Functions

- static TreeNode UnlabelledTree (double treeAge, double birthRate, double deathRate=0, bool keepDead↩ Lineages=false, CancellationToken cancellationToken=default)

  Simulate an unlabelled birth-death tree, stopping when the age of the tree reaches a certain value.
- static TreeNode LabelledTree (double treeAge, double birthRate, double deathRate=0, bool keepDead↩ Lineages=false, CancellationToken cancellationToken=default)

  Simulate a labelled birth-death tree, stopping when the age of the tree reaches a certain value.
- static TreeNode UnlabelledTree (int leafCount, double birthRate, double deathRate=0, bool keepDead↩ Lineages=false, CancellationToken cancellationToken=default)

  Simulate an unlabelled birth-death tree, stopping when the number of lineages that are alive in the tree reaches a certain value.
- static TreeNode LabelledTree (IReadOnlyList< string > leafNames, double birthRate, double deathRate=0, bool keepDeadLineages=false, TreeNode constraint=null, CancellationToken cancellationToken=default)

  Simulate a labelled birth-death tree, stopping when the number of lineages that are alive in the tree reaches a certain value.
- static TreeNode LabelledTree (int leafCount, double birthRate, double deathRate=0, bool keepDead↩ Lineages=false, TreeNode constraint=null, CancellationToken cancellationToken=default)

  Simulate a labelled birth-death tree, stopping when the number of lineages that are alive in the tree reaches a certain value.

7.5.1 Detailed Description

Contains methods to simulate birth-death trees.

Definition at line 12 of file BirthDeathTree.cs.

7.5.2 Member Function Documentation

7.5.2.1 LabelledTree() [1/3]

```
static TreeNode PhyloTree.TreeBuilding.BirthDeathTree.LabelledTree (
            double treeAge,
            double birthRate,
            double deathRate = 0,
            bool keepDeadLineages = false,
            CancellationToken cancellationToken = default )  [static]
```

Simulate a labelled birth-death tree, stopping when the age of the tree reaches a certain value.

Parameters

treeAge	The final age of the tree. Note that the actual age of the tree may be smaller than this; this can happen either if *keepDeadLineages* is `false` and one of the two clades descending from the root node goes exinct, or if *keepDeadLineages* is `true` and all the clades go extinct. If all the clades go extinct and *keepDeadLineages* is `false`, this method will return `null`.
birthRate	The birth rate of the tree.
deathRate	The death rate of the tree.
keepDeadLineages	If this is `true`, dead lineages are kept in the tree. If this is `false`, they are pruned from the tree.
cancellationToken	A CancellationToken used to cancel the simulation if it takes too long.

Returns

A TreeNode object containing the labelled birth-death tree, or `null` if all the lineages went extinct and *keep↩ DeadLineages* is `false`.

Definition at line 126 of file BirthDeathTree.cs.

7.5.2.2 LabelledTree() [2/3]

```
static TreeNode PhyloTree.TreeBuilding.BirthDeathTree.LabelledTree (
            int leafCount,
            double birthRate,
            double deathRate = 0,
            bool keepDeadLineages = false,
            TreeNode constraint = null,
            CancellationToken cancellationToken = default )  [static]
```

Simulate a labelled birth-death tree, stopping when the number of lineages that are alive in the tree reaches a certain value.

Parameters

leafCount	The final number of lineages that are alive in the tree (their names will be in the form `t1, t2, ..., tN`, where `N` is *leafCount*). Note that, if *keepDeadLineages* is `true`, the actual number of leaves in the tree may be larger than this, because the leaves corresponding to dead lineages are kept (their names will be empty). If all the lineages go extinct before the target number of lineages is reaced, the method will return a smaller tree (without any leaf names) if *keepDeadLineages* is `true`, or `null` if it is `false`.
birthRate	The birth rate of the tree.
deathRate	The death rate of the tree.
keepDeadLineages	If this is `true`, dead lineages are kept in the tree. If this is `false`, they are pruned from the tree.
constraint	A tree to constrain the sampling. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the tree, this will bias the sampled topology distribution.
cancellationToken	A CancellationToken used to cancel the simulation if it takes too long.

Returns

A TreeNode object containing the unlabelled birth-death tree, or `null` if all the lineages went extinct and *keepDeadLineages* is `false`.

Definition at line 489 of file BirthDeathTree.cs.

7.5.2.3 LabelledTree() [3/3]

```
static TreeNode PhyloTree.TreeBuilding.BirthDeathTree.LabelledTree (
            IReadOnlyList< string > leafNames,
            double birthRate,
            double deathRate = 0,
            bool keepDeadLineages = false,
            TreeNode constraint = null,
            CancellationToken cancellationToken = default )  [static]
```

Simulate a labelled birth-death tree, stopping when the number of lineages that are alive in the tree reaches a certain value.

Parameters

leafNames	The names for the terminal nodes of the tree. Note that, if *keepDeadLineages* is `true`, the actual number of leaves in the tree may be larger than this, because the leaves corresponding to dead lineages are kept (their names will be empty). If all the lineages go extinct before the target number of lineages is reaced, the method will return a smaller tree (without any leaf names) if *keepDeadLineages* is `true`, or `null` if it is `false`.
birthRate	The birth rate of the tree.
deathRate	The death rate of the tree.
keepDeadLineages	If this is `true`, dead lineages are kept in the tree. If this is `false`, they are pruned from the tree.
constraint	A tree to constrain the sampling. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the tree, this will bias the sampled topology distribution.
cancellationToken	A CancellationToken used to cancel the simulation if it takes too long.

Returns

A TreeNode object containing the unlabelled birth-death tree, or `null` if all the lineages went extinct and *keepDeadLineages* is `false`.

Definition at line 267 of file BirthDeathTree.cs.

7.5.2.4 UnlabelledTree() [1/2]

```
static TreeNode PhyloTree.TreeBuilding.BirthDeathTree.UnlabelledTree (
            double treeAge,
```

```
            double birthRate,
            double deathRate = 0,
            bool keepDeadLineages = false,
            CancellationToken cancellationToken = default )  [static]
```

Simulate an unlabelled birth-death tree, stopping when the age of the tree reaches a certain value.

Parameters

treeAge	The final age of the tree. Note that the actual age of the tree may be smaller than this; this can happen either if *keepDeadLineages* is `false` and one of the two clades descending from the root node goes exinct, or if *keepDeadLineages* is `true` and all the clades go extinct. If all the clades go extinct and *keepDeadLineages* is `false`, this method will return `null`.
birthRate	The birth rate of the tree.
deathRate	The death rate of the tree.
keepDeadLineages	If this is `true`, dead lineages are kept in the tree. If this is `false`, they are pruned from the tree.
cancellationToken	A CancellationToken used to cancel the simulation if it takes too long.

Returns

A TreeNode object containing the unlabelled birth-death tree, or `null` if all the lineages went extinct and *keepDeadLineages* is `false`.

Definition at line 29 of file BirthDeathTree.cs.

7.5.2.5 UnlabelledTree() [2/2]

```
static TreeNode PhyloTree.TreeBuilding.BirthDeathTree.UnlabelledTree (
            int leafCount,
            double birthRate,
            double deathRate = 0,
            bool keepDeadLineages = false,
            CancellationToken cancellationToken = default )  [static]
```

Simulate an unlabelled birth-death tree, stopping when the number of lineages that are alive in the tree reaches a certain value.

Parameters

leafCount	The final number of lineages that are alive in the tree. Note that, if *keepDeadLineages* is `true`, the actual number of leaves in the tree may be larger than this, because the leaves corresponding to dead lineages are kept. If all the lineages go extinct before the target number of lineages is reaced, the method will return a smaller tree if *keepDeadLineages* is `true`, or `null` if it is `false`.
birthRate	The birth rate of the tree.
deathRate	The death rate of the tree.
keepDeadLineages	If this is `true`, dead lineages are kept in the tree. If this is `false`, they are pruned from the tree.
cancellationToken	A CancellationToken used to cancel the simulation if it takes too long.

Returns

A TreeNode object containing the unlabelled birth-death tree, or `null` if all the lineages went extinct and *keepDeadLineages* is `false`.

Definition at line 164 of file BirthDeathTree.cs.

The documentation for this class was generated from the following file:

- TreeBuilding/BirthDeathTree.cs

7.6 PhyloTree.TreeBuilding.CoalescentTree Class Reference

Contains methods to simulate coalescent trees.

Static Public Member Functions

- static TreeNode UnlabelledTree (int leafCount)

  Simulate an unlabelled coalescent tree.
- static TreeNode LabelledTree (IReadOnlyList< string > leafNames, TreeNode constraint=null)

  Simulate a labelled coalescent tree with the supplied tip labels.
- static TreeNode LabelledTree (int leafCount, TreeNode constraint=null)

  Simulate a labelled coalescent tree with the specified number of terminal nodes.

7.6.1 Detailed Description

Contains methods to simulate coalescent trees.

Definition at line 12 of file CoalescentTree.cs.

7.6.2 Member Function Documentation

7.6.2.1 LabelledTree() [1/2]

```
static TreeNode PhyloTree.TreeBuilding.CoalescentTree.LabelledTree (
            int leafCount,
            TreeNode constraint = null )  [static]
```

Simulate a labelled coalescent tree with the specified number of terminal nodes.

Parameters

leafCount	The number of terminal nodes in the tree. Their names will be in the form `t1, t2, ..., tN`, where `N` is *leafCount* .
constraint	A tree to constrain the sampling. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the tree, this will bias the sampled topology distribution.

Returns

A TreeNode object containing the labelled coalescent tree.

Definition at line 212 of file CoalescentTree.cs.

7.6.2.2 LabelledTree() [2/2]

```
static TreeNode PhyloTree.TreeBuilding.CoalescentTree.LabelledTree (
            IReadOnlyList< string > leafNames,
            TreeNode constraint = null )  [static]
```

Simulate a labelled coalescent tree with the supplied tip labels.

Parameters

leafNames	The labels for the terminal nodes of the tree.
constraint	A tree to constrain the sampling. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the tree, this will bias the sampled topology distribution.

Returns

A TreeNode object containing the labelled coalescent tree.

Definition at line 38 of file CoalescentTree.cs.

7.6.2.3 UnlabelledTree()

```
static TreeNode PhyloTree.TreeBuilding.CoalescentTree.UnlabelledTree (
            int leafCount )  [static]
```

Simulate an unlabelled coalescent tree.

Parameters

leafCount	The number of terminal nodes in the tree.

Returns

A TreeNode object containing the unlabelled coalescent tree.

Definition at line 19 of file CoalescentTree.cs.

The documentation for this class was generated from the following file:

- TreeBuilding/CoalescentTree.cs

7.7 PhyloTree.TreeBuilding.DistanceMatrix Class Reference

Contains methods to compute distance matrices.

Static Public Member Functions

- static float[ ][ ] BuildFromAlignment (IReadOnlyList< byte[ ]> sequences, EvolutionModel evolution↩ Model=EvolutionModel.Kimura, int numCores=-1, Action< double > progressCallback=null)

  Build a distance matrix from aligned DNA sequences stored as a T:byte[ ] array where each byte corresponds to three positions.
- static float[ ][ ] BuildFromAlignment (IReadOnlyList< ushort[ ]> sequences, EvolutionModel evolution↩ Model=EvolutionModel.Kimura, int numCores=-1, Action< double > progressCallback=null)

  Build a distance matrix from aligned protein sequences stored as a T:ushort[ ] array where each ushort corresponds to two positions.
- static void BuildFromAlignment (float[ ][ ] allocatedMatrix, IReadOnlyList< byte[ ]> sequences, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=-1, Action< double > progressCallback=null)

  Build a distance matrix from aligned DNA sequences stored as a T:byte[ ] array where each byte corresponds to three positions.
- static void BuildFromAlignment (float[ ][ ] allocatedMatrix, IReadOnlyList< ushort[ ]> sequences, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=0, Action< double > progress↩ Callback=null)

  Build a distance matrix from aligned protein sequences stored as a T:ushort[ ] array where each ushort corresponds to two positions.
- static void BuildFromAlignment (float[ ][ ] allocatedMatrix, IReadOnlyList< string > sequences, AlignmentType alignmentType=AlignmentType.Autodetect, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a distance matrix from aligned DNA or protein sequences.
- static float[ ][ ] BuildFromAlignment (IReadOnlyList< string > sequences, AlignmentType alignment↩ Type=AlignmentType.Autodetect, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a distance matrix from aligned DNA or protein sequences.
- static void BootstrapReplicateFromAlignment (float[ ][ ] allocatedMatrix, IReadOnlyList< string > sequences, AlignmentType alignmentType=AlignmentType.Autodetect, EvolutionModel evolutionModel=Evolution↩ Model.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
- static float[ ][ ] BootstrapReplicateFromAlignment (IReadOnlyList< string > sequences, AlignmentType alignmentType=AlignmentType.Autodetect, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
- static float[ ][ ] BuildFromAlignment (Dictionary< string, string > alignment, AlignmentType alignment↩ Type=AlignmentType.Autodetect, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a distance matrix from aligned DNA or protein sequences.
- static void BuildFromAlignment (float[ ][ ] allocatedMatrix, Dictionary< string, string > alignment, AlignmentType alignmentType=AlignmentType.Autodetect, EvolutionModel evolutionModel=Evolution↩ Model.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a distance matrix from aligned DNA or protein sequences.
- static float[ ][ ] BootstrapReplicateFromAlignment (Dictionary< string, string > alignment, AlignmentType alignmentType=AlignmentType.Autodetect, EvolutionModel evolutionModel=EvolutionModel.Kimura, int numCores=0, Action< double > progressCallback=null)

  Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
- static void BootstrapReplicateFromAlignment (float[ ][ ] allocatedMatrix, Dictionary< string, string > alignment, AlignmentType alignmentType=AlignmentType.Autodetect, EvolutionModel evolution↩ Model=EvolutionModel.Kimura, int numCores=0, Action< double > progressCallback=null)

Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.

- static IEnumerable< byte[ ]> ConvertDNASequences (IEnumerable< string > sequences)

  Convert DNA sequences into T:byte[ ] arrays in which each byte corresponds to 3 positions.
- static List< byte[ ]> ConvertDNASequences (IReadOnlyList< string > sequences)

  Convert DNA sequences into T:byte[ ] arrays in which each byte corresponds to 3 positions.
- static IEnumerable< byte[ ]> BootstrapDNASequences (IEnumerable< string > sequences)

  Computes a bootstrap replicate of a DNA sequence alignment.
- static List< byte[ ]> BootstrapDNASequences (IReadOnlyList< string > sequences)

  Computes a bootstrap replicate of a DNA sequence alignment.
- static IEnumerable< ushort[ ]> ConvertProteinSequences (IEnumerable< string > sequences)

  Convert protein sequences into T:ushort[ ] arrays in which each ushort corresponds to 2 positions.
- static List< ushort[ ]> ConvertProteinSequences (IReadOnlyList< string > sequences)

  Convert protein sequences into T:ushort[ ] arrays in which each ushort corresponds to 2 positions.
- static IEnumerable< ushort[ ]> BootstrapProteinSequences (IEnumerable< string > sequences)

  Computes a bootstrap replicate of a protein sequence alignment.
- static List< ushort[ ]> BootstrapProteinSequences (IReadOnlyList< string > sequences)

  Computes a bootstrap replicate of a protein sequence alignment.
- static ushort[ ] ConvertProteinSequence (string sequence)

  Converts a protein sequence stored as a string into a T:ushort[ ] array. Each ushort contains 2 amino acid positions. The allowed symbols are the usual 20 1-letter amino acid abbreviations, plus U for Sec, O for Pyl, B for Asn or Asp, Z for Gln or Glu, J for Ile or Leu, X for any amino acid, ∗ for stop codons and - for gaps (uppercase and lowercase). All other characters are treated as gaps. If the sequence length is not a multiple of 2, it is padded with gaps.
- static float CompareProteinSequencesBLOSUM62 (ushort[ ] sequence1, ushort[ ] sequence2, int selfScore1, int selfScore2)

  Compares two protein sequences using the Scoredist correction with the BLOSUM62 matrix.

7.7.1 Detailed Description

Contains methods to compute distance matrices.

Definition at line 66 of file DistanceMatrix.cs.

7.7.2 Member Function Documentation

7.7.2.1 BootstrapDNASequences() [1/2]

```
static IEnumerable< byte[]> PhyloTree.TreeBuilding.DistanceMatrix.BootstrapDNASequences (
            IEnumerable< string > sequences )  [static]
```

Computes a bootstrap replicate of a DNA sequence alignment.

Parameters

sequences	The aligned DNA sequences.

Returns

An T:IEnumerable<byte[]> that, when enumerated, will contain the bootstrapped sequences, converted into T:byte[] arrays where each byte corresponds to 3 positions.

Exceptions

ArgumentOutOfRangeException	Thrown if not all of the sequences have the same length.

Definition at line 533 of file DistanceMatrix.cs.

7.7.2.2 BootstrapDNASequences() [2/2]

```
static List< byte[]> PhyloTree.TreeBuilding.DistanceMatrix.BootstrapDNASequences (
            IReadOnlyList< string > sequences )  [static]
```

Computes a bootstrap replicate of a DNA sequence alignment.

Parameters

sequences	The aligned DNA sequences.

Returns

An T:List<byte[]> that contains the bootstrapped sequences, converted into T:byte[] arrays where each byte corresponds to 3 positions.

Exceptions

ArgumentOutOfRangeException	Thrown if not all of the sequences have the same length.

Definition at line 562 of file DistanceMatrix.cs.

7.7.2.3 BootstrapProteinSequences() [1/2]

```
static IEnumerable< ushort[]> PhyloTree.TreeBuilding.DistanceMatrix.BootstrapProteinSequences
(
            IEnumerable< string > sequences )  [static]
```

Computes a bootstrap replicate of a protein sequence alignment.

Parameters

sequences	The aligned protein sequences.

Returns

An T:IEnumerable<ushort[]> that, when enumerated, will contain the bootstrapped sequences, converted into T:ushort[] arrays where each ushort corresponds to 2 positions.

Exceptions

ArgumentOutOfRangeException	Thrown if not all of the sequences have the same length.

Definition at line 596 of file DistanceMatrix.cs.

7.7.2.4 BootstrapProteinSequences() [2/2]

```
static List< ushort[]> PhyloTree.TreeBuilding.DistanceMatrix.BootstrapProteinSequences (
            IReadOnlyList< string > sequences )  [static]
```

Computes a bootstrap replicate of a protein sequence alignment.

Parameters

sequences	The aligned protein sequences.

Returns

A T:List<ushort[]> that contains the bootstrapped sequences, converted into T:ushort[] arrays where each ushort corresponds to 2 positions.

Exceptions

ArgumentOutOfRangeException	Thrown if not all of the sequences have the same length.

Definition at line 625 of file DistanceMatrix.cs.

7.7.2.5 BootstrapReplicateFromAlignment() [1/4]

```
static float[][] PhyloTree.TreeBuilding.DistanceMatrix.BootstrapReplicateFromAlignment (
            Dictionary< string, string > alignment,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.

Parameters

alignment	The aligned sequences. These will be resampled randomly.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the lower-triangular distance matrix.

Definition at line 462 of file DistanceMatrix.cs.

7.7.2.6 BootstrapReplicateFromAlignment() [2/4]

```
static void PhyloTree.TreeBuilding.DistanceMatrix.BootstrapReplicateFromAlignment (
            float allocatedMatrix[][],
            Dictionary< string, string > alignment,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.

Parameters

allocatedMatrix	A pre-allocated T:float[ ][ ] jagged array that will contain the lower-triangular distance matrix.
alignment	The aligned sequences. These will be resampled randomly.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Definition at line 476 of file DistanceMatrix.cs.

7.7.2.7 BootstrapReplicateFromAlignment() [3/4]

```
static void PhyloTree.TreeBuilding.DistanceMatrix.BootstrapReplicateFromAlignment (
            float allocatedMatrix[][],
            IReadOnlyList< string > sequences,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
```

```
int numCores = 0,
Action< double > progressCallback = null )  [static]
```

Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.

Parameters

allocatedMatrix	A pre-allocated T:float[ ][ ] jagged array that will contain the lower-triangular distance matrix.
sequences	The aligned sequences. These will be resampled randomly.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Definition at line 355 of file DistanceMatrix.cs.

7.7.2.8 BootstrapReplicateFromAlignment() [4/4]

```
static float[][] PhyloTree.TreeBuilding.DistanceMatrix.BootstrapReplicateFromAlignment (
            IReadOnlyList< string > sequences,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.

Parameters

sequences	The aligned sequences. These will be resampled randomly.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the lower-triangular distance matrix.

Definition at line 411 of file DistanceMatrix.cs.

7.7.2.9 BuildFromAlignment() [1/8]

```
static float[][] PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            Dictionary< string, string > alignment,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned DNA or protein sequences.

Parameters

alignment	The aligned sequences.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the lower-triangular distance matrix.

Definition at line 434 of file DistanceMatrix.cs.

7.7.2.10 BuildFromAlignment() [2/8]

```
static void PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            float allocatedMatrix[][],
            Dictionary< string, string > alignment,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned DNA or protein sequences.

Parameters

allocatedMatrix	A pre-allocated T:float[ ][ ] jagged array that will contain the lower-triangular distance matrix.
alignment	The aligned sequences.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Definition at line 448 of file DistanceMatrix.cs.

7.7.2.11 BuildFromAlignment() [3/8]

```
static void PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            float allocatedMatrix[][],
            IReadOnlyList< byte[]> sequences,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned DNA sequences stored as a T:byte[ ] array where each byte corresponds to three positions.

Parameters

allocatedMatrix	A pre-allocated T:float[ ][ ] jagged array that will contain the lower-triangular distance matrix.
sequences	The aligned sequences.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Definition at line 171 of file DistanceMatrix.cs.

7.7.2.12 BuildFromAlignment() [4/8]

```
static void PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            float allocatedMatrix[][],
            IReadOnlyList< string > sequences,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned DNA or protein sequences.

Parameters

allocatedMatrix	A pre-allocated T:float[ ][ ] jagged array that will contain the lower-triangular distance matrix.
sequences	The aligned sequences.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Definition at line 252 of file DistanceMatrix.cs.

7.7.2.13 BuildFromAlignment() [5/8]

```
static void PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            float allocatedMatrix[][],
            IReadOnlyList< ushort[]> sequences,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned protein sequences stored as a T:ushort[ ] array where each ushort corresponds to two positions.

Parameters

allocatedMatrix	A pre-allocated T:float[ ][ ] jagged array that will contain the lower-triangular distance matrix.
sequences	The aligned sequences.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Definition at line 211 of file DistanceMatrix.cs.

7.7.2.14 BuildFromAlignment() [6/8]

```
static float[][] PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            IReadOnlyList< byte[]> sequences,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned DNA sequences stored as a T:byte[ ] array where each byte corresponds to three positions.

Parameters

sequences	The aligned sequences.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the lower-triangular distance matrix.

Definition at line 127 of file DistanceMatrix.cs.

7.7.2.15 BuildFromAlignment() [7/8]

```
static float[][] PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            IReadOnlyList< string > sequences,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = 0,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned DNA or protein sequences.

Parameters

sequences	The aligned sequences.
alignmentType	The type of sequences in the alignment.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the lower-triangular distance matrix.

Definition at line 332 of file DistanceMatrix.cs.

7.7.2.16 BuildFromAlignment() [8/8]

```
static float[][] PhyloTree.TreeBuilding.DistanceMatrix.BuildFromAlignment (
            IReadOnlyList< ushort[]> sequences,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Build a distance matrix from aligned protein sequences stored as a T:ushort[ ] array where each ushort corresponds to two positions.

Parameters

sequences	The aligned sequences.
evolutionModel	The evolutionary model to use to compute the distance matrix.
numCores	The maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the lower-triangular distance matrix.

Definition at line 149 of file DistanceMatrix.cs.

7.7.2.17 CompareProteinSequencesBLOSUM62()

```
static float PhyloTree.TreeBuilding.DistanceMatrix.CompareProteinSequencesBLOSUM62 (
            ushort[] sequence1,
            ushort[] sequence2,
            int selfScore1,
            int selfScore2 )  [static]
```

Compares two protein sequences using the Scoredist correction with the BLOSUM62 matrix.

Parameters

sequence1	The first sequence.
sequence2	The second sequence.
selfScore1	The score obtained by comparing *sequence1* with itself.
selfScore2	The score obtained by comparing *sequence2* with itself.

Returns

The distance between the two sequences.

Definition at line 779 of file DistanceMatrix.Protein.cs.

7.7.2.18 ConvertDNASequences() [1/2]

```
static IEnumerable< byte[]> PhyloTree.TreeBuilding.DistanceMatrix.ConvertDNASequences (
            IEnumerable< string > sequences )  [static]
```

Convert DNA sequences into T:byte[] arrays in which each byte corresponds to 3 positions.

Parameters

sequences	The sequences to convert.

Returns

An T:IEnumerable<byte[]> that, when enumerated, will contain the converted sequences.

Definition at line 486 of file DistanceMatrix.cs.

7.7.2.19 ConvertDNASequences() [2/2]

```
static List< byte[]> PhyloTree.TreeBuilding.DistanceMatrix.ConvertDNASequences (
            IReadOnlyList< string > sequences )  [static]
```

Convert DNA sequences into T:byte[] arrays in which each byte corresponds to 3 positions.

Parameters

sequences	The sequences to convert.

Returns

A T:List<byte[]> that contains the converted sequences.

Definition at line 499 of file DistanceMatrix.cs.

7.7.2.20 ConvertProteinSequence()

```
static ushort[] PhyloTree.TreeBuilding.DistanceMatrix.ConvertProteinSequence (
            string sequence )  [static]
```

Converts a protein sequence stored as a string into a T:ushort[] array. Each ushort contains 2 amino acid positions. The allowed symbols are the usual 20 1-letter amino acid abbreviations, plus U for Sec, O for Pyl, B for Asn or Asp, Z for Gln or Glu, J for Ile or Leu, X for any amino acid, * for stop codons and - for gaps (uppercase and lowercase). All other characters are treated as gaps. If the sequence length is not a multiple of 2, it is padded with gaps.

Parameters

sequence	The sequence to convert.

Returns

A T:ushort[] array representing the sequence.

Definition at line 53 of file DistanceMatrix.Protein.cs.

7.7.2.21 ConvertProteinSequences() [1/2]

```
static IEnumerable< ushort[]> PhyloTree.TreeBuilding.DistanceMatrix.ConvertProteinSequences (
            IEnumerable< string > sequences )  [static]
```

Convert protein sequences into T:ushort[] arrays in which each ushort corresponds to 2 positions.

Parameters

sequences	The sequences to convert.

Returns

An T:IEnumerable<ushort[]> that, when enumerated, will contain the converted sequences.

Definition at line 572 of file DistanceMatrix.cs.

7.7.2.22 ConvertProteinSequences() [2/2]

```
static List< ushort[]> PhyloTree.TreeBuilding.DistanceMatrix.ConvertProteinSequences (
            IReadOnlyList< string > sequences )  [static]
```

Convert protein sequences into T:ushort[] arrays in which each ushort corresponds to 2 positions.

Parameters

sequences	The sequences to convert.

Returns

A T:List<ushort[]> that contains the converted sequences.

Definition at line 585 of file DistanceMatrix.cs.

The documentation for this class was generated from the following files:

- TreeBuilding/DistanceMatrix.cs
- TreeBuilding/DistanceMatrix.Protein.cs
- TreeBuilding/DistanceMatrix.DNA.cs

7.8 PhyloTree.SequenceSimulation.RateMatrix.DNA Class Reference

Contains rate matrices for DNA sequence evolution.

Static Public Member Functions

- static ImmutableRateMatrix K80Matrix (double kappa)

  Kimura 2-parameter DNA sequence evolution matrix, from Kimura 1980.
- static ImmutableRateMatrix HKY85Matrix (double kappa, double piA, double piC, double piG, double piT)

  Hasegawa-Kishino-Yano DNA sequence evolution matrix, from Hasegawa, Kishino & Yano, 1985.
- static ImmutableRateMatrix GTRMatrix (double AC, double AG, double AT, double CG, double CT, double GT, double piA, double piC, double piG, double piT)

  General Time Reversible DNA Evolution matrix, from Tavaré, 1986.

Static Public Attributes

- static readonly ImmutableRateMatrix JC69Matrix

  DNA sequence evolution matrix from Jukes & Cantor, 1969.

7.8.1 Detailed Description

Contains rate matrices for DNA sequence evolution.

Definition at line 13 of file RateMatrices.cs.

7.8.2 Member Function Documentation

7.8.2.1 GTRMatrix()

```
static ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.DNA.GTRMatrix (
            double AC,
            double AG,
            double AT,
            double CG,
            double CT,
            double GT,
            double piA,
            double piC,
            double piG,
            double piT )  [static]
```

General Time Reversible DNA Evolution matrix, from Tavaré, 1986.

Parameters

AC	The weight for A-C and C-A transversions.
AG	The weight for A-G and G-A transitions.
AT	The weight for A-T and T-A transversions.
CG	The weight for C-G and G-C transversions.
CT	The weight for C-T and T-C transitions.
GT	The weight for G-T and T-G transversions.
piA	The equilibrium frequency for A.
piC	The equilibrium frequency for C.
piG	The equilibrium frequency for G.
piT	The equilibrium frequency for T.

Returns

The GTR DNA sequence evolution matrix with the specified parameters.

Definition at line 79 of file RateMatrices.cs.

7.8.2.2 HKY85Matrix()

```
static ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.DNA.HKY85Matrix (
            double kappa,
            double piA,
            double piC,
            double piG,
            double piT )  [static]
```

Hasegawa-Kishino-Yano DNA sequence evolution matrix, from Hasegawa, Kishino & Yano, 1985.

Parameters

kappa	The transition/transversion rate ratio.
piA	The equilibrium frequency for A.
piC	The equilibrium frequency for C.
piG	The equilibrium frequency for G.
piT	The equilibrium frequency for C.

Returns

The HKY85 DNA sequence evolution matrix with the specified parameters.

See also .

Definition at line 48 of file RateMatrices.cs.

7.8.2.3 K80Matrix()

```
static ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.DNA.K80Matrix (
            double kappa )  [static]
```

Kimura 2-parameter DNA sequence evolution matrix, from Kimura 1980.

Parameters

kappa	The transition/transversion rate ratio.

Returns

The Kimura 2-parameter DNA sequence evolution matrix with the specified transition/transversion rate ratio.

See also .

Definition at line 28 of file RateMatrices.cs.

7.8.3 Member Data Documentation

7.8.3.1 JC69Matrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.DNA.JC69Matrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[4] { 'A', 'C', 'G', 'T' },
                new double[4, 4] { { -0.999999999, 0.333333333, 0.333333333, 0.333333333 }, { 0.333333333,
      -0.999999999, 0.333333333, 0.333333333 }, { 0.333333333, 0.333333333, -0.999999999, 0.333333333 }, {
      0.333333333, 0.333333333, 0.333333333, -0.999999999 } },
                new double[4] { 0.25, 0.25, 0.25, 0.25 })
```

DNA sequence evolution matrix from Jukes & Cantor, 1969.

Definition at line 18 of file RateMatrices.cs.

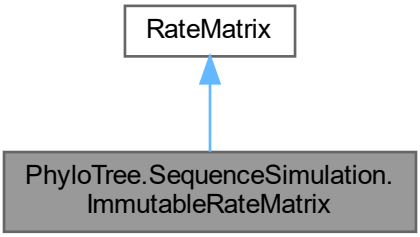
The documentation for this class was generated from the following file:

- SequenceSimulation/RateMatrices.cs

7.9 PhyloTree.SequenceSimulation.ImmutableRateMatrix Class Reference

Represents a rate matrix whose values cannot be changed after initialisation.

Inheritance diagram for PhyloTree.SequenceSimulation.ImmutableRateMatrix:



Public Member Functions

- ImmutableRateMatrix (ReadOnlySpan< char > states, double[,] rates)

  Creates a new ImmutableRateMatrix with the specified states and rates .
- ImmutableRateMatrix (ReadOnlySpan< char > states, double[,] rates, double[ ] equilibriumFrequencies)

  Creates a new ImmutableRateMatrix with the specified states and rates .

Properties

- override ImmutableArray< char > States `[get]`

  Gets the states for the character to which the rate matrix applies.
- override ImmutableArray< double > EquilibriumFrequencies `[get]`

  Gets the equilibrium frequences of the rate matrix.
- override double this[int from, int to] `[get]`

  Gets the rate of going from state number from to state number to . If `from == to` , the negative sum of the elements on the row is returned.
- override double this[char from, char to] `[get]`

  Gets the rate of going from state from to state to . If `from == to` , the negative sum of the elements on the row is returned.

7.9.1 Detailed Description

Represents a rate matrix whose values cannot be changed after initialisation.

Definition at line 371 of file RateMatix.cs.

7.9.2 Constructor & Destructor Documentation

7.9.2.1 ImmutableRateMatrix() [1/2]

```
PhyloTree.SequenceSimulation.ImmutableRateMatrix.ImmutableRateMatrix (
            ReadOnlySpan< char > states,
            double rates[,] )
```

Creates a new ImmutableRateMatrix with the specified *states* and *rates* .

Parameters

states	The possible states of the character described by the ImmutableRateMatrix.
rates	A 2D double> array containing the rates used to initialise the matrix. The number of rows and columns in the array must be equal to the number of states. Diagonal entries are ignored.

Exceptions

ArgumentException	Thrown if the number of rows or columns of the *rates* matrix does not correspond to the number of *states* .

Definition at line 458 of file RateMatix.cs.

7.9.2.2 ImmutableRateMatrix() [2/2]

```
PhyloTree.SequenceSimulation.ImmutableRateMatrix.ImmutableRateMatrix (
            ReadOnlySpan< char > states,
            double rates[,],
            double[] equilibriumFrequencies )
```

Creates a new ImmutableRateMatrix with the specified *states* and *rates* .

Parameters

states	The possible states of the character described by the ImmutableRateMatrix.
rates	A 2D double> array containing the rates used to initialise the matrix. The number of rows and columns in the array must be equal to the number of states. Diagonal entries are ignored.
equilibriumFrequencies	Equilibrium frequencies for the rate matrix. These are not checked, so they better be correct!

Exceptions

ArgumentException	Thrown if the number of rows or columns of the *rates* matrix, or the number of equilibrium frequencies, does not correspond to the number of *states* .

Using this constructor is faster than the ImmutableRateMatrix(ReadOnlySpan<char>, double[,]) constructor, as equilbrium frequencies are not computed. This is especially useful for pre-baked rate matrices.

Definition at line 511 of file RateMatix.cs.

7.9.3 Property Documentation

7.9.3.1 EquilibriumFrequencies

```
override ImmutableArray<double> PhyloTree.SequenceSimulation.ImmutableRateMatrix.Equilibrium↩
Frequencies [get]
```

Gets the equilibrium frequences of the rate matrix.

Definition at line 381 of file RateMatix.cs.

7.9.3.2 States

```
override ImmutableArray<char> PhyloTree.SequenceSimulation.ImmutableRateMatrix.States [get]
```

Gets the states for the character to which the rate matrix applies.

Definition at line 376 of file RateMatix.cs.

7.9.3.3 this[char from, char to]

```
override double PhyloTree.SequenceSimulation.ImmutableRateMatrix.this[char from, char to]
[get]
```

Gets the rate of going from state *from* to state *to* . If `from == to` , the negative sum of the elements on the row is returned.

Parameters

from	The row state.
to	The column state.

Returns

The rate of going from state *from* to state *to* .

Exceptions

ArgumentOutOfRangeException	Thrown if the state is not part of the rate matrix.

Definition at line 429 of file RateMatix.cs.

7.9.3.4 this[int from, int to]

```
override double PhyloTree.SequenceSimulation.ImmutableRateMatrix.this[int from, int to]  [get]
```

Gets the rate of going from state number *from* to state number *to* . If `from == to` , the negative sum of the elements on the row is returned.

Parameters

from	The row number.
to	The column number.

Returns

The rate of going from state number *from* to state number *to* .

Exceptions

ArgumentOutOfRangeException	Thrown if the state index is $<$ 0 or greater than the number of states in the rate matrix.

Definition at line 397 of file RateMatix.cs.

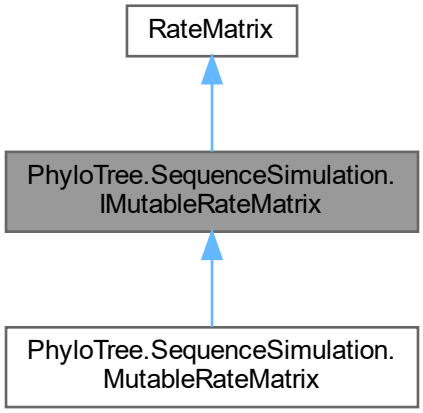
The documentation for this class was generated from the following file:

- SequenceSimulation/RateMatix.cs

7.10 PhyloTree.SequenceSimulation.IMutableRateMatrix Class Reference

Represents a rate matrix whose values can be changed after initialisation.

Inheritance diagram for PhyloTree.SequenceSimulation.IMutableRateMatrix:



Properties

- sealed override double this[int from, int to] `[get]`

  Gets the rate of going from state number from to state number to . If `from == to` , the negative sum of the elements on the row is returned.
- sealed override double this[char from, char to] `[get]`

  Gets the rate of going from state from to state to . If `from == to` , the negative sum of the elements on the row is returned.

7.10.1 Detailed Description

Represents a rate matrix whose values can be changed after initialisation.

Definition at line 61 of file RateMatix.cs.

7.10.2 Property Documentation

7.10.2.1 this[char from, char to]

```
sealed override double PhyloTree.SequenceSimulation.IMutableRateMatrix.this[char from, char
to]  [get]
```

Gets the rate of going from state *from* to state *to* . If `from == to` , the negative sum of the elements on the row is returned.

Parameters

from	The row state.
to	The column state.

Returns

The rate of going from state *from* to state *to* .

Exceptions

ArgumentOutOfRangeException	Thrown if the state is not part of the rate matrix.

Definition at line 95 of file RateMatix.cs.

7.10.2.2 this[int from, int to]

```
sealed override double PhyloTree.SequenceSimulation.IMutableRateMatrix.this[int from, int to]
[get]
```

Gets the rate of going from state number *from* to state number *to* . If `from == to` , the negative sum of the elements on the row is returned.

Parameters

from	The row number.
to	The column number.

Returns

The rate of going from state number *from* to state number *to* .

Definition at line 72 of file RateMatix.cs.

The documentation for this class was generated from the following file:

- SequenceSimulation/RateMatix.cs

7.11 PhyloTree.SequenceSimulation.IndelModel Class Reference

Represents a model for sequence insertion/deletion.

Public Member Functions

- IndelModel (double insertionRate, double deletionRate, IDiscreteDistribution insertionSizeDistribution, IDiscreteDistribution deletionSizeDistribution)

  Creates a new IndelModel with the specified insertion rate, deletion rate, insertion size distribution and deletion size distribution.
- IndelModel (double indelRate, IDiscreteDistribution indelSizeDistribution)

  Creates a new IndelModel with the specified rate and size distribution for insertions and deletions.
- IndelModel (double insertionRate, double deletionRate, IDiscreteDistribution indelSizeDistribution)

  Creates a new IndelModel with the specified insertion rate, deletion rate, and size distribution for insertions and deletions.
- IndelModel (double indelRate, IDiscreteDistribution insertionSizeDistribution, IDiscreteDistribution deletion↩ SizeDistribution)

  Creates a new IndelModel with the specified rate for insertions and deletions, insertion size distribution and deletion size distribution.

Properties

- double InsertionRate `[get]`

  The rate of insertions, expressed as a multiple of the rate of sequence mutation.
- double DeletionRate `[get]`

  The rate of deletions, expressed as a multiple of the rate of sequence mutation.
- IDiscreteDistribution InsertionSizeDistribution `[get]`

  The size distribution for insertions.
- IDiscreteDistribution DeletionSizeDistribution `[get]`

  The size distribution for deletions.

7.11.1 Detailed Description

Represents a model for sequence insertion/deletion.

Definition at line 8 of file IndelModel.cs.

7.11.2 Constructor & Destructor Documentation

7.11.2.1 IndelModel() [1/4]

```
PhyloTree.SequenceSimulation.IndelModel.IndelModel (
            double insertionRate,
            double deletionRate,
            IDiscreteDistribution insertionSizeDistribution,
            IDiscreteDistribution deletionSizeDistribution )
```

Creates a new IndelModel with the specified insertion rate, deletion rate, insertion size distribution and deletion size distribution.

Parameters

insertionRate	The insertion rate, expressed as a multiple of the rate of sequence mutation.
deletionRate	The deletion rate, expressed as a multiple of the rate of sequence mutation.
insertionSizeDistribution	The size distribution for insertions.
deletionSizeDistribution	The size distribution for deletions.

Definition at line 37 of file IndelModel.cs.

7.11.2.2 IndelModel() [2/4]

```
PhyloTree.SequenceSimulation.IndelModel.IndelModel (
            double indelRate,
            IDiscreteDistribution indelSizeDistribution )
```

Creates a new IndelModel with the specified rate and size distribution for insertions and deletions.

Parameters

indelRate	The insertion/deletion rate, expressed as a multiple of the rate of sequence mutation.
indelSizeDistribution	The size distribution for insertions and deletions.

Definition at line 50 of file IndelModel.cs.

7.11.2.3 IndelModel() [3/4]

```
PhyloTree.SequenceSimulation.IndelModel.IndelModel (
            double insertionRate,
            double deletionRate,
            IDiscreteDistribution indelSizeDistribution )
```

Creates a new IndelModel with the specified insertion rate, deletion rate, and size distribution for insertions and deletions.

Parameters

insertionRate	The insertion rate, expressed as a multiple of the rate of sequence mutation.
deletionRate	The deletion rate, expressed as a multiple of the rate of sequence mutation.
indelSizeDistribution	The size distribution for insertions and deletions.

Definition at line 58 of file IndelModel.cs.

7.11.2.4 IndelModel() [4/4]

```
PhyloTree.SequenceSimulation.IndelModel.IndelModel (
            double indelRate,
            IDiscreteDistribution insertionSizeDistribution,
            IDiscreteDistribution deletionSizeDistribution )
```

Creates a new IndelModel with the specified rate for insertions and deletions, insertion size distribution and deletion size distribution.

Parameters

indelRate	The insertion/deletion rate, expressed as a multiple of the rate of sequence mutation.
insertionSizeDistribution	The size distribution for insertions.
deletionSizeDistribution	The size distribution for deletions.

Definition at line 66 of file IndelModel.cs.

7.11.3 Property Documentation

7.11.3.1 DeletionRate

```
double PhyloTree.SequenceSimulation.IndelModel.DeletionRate  [get]
```

The rate of deletions, expressed as a multiple of the rate of sequence mutation.

Definition at line 18 of file IndelModel.cs.

7.11.3.2 DeletionSizeDistribution

```
IDiscreteDistribution PhyloTree.SequenceSimulation.IndelModel.DeletionSizeDistribution  [get]
```

The size distribution for deletions.

Definition at line 28 of file IndelModel.cs.

7.11.3.3 InsertionRate

```
double PhyloTree.SequenceSimulation.IndelModel.InsertionRate  [get]
```

The rate of insertions, expressed as a multiple of the rate of sequence mutation.

Definition at line 13 of file IndelModel.cs.

7.11.3.4 InsertionSizeDistribution

```
IDiscreteDistribution PhyloTree.SequenceSimulation.IndelModel.InsertionSizeDistribution [get]
```

The size distribution for insertions.

Definition at line 23 of file IndelModel.cs.

The documentation for this class was generated from the following file:

- SequenceSimulation/IndelModel.cs

7.12 PhyloTree.SequenceSimulation.Insertion Struct Reference

Represents an insertion event.

Public Member Functions

- Insertion (int start, int length)
  Creates a new Insertion

Properties

- int Start [get]
  The position in the ancestral sequence at which the insertion occurred.
- int Length [get]
  The length of the insertion.
- int End [get]
  The end of the insertion in the new sequence.

7.12.1 Detailed Description

Represents an insertion event.

Definition at line 72 of file IndelModel.cs.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 Insertion()

```
PhyloTree.SequenceSimulation.Insertion.Insertion (
            int start,
            int length )
```

Creates a new Insertion

Parameters

start	The position in the ancestral sequence at which the insertion occurred.
length	The length of the insertion.

Definition at line 94 of file IndelModel.cs.

7.12.3 Property Documentation

7.12.3.1 End

```
int PhyloTree.SequenceSimulation.Insertion.End  [get]
```

The end of the insertion in the new sequence.

Definition at line 87 of file IndelModel.cs.

7.12.3.2 Length

```
int PhyloTree.SequenceSimulation.Insertion.Length  [get]
```

The length of the insertion.

Definition at line 82 of file IndelModel.cs.

7.12.3.3 Start

```
int PhyloTree.SequenceSimulation.Insertion.Start  [get]
```

The position in the ancestral sequence at which the insertion occurred.

Definition at line 77 of file IndelModel.cs.

The documentation for this struct was generated from the following file:

- SequenceSimulation/IndelModel.cs

7.13 PhyloTree.SequenceScores.LikelihoodScores Class Reference

Contains methods to compute likelihood scores on a tree.

Static Public Member Functions

- static double GetLogLikelihood (this TreeNode tree, Dictionary< string, char > tipStates, RateMatrix rate↩ Matrix, double rate)

  Computes the likelihood for the specified character using the specified rate matrix and evolutionary rate.
- static double double logLikelihood GetLogLikelihood (this TreeNode tree, Dictionary< string, char > tipStates, RateMatrix rateMatrix)
- static double[ ] GetLogLikelihoods (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tip↩ States, RateMatrix rateMatrix, double rate, int maxParallelism=-1)

  Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the likelihood for each site.
- static double GetLogLikelihood (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tipStates, RateMatrix rateMatrix, double rate, int maxParallelism=-1)

  Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the total likelihood for the tree.
- static double[ ] GetLogLikelihoods (this TreeNode tree, Dictionary< string, Sequence > tipStates, RateMatrix rateMatrix, double rate, int maxParallelism)

  Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the likelihood for each site.
- static double GetLogLikelihood (this TreeNode tree, Dictionary< string, Sequence > tipStates, RateMatrix rateMatrix, double rate, int maxParallelism=-1)

  Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the total likelihood for the tree.
- static double[ ] GetLogLikelihoods (this TreeNode tree, Dictionary< string, string > tipStates, RateMatrix rateMatrix, double rate, int maxParallelism=-1)

  Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the likelihood for each site.
- static double GetLogLikelihood (this TreeNode tree, Dictionary< string, string > tipStates, RateMatrix rate↩ Matrix, double rate, int maxParallelism=-1)

  Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the total likelihood for the tree.
- static double double[ ] logLikelihoods GetLogLikelihoods (this TreeNode tree, Dictionary< string, IRead↩ OnlyList< char > > tipStates, RateMatrix rateMatrix, int maxParallelism=-1)
- static double double logLikelihood GetLogLikelihood (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tipStates, RateMatrix rateMatrix, int maxParallelism=-1)
- static double double[ ] logLikelihoods GetLogLikelihoods (this TreeNode tree, Dictionary< string, Sequence > tipStates, RateMatrix rateMatrix, int maxParallelism=-1)
- static double double logLikelihood GetLogLikelihood (this TreeNode tree, Dictionary< string, Sequence > tipStates, RateMatrix rateMatrix, int maxParallelism=-1)
- static double double[ ] logLikelihoods GetLogLikelihoods (this TreeNode tree, Dictionary< string, string > tipStates, RateMatrix rateMatrix, int maxParallelism=-1)
- static double double logLikelihood GetLogLikelihood (this TreeNode tree, Dictionary< string, string > tip↩ States, RateMatrix rateMatrix, int maxParallelism=-1)

Static Public Attributes

- static double rateMLE

  Estimates the maximum-likelihood evolutionary rate for the specified character under the specified rate matrix and computes the likelihood.

7.13.1 Detailed Description

Contains methods to compute likelihood scores on a tree.

Definition at line 17 of file LikelihoodScores.cs.

7.13.2 Member Function Documentation

7.13.2.1 GetLogLikelihood() [1/8]

```
static double double logLikelihood PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood
(
            this TreeNode tree,
            Dictionary< string, char > tipStates,
            RateMatrix rateMatrix )  [static]
```

Definition at line 255 of file LikelihoodScores.cs.

7.13.2.2 GetLogLikelihood() [2/8]

```
static double PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood (
            this TreeNode tree,
            Dictionary< string, char > tipStates,
            RateMatrix rateMatrix,
            double rate )  [static]
```

Computes the likelihood for the specified character using the specified rate matrix and evolutionary rate.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The character states at the tips of the tree. This Dictionary<String, Char> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.

Returns

The log-likelihood for the specified character.

Exceptions

MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 164 of file LikelihoodScores.cs.

7.13.2.3 GetLogLikelihood() [3/8]

```
static double PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood (
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            RateMatrix rateMatrix,
            double rate,
            int maxParallelism = -1 )  [static]
```

Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the total likelihood for the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The log-likelihood for the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 477 of file LikelihoodScores.cs.

7.13.2.4 GetLogLikelihood() [4/8]

```
static double double logLikelihood PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood
(
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            RateMatrix rateMatrix,
            int maxParallelism = -1 )  [static]
```

Definition at line 709 of file LikelihoodScores.cs.

7.13.2.5 GetLogLikelihood() [5/8]

```
static double PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood (
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            RateMatrix rateMatrix,
            double rate,
            int maxParallelism = -1 )  [static]
```

Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the total likelihood for the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The log-likelihood for the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 520 of file LikelihoodScores.cs.

7.13.2.6 GetLogLikelihood() [6/8]

```
static double double logLikelihood PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood
(
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            RateMatrix rateMatrix,
            int maxParallelism = -1 )  [static]
```

Definition at line 751 of file LikelihoodScores.cs.

7.13.2.7 GetLogLikelihood() [7/8]

```
static double PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood (
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            RateMatrix rateMatrix,
            double rate,
            int maxParallelism = -1 )  [static]
```

Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the total likelihood for the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The log-likelihood for the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 562 of file LikelihoodScores.cs.

7.13.2.8 GetLogLikelihood() [8/8]

```
static double double logLikelihood PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihood
(
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            RateMatrix rateMatrix,
            int maxParallelism = -1 )  [static]
```

Definition at line 794 of file LikelihoodScores.cs.

7.13.2.9 GetLogLikelihoods() [1/6]

```
static double[] PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihoods (
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            RateMatrix rateMatrix,
            double rate,
            int maxParallelism = -1 )  [static]
```

Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the likelihood for each site.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

A T:double[] array containing the log-likelihood for each site in the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 357 of file LikelihoodScores.cs.

7.13.2.10 GetLogLikelihoods() [2/6]

```
static double double[] logLikelihoods PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihoods
(
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            RateMatrix rateMatrix,
            int maxParallelism = -1 )  [static]
```

Definition at line 583 of file LikelihoodScores.cs.

7.13.2.11 GetLogLikelihoods() [3/6]

```
static double[] PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihoods (
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            RateMatrix rateMatrix,
            double rate,
            int maxParallelism )  [static]
```

Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the likelihood for each site.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

A T:double[] array containing the log-likelihood for each site in the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 499 of file LikelihoodScores.cs.

7.13.2.12 GetLogLikelihoods() [4/6]

```
static double double[] logLikelihoods PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihoods
(
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            RateMatrix rateMatrix,
            int maxParallelism = -1 )  [static]
```

Definition at line 731 of file LikelihoodScores.cs.

7.13.2.13 GetLogLikelihoods() [5/6]

```
static double[] PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihoods (
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            RateMatrix rateMatrix,
            double rate,
            int maxParallelism = -1 )  [static]
```

Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and evolutionary rate, and returns the likelihood for each site.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
rate	The overall evolutionary rate of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

A T:double[] array containing the log-likelihood for each site in the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 541 of file LikelihoodScores.cs.

7.13.2.14 GetLogLikelihoods() [6/6]

```
static double double[] logLikelihoods PhyloTree.SequenceScores.LikelihoodScores.GetLogLikelihoods
(
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            RateMatrix rateMatrix,
            int maxParallelism = -1 )  [static]
```

Definition at line 774 of file LikelihoodScores.cs.

7.13.3 Member Data Documentation

7.13.3.1 rateMLE

```
static double PhyloTree.SequenceScores.LikelihoodScores.rateMLE  [static]
```

Estimates the maximum-likelihood evolutionary rate for the specified character under the specified rate matrix and computes the likelihood.

Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the specified rate matrix and returns the total likelihood for the alignment.

Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the specified rate matrix and computes the likelihood.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The character states at the tips of the tree. This Dictionary<String, Char> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state.
rateMatrix	The rate matrix describing the evolution of the character.

Returns

The maximum-likelihood rate estimate and the log-likelihood for the specified character.

Exceptions

MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

A T:double[ ] array containing the log-likelihood for each site in the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The log-likelihood for the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

A T:double[ ] array containing the log-likelihood for each site in the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The log-likelihood for the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

A T:double[ ] array containing the log-likelihood for each site in the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Parameters

tree	The tree on which the likelihood should be computed.
tipStates	The aligned sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
rateMatrix	The rate matrix describing the evolution of the character.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The log-likelihood for the alignment.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Definition at line 255 of file LikelihoodScores.cs.

The documentation for this class was generated from the following file:

- SequenceScores/LikelihoodScores.cs

7.14 PhyloTree.SequenceScores.MissingDataException Class Reference

Exception that is thrown when not enough data has been supplied.

Inheritance diagram for PhyloTree.SequenceScores.MissingDataException:

Public Member Functions

- MissingDataException (string message)

7.14.1 Detailed Description

Exception that is thrown when not enough data has been supplied.

Definition at line 15 of file ParsimonyScore.cs.

7.14.2 Constructor & Destructor Documentation

7.14.2.1 MissingDataException()

```
PhyloTree.SequenceScores.MissingDataException.MissingDataException (
            string message )
```

Definition at line 18 of file ParsimonyScore.cs.

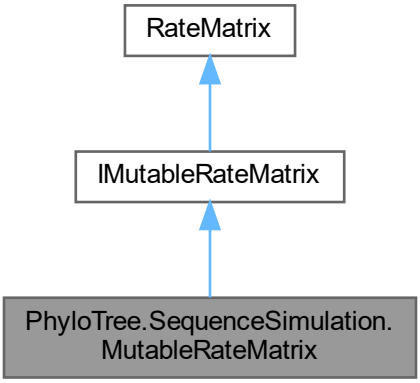
The documentation for this class was generated from the following file:

- SequenceScores/ParsimonyScore.cs

7.15 PhyloTree.SequenceSimulation.MutableRateMatrix Class Reference

Represents a rate matrix whose values can be changed after initialisation.

Inheritance diagram for PhyloTree.SequenceSimulation.MutableRateMatrix:



Public Member Functions

- MutableRateMatrix (ReadOnlySpan< char > states)

  Creates a new MutableRateMatrix with the specified states .
- MutableRateMatrix (ReadOnlySpan< char > states, double[,] rates)

  Creates a new MutableRateMatrix with the specified states and rates .

Properties

- override ImmutableArray< char > States `[get]`

  Gets the states for the character to which the rate matrix applies.
- new double this[int from, int to] `[get, set]`

  Gets or sets the rate of going from state number from to state number to . If `from == to` , the negative sum of the elements on the row is returned, but this value cannot be set.
- new double this[char from, char to] `[get, set]`

  Gets or sets the rate of going from state from to state to . If `from == to` , the negative sum of the elements on the row is returned, but this value cannot be set.
- override ImmutableArray< double > EquilibriumFrequencies `[get]`

  Gets the equilibrium frequences of the rate matrix.

7.15.1 Detailed Description

Represents a rate matrix whose values can be changed after initialisation.

Definition at line 114 of file RateMatrix.cs.

7.15.2 Constructor & Destructor Documentation

7.15.2.1 MutableRateMatrix() [1/2]

```
PhyloTree.SequenceSimulation.MutableRateMatrix.MutableRateMatrix (
            ReadOnlySpan< char > states )
```

Creates a new MutableRateMatrix with the specified *states* .

Parameters

states	The possible states of the character described by the MutableRateMatrix.

Definition at line 297 of file RateMatrix.cs.

7.15.2.2 MutableRateMatrix() [2/2]

```
PhyloTree.SequenceSimulation.MutableRateMatrix.MutableRateMatrix (
            ReadOnlySpan< char > states,
            double rates[,] )
```

Creates a new MutableRateMatrix with the specified *states* and *rates* .

Parameters

states	The possible states of the character described by the MutableRateMatrix.
rates	A 2D double> array containing the rates used to initialise the matrix. The number of rows and columns in the array must be equal to the number of states. Diagonal entries are ignored.

Exceptions

ArgumentException	Thrown if the number of rows or columns of the *rates* matrix does not correspond to the number of *states* .

Definition at line 313 of file RateMatrix.cs.

7.15.3 Property Documentation

7.15.3.1 EquilibriumFrequencies

```
override ImmutableArray<double> PhyloTree.SequenceSimulation.MutableRateMatrix.Equilibrium↩
Frequencies [get]
```

Gets the equilibrium frequences of the rate matrix.

Definition at line 264 of file RateMatix.cs.

7.15.3.2 States

```
override ImmutableArray<char> PhyloTree.SequenceSimulation.MutableRateMatrix.States [get]
```

Gets the states for the character to which the rate matrix applies.

Definition at line 120 of file RateMatix.cs.

7.15.3.3 this[char from, char to]

```
new double PhyloTree.SequenceSimulation.MutableRateMatrix.this[char from, char to] [get],
[set]
```

Gets or sets the rate of going from state *from* to state *to* . If `from == to` , the negative sum of the elements on the row is returned, but this value cannot be set.

Parameters

from	The row state.
to	The column state.

Returns

The rate of going from state *from* to state *to* .

Exceptions

ArgumentOutOfRangeException	Thrown if the state is not part of the rate matrix.
ArgumentException	Thrown when attempting to set the value of a diagonal entry.

Definition at line 216 of file RateMatix.cs.

7.15.3.4 this[int from, int to]

```
new double PhyloTree.SequenceSimulation.MutableRateMatrix.this[int from, int to] [get], [set]
```

Gets or sets the rate of going from state number *from* to state number *to* . If `from == to` , the negative sum of the elements on the row is returned, but this value cannot be set.

Parameters

from	The row number.
to	The column number.

Returns

The rate of going from state number *from* to state number *to* .

Exceptions

ArgumentOutOfRangeException	Thrown if the state index is $<$ 0 or greater than the number of states in the rate matrix.
ArgumentException	Thrown when attempting to set the value of a diagonal entry.

Definition at line 138 of file RateMatix.cs.

The documentation for this class was generated from the following file:

- SequenceSimulation/RateMatix.cs

7.16 PhyloTree.Formats.NcbiAsnBer Class Reference

Contains methods to read and write trees in the NCBI ASN.1 binary format.
Note: this is a hackish reverse-engineering of the NCBI binary ASN format. A lot of this is derived by assumptions and observations.

Static Public Member Functions

- static IEnumerable$<$ TreeNode $>$ ParseTrees (string inputFile)

  Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.
- static IEnumerable$<$ TreeNode $>$ ParseTrees (Stream inputStream, bool keepOpen=false)

  Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.
- static List$<$ TreeNode $>$ ParseAllTrees (string inputFile)

  Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.
- static List$<$ TreeNode $>$ ParseAllTrees (Stream inputStream, bool keepOpen=false)

  Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.
- static TreeNode ParseTree (BinaryReader reader)

  Parses a tree from a BinaryReader reading a stream in NCBI ASN.1 binary format into a TreeNode object.
- static void WriteTree (TreeNode tree, string outputFile, string treeType=null, string label=null)

  Writes a TreeNode to a file in NCBI ASN.1 binary format.

- static void WriteTree (TreeNode tree, Stream outputStream, bool keepOpen=false, string treeType=null, string label=null)

  Writes a TreeNode to a file in NCBI ASN.1 binary format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, Stream outputStream, bool keepOpen=false, string treeType=null, string label=null)

  Writes a collection of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, string outputFile, string treeType=null, string label=null)

  Writes a collection of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.
- static void WriteAllTrees (List< TreeNode > trees, Stream outputStream, bool keepOpen=false, string treeType=null, string label=null)

  Writes a list of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the tree contains more than one tree an exception will be thrown.
- static void WriteAllTrees (List< TreeNode > trees, string outputFile, string treeType=null, string label=null)

  Writes a list of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the list contains more than one tree an exception will be thrown.
- static void WriteTree (TreeNode tree, BinaryWriter writer, string treeType=null, string label=null)

  Writes a TreeNode to a BinaryWriter in NCBI ASN.1 binary format.

7.16.1 Detailed Description

Contains methods to read and write trees in the NCBI ASN.1 binary format.
Note: this is a hackish reverse-engineering of the NCBI binary ASN format. A lot of this is derived by assumptions and observations.

Definition at line 13 of file NcbiAsnBer.cs.

7.16.2 Member Function Documentation

7.16.2.1 ParseAllTrees() [1/2]

```
static List< TreeNode > PhyloTree.Formats.NcbiAsnBer.ParseAllTrees (
            Stream inputStream,
            bool keepOpen = false )  [static]
```

Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.

Returns

A List<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 145 of file NcbiAsnBer.cs.

7.16.2.2 ParseAllTrees() [2/2]

```
static List< TreeNode > PhyloTree.Formats.NcbiAsnBer.ParseAllTrees (
            string inputFile )  [static]
```

Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.

Parameters

inputFile	The path to the input file.

Returns

A List<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 132 of file NcbiAsnBer.cs.

7.16.2.3 ParseTree()

```
static TreeNode PhyloTree.Formats.NcbiAsnBer.ParseTree (
            BinaryReader reader )  [static]
```

Parses a tree from a BinaryReader reading a stream in NCBI ASN.1 binary format into a TreeNode object.

Parameters

reader	The BinaryReader that reads a stream in NCBI ASN.1 binary format.

Returns

The parsed TreeNode object.

Definition at line 156 of file NcbiAsnBer.cs.

7.16.2.4 ParseTrees() [1/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NcbiAsnBer.ParseTrees (
            Stream inputStream,
            bool keepOpen = false )  [static]
```

Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.

Returns

A IEnumerable<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 122 of file NcbiAsnBer.cs.

7.16.2.5 ParseTrees() [2/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NcbiAsnBer.ParseTrees (
            string inputFile )  [static]
```

Parses a tree from an NCBI ASN.1 binary format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.

Parameters

inputFile	The path to the input file.

Returns

A IEnumerable<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 111 of file NcbiAsnBer.cs.

7.16.2.6 WriteAllTrees() [1/4]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            string treeType = null,
            string label = null )  [static]
```

Writes a collection of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.

Parameters

trees	The collection of trees to write. If this contains more than one tree, an exception will be thrown.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 550 of file NcbiAsnBer.cs.

7.16.2.7 WriteAllTrees() [2/4]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            string outputFile,
            string treeType = null,
            string label = null )  [static]
```

Writes a collection of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.

Parameters

trees	The collection of trees to write. If this contains more than one tree, an exception will be thrown.
outputFile	The path to the output file.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 577 of file NcbiAsnBer.cs.

7.16.2.8 WriteAllTrees() [3/4]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteAllTrees (
            List< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            string treeType = null,
            string label = null )  [static]
```

Writes a list of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the tree contains more than one tree an exception will be thrown.

Parameters

trees	The list of trees to write. If this contains more than one tree, an exception will be thrown.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 605 of file NcbiAsnBer.cs.

7.16.2.9 WriteAllTrees() [4/4]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteAllTrees (
            List< TreeNode > trees,
            string outputFile,
            string treeType = null,
            string label = null )  [static]
```

Writes a list of TreeNodes to a file in NCBI ASN.1 binary format. Note that only one tree can be saved in each file; if the list contains more than one tree an exception will be thrown.

Parameters

trees	The list of trees to write. If this contains more than one tree, an exception will be thrown.
outputFile	The path to the output file.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 624 of file NcbiAsnBer.cs.

7.16.2.10 WriteTree() [1/3]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteTree (
            TreeNode tree,
            BinaryWriter writer,
            string treeType = null,
            string label = null )  [static]
```

Writes a TreeNode to a BinaryWriter in NCBI ASN.1 binary format.

Parameters

tree	The tree to write.
writer	The BinaryWriter on which the tree will be written..
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 643 of file NcbiAsnBer.cs.

7.16.2.11 WriteTree() [2/3]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteTree (
            TreeNode tree,
```

```
            Stream outputStream,
            bool keepOpen = false,
            string treeType = null,
            string label = null )  [static]
```

Writes a TreeNode to a file in NCBI ASN.1 binary format.

Parameters

tree	The tree to write.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 536 of file NcbiAsnBer.cs.

7.16.2.12 WriteTree() [3/3]

```
static void PhyloTree.Formats.NcbiAsnBer.WriteTree (
            TreeNode tree,
            string outputFile,
            string treeType = null,
            string label = null )  [static]
```

Writes a TreeNode to a file in NCBI ASN.1 binary format.

Parameters

tree	The tree to write.
outputFile	The path to the output file.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 522 of file NcbiAsnBer.cs.

The documentation for this class was generated from the following file:

- NcbiAsnBer.cs

7.17 PhyloTree.Formats.NcbiAsnText Class Reference

Contains methods to read and write trees in the NCBI ASN.1 text format.

Static Public Member Functions

- static IEnumerable< TreeNode > ParseTrees (string inputFile)

  Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.
- static IEnumerable< TreeNode > ParseTrees (Stream inputStream, bool keepOpen=false)

  Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.
- static List< TreeNode > ParseAllTrees (string inputFile)

  Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.
- static List< TreeNode > ParseAllTrees (Stream inputStream, bool keepOpen=false)

  Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.
- static TreeNode ParseTree (string source)

  Parses a tree from an NCBI ASN.1 format string into a TreeNode object.
- static TreeNode ParseTree (TextReader reader)

  Parses a tree from a TextReader that reads an NCBI ASN.1 format string into a TreeNode object.
- static void WriteTree (TreeNode tree, string outputFile, string treeType=null, string label=null)

  Writes a TreeNode to a file in NCBI ASN.1 text format.
- static void WriteTree (TreeNode tree, Stream outputStream, bool keepOpen=false, string treeType=null, string label=null)

  Writes a TreeNode to a file in NCBI ASN.1 text format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, Stream outputStream, bool keepOpen=false, string treeType=null, string label=null)

  Writes a collection of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, string outputFile, string treeType=null, string label=null)

  Writes a collection of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.
- static void WriteAllTrees (List< TreeNode > trees, Stream outputStream, bool keepOpen=false, string tree↩ Type=null, string label=null)

  Writes a list of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the tree contains more than one tree an exception will be thrown.
- static void WriteAllTrees (List< TreeNode > trees, string outputFile, string treeType=null, string label=null)

  Writes a list of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the list contains more than one tree an exception will be thrown.
- static string WriteTree (TreeNode tree, string treeType=null, string label=null)

  Writes a TreeNode to a string in NCBI ASN.1 text format.

7.17.1 Detailed Description

Contains methods to read and write trees in the NCBI ASN.1 text format.

Definition at line 12 of file NcbiAsnText.cs.

7.17.2 Member Function Documentation

7.17.2.1 ParseAllTrees() [1/2]

```
static List< TreeNode > PhyloTree.Formats.NcbiAsnText.ParseAllTrees (
            Stream inputStream,
            bool keepOpen = false )  [static]
```

Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.

Returns

A List<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 52 of file NcbiAsnText.cs.

7.17.2.2 ParseAllTrees() [2/2]

```
static List< TreeNode > PhyloTree.Formats.NcbiAsnText.ParseAllTrees (
            string inputFile )  [static]
```

Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a list with a single element.

Parameters

inputFile	The path to the input file.

Returns

A List<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 40 of file NcbiAsnText.cs.

7.17.2.3 ParseTree() [1/2]

```
static TreeNode PhyloTree.Formats.NcbiAsnText.ParseTree (
            string source )  [static]
```

Parses a tree from an NCBI ASN.1 format string into a TreeNode object.

Parameters

source	The NCBI ASN.1 format tree string.

Returns

The parsed TreeNode object.

Definition at line 63 of file NcbiAsnText.cs.

7.17.2.4 ParseTree() [2/2]

```
static TreeNode PhyloTree.Formats.NcbiAsnText.ParseTree (
            TextReader reader )  [static]
```

Parses a tree from a TextReader that reads an NCBI ASN.1 format string into a TreeNode object.

Parameters

reader	The TextReader that reads the NCBI ASN.1 format string.

Returns

The parsed TreeNode object.

Definition at line 74 of file NcbiAsnText.cs.

7.17.2.5 ParseTrees() [1/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NcbiAsnText.ParseTrees (
            Stream inputStream,
            bool keepOpen = false )  [static]
```

Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.

Returns

A IEnumerable<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 30 of file NcbiAsnText.cs.

7.17.2.6 ParseTrees() [2/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NcbiAsnText.ParseTrees (
            string inputFile )  [static]
```

Parses a tree from an NCBI ASN.1 text format file. Note that the tree can only contain a single file, and this method will always return a collection with a single element.

Parameters

inputFile	The path to the input file.

Returns

A IEnumerable<T> containing the tree defined in the file. This will always consist of a single element.

Definition at line 19 of file NcbiAsnText.cs.

7.17.2.7 WriteAllTrees() [1/4]

```
static void PhyloTree.Formats.NcbiAsnText.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            string treeType = null,
            string label = null )  [static]
```

Writes a collection of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.

Parameters

trees	The collection of trees to write. If this contains more than one tree, an exception will be thrown.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 359 of file NcbiAsnText.cs.

7.17.2.8 WriteAllTrees() [2/4]

```
static void PhyloTree.Formats.NcbiAsnText.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            string outputFile,
            string treeType = null,
            string label = null )  [static]
```

Writes a collection of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the collection contains more than one tree an exception will be thrown.

Parameters

trees	The collection of trees to write. If this contains more than one tree, an exception will be thrown.
outputFile	The path to the output file.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 386 of file NcbiAsnText.cs.

7.17.2.9 WriteAllTrees() [3/4]

```
static void PhyloTree.Formats.NcbiAsnText.WriteAllTrees (
            List< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            string treeType = null,
            string label = null )  [static]
```

Writes a list of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the tree contains more than one tree an exception will be thrown.

Parameters

trees	The list of trees to write. If this contains more than one tree, an exception will be thrown.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 414 of file NcbiAsnText.cs.

7.17.2.10 WriteAllTrees() [4/4]

```
static void PhyloTree.Formats.NcbiAsnText.WriteAllTrees (
            List< TreeNode > trees,
```

```
            string outputFile,
            string treeType = null,
            string label = null )  [static]
```

Writes a list of TreeNodes to a file in NCBI ASN.1 text format. Note that only one tree can be saved in each file; if the list contains more than one tree an exception will be thrown.

Parameters

trees	The list of trees to write. If this contains more than one tree, an exception will be thrown.
outputFile	The path to the output file.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 433 of file NcbiAsnText.cs.

7.17.2.11 WriteTree() [1/3]

```
static void PhyloTree.Formats.NcbiAsnText.WriteTree (
            TreeNode tree,
            Stream outputStream,
            bool keepOpen = false,
            string treeType = null,
            string label = null )  [static]
```

Writes a TreeNode to a file in NCBI ASN.1 text format.

Parameters

tree	The tree to write.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 345 of file NcbiAsnText.cs.

7.17.2.12 WriteTree() [2/3]

```
static void PhyloTree.Formats.NcbiAsnText.WriteTree (
            TreeNode tree,
            string outputFile,
            string treeType = null,
            string label = null )  [static]
```

Writes a TreeNode to a file in NCBI ASN.1 text format.

Parameters

tree	The tree to write.
outputFile	The path to the output file.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Definition at line 332 of file NcbiAsnText.cs.

7.17.2.13 WriteTree() [3/3]

```
static string PhyloTree.Formats.NcbiAsnText.WriteTree (
            TreeNode tree,
            string treeType = null,
            string label = null )  [static]
```

Writes a TreeNode to a string in NCBI ASN.1 text format.

Parameters

tree	The tree to write.
treeType	An optional value for the `treetype` property defined in the NCBI ASN.1 tree format.
label	An optional value for the `label` property defined in the NCBI ASN.1 tree format.

Returns

A string containing the NCBI ASN.1 representation of the TreeNode.

Definition at line 452 of file NcbiAsnText.cs.

The documentation for this class was generated from the following file:

- NcbiAsnText.cs

7.18 PhyloTree.TreeBuilding.NeighborJoining Class Reference

Contains methods to compute neighbour-joining trees.

Static Public Member Functions

- static TreeNode BuildTree (Dictionary< string, string > alignment, EvolutionModel evolutionModel=EvolutionModel.Kimura, int bootstrapReplicates=0, AlignmentType alignmentType=AlignmentType.Autodetect, TreeNode constraint=null, bool allowNegativeBranches=true, int numCores=-1, Action< double > progressCallback=null)

  Builds a neighbour-joining tree using data from a sequence alignment. This method first computes a distance matrix from the sequence alignment, and then uses the distance matrix to compute the tree.
- static TreeNode BuildTree (float[ ][ ] distanceMatrix, IReadOnlyList< string > sequenceNames, TreeNode constraint=null, bool copyMatrix=true, bool allowNegativeBranches=true, int numCores=-1, Action< double > progressCallback=null)

  Builds a neighbour-joining tree using data from a distance matrix.

7.18.1 Detailed Description

Contains methods to compute neighbour-joining trees.

Definition at line 13 of file NeighborJoining.cs.

7.18.2 Member Function Documentation

7.18.2.1 BuildTree() [1/2]

```
static TreeNode PhyloTree.TreeBuilding.NeighborJoining.BuildTree (
            Dictionary< string, string > alignment,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int bootstrapReplicates = 0,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            TreeNode constraint = null,
            bool allowNegativeBranches = true,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Builds a neighbour-joining tree using data from a sequence alignment. This method first computes a distance matrix from the sequence alignment, and then uses the distance matrix to compute the tree.

Parameters

alignment	The sequence alignment.
evolutionModel	The evolutionary model to use when computing the distance matrix.
bootstrapReplicates	The number of bootstrap replicates to perform.
alignmentType	The type of sequence alignment (DNA, protein, or autodetect).
constraint	An optional tree to constrain the search. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating.
allowNegativeBranches	If this is `true` negative branches produced by the neighbour-joining algorithm are left untouched; otherwise, their (absolute) length is added to the sibling branch, and the negative length is set to 0.
numCores	Maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

The neighbour-joining tree built from the supplied *alignment* .

Definition at line 28 of file NeighborJoining.cs.

7.18.2.2 BuildTree() [2/2]

```
static TreeNode PhyloTree.TreeBuilding.NeighborJoining.BuildTree (
            float distanceMatrix[][],
            IReadOnlyList< string > sequenceNames,
            TreeNode constraint = null,
            bool copyMatrix = true,
            bool allowNegativeBranches = true,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Builds a neighbour-joining tree using data from a distance matrix.

Parameters

distanceMatrix	The distance matrix containg distances between the taxa. This can be a lower triangular matrix or a full matrix; values above the diagonal will not be used.
sequenceNames	The names of the taxa. The indices of this list should correspond to the rows and columns of the *distanceMatrix* .
constraint	An optional tree to constrain the search. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating.
copyMatrix	If this is `true`, the matrix is copied before using it to compute the tree. If this is `false`, the matrix is not copied. Copying the matrix increases the memory used by the method, but note that if the matrix is not copied, it will be modified in-place!
allowNegativeBranches	If this is `true` negative branches produced by the neighbour-joining algorithm are left untouched; otherwise, their (absolute) length is added to the sibling branch, and the negative length is set to 0.
numCores	Maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

The neighbour-joining tree built from the supplied *distanceMatrix* .

Definition at line 131 of file NeighborJoining.cs.

The documentation for this class was generated from the following file:

- TreeBuilding/NeighborJoining.cs

7.19 PhyloTree.Formats.NEXUS Class Reference

Contains methods to read and write trees in NEXUS format.

Static Public Member Functions

- static List< TreeNode > ParseAllTrees (string sourceString=null, Stream sourceStream=null, bool keepOpen=false, Action< double > progressAction=null)

  Parses a NEXUS file and completely loads it into memory. Can be used to parse a string or a file.
- static IEnumerable< TreeNode > ParseTrees (string inputFile, Action< double > progressAction=null)

  Lazily parses a NEXUS file. Each tree in the NEXUS file is not read and parsed until it is requested. Can be used to parse a string or a Stream.
- static IEnumerable< TreeNode > ParseTrees (string sourceString=null, Stream sourceStream=null, bool keepOpen=false, Action< double > progressAction=null)

  Lazily parses a NEXUS file. Each tree in the NEXUS file is not read and parsed until it is requested. Can be used to parse a string or a Stream.
- static IEnumerable< TreeNode > ParseTrees (Stream inputStream, bool keepOpen=false, Action< double > progressAction=null)

  Lazily parses trees from a file in NEXUS format. Each tree in the file is not read and parsed until it is requested.
- static List< TreeNode > ParseAllTrees (string inputFile, Action< double > progressAction=null)

  Parses trees from a file in NEXUS format and completely loads them in memory.
- static List< TreeNode > ParseAllTrees (Stream inputStream, bool keepOpen=false, Action< double > progressAction=null)

  Parses trees from a file in NEXUS format and completely loads them in memory.
- static void WriteTree (TreeNode tree, Stream outputStream, bool keepOpen=false, bool translate=true, bool translateQuotes=true, TextReader additionalNexusBlocks=null)

  Writes a single tree in NEXUS format.
- static void WriteTree (TreeNode tree, string outputFile, bool append=false, bool translate=true, bool translateQuotes=true, TextReader additionalNexusBlocks=null)

  Writes a single tree in NEXUS format.
- static void WriteAllTrees (IList< TreeNode > trees, string outputFile, bool append=false, Action< double > progressAction=null, bool translate=true, bool translateQuotes=true, TextReader additionalNexusBlocks=null)

  Writes trees in NEXUS format.
- static void WriteAllTrees (IList< TreeNode > trees, Stream outputStream, bool keepOpen=false, Action< double > progressAction=null, bool translate=true, bool translateQuotes=true, TextReader additionalNexusBlocks=null)

  Writes trees in NEXUS format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, string outputFile, bool append=false, Action< int > progressAction=null, TextReader additionalNexusBlocks=null)

  Writes trees in NEXUS format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, Stream outputStream, bool keepOpen=false, Action< int > progressAction=null, TextReader additionalNexusBlocks=null)

  Writes trees in NEXUS format.

7.19.1 Detailed Description

Contains methods to read and write trees in NEXUS format.

Definition at line 14 of file NEXUS.cs.

7.19.2 Member Function Documentation

7.19.2.1 ParseAllTrees() [1/3]

```
static List< TreeNode > PhyloTree.Formats.NEXUS.ParseAllTrees (
            Stream inputStream,
            bool keepOpen = false,
            Action< double > progressAction = null )  [static]
```

Parses trees from a file in NEXUS format and completely loads them in memory.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A List<T> containing the trees defined in the file.

Definition at line 423 of file NEXUS.cs.

7.19.2.2 ParseAllTrees() [2/3]

```
static List< TreeNode > PhyloTree.Formats.NEXUS.ParseAllTrees (
            string inputFile,
            Action< double > progressAction = null )  [static]
```

Parses trees from a file in NEXUS format and completely loads them in memory.

Parameters

inputFile	The path to the input file.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A List<T> containing the trees defined in the file.

Definition at line 410 of file NEXUS.cs.

7.19.2.3 ParseAllTrees() [3/3]

```
static List< TreeNode > PhyloTree.Formats.NEXUS.ParseAllTrees (
            string sourceString = null,
```

```
            Stream sourceStream = null,
            bool keepOpen = false,
            Action< double > progressAction = null )  [static]
```

Parses a NEXUS file and completely loads it into memory. Can be used to parse a string or a file.

Parameters

sourceString	The NEXUS file content. If this parameter is specified, *sourceStream* is ignored.
sourceStream	The stream to parse.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A List<T> containing the trees defined in the "Trees" blocks of the NEXUS file.

Definition at line 80 of file NEXUS.cs.

7.19.2.4 ParseTrees() [1/3]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NEXUS.ParseTrees (
            Stream inputStream,
            bool keepOpen = false,
            Action< double > progressAction = null )  [static]
```

Lazily parses trees from a file in NEXUS format. Each tree in the file is not read and parsed until it is requested.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A lazy IEnumerable<T> containing the trees defined in the file.

Definition at line 399 of file NEXUS.cs.

7.19.2.5 ParseTrees() [2/3]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NEXUS.ParseTrees (
            string inputFile,
            Action< double > progressAction = null )  [static]
```

Lazily parses a NEXUS file. Each tree in the NEXUS file is not read and parsed until it is requested. Can be used to parse a string or a Stream.

Parameters

inputFile	The path to the input file.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A lazy IEnumerable<T> containing the trees defined in the "Trees" blocks of the NEXUS file.

Definition at line 93 of file NEXUS.cs.

7.19.2.6 ParseTrees() [3/3]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NEXUS.ParseTrees (
            string sourceString = null,
            Stream sourceStream = null,
            bool keepOpen = false,
            Action< double > progressAction = null )  [static]
```

Lazily parses a NEXUS file. Each tree in the NEXUS file is not read and parsed until it is requested. Can be used to parse a string or a Stream.

Parameters

sourceString	The NEXUS file content. If this parameter is specified, *sourceStream* is ignored.
sourceStream	The stream to parse.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that might be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.

Returns

A lazy IEnumerable<T> containing the trees defined in the "Trees" blocks of the NEXUS file.

Definition at line 107 of file NEXUS.cs.

7.19.2.7 WriteAllTrees() [1/4]

```
static void PhyloTree.Formats.NEXUS.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            Action< int > progressAction = null,
            TextReader additionalNexusBlocks = null )  [static]
```

Writes trees in NEXUS format.

Parameters

trees	An IEnumerable<T> containing the trees to be written. It will only be enumerated once.
outputStream	The Stream on which the trees should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
progressAction	An Action that will be invoked after each tree is written, with the number of trees written so far.
additionalNexusBlocks	A TextReader that can read additional NEXUS blocks that will be placed at the end of the file.

Definition at line 627 of file NEXUS.cs.

7.19.2.8 WriteAllTrees() [2/4]

```
static void PhyloTree.Formats.NEXUS.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            string outputFile,
            bool append = false,
            Action< int > progressAction = null,
            TextReader additionalNexusBlocks = null )  [static]
```

Writes trees in NEXUS format.

Parameters

trees	An IEnumerable<T> containing the trees to be written. It will only be enumerated once.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
progressAction	An Action that will be invoked after each tree is written, with the number of trees written so far.
additionalNexusBlocks	A TextReader that can read additional NEXUS blocks that will be placed at the end of the file.

Definition at line 613 of file NEXUS.cs.

7.19.2.9 WriteAllTrees() [3/4]

```
static void PhyloTree.Formats.NEXUS.WriteAllTrees (
            IList< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            Action< double > progressAction = null,
            bool translate = true,
            bool translateQuotes = true,
            TextReader additionalNexusBlocks = null )  [static]
```

Writes trees in NEXUS format.

Parameters

trees	A collection of trees to be written. If *translate* is `true`, each tree will be accessed twice. Otherwise, each tree will be accessed once.
outputStream	The Stream on which the trees should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
progressAction	An Action that will be invoked after each tree is written, with a value between 0 and 1 depending on how many trees have been written so far.
translate	If this is `true`, a `Taxa` block and a `Translate` statement in the `Trees` block are added to the NEXUS file.
translateQuotes	If this is `true`, entries in the `Taxa` block and a `Translate` statement in the `Trees` block are placed between single quotes. Otherwise, they are not. This has no effect if *translate* is `false`.
additionalNexusBlocks	A TextReader that can read additional NEXUS blocks that will be placed at the end of the file.

Definition at line 483 of file NEXUS.cs.

7.19.2.10 WriteAllTrees() [4/4]

```
static void PhyloTree.Formats.NEXUS.WriteAllTrees (
              IList< TreeNode > trees,
              string outputFile,
              bool append = false,
              Action< double > progressAction = null,
              bool translate = true,
              bool translateQuotes = true,
              TextReader additionalNexusBlocks = null )  [static]
```

Writes trees in NEXUS format.

Parameters

trees	A collection of trees to be written. If *translate* is `true`, each tree will be accessed twice. Otherwise, each tree will be accessed once.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
progressAction	An Action that will be invoked after each tree is written, with a value between 0 and 1 depending on how many trees have been written so far.
translate	If this is `true`, a `Taxa` block and a `Translate` statement in the `Trees` block are added to the NEXUS file.
translateQuotes	If this is `true`, entries in the `Taxa` block and a `Translate` statement in the `Trees` block are placed between single quotes. Otherwise, they are not. This has no effect if *translate* is `false`.
additionalNexusBlocks	A TextReader that can read additional NEXUS blocks that will be placed at the end of the file.

Definition at line 467 of file NEXUS.cs.

7.19.2.11 WriteTree() [1/2]

```
static void PhyloTree.Formats.NEXUS.WriteTree (
            TreeNode tree,
            Stream outputStream,
            bool keepOpen = false,
            bool translate = true,
            bool translateQuotes = true,
            TextReader additionalNexusBlocks = null )  [static]
```

Writes a single tree in NEXUS format.

Parameters

tree	The tree to be written.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
translate	If this is `true`, a `Taxa` block and a `Translate` statement in the `Trees` block are added to the NEXUS file.
translateQuotes	If this is `true`, entries in the `Taxa` block and a `Translate` statement in the `Trees` block are placed between single quotes. Otherwise, they are not. This has no effect if *translate* is `false`.
additionalNexusBlocks	A TextReader that can read additional NEXUS blocks that will be placed at the end of the file.

Definition at line 437 of file NEXUS.cs.

7.19.2.12 WriteTree() [2/2]

```
static void PhyloTree.Formats.NEXUS.WriteTree (
            TreeNode tree,
            string outputFile,
            bool append = false,
            bool translate = true,
            bool translateQuotes = true,
            TextReader additionalNexusBlocks = null )  [static]
```

Writes a single tree in NEXUS format.

Parameters

tree	The tree to be written.
outputFile	The file on which the tree should be written.
append	Specifies whether the file should be overwritten or appended to.
translate	If this is `true`, a `Taxa` block and a `Translate` statement in the `Trees` block are added to the NEXUS file.
translateQuotes	If this is `true`, entries in the `Taxa` block and a `Translate` statement in the `Trees` block are placed between single quotes. Otherwise, they are not. This has no effect if *translate* is `false`.
additionalNexusBlocks	A TextReader that can read additional NEXUS blocks that will be placed at the end of the file.

Definition at line 451 of file NEXUS.cs.

The documentation for this class was generated from the following file:

- NEXUS.cs

7.20 PhyloTree.Formats.NWKA Class Reference

Contains methods to read and write trees in Newick and Newick-with-Attributes (NWKA) format.

Static Public Member Functions

- static TreeNode ParseTree (string source, bool debug=false, TreeNode parent=null)

  Parse a Newick-with-Attributes string into a TreeNode object.
- static IEnumerable< TreeNode > ParseTreesFromSource (string source, bool debug=false)

  Lazily parses trees from a string in Newick-with-Attributes (NWKA) format. Each tree in the string is not read and parsed until it is requested.
- static List< TreeNode > ParseAllTreesFromSource (string source, bool debug=false)

  Parses trees from a string in Newick-with-Attributes (NWKA) format and completely loads them in memory.
- static IEnumerable< TreeNode > ParseTrees (string inputFile, Action< double > progressAction=null, bool debug=false)

  Lazily parses trees from a file in Newick-with-Attributes (NWKA) format. Each tree in the file is not read and parsed until it is requested.
- static IEnumerable< TreeNode > ParseTrees (Stream inputStream, bool keepOpen=false, Action< double > progressAction=null, bool debug=false)

  Lazily parses trees from a file in Newick-with-Attributes (NWKA) format. Each tree in the file is not read and parsed until it is requested.
- static List< TreeNode > ParseAllTrees (string inputFile, Action< double > progressAction=null, bool debug=false)

  Parses trees from a file in Newick-with-Attributes (NWKA) format and completely loads them in memory.
- static List< TreeNode > ParseAllTrees (Stream inputStream, bool keepOpen=false, Action< double > progressAction=null, bool debug=false)

  Parses trees from a file in Newick-with-Attributes (NWKA) format and completely loads them in memory.
- static string WriteTree (TreeNode tree, bool nwka, bool singleQuoted=false)

  Writes a TreeNode to a string.
- static void WriteTree (TreeNode tree, Stream outputStream, bool keepOpen=false, bool nwka=true, bool singleQuoted=false)

  Writes a single tree in Newick o Newick-with-Attributes format.
- static void WriteTree (TreeNode tree, string outputFile, bool append=false, bool nwka=true, bool singleQuoted=false)

  Writes a single tree in Newick o Newick-with-Attributes format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, string outputFile, bool append=false, Action< int > progressAction=null, bool nwka=true, bool singleQuoted=false)

  Writes trees in Newick o Newick-with-Attributes format.
- static void WriteAllTrees (IEnumerable< TreeNode > trees, Stream outputStream, bool keepOpen=false, Action< int > progressAction=null, bool nwka=true, bool singleQuoted=false)

  Writes trees in Newick o Newick-with-Attributes format.
- static void WriteAllTrees (IList< TreeNode > trees, string outputFile, bool append=false, Action< double > progressAction=null, bool nwka=true, bool singleQuoted=false)

  Writes trees in Newick o Newick-with-Attributes format.
- static void WriteAllTrees (IList< TreeNode > trees, Stream outputStream, bool keepOpen=false, Action< double > progressAction=null, bool nwka=true, bool singleQuoted=false)

  Writes trees in Newick o Newick-with-Attributes format.

7.20.1 Detailed Description

Contains methods to read and write trees in Newick and Newick-with-Attributes (NWKA) format.

Definition at line 15 of file NWKA.cs.

7.20.2 Member Function Documentation

7.20.2.1 ParseAllTrees() [1/2]

```
static List< TreeNode > PhyloTree.Formats.NWKA.ParseAllTrees (
            Stream inputStream,
            bool keepOpen = false,
            Action< double > progressAction = null,
            bool debug = false )  [static]
```

Parses trees from a file in Newick-with-Attributes (NWKA) format and completely loads them in memory.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

A List<T> containing the trees defined in the file.

Definition at line 362 of file NWKA.cs.

7.20.2.2 ParseAllTrees() [2/2]

```
static List< TreeNode > PhyloTree.Formats.NWKA.ParseAllTrees (
            string inputFile,
            Action< double > progressAction = null,
            bool debug = false )  [static]
```

Parses trees from a file in Newick-with-Attributes (NWKA) format and completely loads them in memory.

Parameters

inputFile	The path to the input file.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

A List<T> containing the trees defined in the file.

Definition at line 348 of file NWKA.cs.

7.20.2.3 ParseAllTreesFromSource()

```
static List< TreeNode > PhyloTree.Formats.NWKA.ParseAllTreesFromSource (
            string source,
            bool debug = false )  [static]
```

Parses trees from a string in Newick-with-Attributes (NWKA) format and completely loads them in memory.

Parameters

source	The string from which the trees should be read.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

A List<T> containing the trees defined in the string.

Definition at line 258 of file NWKA.cs.

7.20.2.4 ParseTree()

```
static TreeNode PhyloTree.Formats.NWKA.ParseTree (
            string source,
            bool debug = false,
            TreeNode parent = null )  [static]
```

Parse a Newick-with-Attributes string into a TreeNode object.

Parameters

source	The Newick-with-Attributes string. This string must specify only a single tree.
parent	The parent node of this node. If parsing a whole tree, this parameter should be left equal to `null`.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

The parsed TreeNode object.

Definition at line 24 of file NWKA.cs.

7.20.2.5 ParseTrees() [1/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NWKA.ParseTrees (
            Stream inputStream,
            bool keepOpen = false,
            Action< double > progressAction = null,
            bool debug = false )  [static]
```

Lazily parses trees from a file in Newick-with-Attributes (NWKA) format. Each tree in the file is not read and parsed until it is requested.

Parameters

inputStream	The Stream from which the file should be read.
keepOpen	Determines whether the stream should be disposed at the end of this method or not.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

A lazy IEnumerable<T> containing the trees defined in the file.

Definition at line 285 of file NWKA.cs.

7.20.2.6 ParseTrees() [2/2]

```
static IEnumerable< TreeNode > PhyloTree.Formats.NWKA.ParseTrees (
            string inputFile,
            Action< double > progressAction = null,
            bool debug = false )  [static]
```

Lazily parses trees from a file in Newick-with-Attributes (NWKA) format. Each tree in the file is not read and parsed until it is requested.

Parameters

inputFile	The path to the input file.
progressAction	An Action that will be called after each tree is parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to 1.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

A lazy IEnumerable<T> containing the trees defined in the file.

Definition at line 270 of file NWKA.cs.

7.20.2.7 ParseTreesFromSource()

```
static IEnumerable< TreeNode > PhyloTree.Formats.NWKA.ParseTreesFromSource (
            string source,
            bool debug = false )  [static]
```

Lazily parses trees from a string in Newick-with-Attributes (NWKA) format. Each tree in the string is not read and parsed until it is requested.

Parameters

source	The string from which the trees should be read.
debug	When this is `true`, debug information is printed to the standard output during the parsing.

Returns

A lazy IEnumerable<T> containing the trees defined in the string.

Definition at line 197 of file NWKA.cs.

7.20.2.8 WriteAllTrees() [1/4]

```
static void PhyloTree.Formats.NWKA.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            Action< int > progressAction = null,
            bool nwka = true,
            bool singleQuoted = false )  [static]
```

Writes trees in Newick o Newick-with-Attributes format.

Parameters

trees	An IEnumerable<T> containing the trees to be written. It will ony be enumerated once.
outputStream	The Stream on which the trees should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
progressAction	An Action that will be invoked after each tree is written, with the number of trees written so far.
nwka	If this is false, a Newick-compliant string is produced for each tree, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Definition at line 995 of file NWKA.cs.

7.20.2.9 WriteAllTrees() [2/4]

```
static void PhyloTree.Formats.NWKA.WriteAllTrees (
            IEnumerable< TreeNode > trees,
            string outputFile,
            bool append = false,
            Action< int > progressAction = null,
            bool nwka = true,
            bool singleQuoted = false )  [static]
```

Writes trees in Newick o Newick-with-Attributes format.

Parameters

trees	An IEnumerable<T> containing the trees to be written. It will ony be enumerated once.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
progressAction	An Action that will be invoked after each tree is written, with the number of trees written so far.
nwka	If this is false, a Newick-compliant string is produced for each tree, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Definition at line 979 of file NWKA.cs.

7.20.2.10 WriteAllTrees() [3/4]

```
static void PhyloTree.Formats.NWKA.WriteAllTrees (
            IList< TreeNode > trees,
            Stream outputStream,
            bool keepOpen = false,
            Action< double > progressAction = null,
            bool nwka = true,
            bool singleQuoted = false )  [static]
```

Writes trees in Newick o Newick-with-Attributes format.

Parameters

trees	A collection of trees to be written. Each tree will only be accessed once.
outputStream	The Stream on which the trees should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
progressAction	An Action that will be invoked after each tree is written, with a value between 0 and 1 depending on how many trees have been written so far.
nwka	If this is false, a Newick-compliant string is produced for each tree, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Definition at line 1034 of file NWKA.cs.

7.20.2.11 WriteAllTrees() [4/4]

```
static void PhyloTree.Formats.NWKA.WriteAllTrees (
            IList< TreeNode > trees,
            string outputFile,
            bool append = false,
            Action< double > progressAction = null,
            bool nwka = true,
            bool singleQuoted = false )  [static]
```

Writes trees in Newick o Newick-with-Attributes format.

Parameters

trees	A collection of trees to be written. Each tree will only be accessed once.
outputFile	The file on which the trees should be written.
append	Specifies whether the file should be overwritten or appended to.
progressAction	An Action that will be invoked after each tree is written, with a value between 0 and 1 depending on how many trees have been written so far.
nwka	If this is false, a Newick-compliant string is produced for each tree, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Definition at line 1018 of file NWKA.cs.

7.20.2.12 WriteTree() [1/3]

```
static string PhyloTree.Formats.NWKA.WriteTree (
            TreeNode tree,
            bool nwka,
            bool singleQuoted = false )  [static]
```

Writes a TreeNode to a string.

Parameters

tree	The tree to write.
nwka	If this is false, a Newick-compliant string is produced, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Returns

A string containing the Newick or NWKA representation of the TreeNode.

Definition at line 722 of file NWKA.cs.

7.20.2.13 WriteTree() [2/3]

```
static void PhyloTree.Formats.NWKA.WriteTree (
            TreeNode tree,
            Stream outputStream,
            bool keepOpen = false,
            bool nwka = true,
            bool singleQuoted = false )  [static]
```

Writes a single tree in Newick o Newick-with-Attributes format.

Parameters

tree	The tree to be written.
outputStream	The Stream on which the tree should be written.
keepOpen	Determines whether the *outputStream* should be kept open after the end of this method.
nwka	If this is false, a Newick-compliant string is produced for each tree, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Definition at line 949 of file NWKA.cs.

7.20.2.14 WriteTree() [3/3]

```
static void PhyloTree.Formats.NWKA.WriteTree (
            TreeNode tree,
            string outputFile,
            bool append = false,
            bool nwka = true,
            bool singleQuoted = false )  [static]
```

Writes a single tree in Newick o Newick-with-Attributes format.

Parameters

tree	The tree to be written.
outputFile	The file on which the tree should be written.
append	Specifies whether the file should be overwritten or appended to.
nwka	If this is false, a Newick-compliant string is produced for each tree, only including the TreeNode.Name, TreeNode.Length and TreeNode.Support attributes of each branch. Otherwise, a Newick-with-Attributes string is produced, including all attributes.
singleQuoted	If *nwka* is false, this determines whether the names of the nodes are placed between single quotes.

Definition at line 963 of file NWKA.cs.

The documentation for this class was generated from the following file:

- NWKA.cs

7.21 PhyloTree.SequenceScores.ParsimonyScore Class Reference

Contains methods to compute parsimony scores for a tree.

Static Public Member Functions

- static int GetParsimonyScore (this TreeNode tree, Dictionary< string, char > tipStates)

  Computes the parsimony score (minimum number of state changes) for a character on a tree.
- static int[ ] GetParsimonyScores (this TreeNode tree, Dictionary< string, string > tipStates, int max↩ Parallelism=-1)

  Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the score for each character in the sequence.
- static int GetParsimonyScore (this TreeNode tree, Dictionary< string, string > tipStates, int maxParallelism=-1)

  Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the total score for the sequence.
- static int[ ] GetParsimonyScores (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tipStates, int maxParallelism=-1)

  Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the score for each character in the sequence.
- static int GetParsimonyScore (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tipStates, int maxParallelism=-1)

  Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the total score for the sequence.
- static int[ ] GetParsimonyScores (this TreeNode tree, Dictionary< string, Sequence > tipStates, int max↩ Parallelism=-1)

  Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the score for each character in the sequence.
- static int GetParsimonyScore (this TreeNode tree, Dictionary< string, Sequence > tipStates, int max↩ Parallelism=-1)

  Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the total score for the sequence.
- static double GetSankoffParsimonyScore (this TreeNode tree, Dictionary< string, char > tipStates, char[ ] states, double[,] transitionCosts)

  Computes the Sankoff parsimony score for a character on a tree.
- static double[ ] GetSankoffParsimonyScores (this TreeNode tree, Dictionary< string, string > tipStates, char[ ] states, double[,] transitionCosts, int maxParallelism=-1)

  Computes the Sankoff parsimony score for a character sequence on a tree, returning the score for each character in the sequence.
- static double GetSankoffParsimonyScore (this TreeNode tree, Dictionary< string, string > tipStates, char[ ] states, double[,] transitionCosts, int maxParallelism=-1)

  Computes the Sankoff parsimony score for a character sequence on a tree, returning the total score for the sequence.
- static double[ ] GetSankoffParsimonyScores (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tipStates, char[ ] states, double[,] transitionCosts, int maxParallelism=-1)

Computes the Sankoff parsimony score for a character sequence on a tree, returning the score for each character in the sequence.

- static double GetSankoffParsimonyScore (this TreeNode tree, Dictionary< string, IReadOnlyList< char > > tipStates, char[ ] states, double[,] transitionCosts, int maxParallelism=-1)

  Computes the Sankoff parsimony score for a character sequence on a tree, returning the total score for the sequence.
- static double[ ] GetSankoffParsimonyScores (this TreeNode tree, Dictionary< string, Sequence > tipStates, char[ ] states, double[,] transitionCosts, int maxParallelism=-1)

  Computes the Sankoff parsimony score for a character sequence on a tree, returning the score for each character in the sequence.
- static double GetSankoffParsimonyScore (this TreeNode tree, Dictionary< string, Sequence > tipStates, char[ ] states, double[,] transitionCosts, int maxParallelism=-1)

  Computes the Sankoff parsimony score for a character sequence on a tree, returning the total score for the sequence.

7.21.1 Detailed Description

Contains methods to compute parsimony scores for a tree.

Definition at line 24 of file ParsimonyScore.cs.

7.21.2 Member Function Documentation

7.21.2.1 GetParsimonyScore() [1/4]

```
static int PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScore (
            this TreeNode tree,
            Dictionary< string, char > tipStates )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character on a tree.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character states at the tips of the tree. This Dictionary<String, Char> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state.

Returns

The parsimony score (minimum number of state changes) for the specified character.

Exceptions

MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is

whether two states are the same or not. Note that this is case sensitive.

Definition at line 39 of file ParsimonyScore.cs.

7.21.2.2 GetParsimonyScore() [2/4]

```
static int PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScore (
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            int maxParallelism = -1 )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the total score for the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The total parsimony score (minimum number of state changes) for the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is whether two states are the same or not. Note that this is case sensitive.

Definition at line 374 of file ParsimonyScore.cs.

7.21.2.3 GetParsimonyScore() [3/4]

```
static int PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScore (
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            int maxParallelism = -1 )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the total score for the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The total parsimony score (minimum number of state changes) for the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is whether two states are the same or not. Note that this is case sensitive.

Definition at line 507 of file ParsimonyScore.cs.

7.21.2.4 GetParsimonyScore() [4/4]

```
static int PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScore (
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            int maxParallelism = -1 )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the total score for the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The total parsimony score (minimum number of state changes) for the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.

Exceptions

MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is whether two states are the same or not. Note that this is case sensitive.

Definition at line 241 of file ParsimonyScore.cs.

7.21.2.5 GetParsimonyScores() [1/3]

```
static int[] PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScores (
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            int maxParallelism = -1 )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the score for each character in the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
maxParallelism	The maximum number of concurrent computations to run.

Returns

An T:int[ ] array containing the parsimony score (minimum number of state changes) for each character in the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is whether two states are the same or not. Note that this is case sensitive.

Definition at line 263 of file ParsimonyScore.cs.

7.21.2.6 GetParsimonyScores() [2/3]

```
static int[] PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScores (
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            int maxParallelism = -1 )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the score for each character in the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
maxParallelism	The maximum number of concurrent computations to run.

Returns

An T:int[ ] array containing the parsimony score (minimum number of state changes) for each character in the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is whether two states are the same or not. Note that this is case sensitive.

Definition at line 396 of file ParsimonyScore.cs.

7.21.2.7 GetParsimonyScores() [3/3]

```
static int[] PhyloTree.SequenceScores.ParsimonyScore.GetParsimonyScores (
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            int maxParallelism = -1 )  [static]
```

Computes the parsimony score (minimum number of state changes) for a character sequence on a tree, returning the score for each character in the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
maxParallelism	The maximum number of concurrent computations to run.

Returns

An T:int[ ] array containing the parsimony score (minimum number of state changes) for each character in the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

The character states can be anything; the only thing that is taken into account when computing parsimony score is whether two states are the same or not. Note that this is case sensitive.

Definition at line 130 of file ParsimonyScore.cs.

7.21.2.8 GetSankoffParsimonyScore() [1/4]

```
static double PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScore (
            this TreeNode tree,
            Dictionary< string, char > tipStates,
            char[] states,
            double transitionCosts[,] )  [static]
```

Computes the Sankoff parsimony score for a character on a tree.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character states at the tips of the tree. This Dictionary<String, Char> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.

Returns

The Sankoff parsimony score for the specified character.

Exceptions

MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a branch will incur a cost. This may or may not be useful.

Definition at line 528 of file ParsimonyScore.cs.

7.21.2.9 GetSankoffParsimonyScore() [2/4]

```
static double PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScore (
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            char[] states,
            double transitionCosts[,],
            int maxParallelism = -1 )  [static]
```

Computes the Sankoff parsimony score for a character sequence on a tree, returning the total score for the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The total Sankoff parsimony score for the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a branch will incur a cost. This may or may not be useful.

Definition at line 903 of file ParsimonyScore.cs.

7.21.2.10 GetSankoffParsimonyScore() [3/4]

```
static double PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScore (
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            char[] states,
            double transitionCosts[,],
            int maxParallelism = -1 )  [static]
```

Computes the Sankoff parsimony score for a character sequence on a tree, returning the total score for the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The total Sankoff parsimony score for the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a branch will incur a cost. This may or may not be useful.

Definition at line 1054 of file ParsimonyScore.cs.

7.21.2.11 GetSankoffParsimonyScore() [4/4]

```
static double PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScore (
            this TreeNode tree,
            Dictionary< string, string > tipStates,
            char[] states,
            double transitionCosts[,],
            int maxParallelism = -1 )  [static]
```

Computes the Sankoff parsimony score for a character sequence on a tree, returning the total score for the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.
maxParallelism	The maximum number of concurrent computations to run.

Returns

The total Sankoff parsimony score for the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a branch will incur a cost. This may or may not be useful.

Definition at line 753 of file ParsimonyScore.cs.

7.21.2.12 GetSankoffParsimonyScores() [1/3]

```
static double[] PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScores (
            this TreeNode tree,
            Dictionary< string, IReadOnlyList< char > > tipStates,
            char[] states,
            double transitionCosts[,],
            int maxParallelism = -1 )  [static]
```

Computes the Sankoff parsimony score for a character sequence on a tree, returning the score for each character in the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<TKey, TValue> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.
maxParallelism	The maximum number of concurrent computations to run.

Returns

An T:double[] array containing the Sankoff parsimony score for each character in the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a

branch will incur a cost. This may or may not be useful.

Definition at line 778 of file ParsimonyScore.cs.

7.21.2.13 GetSankoffParsimonyScores() [2/3]

```
static double[] PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScores (
            this TreeNode tree,
            Dictionary< string, Sequence > tipStates,
            char[] states,
            double transitionCosts[,],
            int maxParallelism = -1 )  [static]
```

Computes the Sankoff parsimony score for a character sequence on a tree, returning the score for each character in the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, Sequence> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.
maxParallelism	The maximum number of concurrent computations to run.

Returns

An T:double[] array containing the Sankoff parsimony score for each character in the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a branch will incur a cost. This may or may not be useful.

Definition at line 929 of file ParsimonyScore.cs.

7.21.2.14 GetSankoffParsimonyScores() [3/3]

```
static double[] PhyloTree.SequenceScores.ParsimonyScore.GetSankoffParsimonyScores (
            this TreeNode tree,
```

```
Dictionary< string, string > tipStates,
char[] states,
double transitionCosts[,],
int maxParallelism = -1 )  [static]
```

Computes the Sankoff parsimony score for a character sequence on a tree, returning the score for each character in the sequence.

Parameters

tree	The tree for which the parsimony score is computed.
tipStates	The character state sequences at the tips of the tree. This Dictionary<String, String> should contain an entry for each terminal node of the tree, where the key is either the TreeNode.Name or TreeNode.Id and the value is the character state sequence.
states	The character states.
transitionCosts	The transition cost matrix. Indices in this matrix should correspond to the states in the *states* array.
maxParallelism	The maximum number of concurrent computations to run.

Returns

An T:double[ ] array containing the Sankoff parsimony score for each character in the sequence.

Exceptions

ArgumentException	Thrown when the sequences do not all have the same length.
MissingDataException	Thrown when the *tipStates* dictionary does not contain an entry for one of the tips in the tree.

Note that diagonal entries in the cost matrix may not be 0, in which case even retaining the same state across a branch will incur a cost. This may or may not be useful.

Definition at line 628 of file ParsimonyScore.cs.

The documentation for this class was generated from the following file:

- SequenceScores/ParsimonyScore.cs

7.22 PhyloTree.SequenceSimulation.RateMatrix.Protein Class Reference

Contains rate matrices for protein sequence evolution.

Static Public Member Functions

- static ImmutableRateMatrix ParsePAMLAminoAcidMatrix (TextReader pamlMatrixReader)

  Read an amino acid substitution matrix in PAML format.
- static ImmutableRateMatrix ParsePAMLAminoAcidMatrix (string fileName)

  Read an amino acid substitution matrix in PAML format.

Static Public Attributes

- static readonly ImmutableRateMatrix cpREV10Matrix
  cpREV10 protein sequence evolution matrix.
- static readonly ImmutableRateMatrix cpREV64Matrix
  cpREV64 protein sequence evolution matrix.
- static readonly ImmutableRateMatrix DayhoffDCMutMatrix
  Dayhoff rate matrix prepared using the DCMut method.
- static readonly ImmutableRateMatrix DayhoffMatrix
  Dayhoff protein sequence evolution matrix.
- static readonly ImmutableRateMatrix JTTDCMutMatrix
  JTT rate matrix prepared using the DCMut method.
- static readonly ImmutableRateMatrix JTTMatrix
  JTT protein sequence evolution matrix.
- static readonly ImmutableRateMatrix LGMatrix
  LG protein sequence evolution matrix.
- static readonly ImmutableRateMatrix mtArtMatrix
  mtArt protein sequence evolution matrix.
- static readonly ImmutableRateMatrix mtmamMatrix
  mtmam protein sequence evolution matrix.
- static readonly ImmutableRateMatrix mtREV24Matrix
  mtREV24 protein sequence evolution matrix.
- static readonly ImmutableRateMatrix MtZoaMatrix
  MTZoa protein sequence evolution matrix.
- static readonly ImmutableRateMatrix WAGMatrix
  WAG protein sequence evolution matrix.

7.22.1 Detailed Description

Contains rate matrices for protein sequence evolution.

Definition at line 108 of file RateMatrices.cs.

7.22.2 Member Function Documentation

7.22.2.1 ParsePAMLAminoAcidMatrix() [1/2]

```
static ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.ParsePAMLAmino↩
AcidMatrix (
            string fileName )  [static]
```

Read an amino acid substitution matrix in PAML format.

Parameters

fileName	The path to the file containing the matrix in PAML format to read.

Returns

An ImmutableRateMatrix corresponding to the matrix contained in the input file, normalised so that the average substitution rate is 1.

Definition at line 544 of file RateMatrices.cs.

7.22.2.2 ParsePAMLAminoAcidMatrix() [2/2]

```
static ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.ParsePAMLAmino↩
AcidMatrix (
            TextReader pamlMatrixReader )  [static]
```

Read an amino acid substitution matrix in PAML format.

Parameters

pamlMatrixReader	A TextReader that contains the matrix in PAML format to read.

Returns

An ImmutableRateMatrix corresponding to the matrix contained in the input file, normalised so that the average substitution rate is 1.

Definition at line 465 of file RateMatrices.cs.

7.22.3 Member Data Documentation

7.22.3.1 cpREV10Matrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.cpREV10Matrix
[static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
    new double[20, 20] {
        { -0.9437513720455278, 0.012847710436231344, 0.018338109198681868, 0.012792540450461161,
      0.011995334156082019, 0.011817016880646248, 0.04866879405415725, 0.10980206417910651,
      0.0031990710320167486, 0.023027557989147417, 0.03924300198532149, 0.0234362099551737,
      0.008019351503023013, 0.006779603108501619, 0.041611961767160456, 0.2990370857274685,
      0.14336708945035886, 0.0004992883712201553, 0.0033874371262892305, 0.1258821446744802,  },
        { 0.015620002221183033, -0.9290073903841098, 0.028840110061363115, 0.0031433099392561706,
      0.014756591943879672, 0.13134200290909362, 0.014824963319102005, 0.040123160294019376,
      0.03465660284684811, 0.02159826128637275, 0.04043822031989981, 0.44508937719952757,
      0.005418480745285821, 0.005284102422802733, 0.007388246272944815, 0.04718413032994893,
      0.03359497469209902, 0.008202594670045409, 0.01953825342484681, 0.011964005485591093,  },
        { 0.03376895718293856, 0.04368221548318657, -1.5278289181105693, 0.32419952513025857,
      0.009646472011916481, 0.057805534804689924, 0.10289694935297775, 0.1078206735473031,
      0.06810143636338685, 0.026680205118460455, 0.022509945301225017, 0.24131351775878002,
      0.0026442186036994804, 0.009670904434186132, 0.014691570174936242, 0.25552964087777535,
      0.14903757880921634, 0.001426538203486158, 0.04560942130753714, 0.010793613644609356,  },
```

```
        { 0.026033337035305058, 0.005261443321504264, 0.35827979866147175, -1.0583849477284581,
      0.00017930245375309442, 0.030107049377442668, 0.35999302375529935, 0.07116496485893972,
      0.01604382593329612, 0.001588107447527408, 0.0019920305576305326, 0.040914061447167645,
      0.0020373487602274687, 0.0021934010056917004, 0.014436803062076077, 0.0723081477783633,
      0.02845943715954885, 0.0006419421915687711, 0.016997675580129885, 0.009753265341514479,  },
        { 0.09952172843782332, 0.10070157799065138, 0.04346212664709623, 0.00073100231145492235,
     -1.0776282387650977, 0.0007526762344360668, 0.0009753265341514478, 0.050030113453036505,
      0.021375610986657363, 0.044467008530767425, 0.07888441008216908, 0.004766686770543803,
      0.006892307508003564, 0.0723822331878261, 0.024202875721715776, 0.2856784618158726,
      0.06162645039060202, 0.015513602962911969, 0.08867826477035735, 0.07698577442902095,  },
        { 0.02335562236881654, 0.21351671153546378, 0.06204258971183997, 0.029240092458196938,
      0.00017930245375309442, -1.2609486437675692, 0.30449694396208205, 0.021960412835821307,
      0.06150941120650385, 0.014610588517252153, 0.05697207394823323, 0.3290006931419087,
      0.008756264884381887, 0.000997000457132591, 0.027429925817944546, 0.04853224833937604,
      0.025784678028012303, 0.0018901631196191592, 0.023651569935340875, 0.007022351045890424,  },
        { 0.07423220103209842, 0.018598590345782515, 0.08522777623175934, 0.2698129531580123,
      0.00017930245375309442, 0.23498552039094, -1.2353834634686571, 0.062578920778779154,
      0.007852265260404747, 0.023503990223405637, 0.01633465057257037, 0.2610754066616595,
      0.004898306593738381, 0.01445650662842257, 0.015710638626376904, 0.06961191175950907,
      0.03947944478147942, 0.002246797670490699, 0.008589572713090549, 0.02600870757737194,  },
        { 0.09892668073415921, 0.029733272723849683, 0.0527523581794681, 0.0315061996237072,
      0.005432864348718762, 0.010010593917999687, 0.03696487564433987, -0.4183750849112877,
      0.0009209446910351246, 0.006352429790109632, 0.0039840611152610165, 0.026117471263604586,
      0.0009103047652080178, 0.0024925011428314774, 0.0023778263866948827, 0.08468632222855768,
      0.009843113604054491, 0.002924403317146624, 0.0006048994868373625, 0.011833961947704233,  },
        { 0.009818287110457907, 0.08748679011338487, 0.11350239393897808, 0.02419617650915797,
      0.007907238210511466, 0.09551461414993687, 0.015800289853253453, 0.0031372018336688758,
     -0.5969035825782917, 0.0046055115978294835, 0.013147401680361517, 0.030288322187830414,
      0.0004334784596228656, 0.012661905805583906, 0.012908200384915079, 0.037134523350583176,
      0.003423691688366779, 0.0024607784010136228, 0.11922568885564416, 0.0032510884471714926,  },
        { 0.021570479257824419, 0.016640843993594886, 0.013571816499464994, 0.0007310023114549235,
      0.005020468705086645, 0.006924621356811813, 0.014434832705441427, 0.006604635439344753,
      0.0014056524231588746, -1.3692288044820726, 0.34760933230652796, 0.03426056116328358,
      0.07681238304517178, 0.045263820753819635, 0.009935917401546476, 0.026472135457841477,
      0.11126997987192032, 0.001497865113660466, 0.005383605432852527, 0.6238188512432661,  },
        { 0.029306099405457694, 0.0248389068433806, 0.009128662288330062, 0.0007310023114549235,
      0.007100377716862254, 0.02152654030487151, 0.007997677580041872, 0.0033023177196723763,
      0.003199071032016748, 0.2771247495935327, -0.8190049409352597, 0.021648702416219777,
      0.05856293989504915, 0.12641965796441254, 0.018597999238792122, 0.06323899026039909,
      0.016690496980788047, 0.0056704893588574778, 0.011432600301226153, 0.112487660272133655,  },
        { 0.03510781451618281, 0.5484136969065608, 0.19630663151011868, 0.030117295231942846,
      0.0008606517780148533, 0.24936163646866888, 0.25641334582841563, 0.04342547801369175,
      0.01478358582977437, 0.05478970693969557, 0.043426266156345615, -1.7664716174510826,
      0.008366134270721307, 0.007178403291354655, 0.02564655602792338, 0.10637876656206667,
      0.09821715531002197, 0.0003566345508715395, 0.014941017324882856, 0.032380840933828064,  },
        { 0.027520956294465344, 0.015294893376465885, 0.004927861943258123, 0.003435710863838314,
      0.0028509090146742015, 0.015204059935608546, 0.01102118983591136, 0.0034674336056559956,
      0.0004847077321237498, 0.28141263970185674, 0.2691233283358849, 0.019166053056556154,
     -0.8532534829230717, 0.03260191494823572, 0.008492237095338867, 0.011397724988792858,
      0.06900878559364289, 0.003067057137495524, 0.013005338967003295, 0.06177068049625836,  },
        { 0.010115810962289964, 0.0064850347916215354, 0.007836108336000623, 0.0016082050852008318,
      0.013017358142474657, 0.0007526762344360668, 0.014142234745195994, 0.004127897149590471,
      0.006155788197971622, 0.07210007811774433, 0.2525894747075515, 0.007150030155815704,
      0.014174745629667708, -0.690702016099449, 0.0036516619509957132, 0.05968486096281851,
      0.015834574058696354, 0.016690496980788047, 0.14336117838045492, 0.04122380151013453,  },
        { 0.07289334369885415, 0.010645245790020257, 0.013975739609568119, 0.0124270392947337,
      0.005110119931963191, 0.024311442372284955, 0.018043540881801785, 0.0046232448075413275,
      0.0073675575282809965, 0.018580857136070675, 0.043625469212108664, 0.02999040426467143,
      0.004334784596228656, 0.004287101965670142, -0.46882626357714885, 0.1473125315755808,
      0.02781749496798008, 0.0017475092992705434, 0.005867525022322417, 0.015865311622196884,  },
        { 0.3629790992351105, 0.047108271599514924, 0.16843593691300304, 0.04312913637584048,
      0.04179540196984631, 0.02980597888366824, 0.05539854713980223, 0.11409507721468061,
      0.014686644283349619, 0.03430312086659201, 0.10278877677373549, 0.08619758576733377,
      0.00403134967449265, 0.048553922262357184, 0.1020766898859732, -1.541418270784083,
      0.23013627567740444, 0.0026034322213622386, 0.031575753212910326, 0.021717270827105572,  },
        { 0.19934098072747872, 0.038420772161682305, 0.11253297847473058, 0.019444661484700965,
      0.010327821336178239, 0.018139497249909205, 0.035989549110188424, 0.015190661510492932,
      0.0015510647427959994, 0.16516317454285043, 0.031075676699036308, 0.09116288448665023,
      0.027959360645674836, 0.0147556067655623345, 0.022079816447881057, 0.26361834893433805,
     -1.1709149706682367, 0.0010342401975274645, 0.004294786356545274, 0.09883308879401338,  },
        { 0.0020826669628244044, 0.028142603812697223, 0.0032313848808249988, 0.0013158041606188625,
      0.007799656738259608, 0.003989184042511154, 0.006144557165154206, 0.013539502650656744,
      0.0033344833516538736, 0.006670051279615114, 0.031673285866632547, 0.0009930597438632923,
      0.0037279147527566445, 0.04665962139380526, 0.004161196176716045, 0.008946601335289017,
      0.0031027205925823935, -0.19775425252959558, 0.020929522244572746, 0.001300435378868597,  },
        { 0.008330667851297617, 0.03952200448478785, 0.06091160500355123, 0.02054116495188335,
      0.026285739720203646, 0.029429640766455021, 0.013849636784950558, 0.0016511588598361882,
      0.09553589400159108, 0.014134156282993932, 0.03764937753921707, 0.02452857567342332,
      0.00931978688189161, 0.23628910834042408, 0.008237469982478702, 0.06397432735645024,
      0.007596315933563791 6, 0.012339555460155268, -0.725601366883686, 0.015475181008536305,  },
        { 0.144001544286716, 0.011257041525078892, 0.006705123627711872, 0.0054825173359111926,
      0.01061470526218319, 0.004064451665954 76, 0.019506530683028957, 0.015025545624509314,
      0.0012117693303093744, 0.7618151425788976, 0.17231064323504106, 0.024727187622195976,
      0.02059022683208612, 0.031604914491103135, 0.010360529256313419, 0.020466882506757066,
      0.081312677598711, 0.0003566345508715395, 0.007198303893364615, -1.3486123719067455,  },
    }, new double[20] {
0.0755, 0.0621, 0.041, 0.0371, 0.0091, 0.0382, 0.0495, 0.0838, 0.0246, 0.0806, 0.1011, 0.0504, 0.022,
```

```
      0.0506, 0.0431, 0.0622, 0.0543, 0.0181, 0.0307, 0.066,  })
```

cpREV10 protein sequence evolution matrix.

Citation: Adachi, J., P. J. Waddell, W. Martin, and M. Hasegawa. 2000. Plastid genome phylogeny and a model of amino acid substitution for proteins encoded by chloroplast DNA.Journal of Molecular Evolution 50:348-358.

Definition at line 114 of file RateMatrices.cs.

7.22.3.2 cpREV64Matrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.cpREV64Matrix
[static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
            new double[20, 20] {
        { -0.9240449511486136, 0.004224894241913516, 0.0020699970330603783, 0.034710712310833336,
      0.002355069832607296, 0.002308940541097105, 0.040431342068098494, 0.0931133780673936,
      0.0007366446632292768, 0.0013731506301627398, 0.004458309590169038, 0.0020970530508021548,
      0.003349596574850681, 0.002580571926424513, 0.042206412685540914, 0.1986320236568338,
      0.26028069740380083, 0.0006551067118634918, 0.000847714597924398, 0.227613335562000802,  },
        { 0.004239348065205315, -1.131555515858714, 0.00487599301120889, 0.003911646108575524,
      0.04271734357913849, 0.18407176805500367, 0.011668105429048559, 0.14499195788817684,
      0.09118608581259549, 0.04632094792415643, 0.042182467660830135, 0.3555503517800051,
      0.013373389310038168, 0.0030006650307261777, 0.03193248328182372, 0.0655778365637257,
      0.04593188777714132, 0.033102156793572916, 0.0016568967141249592, 0.005264185073570671,  },
        { 0.0029349332759113713, 0.006889827532966657, -1.1128393338439788, 0.2648390291616608,
      0.0013164236500215142, 0.006813267170450474, 0.007055133515238663, 0.013718651806711675,
      0.09084407221895334, 0.023435104088110763, 0.002743575132411716, 0.3191515023982708,
      0.002474701947091175, 0.00522115715346355, 0.0016197636447301885, 0.21469028761634684,
      0.11997965837544187, 0.0011753385124609707, 0.026934204724961552, 0.0010027019187753658,  },
        { 0.05498108336873969, 0.006174845430488986, 0.29587157592543006, -0.9176986632375372,
      0.0012922690876357981, 0.004390772176512528, 0.23737810765887624, 0.11850076819245775,
      0.05324888565628773, 0.005126429019274229, 0.006744622200512135, 0.005042913288833753,
      0.0009498855958531784, 0.004500997546089267, 0.006201380811252722, 0.027528452502022367,
      0.004454001238995522, 0.0006743745563300651, 0.07818240541766379, 0.006454893602116418,  },
        { 0.012718044195615943, 0.2298992451330478, 0.005013992813412915, 0.004405748774921906,
      -1.2489795800538401, 3.7851484280280415E-05, 5.427025780952818E-05, 0.06843557338060767,
      0.029413169053226128, 0.00311247476170221, 0.02320607299498243, 4.992983454290845E-05,
      0.002624683883278519, 0.14283165546256607, 0.0030081324830703504, 0.3732953084972516,
      0.004119951146070859, 0.09992304140364908, 0.22314160166273583, 0.023688832831068016,  },
        { 0.003978465107346525, 0.3160870876680836, 0.008279988132241513, 0.004776325774681693,
      1.2077281192857928E-05, -1.0188696394432908, 0.18592990325544353, 0.0055978406797501665,
      0.11841562961410627, 0.0032955561512390576, 0.029836379564977415, 0.249898821887256676,
      0.0023747139896329456, 0.0014403192147485655, 0.06738216762077584, 0.008464208096886187,
      0.0030064508363219777, 0.0032755335593174593, 0.0036220532820406093, 0.0031961123660964783,  },
        { 0.04858945090119937, 0.013974650184790863, 0.0059799914288410925, 0.18010042188325562,
      1.2077281192857928E-05, 0.1296791851442407, -0.7793777082361288, 0.12764653606359888,
      0.001631141754293399, 0.003936365139799855, 0.005830097156374896, 0.21300067416004745,
      0.002399710978997503, 0.004621024147318314, 0.003748595863518436, 0.012973178765320887,
      0.006458301796543507, 0.0017533738464581693, 0.0035064558368691, 0.013536475903467439,  },
        { 0.07702569330780733, 0.11953200785967627, 0.008003988527833461, 0.06188635895988435,
      0.010483080075400681, 0.0026874553838999093, 0.08786354739362612, -0.5624017163101381,
      0.0005787922353944319, 0.0022885843836045667, 0.003886731437583265, 0.014629441521072175,
      0.0009498855958531784, 0.0018604123190502303, 0.00120325299322814, 0.10813619129386373,
      0.0035075259757089737, 0.009441243788620912, 0.0008091821162005616, 0.047628341142182988,  },
        { 0.0018261807050115198, 0.2252843606534192, 0.15883777233683302, 0.08333864972375643,
      0.013502400373615163, 0.17036953074554215, 0.0033647559841907472, 0.0017345421824577982,
      -1.121509083433005, 0.007689643528911344, 0.01977660407946779, 0.004593544777947577,
      0.0008998916171240637, 0.006601463067597591, 0.061967529151249215, 0.01518811172525372,
      0.0021156505885522873, 0.0010982671345946774, 0.3426308274883521, 0.0006893575691580064,  },
        { 0.000978311091970457, 0.032889176713972915, 0.011775983121410151, 0.0023058124429497823,
      0.0004106275605571696, 0.001362653434090095, 0.0023336210858097115, 0.0019710706618838616,
      0.0022099339896878307, -1.0353164407747015, 0.23434704256016742, 0.018923407291762298,
      0.133708896111101713, 0.03660811337485937, 0.0009718581868381133, 0.028635918981988879,
      0.148206891227576, 0.0006358388673969185, 0.004161508026174317, 0.3728797760445892,  },
        { 0.002543608839123188, 0.023984399619478267, 0.0011039984176322015, 0.0024293381095363783,
      0.0024516880821501595, 0.009879237397153188, 0.002767783148285937, 0.0026806561001620514,
      0.004551411669238032, 0.18766391945557445, -0.9318349012177747, 0.005392422130634112,
      0.03569570081258786, 0.3254521292325612, 0.07210262167113239, 0.12680491195615476,
      0.00595722665715651, 0.01903663033297441, 0.003699118245488281, 0.09763809934075125,  },
        { 0.00273927105751728, 0.4628534137948639, 0.29403157856270973, 0.004158697441748715,
      1.2077281192857928E-05, 0.18944667882280347, 0.23151691981544725, 0.023100948157278856,
```

```
      0.002420403893467624, 0.03469493925544523, 0.012346088095852723, -1.4045870212393752,
      0.02089748310876992, 0.0056412502577652215, 0.009811139790937142, 0.01890603490799812,
      0.07805637171339652, 0.0004431604227311857, 0.00774502882649109, 0.005765536032958353,  },
        { 0.008739579088269419, 0.03477412952959587, 0.004553993472732832, 0.0015646584434302095,
      0.0012681145252500825, 0.0035958910066266393, 0.005209944749714705, 0.002996027406063469,
      0.0009471145670090703, 0.48966551471603303, 0.16324272037849713, 0.04174134167787146,
      -1.0584063625463715, 0.002280505423351895, 0.004859290934190566, 0.0049044944112798475,
      0.16435264571893476, 0.002138730735789635, 0.001310104378610433, 0.12026156138312045,  },
        { 0.0028044917969819775, 0.003249918647625782, 0.0040019942639167305, 0.0030881416646648878,
      0.028743929239001872, 0.0009084356227267297, 0.00417880985133367, 0.002444127620735988,
      0.0028939611769721587, 0.05584145895995142, 0.6199336642945307, 0.004693404447033395,
      0.0009498855958531784, -1.1080774565472404, 0.005831149121028678, 0.1972872429311603,
      0.002004300557547985, 0.006570334963101492, 0.09860462073129701, 0.06404758506177649,  },
        { 0.05948131439180379, 0.04484887733723579, 0.0016099976923802939, 0.005517479774201266,
      0.0007850232775357653, 0.05511176111208828, 0.0043958908825717825, 0.002049913488359216,
      0.035227400145142924, 0.001922410882227836, 0.17810375234572726, 0.01058512492309659,
      0.002624683883278519, 0.007561675877429968, -0.7936628558377392, 0.2757591535116379,
      0.10088312806324856, 0.00021194628913230622, 0.002157818976534831, 0.004825502984106448,  },
        { 0.1637692767958545, 0.05388365117763547, 0.12484382106057478, 0.014328977324045077,
      0.056992689949096564, 0.004050108817990004, 0.00890032228076262, 0.10777814379180953,
      0.005051277690715041, 0.033138701874594126, 0.18324795571899924, 0.011933230455755117,
      0.001549813340602554, 0.14967317173262176, 0.1613284590151268, -1.2801057249450514,
      0.08061742242581896, 0.010847796434680762, 0.10184134919609925, 0.006329555862269497,  },
        { 0.3049069569974591, 0.0536236576858254, 0.0991298579165581, 0.0032940177756425465,
      0.0008937188082714867, 0.0020439801511351425, 0.006295349905905268, 0.004967098067947331,
      0.0009997320429540185, 0.24368846516621423, 0.012231772465335567, 0.07000162802915763,
      0.07379111260417322, 0.002160478822122848, 0.08385747783574576, 0.11454367592795515,
      -1.1010517138981215, 0.00028901766699859934, 0.0012715718968865967, 0.023062144131833412,  },
        { 0.002217505141799703, 0.11166720473242188, 0.002805995978148512, 0.001441132776843614,
      0.06263278026616122, 0.0064347523276476705, 0.004938593460667064, 0.03863298497292368,
      0.0014995980644310278, 0.0030209313863580277, 0.11294384295094899, 0.0011483861944868942,
      0.0027746658194658627, 0.020464535509552535, 0.0005090685740580593, 0.044535973443363774,
      0.0008351252323116603, -0.4299980688628356, 0.008361548534072468, 0.003133443496173018,  },
        { 0.0014348562682233374, 0.0027949300369581724, 0.032153953913537876, 0.0835445258347341,
      0.06993953538784027, 0.0035580395223463587, 0.004938593460667064, 0.0016556993559824439,
      0.23393729805124036, 0.009886684537171727, 0.010974300529646864, 0.010035896743124597,
      0.000849897638394949, 0.15357403627256577, 0.0025916218315683013, 0.20907385046794577,
      0.001837275511085653, 0.004181122249246404, -0.8414742762467692, 0.004512158634489147,  },
        { 0.23688172573578004, 0.0054598633280011314, 0.0007359989450881345, 0.0042410478861397 79,
      0.004565212290900297, 0.0019304256982943 01, 0.011722375686858088, 0.059920548121269 39,
      0.00028939611769721594, 0.5446830832978868, 0.17810375234572726, 0.004593544777947577,
      0.04796922259058551, 0.06133359322804308, 0.003563480018406415, 0.00798957960547201,
      0.020488405699379403, 0.0009633922233286645, 0.0027743386841162115, -1.1982089862809313,  },
                }, new double[20] {
0.061007, 0.060799, 0.043028, 0.038515, 0.011297, 0.035406, 0.050764, 0.073749, 0.024609, 0.085629, 0.10693,
      0.046704, 0.023382, 0.056136, 0.043289, 0.073994, 0.052078, 0.018023, 0.036043, 0.05862,  })
```

cpREV64 protein sequence evolution matrix.

Citation: Zhong B, Yonezawa T, Zhong Y and Hasegawa M. 2010. The position of Gnetales among seed plants: Overcoming pitfalls of chloroplast phygenomics. MBE Advance Access published July 2,2010.

Definition at line 144 of file RateMatrices.cs.

7.22.3.3 DayhoffDCMutMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.DayhoffDCMut↩
Matrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
        { -1.3623986751406103, 0.011216122929753615, 0.04075214548621089, 0.057576485934644386,
      0.012338160536299818, 0.034769733 14298245, 0.09944979848871842, 0.21647322349976,
      0.007851661018147275, 0.024675861797894636, 0.03551787472812525, 0.021310893678932345,
      0.010842489238126737, 0.007477700770853093, 0.12898664811681773, 0.28863048867549157,
      0.22058676413223588, 0, 0.007477590682196854, 0.13646503228341925,  },
        { 0.02389074766528073, -0.8855562289329317, 0.013538555564265432, 0, 0.007963723046926064,
      0.09556264852403254, 0, 0.007964592076507517, 0.08202699614264515, 0.023890853259394145,
      0.013538758668458058, 0.3798629822363342, 0.013538324412692237, 0.00557469248008023,
      0.05335595959843808, 0.1091011237158389, 0.015927721743446917, 0.021502578062410793,
      0.0023893839341503856, 0.01592658780203039,  },
        { 0.08781688216702353, 0.013696603601125673, -1.8256124276959287, 0.4286073551723533, 0,
      0.04028258250341676, 0.07573002406793428, 0.125682172003666, 0.18208049950842342,
```

```
      0.029003963908851682, 0.02980975565873418, 0.25941809748423034, 0, 0.005639720922154997,
      0.021753265713734603, 0.34804112559773015, 0.13615668291566402, 0.0024170342797049187,
      0.02900327158135115, 0.01047339060982967,  },
        { 0.1070248013745469, 0, 0.369718650459306, -1.4441348614583254, 0, 0.052817347167176155,
      0.5775132064769238, 0.11258451822733964, 0.029884506190084027, 0.009035056661334731, 0,
      0.05907188400659008, 0, 0, 0.006949729536254814, 0.06810651485145659, 0.039613573658568824, 0, 0,
      0.011815072848743806,  },
        { 0.03211408594868239, 0.009731377412662478, 0, 0, -0.27248134844081473, 0, 0, 0.009732495458345085,
      0.009731418523907841, 0.016543600832013443, 0, 0, 0, 0, 0.00973138550488719, 0.11385712605720422,
      0.009731586553261595, 0, 0.029193902838159744, 0.03211436931169077,  },
        { 0.07918919198924669, 0.10217996536993929, 0.04257496734487378, 0.06471453918232599, 0,
      -1.2670296761329414, 0.35932863442359847, 0.02554564592606376, 0.20691730281211046,
      0.006812433024761571, 0.0638619306864426, 0.12516832504496758, 0.017030112244370135, 0,
      0.07918914306015874, 0.04002119766714481, 0.03150547653639623, 0, 0, 0.022990810820541465,  },
        { 0.17493968489655903, 0, 0.06181942929769269, 0.5465212803146855, 0, 0.277531130827271 54,
      -1.3896432373759031, 0.07365798383730666, 0.015126346314963283, 0.023018382375429323,
      0.009864546993981213, 0.06839640421142074, 0.004603843819303388, 0, 0.026306806958860677,
      0.05655964267803548, 0.020387627678277787, 0, 0.00657644144732821, 0.02433368572478725,  },
        { 0.21284546724894585, 0.003676518691570707, 0.05734642687731034, 0.059552448182547096,
      0.0036765398927080238, 0.01102840117480216, 0.04117139822441429, -0.6675757464664498,
      0.0036760819059608496, 0, 0.0062495855220037318, 0.022056426827991756, 0.002573354196587162,
      0.006249489813870492, 0.0180126882014295, 0.16542241777580216, 0.018380127585801872, 0, 0,
      0.0356583743446702,  },
        { 0.020348364601211148, 0.09980166721850016, 0.21897970659819077, 0.04166532538569316,
      0.009689446553118505, 0.23545082896806221, 0.02228525336803984, 0.009689311694309847,
      -0.899177679102265, 0.0029071411123445506, 0.038757672307039734, 0.022286480823590103, 0,
      0.019379330839898495, 0.048447252617343 18, 0.02519136890045013, 0.013565519747726473,
      0.0029069132625710397, 0.038757196716837125, 0.029068898387338428,  },
        { 0.058285902804998256, 0.02649328909944852, 0.0317922319786014, 0.011481081598169537,
      0.015013297572271806, 0.007065272064258903, 0.03090875885308828, 0, 0.002649655073900977,
      -1.307905911011769, 0.22342788209874964, 0.037973493642359794, 0.05033866995926752,
      0.07948173609034728, 0.006182426263280824, 0.017662800835054313, 0.1139229031549702, 0,
      0.011480571419439645, 0.5837459385035623,  },
        { 0.03625438887774136, 0.006487919966430503, 0.0141203186709226, 0, 0, 0.028621415448174856,
      0.005724088388906469, 0.006487906935327751, 0.01526522939290707, 0.0965516695654074,
      -0.544962428880948, 0.014883389387870 44, 0.07899736097412406, 0.06373194514573775,
      0.016409642852597245, 0.012211769364421086, 0.01984424689848479, 0.004961276366172161,
      0.0087772462284949, 0.11563261441722752,  },
        { 0.023070715244148787, 0.19306315062430904, 0.1303263194727004, 0.034403366597791904, 0,
      0.059496207484937245, 0.042092840553567545, 0.024284788882866 83, 0.009309640769973 98,
      0.017403987108660222, 0.01578511037363424, -0.7629424484914241, 0.03642769162023646, 0,
      0.017403847478879002, 0.06799681465067023, 0.08094965120313709, 0, 0.004047300054177565,
      0.0068810163717335756,  },
        { 0.06403264148649551, 0.037536204282299424, 0, 0, 0, 0.04415962474807696, 0.01545640780655438,
      0.01545652152565455, 0, 0.12585861723836114, 0.45705807230179, 0.19872141593494547,
      -1.280653993800121, 0.037537332254881164, 0.008831407463926015, 0.0441615254400708,
      0.061826136036067555, 0, 0, 0.17001808728099788,  },
        { 0.016381113221917865, 0.0057333606860404 73, 0.005733309773825074, 0, 0, 0, 0,
      0.013923860791177009, 0.01638121601897183, 0.07371425418456577, 0.1367788303782746, 0,
      0.013924073789506735, -0.5585836389587654, 0.005733812245284265, 0.032760540286916225,
      0.008190569210524143, 0.008190657121324649, 0.21294800841803904, 0.008190032832397692,  },
        { 0.22174861267707138, 0.04306377607368807, 0.017354539055598216, 0.006427539913641191,
      0.006427553243697588, 0.05977467773808943, 0.02570986875833404, 0.03149448158849784,
      0.03213788843217168, 0.004499703534873253, 0.027637685180922314, 0.027637708148138533,
      0.0025708317741772, 0.00449970758917612, -0.7629424613771453, 0.172899056330566, 0.04692140383039991,
      0, 0, 0.03213742750810255,  },
        { 0.36143421801499853, 0.06414005151806881, 0.20225072639187414, 0.04588137695096761,
      0.05477749022865104, 0.02200455490688913, 0.04026329249382839, 0.21067897845479658,
      0.012172249896722092, 0.009363871273579105, 0.014981387493552335, 0.07865317044282724,
      0.009363941889092151, 0.018726766148170117, 0.1259399539335281, -1.6348763369454706,
      0.32585309481582925, 0.007959287178873075, 0.010299999151816374, 0.020131925761406284,  },
        { 0.3282952922440182, 0.011128890885073158, 0.09403653793252927, 0.031716843027645755,
      0.005564468728158905, 0.02058764655973212, 0.017249140769107628, 0.02782104925750872,
      0.007790290874907184, 0.0717802638409045, 0.028933848903590 01, 0.111286066046251 87,
      0.015580625618190439, 0.00556447197979171, 0.040620012061847 36, 0.387275473642870 96,
      -1.321525317126502, 0, 0.012797746419287583, 0.10349664833508647,  },
        { 0, 0.08381374624212419, 0.009312514770061871, 0, 0, 0, 0, 0, 0.009312704114196284, 0,
      0.04035445652633477, 0, 0, 0.03104238755758757, 0, 0.05277142405607509, 0, -0.24523145375603297,
      0.018624220489653193, 0,  },
        { 0.02177764551971404, 0.003266992928282102, 0.039198431494089775, 0, 0.032666021647431445, 0,
      0.01088819176648503, 0, 0.04355455931352278, 0.014155380310785223, 0.025043435162643377,
      0.010888178755858558, 0, 0.2831049669341572, 0, 0.02395517585860168, 0.025043644567386468,
      0.0065330448528687 18, -0.5585870939834543, 0.018511424871627873,  },
        { 0.18371687734104064, 0.010066150799688667, 0.00654315845880023, 0.0085570644112352,
      0.016610470013590294, 0.013589935843811825, 0.018623063969045898, 0.048823509184924 06,
      0.01510039393806871, 0.3327057802719861, 0.15250862308494223, 0.008556986906478721,
      0.038757020329067055, 0.005033128122162629, 0.025166488861068 59, 0.021643422211177053,
      0.093620025137251 34, 0, 0.008556936037263504, -1.0081790349221962,  },
              }, new double[20] {
0.087127, 0.040904, 0.040432, 0.046872, 0.033474, 0.038255, 0.04953, 0.088612, 0.033619, 0.036886, 0.085357,
      0.080481, 0.014753, 0.039772, 0.05068, 0.069577, 0.058542, 0.010494, 0.029916, 0.064718,  })
```

Dayhoff rate matrix prepared using the DCMut method.

Citation: Kosiol, C., and Goldman, N. 2005. Different versions of the Dayhoff rate matrix. Molecular Biology and Evolution 22:193-199.

Definition at line 173 of file RateMatrices.cs.

7.22.3.4 DayhoffMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.DayhoffMatrix
[static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
      { -1.3634324059950833, 0.011242118443529615, 0.0403338717440125, 0.05725499007813634,
      0.012266727712977061, 0.034657467579094915, 0.09982794497014537, 0.21648229991482412,
      0.007870789934693798, 0.024405827157193387, 0.03562391223475259, 0.02130054526124033,
      0.010812632726865662, 0.007287331878480668, 0.12897193852228542, 0.28967273494278667,
      0.22108540714005337, 0, 0.007308586291080078, 0.137027279462931,  },
      { 0.02394611905020058, -0.8852989321897626, 0.013170243834779593, 0, 0.007837076038846456,
      0.09579479802761066, 0.0005041815402532595, 0.008118086246805904, 0.08212998192723962,
      0.02403035289323704, 0.013033138622470464, 0.3801328077390582, 0.013515790908582079,
      0.005667924794373852, 0.0531364386711816, 0.10906992953835976, 0.015493856025987567,
      0.021471170962385596, 0.0024361954303600258, 0.0158108399380305,  },
      { 0.08691554321924656, 0.013323992229368432, -1.8300026751803307, 0.4317980501726116, 0,
      0.04010920405221097, 0.07461886795748239, 0.12537933203400228, 0.1830814180461383,
      0.028911518324675818, 0.029541780877599715, 0.2605220535797856, 0.0001501754545398008,
      0.005667924794373852, 0.021667285671743954, 0.3505819163732993, 0.1364651165365828,
      0.002456900159874969, 0.02892982073552531, 0.009881774961269062,  },
      { 0.10642719577866926, 0, 0.37247095845236033, -1.4469383668345417, 0, 0.05218090624268223,
      0.5813213159120081, 0.11275119787230423, 0.0294299101905942, 0.00901138233496389, 0,
      0.05816687359800243, 0, 0, 0.006706540803158843, 0.06728339809184532, 0.03933055760442998, 0, 0,
      0.011858129953522875,  },
      { 0.031928158733600775, 0.00957661941485856, 0, 0, -0.2724085940053223, 0, 0, 0.009922105412762 77,
      0.009581831224844623, 0.016520867614100467, 0, 0, 0, 0, 0.0098018673276936 92, 0.11402765360828522,
      0.009534680631376963, 0, 0.029234345164320313, 0.03228046487347894,  },
      { 0.07893350353584636, 0.10242819026326982, 0.04239172234319682, 0.06393473892058558, 0,
      -1.267202330219789, 0.3609939828213337, 0.025256268323396145, 0.20737820436628004,
      0.006758536751222917, 0.06342794129602292, 0.12534551634499116, 0.017120001817537298, 0,
      0.07893082637563887, 0.03966179255940355, 0.031583629591436194, 0, 0, 0.023057474909627815,  },
      { 0.17560487303480427, 0.0004163747571677635, 0.060912377735855616, 0.5501250296674267, 0,
      0.2788173796250782, -1.3922160008508242, 0.07306277622125314, 0.0147149550952971,
      0.022903930101366557, 0.009557634989811672, 0.06799789448780566, 0.004505263636194026, 0,
      0.026310275454854623, 0.05595145736058715, 0.020261196341676047, 0, 0.006699537433490071,
      0.024375044904463686,  },
      { 0.2128543915573385, 0.0037473728145098716, 0.05720824665732386, 0.05964061466472536,
      0.0037481668011874352, 0.010903472946232107, 0.04083870476051401, -0.6659124733120019,
      0.0034220825803016513, 0, 0.006082131357152882, 0.022119797002057264, 0.0025529827271766146,
      0.006072776565400556, 0.01754018363903082, 0.16572963319465053, 0.01787752618383181, 0, 0,
      0.03557438986056862,  },
      { 0.020398545857578276, 0.09992994172026323, 0.22019001411272132, 0.04103274288933104,
      0.009540788221204 38, 0.2359823073363092, 0.021679806230890154, 0.009020095829784339,
      -0.8963041168652541, 0.0026283198476978018, 0.03823053995924669, 0.02130054526124033, 0,
      0.01943288500928178, 0.04849344888437933, 0.02478862034962722, 0.013110185868143324,
      0.0028841871442010503, 0.03867460245696541, 0.02898653988638925,  },
      { 0.05764806438011251, 0.026647984458736864, 0.03169089922743839, 0.011450998015627269,
      0.014992667204749741, 0.007009375465434926, 0.030755073955448825, 0, 0.0023954578062111557,
      -1.307919107128715, 0.22330110839832726, 0.037685580077579044, 0.05045895272537309,
      0.07935094712123393, 0.0061906530490697005, 0.016997911096887235, 0.11441616757652358, 0,
      0.011267403865415119, 0.5856598627045464,  },
      { 0.03636262522437866, 0.006245621357516452, 0.013993384074453318, 0, 0, 0.028426911609819423,
      0.005545996942785853, 0.006314067080849036, 0.015057163353327263, 0.09649688583690501,
      -0.5428351122425857, 0.014746531334704842, 0.07914246454247505, 0.06356172805119248,
      0.016508408130852536, 0.012040187026961791, 0.01966527880221499, 0.004918300319749938,
      0.008526684006260091, 0.11528737454813906,  },
      { 0.023059225752045002, 0.19319788732584228, 0.13087929810812218, 0.03387586912956401, 0,
      0.05957969145619687, 0.04184706784102053, 0.02435425874041771, 0.008897414708784293,
      0.017271816142014125, 0.015639766346964557, -0.7617023241918524, 0.036492635453171614, 0,
      0.017024295884941677, 0.06799164438754894, 0.08104478536670419, 0, 0.003958817574335042,
      0.006587849974179375,  },
      { 0.06385631746720155, 0.037473728145098716, 0.00041157011983686227, 0, 0, 0.04439271128108787,
      0.015125446207597783, 0.015334162910633373, 0, 0.12615935268949446, 0.4578976036027956,
      0.19907817301851538, -1.2815987439676373, 0.03724636293445674, 0.008770091819515 41,
      0.043911270333625366, 0.061975424103950266, 0, 0, 0.16996652933382786,  },
      { 0.015964079366800388, 0.005829246600348689, 0.005761981677716071, 0, 0, 0, 0,
      0.013530143744676508, 0.016425996385447925, 0.07359295573553844, 0.13641351758185752, 0,
      0.013816141817661 68, -0.5559170097905118, 0.005674765294980558, 0.0325793296023672,
      0.007746928012993783, 0.00811845270219555, 0.21255805129891228, 0.00790541996901525,  },
      { 0.2217233245388943, 0.04288659998827964, 0.017285945033148214, 0.006202623925131438,
      0.006474106292960116, 0.05957969145619687, 0.025713258552916233, 0.030668325821266745,
      0.03216757625483552, 0.004505691167481945, 0.02780402906127032, 0.027035307446958878,
```

```
      0.0025529827271766146, 0.0044533694812937405, -0.7606764221492248, 0.17352034244739054,
      0.04648156807796269, 0, 0, 0.031621679876061,  },
        { 0.36273935894563103, 0.06412171260383558, 0.20372720931924682, 0.045326867145191276,
      0.054859532271925186, 0.021806945892464215, 0.03983034168000749, 0.21107024241695352,
      0.011977289031055778, 0.00901138233496389, 0.014770890438799858, 0.07864816711842583,
      0.009310878181467653, 0.01862318146722837, 0.12639249975183972, -1.6381001567603015,
      0.3277546467035831, 0.008011630956114029, 0.01035383057903011, 0.019763549922538124,  },
        { 0.32903741361571914, 0.01082574368636185, 0.09424955744264146, 0.03149024454297499,
      0.00545187898354536, 0.02063871664822506, 0.01714217236861082, 0.027060287489353015,
      0.007528581676663632, 0.07209105867971112, 0.02867290496943502, 0.11141823675110325,
      0.015618247272139 29, 0.005263073023347149, 0.040239244818953 05, 0.3895354626369992,
      -1.3224820952097895, 0, 0.012790026009390136, 0.1034292445946162,  },
        { 0, 0.08369132619072046, 0.009466112756247832, 0, 0, 0, 0, 0, 0.009239622966814458, 0,
      0.039968291775576086, 0, 0, 0.030768734598029483, 0, 0.053118472177772616, 0, -0.24482855062165615,
      0.0185759901564952, 0,  },
        { 0.02128543915573385, 0.003330998057342108, 0.03909916138450192, 0, 0.03271127390127216, 0,
      0.011091993885571706, 0, 0.04346044876983097, 0.013892547766402665, 0.024328525422861153,
      0.010650272630620164, 0, 0.2825865361766392, 0, 0.024080374053923582, 0.02502853665736453,
      0.006516126510972743, -0.5565082143064893, 0.01844597992770225,  },
        { 0.18447380601636001, 0.009992994172026325, 0.006173551797552934, 0.008588248511720452,
      0.016696379387107663, 0.013629341182790135, 0.018654716989370598, 0.04870851748083542,
      0.015057163353327263, 0.3337966206576208, 0.15205328392882206, 0.008192517408169358,
      0.038745267271268626, 0.004858221252320445, 0.024762612196278802, 0.021247388871109045,
      0.09355905369538646, 0, 0.008526684006260091, -1.0077163681783265,  },
                }, new double[20] {
0.087127, 0.040904, 0.040432, 0.046872, 0.033474, 0.038255, 0.04953, 0.088612, 0.033618, 0.036886, 0.085357,
      0.080482, 0.014753, 0.039772, 0.05068, 0.069577, 0.058542, 0.010494, 0.029916, 0.064718,  })
```

Dayhoff protein sequence evolution matrix.

Citation: Dayhoff, M. O., Schwartz, R. M., and Orcutt, B. C. (1978) A model of evolutionary change in proteins. Matrices for detecting distant relationships In M. O. Dayhoff, (ed.), Atlas of protein sequence and structure, volume 5, pp. 345-358 National biomedical research foundation Washington DC.

Definition at line 202 of file RateMatrices.cs.

7.22.3.5 JTTDCMutMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.JTTDCMutMatrix
[static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
        { -1.2928238294558667, 0.0281468321036338, 0.024614600367361905, 0.04398651167844037,
      0.012079403154917593, 0.023702588958263795, 0.06837370272266732, 0.13480824638984204,
      0.005241984242941388, 0.019714444850004956, 0.029286525644394715, 0.02279125790163828,
      0.011395663392627488, 0.005811688665702252, 0.10267370547483 52, 0.27497565072872 24,
      0.27805047906557645, 0.001253517732522138, 0.004672159901677867, 0.2012448664800967,  },
        { 0.042372677774830114, -1.0730468079009121, 0.019899963891619254, 0.008234343880836473,
      0.02144398001615488, 0.1286615569965933, 0.020414596270321 92, 0.105330778291203, 0.07651173701536058,
      0.013037891186496875, 0.03516769402919296, 0.402801924578646, 0.01046462729060837,
      0.0027447855893767354, 0.037226258193186555, 0.07085037488484398, 0.0394563353990006,
      0.018699100194728757, 0.007891244981756711, 0.01183693743615493,  },
        { 0.04446780927551758, 0.023880798580698637, -1.344537769632745, 0.295009899334524,
      0.0065879109067194675, 0.03273313804685938, 0.037056873750929115, 0.05990772650112313,
      0.09593610420320996, 0.026763283874008 76, 0.012969764615616119, 0.1560506238237598,
      0.008029002859446 23, 0.0030879993553142035, 0.006381952645238555, 0.35780369202771006,
      0.14266751262276905, 0.00041174970876997505, 0.023469085954652 38, 0.011322841545878 6,  },
        { 0.0659441623715 7509, 0.008200294437649804, 0.24481638372284728, -1.1118339420349186,
      0.0022209500768317863, 0.022209099141806955, 0.4978323036565298, 0.09857409362409543,
      0.024601175148095643, 0.0063211111081400822, 0.005808615017632677, 0.017425854623235242,
      0.004612719437281214, 0.001366719492569896, 0.006662791256273321, 0.041685204954757415,
      0.025796832915417768, 0.0008542097026778119, 0.015204716626662972, 0.021696704747577775,  },
        { 0.04578367203970986, 0.053990102454993825, 0.013821650842609915, 0.00561496570289900 16,
      -0.5688423247061868, 0.003887271421698895, 0.003455410936044447, 0.04232816825169 39,
      0.017277029104714372, 0.008206575635560036, 0.015549187971200196, 0.003023456661085355,
      0.009934337288616 1, 0.028506662166914717, 0.006478831486992899, 0.15246952911127407,
      0.028506853743706 05, 0.016413220403586278, 0.07083507808607835, 0.042760321396808555,  },
        { 0.04436882668493392, 0.15998327160992334, 0.033916955505819, 0.027730408511758134,
      0.0019198260432194024, -1.085975363440827, 0.21907319436408487, 0.017918097449998605,
      0.1354541882784847, 0.004266292118008781, 0.06697998376803889, 0.18302323301954904,
```

```
      0.011092342269847127, 0.0019198034124306795, 0.08425818511359968, 0.038822967267195155,
      0.03178346454366181, 0.002559775102831853, 0.008532456156288018, 0.012372092676153421,  },
        { 0.0850103451742099, 0.016860369488415176, 0.02550342527672323, 0.41286581003181216,
      0.0011334888122297854, 0.14550896851801504, -0.9954711393982095, 0.08642681130654857,
      0.005809101308966388, 0.006092452649881123, 0.009209459632988349, 0.10683081729263876,
      0.004250574312540166, 0.0018418900633370018, 0.01005957617294942, 0.022102847266284607,
      0.020119101431290l4, 0.0017002290050114987, 0.0021252680446281077, 0.03202060360974005,  },
        { 0.13868393385464625, 0.0719794623124709, 0.0341145452220037, 0.06764187710487657,
      0.011488782878390938, 0.009847351224595027, 0.07151143661122189, -0.6464123500849578,
      0.004806506586645025, 0.002930806961712203, 0.006564946082l216215, 0.016646933123044185,
      0.0031652556960925855, 0.0021101321923903704, 0.010902458772896487, 0.13258892881656903,
      0.01922583332646164, 0.008089023701026895, 0.0017584405904144183, 0.032355695027378,  },
        { 0.0175307572066728, 0.16997170764448788, 0.17759637512203677, 0.054878721170766025,
      0.015244349006417905, 0.24199993146686077, 0.01562540325111178, 0.01562517221923145,
      -1.1764684141862969, 0.009908786400262l4, 0.05106802896101087, 0.032394152072257l66,
      0.008003268875922363, 0.019055066910921306, 0.05983334721937921, 0.05259270672300895,
      0.028963863346040532, 0.0019055390042003399, 0.19588693236151777, 0.008384305224189346,  },
        { 0.0288248142452982, 0.012662892775380375, 0.021660497169502496, 0.006164793776414593,
      0.003165765894605603, 0.003332348354687336, 0.007164591733067988, 0.004165426607646437,
      0.004332089973886571, -1.3316082253048758, 0.22060181933545445, 0.012496427761896219,
      0.11730001248756994, 0.03265678772772709, 0.0051651250514565756, 0.028658383869591374,
      0.15495404467597154, 0.001999438748254489, 0.010163619142062917, 0.6561393459744018,  },
        { 0.024706357454966665, 0.01970735645584512, 0.0060564762250003l2, 0.0032685612422101587,
      0.0034608552520328916, 0.03018587342362003, 0.006248738291878475, 0.005383470509374663,
      0.01288205057140096, 0.12728229347329637, -0.6981256368991662, 0.009036686224416l2,
      0.0936354296765098, 0.10507350553334836, 0.05556538626059955, 0.04191461690257789, 0.0165349860290201,
      0.007883059658979l6, 0.008075228108673786, 0.1212247056054147,  },
        { 0.029442698323233072, 0.34565628866872716, 0.11158912637745276, 0.015015733985657l6,
      0.0010304998089036592, 0.12630873257951028, 0.11100005252329369, 0.020904214618224l3,
      0.012513274346645l3, 0.011041122575802921, 0.013838137728878968, -0.9308324462978204,
      0.015163167135207383, 0.0010304817870151412, 0.011335453228417254, 0.033564885037917l9,
      0.058590973102479134, 0.0013249421856138253, 0.0029442659363769336, 0.008583963484636l4,  },
        { 0.03740896385428094, 0.022819359168727748, 0.014589645325787961, 0.010100346494830895,
      0.008604186635168952, 0.019452578198607365, 0.011222794225729608, 0.010100320922433648,
      0.007855946381452279, 0.2633614229289768, 0.364363954610766l7, 0.03853156736185l91,
      -1.2023274819127545, 0.018330271046941446, 0.008604088155051141, 0.020200984726305084,
      0.128312666267339, 0.0029927514752885976, 0.006359525998133059, 0.20911610813507453,  },
        { 0.011021416585818072, 0.0034576984415694l5, 0.0032415993232469l3l, 0.00172885126238751l6,
      0.014263177944309487, 0.0019449555370791053, 0.0028094163265604l1, 0.003889869642789394,
      0.010805393605075363, 0.042357134815171116, 0.23620409974460654, 0.00151274624633177l9,
      0.010589315723984385, -0.6593402855810772, 0.007779805873558802, 0.06677127905040l4,
      0.008428101673216818, 0.007996000968987817, 0.18368958456784779, 0.040843989393496l6,  },
        { 0.15617245211364647, 0.03761301877166006, 0.0053733586092836l2, 0.006759966851062235l,
      0.0026000202589394634, 0.06846602823853234, 0.012306716516499ll, 0.016119811302900897,
      0.027213445324606l4, 0.0053733368722793l2, 0.10018637512050751, 0.013346726751056158,
      0.00398670387204875, 0.006239917914496521, -0.7498377485952056, 0.19725366998899535,
      0.0714129684775884, 0.0010400024683787474, 0.0038133040232129814, 0.014559925119511456,  },
        { 0.30978641944025004, 0.0530217308976984l5, 0.22313104992320928, 0.03132515606926285,
      0.04531959810696265, 0.02336547979418542, 0.020027819978039055, 0.14519969553098042,
      0.017716939224843017, 0.022081972614738717, 0.05597482829770282, 0.029271433198768l4,
      0.006932735161329532, 0.039669871659820964, 0.14609926642556123, -1.5126034362160459,
      0.28988722192837724, 0.004621808725585633, 0.021054663288747173, 0.028115745949982l3,  },
        { 0.3652126853607153, 0.03442568297731935, 0.10372760504542759, 0.022601213758852892,
      0.009878849021986652, 0.022301870153239985, 0.021254363622857177, 0.024546958391490736,
      0.011375584799242108, 0.13920125729811592, 0.02574454205697478, 0.059572195181846l7,
      0.051339976895715l3, 0.005837365610845853, 0.061667181433225l3, 0.337973883524104l3,
      -1.3833244226535613, 0.0011974335573889572, 0.006735463849310419, 0.07873031011490168,  },
        { 0.006720694751472976, 0.06659597925797058, 0.0012219798485858927, 0.0030548602990087ll,
      0.023217333744721406, 0.007331677280788136, 0.007331763189858459, 0.042157039397218l4,
      0.003054896968020ll77, 0.007331786799455233, 0.05009998943842553, 0.005498842784573897,
      0.004887854564900059, 0.02260588164572239, 0.0036658345934789944, 0.021995179987659024,
      0.004887794148387765, -0.3232055466560041, 0.025049873804275216, 0.01649628415148131,  },
        { 0.011116972242911317, 0.012472627775548787, 0.030910928738093684, 0.024131833474673684,
      0.044468456443908705, 0.010845778479811236, 0.004067124052004178, 0.00406711854230947l,
      0.13937000793934815, 0.016539989929081064, 0.022776277999237756, 0.005422961789386583l,
      0.004609539105355152, 0.23047205716295296, 0.005965200721326142, 0.04446814236680109,
      0.012201525354733218, 0.011117078626074653, -0.6464118214487702, 0.011388084237174712,  },
        { 0.2330443084248831, 0.0091053502075777, 0.007257986808252491, 0.01675909777478478,
      0.013064401084850704, 0.007653757455864278, 0.029823631469462892, 0.036421240218692856,
      0.0029031923187926557, 0.51967019131781, 0.16640407618065717, 0.007653863047893564l,
      0.07376750769690897, 0.024940592553083965, 0.011084794289016073, 0.028899821728953477,
      0.06941182220905501, 0.0035630025250193756, 0.005542370282240858, -1.2669710075937994,  },
                }, new double[20] {
0.076862, 0.051057, 0.042546, 0.051269, 0.020279, 0.041061, 0.06182, 0.074714, 0.022983, 0.052569, 0.091111,
      0.059498, 0.023414, 0.04053, 0.050532, 0.068225, 0.058518, 0.014336, 0.032303, 0.066374,  })
```

JTT rate matrix prepared using the DCMut method.

Citation: Kosiol, C., and Goldman, N. 2005. Different versions of the Dayhoff rate matrix. Molecular Biology and Evolution 22:193-199.

Definition at line 231 of file RateMatrices.cs.

7.22.3.6 JTTMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.JTTMatrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
  { -1.2950937799320588, 0.030917490151747835, 0.023747789032223136, 0.0430550839914495,
0.01143616250764773, 0.023954388392780166, 0.0669498871239073, 0.1350331935609873,
0.006388430901105962, 0.019958649097766023, 0.02843262598477658, 0.021178239133551047,
0.01326802254617771, 0.0062069526368011465, 0.10183318987882857, 0.26805292506989725,
0.286875236434852, 0.001323590961771736, 0.0036415483377568894, 0.20284037418803064,  },
  { 0.045904616551553326, -1.064824819239263, 0.01978982419351928, 0.008504707948928296,
0.023076542202932027, 0.13027825266248863, 0.018490921205650587, 0.10334942747405174,
0.07760760502084281, 0.01219695222641257, 0.036014659580717, 0.3908897851506851, 0.010810981333922578,
0.002068984212267049, 0.038843587891924304, 0.0716226069631207, 0.038652663435432694,
0.018530273464804306, 0.0066209969777397985, 0.011571430742270204,  },
  { 0.042738780927308227, 0.023987707876356082, -1.3290708876081345, 0.2806553623146338,
0.006943384379643266, 0.036141708803142006, 0.036981842411301175, 0.061104402602480432,
0.09251394379009006, 0.026057125210972306, 0.011373050393910632, 0.1591393397749693,
0.007371123636765393, 0.004137968424534098, 0.007873700248363036, 0.3566947653707892,
0.14011590495344435, 0.0011765252993526543, 0.023173489422089296, 0.010890758345666074,  },
  { 0.0641081713909624, 0.008528962800482163, 0.23220060387062622, -1.0844802001931104,
0.0020421718763656664, 0.020592368969232073, 0.48905298499082756, 0.09806879979289582,
0.026500157811995106, 0.006098476113206285, 0.006634279396447868, 0.015732406213495066,
0.0036855618183826966, 0.001655187369813639, 0.007873700248363036, 0.0418389482622021,
0.022950018914778816, 0.0005882626496763272, 0.015228293048801536, 0.021100844294728017,  },
  { 0.044321698739430794, 0.06023579977840527, 0.014952311612881235, 0.005315442468080185,
-0.5871369440659445, 0.0037822718514916058, 0.0031880898630432045, 0.04450814759831426,
0.016325990080604126, 0.009424917629500621, 0.021798346588328712, 0.0042356478267102095,
0.007616827757990907, 0.03227615371136596, 0.007348786898472166, 0.15813704309679125,
0.025365810379502703, 0.016912551178194405, 0.0691894184173809, 0.042201688589456034,  },
  { 0.04511315764549206, 0.16524865425934188, 0.03782055290317018, 0.02604566809359291,
0.0018379546887291, -1.0836982081041098, 0.205950605152591, 0.019613759958579162, 0.1412553054800096,
0.004989662274441506, 0.06823830236346379, 0.17668702362848304, 0.010565277212697064,
0.001655187369813639, 0.086085789382102 51, 0.037584140287578185, 0.030801341175110426,
0.002647181923543472, 0.007945196373287758, 0.013613447932082592,  },
  { 0.08310318513643274, 0.015458745075873918, 0.025506884516091517, 0.407694437301750 15,
0.0010210859381828332, 0.13574153422575427, -0.9801115223946416, 0.089770670579650 78,
0.006151822349213149, 0.006652883032588674, 0.008529787795432973, 0.10952175094779257,
0.004422674182059236, 0.002068984212267049, 0.009448440298035642, 0.021274041672214 07,
0.019326331717716347, 0.0014706566241908178, 0.0023173489422089297, 0.030630257847185833,  },
  { 0.1416711441849663, 0.07302924397912851, 0.035621683548334705, 0.0691007520850424,
0.012048814070557431, 0.010926563126531305, 0.07587653874042827, -0.672368935861015,
0.005441996693534709, 0.003326441516294337, 0.005686525196955316, 0.016337498760167952,
0.0034398576971571837, 0.002068984212267049, 0.012597920397380856, 0.14253607920383424,
0.01993027958389498, 0.008088611433049498, 0.002648398791095919 4, 0.031991602640394 09,  },
  { 0.021369390463654134, 0.17484373740988432, 0.1719515835481342, 0.059532955642498074,
0.0140909859469231, 0.2508906994822765, 0.016578067287824662, 0.017350633809512336,
-1.178711030662898, 0.008870510710118231, 0.05307423517158295, 0.02722916460027992,
0.0081082360004419 34, 0.016551873698136 39, 0.060365033523744993, 0.05176683473572089,
0.027781601844217244, 0.0011765252993526543, 0.189691563412245 22, 0.007487396362645425,  },
  { 0.028492520618205515, 0.011727323850662974, 0.02066937193545347, 0.005846986714888204,
0.0034716921898216 33, 0.0037822718514916058, 0.0076514156713036905, 0.004526252298133653,
0.0037857368302850153, -1.321841529041229, 0.21703571168379457, 0.012706943480130 63,
0.117692274067020 79, 0.036827918978353 47, 0.005249133498908689, 0.028365388896285423,
0.14796722721376576, 0.001323590961771736, 0.010593595164383678, 0.6541261731365685,  },
  { 0.023743767181837926, 0.020256286651145136, 0.005277286451605142, 0.0037208097276561296,
0.0046969953156410 32, 0.030258174811932846, 0.005738561753477768, 0.004526252298133653,
0.013250078905997553, 0.1269591845385672, -0.6934459584265384, 0.008471295653420419,
0.095333199035499 1, 0.102621616928445 61, 0.053541161688868 63, 0.041838948622021, 0.015098696654465895,
0.007647414445792253, 0.007945196373287758, 0.122521031388743 33,  },
  { 0.027701061712144245, 0.34435687306946733, 0.11566052806434601, 0.013820150417008481,
0.0014295203134559664, 0.12271370895950542, 0.1154088530421 64, 0.020368135341601437,
0.010647384835176606, 0.011642545307030 18, 0.013268558792895737, -0.934850886589255,
0.015970767879658353, 0.001655187369813639, 0.011023180347708248, 0.033329331953135 37,
0.062206630216399 49, 0.0014706566241908178, 0.0026483987910959194, 0.009529413552457816,  },
  { 0.04273878092730827, 0.023454647701325947, 0.013193216129012853, 0.007973163702120277,
0.006330732816733566, 0.018070854401571003, 0.011477123506955535, 0.010561255362311856,
0.007808082212462843, 0.26556091438416457, 0.3677286294031104, 0.039331015533737 66,
-1.2268230758919016, 0.017793264225496618, 0.008398613598253904, 0.020564906949806932,
0.13649222177563717, 0.0035295758980579 63, 0.005958879279965819, 0.219857184103133 87,  },
  { 0.011871883590918963, 0.0026653008751506757, 0.0043977387096709505, 0.002126176987232074,
0.015928940635652197, 0.0016810097117740468, 0.0031880898630432045, 0.0037718769151113773,
0.009464342075712538, 0.049342215825032 67, 0.235043041474153 06, 0.0024203701866915483,
0.010565277212697064, -0.6613164475676792, 0.008923526948144772, 0.065240394461456 46,
0.007247374394143629, 0.007794480108211334, 0.177442719003426 6, 0.042201688589456034,  },
  { 0.15354302777588524, 0.03944645295223, 0.006596608064506427, 0.007973163702120277,
0.0028590406269119327, 0.06892139818273592, 0.011477123506955535, 0.018105009192534 61,
0.027209983467673546, 0.005544069193823895, 0.096670928342403 6, 0.012706943480130 63,
0.0039312659396082104, 0.007034546321707966, -0.7552371624354566, 0.202103395886033 65,
0.07126584820907902, 0.0008823939745144908, 0.0033104984888698992, 0.015655465121894 98,  },
  { 0.2991714664911579, 0.05383907767804365, 0.22120625709644887, 0.031361110561673095,
0.04554043284295436, 0.022273378681006123, 0.019128539178259227, 0.151629451987477 38,
```

```
      0.01727242428817538, 0.02217627677529558, 0.0559174977700606, 0.028439349693625694,
      0.007125419515539881, 0.038069309505713696, 0.14960030471889765, -1.5027024186870435,
      0.28808313216720927, 0.005147298184667862, 0.020856140479880367, 0.02586555107095693,  },
        { 0.3759429803791005, 0.03411585120192865, 0.10202753806436607, 0.020198681378704703,
      0.008577121880735799, 0.021432873825119096, 0.02040377512347651, 0.02489438763973509,
      0.010883993387069418, 0.13582969524868543, 0.023693854987313818, 0.062324532307307376,
      0.05552913139696596, 0.004965562109440917, 0.061939775287122543, 0.33825726258820366,
      -1.3859691600005937, 0.0017647879490289816, 0.0069520468266267895, 0.07623530841966253,  },
        { 0.007123130154551379, 0.06716558205379702, 0.003518190967736761, 0.002126176987232074,
      0.023484976578205162, 0.007564543702983211 5, 0.006376179726086409, 0.04149064606622515,
      0.0018928684151425076, 0.004989662274441506, 0.049283218373612735, 0.00605092546672887,
      0.005896898909412315, 0.021931232650030715, 0.003149480099345214, 0.024819715284249744,
      0.007247374394143629, -0.3246321512900039, 0.023504539270976288, 0.01701680991510324,  },
        { 0.008706047966673906, 0.010661203500602703, 0.030784170967696655, 0.024451053553168848,
      0.042681392216042426, 0.010086058270644281, 0.004463325808260486, 0.006035003064178204,
      0.1355767002345821, 0.017741021420236463, 0.022746100787821265, 0.004840740373383097,
      0.004422674182059236, 0.22179510755502763, 0.005249133498908689, 0.04467548751164954,
      0.012682905189751351, 0.010441662031754806, -0.6289305282781077, 0.010890758345666074,  },
        { 0.23585475400625674, 0.009062022975512297, 0.007036381935473522, 0.016477871651048574,
      0.012661465633467131, 0.008405048558870235, 0.02869280876738884, 0.03545564300204695,
      0.002602694070820948, 0.5327850495264763, 0.17059575590865947, 0.008471295653420419,
      0.07936243115584074, 0.025655404232111403, 0.012073007047489986, 0.026947119451471154,
      0.06764216101200721, 0.0036766415604770446, 0.005296797582191839, -1.2887543537310306,  },
                }, new double[20] {
0.076748, 0.051691, 0.042645, 0.051544, 0.019803, 0.040752, 0.06183, 0.073152, 0.022944, 0.053761, 0.091904,
      0.058676, 0.023826, 0.040126, 0.050901, 0.068765, 0.058565, 0.014261, 0.032102, 0.066005,  })
```

JTT protein sequence evolution matrix.

Citation: David T. Jones, William R. Taylor, Janet M. Thornton, The rapid generation of mutation data matrices from protein sequences, Bioinformatics, Volume 8, Issue 3, June 1992, Pages 275–282

Definition at line 260 of file RateMatrices.cs.

7.22.3.7 LGMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.LGMatrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
        { -1.1067338146342782, 0.02415402492707075, 0.011802691482520505, 0.021292785735778922,
      0.03270758377243398, 0.04016135501449744, 0.0755142209642656, 0.1203231027795738,
      0.008148405583470372, 0.00945941231358669, 0.03978626533223938, 0.035204010843190754,
      0.026203347629781198, 0.010900800991046372, 0.05267920916709539, 0.29383788344986334,
      0.11580014295907595, 0.0022148160423889206, 0.007596130578979939, 0.17894762506741882,  },
        { 0.034138862996438674, -0.8972349426496831, 0.0320578288496216, 0.006679402858433231,
      0.007024219195952 55, 0.1162698089028775, 0.026464824349800683, 0.022724202880760998,
      0.05509959130702056, 0.008017488013846944, 0.030377638870143175, 0.4150893935762662,
      0.01128604118025583, 0.002265312434125001, 0.014875014297072504, 0.05334198546203291,
      0.03133757701980345, 0.007275078196707338, 0.010908559589943561, 0.012002112668580422,  },
        { 0.022231021863329113, 0.04272213363691264, -1.2057263781461713, 0.27353408635810855,
      0.006948228206112117, 0.07021767130072365, 0.03938872140060782, 0.083726310766268,
      0.10238896749242533, 0.012090408037701342, 0.00688641533144923, 0.14075082135453035,
      0.008648793662154066, 0.003846631305053691, 0.007237128158951048, 0.24915635379160933,
      0.10828642483234224, 0.0005561153225177468, 0.021232385138771175, 0.00587776018660377,  },
        { 0.03173368387591601, 0.007043136456752117, 0.21643180922593536, -0.8768633936933968,
      0.0008220114750921946, 0.02167235899553741, 0.3812899372563379, 0.049207236035669275,
      0.021051504757072577, 0.0006749056773159029, 0.0015172314661892029, 0.018566556395458143,
      0.0005955714237062459, 0.0007483153399476692, 0.017644981519325996, 0.07709451019566822,
      0.023049603099298425, 0.0003663233204790077, 0.004687135098964841, 0.0026665820787303476,  },
        { 0.19989625249681264, 0.030373490456889665, 0.02254508583195241, 0.0033709015054951773,
      -0.881465950895653, 0.0035117282879051 72, 0.0002544177278345814, 0.033153148585610186,
      0.014544483215235169, 0.02024256151550664, 0.05978018781750131, 0.0008704580421267664,
      0.02083334389924056, 0.04748945095271597, 0.003372021206141705, 0.1730809437911865,
      0.06189066865163613, 0.008212897761992521, 0.04043466249836566, 0.13760924665150423,  },
        { 0.07789137526863038, 0.1595469222615319, 0.07230179282729847, 0.028203252371556665,
      0.0011144118738349452, -1.2691818974788556, 0.30019626789891457, 0.015605534402873038,
      0.10929780308105866, 0.004599586362504471, 0.05861780897467263, 0.212220505269286,
      0.038990695967470325, 0.0015405860423647036, 0.027926197326510697, 0.076072177721669 98,
      0.058462184974554555, 0.002894787620402179, 0.008927506330739462, 0.014772500902981838,  },
        { 0.08340467961278217, 0.020680981462188136, 0.023096979272948 82, 0.2825719239980337,
      4.597829387025368E-05, 0.17095662913747173, -0.8557924238442528, 0.020316331453091894,
      0.009624849681843527, 0.002794639832215944, 0.007011811352069536, 0.11857920648247577,
```

```
      0.004050086163206694, 0.0008082544705877128, 0.018761190231708975, 0.03803975625403533,
      0.03272089960471838, 0.0009541319219113988, 0.004164326318210327, 0.017209768300882663,  },
        { 0.16592194297521287, 0.02217093034781469, 0.0612968824151182, 0.04552980250386882,
      0.0074803753815518605, 0.011095641924096572, 0.025365207516979196, -0.5120723831682655,
      0.007072708326863787, 0.0005495840899003681, 0.004454376620124722, 0.019463982495425563,
      0.0032528904540912065, 0.0038492522992967324, 0.008810547196716409, 0.1081563982143966,
      0.007027352341146176, 0.003290549168241193, 0.0018969251043713395, 0.005387033793049132,  },
        { 0.028819585589920302, 0.13788084264397393, 0.19226041997000842, 0.04995859675116145,
      0.008416997510869934, 0.1993175369360554, 0.0308210462681481, 0.018140365794560007,
      -1.1104356813772154, 0.00687418895767167, 0.03686572559034427, 0.04575147416594887,
      0.01031484574096407, 0.02930954740908149, 0.0227621496214801, 0.06153836062795258,
      0.03162308553515779, 0.007317323646211892, 0.18410480512319857, 0.008358783494506673,  },
        { 0.012032721881462191, 0.007215700516154446, 0.00816511508275157, 0.0005760428590981431,
      0.004213170171116684, 0.003016737249870807, 0.0032185769427258487, 0.0005069662783373941,
      0.002472328042678221, -1.4372501178721084, 0.4171548210309419, 0.010437425772882753,
      0.09962573216498272, 0.04781067313240412, 0.003501700565625465, 0.00398471595097322,
      0.05595095491068815, 0.001368473133646236, 0.008066693173681614, 0.7479315690120867,  },
        { 0.03174918354436106, 0.0171511742517201, 0.0029175225963428337, 0.0008123874783689061,
      0.007805495400682415, 0.02411837000505121, 0.005066032109579534, 0.0025776949391719014,
      0.008317773292277491, 0.26169590749811017, -0.8062356289837133, 0.009022160469804795,
      0.14715281200107683, 0.11140050501605434, 0.011141062874448184, 0.011330815327276427,
      0.016396361631731245, 0.007594033178823971, 0.01039539519147503, 0.1195909421773568,  },
        { 0.04308731147566613, 0.35945070845278493, 0.09145970941175511, 0.015247568883774697,
      0.0001743206763311769 2, 0.13392559347233715, 0.13140264822375403, 0.017275640314864015,
      0.01583241803374283, 0.010042710120202372, 0.013837843367008185, -0.9917970816959694,
      0.015306661151214025, 0.001027687545984632, 0.017460057589658573, 0.04653754141365703,
      0.06153252460498217, 0.0006116337113357429, 0.004576987927173495, 0.013007515319751833,  },
        { 0.09027030995147402, 0.027508711152659637, 0.015818500786730045, 0.0013766831584882471,
      0.011743321424969507, 0.06925771001289108, 0.012632541853484138, 0.008126485990424273,
      0.010046986037177108, 0.2698111905441519, 0.6352685184034985, 0.043083539295386955,
      -1.5662058090195037, 0.07729153044389553, 0.004466489950015163, 0.021566758386236153,
      0.10935198049903058, 0.008532122369838514, 0.016697478635025367, 0.13335495012412696,  },
        { 0.020374514943928716, 0.0029956938886432478, 0.0038170782065207035, 0.0009384810508936634,
      0.01452345106555923, 0.001484683258216677, 0.0013677770443830553, 0.0052173556589470175,
      0.015488982372701448, 0.07025124130988708, 0.2609255694174195, 0.0015693966117584803,
      0.04193461101644949, -0.8233635354517814, 0.0042256057310361885, 0.022490382039859 29,
      0.008930652235446718, 0.030113774624914682, 0.2707331446415206, 0.04598114033369558,  },
        { 0.09457616603100734, 0.01889471332408113, 0.006898113731341692, 0.021255712070010962,
      0.000990550371113879, 0.02585075582220394, 0.030495880198163457, 0.011470716271982939,
      0.011554220135971 56, 0.004942216213841553, 0.025065114683542243, 0.025611256137419 25,
      0.002327666004604859, 0.004058845904502562, -0.42319902976049834, 0.08317722369405971,
      0.03093061237014482, 0.0011658984208928898, 0.003108856222279647, 0.020824512153333863,  },
        { 0.37963602942704533, 0.04876062566353879, 0.17090439503750812, 0.06683363489877918,
      0.036589181983211 26, 0.0506762499661637, 0.04449750790400465, 0.10133443476671826,
      0.022479697564225038, 0.004047224363361642, 0.018345156027940516, 0.04912536848738081,
      0.008088283277325782, 0.01554631993480281, 0.05985791675223278, -1.450887708044806,
      0.35031104611356817, 0.003049981739078181, 0.013895785580947473, 0.0069088685569738345,  },
        { 0.17182153439304704, 0.03289836913440098, 0.085302967988200 32, 0.022947952476663 73,
      0.015025795791585504, 0.04472625396171046, 0.04395740648006773, 0.007561455912029168,
      0.013266539252321435, 0.06526438914526325, 0.030487133950749024, 0.07459607576860863,
      0.04709849127241636, 0.007089617558951848, 0.025563161160905623, 0.40231172873331256,
      -1.253857044124528, 0.0017259110607713705, 0.008528721530820873, 0.1536835385527021,  },
        { 0.01451323099681107, 0.03372908581153698, 0.0019346969081159837, 0.0016106567875064 08,
      0.008805756534634281, 0.009780524359434413, 0.005660739910653852, 0.015636517293174643,
      0.013557000672225, 0.007049576045752452, 0.06235905862680738, 0.00327461774840784,
      0.016229134801107555, 0.10557540976157308, 0.0042554422721799165, 0.015469064519009399,
      0.007622130175312781, -0.4497155647482195, 0.109342837247099 49, 0.013310084276876978,  },
        { 0.017584414005493425, 0.017866658820700713, 0.02609491526775575, 0.0072803950013258035,
      0.015315568108369393, 0.010655764912465397, 0.008728076820828707, 0.0031844237947398473,
      0.12049957307946432, 0.014680177063285846, 0.030156233376271047, 0.008656812182562 08,
      0.011220138549332957, 0.3353111838567005, 0.004008608637950393, 0.024897683800241 33,
      0.013306104061275123, 0.03862774628088135, -0.7255847780856934, 0.017510300466049 44,  },
        { 0.20461730694868233, 0.009709896087943908, 0.0035682059865658152, 0.0020458951572852 39,
      0.025745886646282707, 0.008709424043152421, 0.017816802950048248, 0.0044669523858327 08,
      0.0027023674927284867, 0.6723239263465 41, 0.17136231711968256, 0.012152161187845727,
      0.04426265001082964, 0.028129842197000456, 0.013263214821074284, 0.006114539012265575,
      0.11843369515463918, 0.0023225805441276933, 0.008649172233327819, -1.3563968363254058,  },
              }, new double[20] {
0.079066, 0.055941, 0.041977, 0.053052, 0.012937, 0.040767, 0.071586, 0.057337, 0.022355, 0.062157,
      0.099081, 0.0646, 0.022951, 0.042302, 0.04404, 0.061197, 0.053287, 0.012066, 0.034155, 0.069147,  })
```

LG protein sequence evolution matrix.

Citation: Le, S. Q., and O. Gascuel. 2008. An improved general amino acid replacement matrix. Mol. Biol. Evol. 25:1307-1320.

Definition at line 289 of file RateMatrices.cs.

7.22.3.8 mtArtMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.mtArtMatrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
  { -1.134670547403153, 3.370446316512181E-05, 7.378664583737619E-05, 0.00018639435381819712,
0.022800818771334448, 3.4728892450987694E-05, 4.4914012498910615E-05, 0.12608061732525924,
4.5337467925739656E-05, 0.02226920654138213, 0.005497819811489196, 4.015984698634529E-05,
0.06874968051825148, 0.01065743002661581, 0.018948927673472008, 0.5664242417108847,
0.11097986395747535, 5.507879189314304E-05, 0.0003646801834152355, 0.18138315640899758,  },
  { 0.00010006861955580907, -0.09804006835306175, 7.3786645837376 19E-05, 0.0007455774152727885,
0.003231612109322992, 0.026741247187260526, 4.4914012498910615E-05, 0.00012608061732525926,
0.00929418092477663, 0.0017130158877986252, 0.002748909905744598, 0.041967040100730824,
0.002840895889183945, 0.004099011548698388, 7.734256193253882E-05, 0.0025249223256057266,
9.096710160448799E-05, 5.507879189314304E-05, 0.001458720733660942, 0.00010669597435823387,  },
  { 0.00010006861955580907, 3.3704463165121 81E-05, -1.0197695620084046, 0.09319717690909855,
0.00879716629760148, 0.04549484911079388, 0.041096321436503214, 0.07627877348178184,
0.04080372113316568, 0.017986666821885565, 0.017867914387339884, 0.09377324271311625,
0.044886155049106334, 0.01639604619479355, 0.006574117764265799, 0.3349730285303597,
0.07550269433172503, 0.002203151675725722, 0.09153472603722412, 0.012270037051196893,  },
  { 0.0005003430977790454, 0.0006740892633024363, 0.18446661459344044, -0.43333720116102487,
0.0009874370334042478, 3.4728892450987694E-05, 0.19357939387030473, 0.0075648370395155545,
4.5337467925739656E-05, 0.005995555607295189, 0.001374454952872299, 0.00040159846986345286,
0.0001136358355673578 1, 0.0001639604619479355 2, 7.734256193253882E-05, 0.03703219410888399,
9.096710160448799E-05, 5.507879189314304E-05, 7.293603668304711E-05, 0.00010669597435823387,  },
  { 0.12708714683587752, 0.006066803369721926, 0.03615545646031433, 0.0020503378920001683,
-1.5771152263270305, 3.4728892450987694E-05, 4.4914012498910615E-05, 0.05106265001673,
0.002720248075544379, 0.053960000465656695, 0.10858194127691162, 4.015984698634529E-05,
0.17727190348507815, 0.1508436249921007, 7.734256193253882E-05, 0.5588494747340674,
0.0832348979681065, 0.006058667108245735, 0.02625697320589696, 0.18671795512690925,  },
  { 0.00010006861955580907, 0.025952436637143795, 0.0966605060469628, 3.727887076363942E-05,
1.7953400607349 96E-05, -0.5070219107370064, 0.0588373563735729, 0.0018912092598788886,
0.07117982464341126, 0.009421587382892439, 0.021991279245956785, 0.07007893299117253,
0.03806800491506487, 0.0001639604619479355 2, 0.01508179957684507, 0.043765320310499255,
0.020012762352987355, 0.0019277577162600066, 0.031727175957125486, 0.00010669597435823387,  },
  { 0.00010006861955580907, 3.3704463165121 81E-05, 0.0675147809411992, 0.16067193299912859,
1.7953400607349 96E-05, 0.04549484911079388, -0.39743610272595964, 0.027737735811557033,
0.003400310094430474, 0.005995555607295189, 0.002748909905744598, 0.021284718902763,
0.0001136358355673578 1, 0.0001639604619479355 2, 0.0030937024773015527, 0.026090864031259174,
0.019557926844964917, 0.0030293335541228673, 0.002917441467321884, 0.007468718205076371,  },
  { 0.10006861955580908, 3.3704463165121 81E-05, 0.044640920731612584, 0.0022367322458183656,
0.007271127245976733, 0.0005209333867648154, 0.009881082749760335, -0.39713290679119967,
4.5337467925739656E-05, 0.0025695238316979377, 0.001374454952872299, 4.015984698634529E-05,
0.03181803395886019, 0.0008198023097396776, 7.734256193253882E-05, 0.1902108151956314,
9.096710160448799E-05, 0.0005507879189314305, 0.003282121650737 12, 0.0016004396153735 08,  },
  { 0.00010006861955580907, 0.006909414948849972, 0.06640798125363856, 3.727887076363942E-05,
0.0010772040364409974, 0.05452436114805068, 0.003368550937418296, 0.00012608061732525926,
-0.24070232554483748, 0.0001713015887798625 4, 0.008246729717233794, 4.015984698634529E-05,
0.0001136358355673578 1, 0.011477232336355487, 0.0003867126096626941, 0.009258048527220997,
0.008641874652426359, 5.507879189314304E-05, 0.06965391503230998, 0.00010669597435823387,  },
  { 0.01300892054225518, 0.00033704463165121814, 0.007747597812924499, 0.0013047604767273797,
0.005655321191315237, 0.0019100890848043233, 0.0015719904374618713, 0.0018912092598788886,
4.5337467925739656E-05, -2.2240595932646205, 0.707844300729234, 0.0006023977047951793,
0.29261227658594635, 0.09673667254928195, 7.734256193253882E-05, 0.005891485426413362,
0.09278644363657775, 5.507879189314304E-05, 0.004376162200982826, 0.9896051621726191,  },
  { 0.0020013723911161815, 0.00033704463165121814, 0.004796131979429452, 0.00018639435381819712,
0.007091593239903233, 0.0027783113960790154, 0.00044914012498910614, 0.0006304030866262962,
0.0013601240377721896, 0.441101591108146, -1.27161033335833, 0.0008031969397269057,
0.5028385723855583, 0.2156080074615352, 0.004640553715952329, 0.00673312620161527,
0.021832104385077114, 0.00578327314878002, 0.007293603668304711, 0.045345789102249394,  },
  { 0.00010006861955580907, 0.035221164007552 29, 0.17229178103027338, 0.0003727887076363942 3,
1.7953400607349 96E-05, 0.06060191732697353, 0.023804426624422625, 0.00012608061732525926,
4.5337467925739656E-05, 0.0025695238316979377, 0.005497819811489196, -0.5904469531358115,
0.060226992850699636, 0.009017825407136453, 0.006574117764265799, 0.12119627162907487,
0.031838485561570797, 0.004406303351451444, 0.042667581459582556, 0.013870476666570401,  },
  { 0.060541514831264485, 0.0008426115791280453, 0.029145725105763593, 3.727887076363942E-05,
0.028007304947465936, 0.011634178971080877, 4.4914012498910615E-05, 0.03530257285107259,
4.5337467925739656E-05, 0.441101591108146, 1.2163926332919845, 0.021284718902763, -2.531354626483312,
0.2639763437361762, 0.0019335640483134704, 0.0942637668226138, 0.13144746181848513,
0.019552971122065783, 0.025892293022481725, 0.14990784397331855,  },
  { 0.00650446027112759, 0.0008426115791280453, 0.007378664583737618, 3.727887076363942E-05,
0.016517128558761 96, 3.4728892450987694E-05, 4.4914012498910615E-05, 0.0006304030866262962,
0.0031736227548017754, 0.10106793738011889, 0.3614816526054146, 0.0022087915842489906,
0.18295369526344607, -1.0567822790168042, 0.005800692144940411, 0.030299067907268717,
0.006367697112314158, 0.014871273811148621, 0.28882670526486653, 0.027740953333140802,  },
  { 0.02451681179117322, 3.3704463165121 81E-05, 0.006271864896176976, 3.727887076363942E-05,
1.7953400607349 96E-05, 0.0067721340279426005, 0.0017965604999564246, 0.0001260806173252592 6,
0.0002266873396286982 6, 0.0001713015887798625 4, 0.016493459434467 59, 0.0034135869938393495,
0.002840895889183945, 0.012297034646095164, -0.19330604876941077, 0.07322274744256607,
0.021377268877054673, 5.507879189314304E-05, 0.00656424330147424, 0.017071355897317418,  },
  { 0.33673090480529755, 0.00050556694747682 72, 0.1468354252163786, 0.008201351568000673,
0.059605290016401864, 0.0090295120372568, 0.006961671937331145, 0.14247109757754295,
```

```
      0.0024935607359156805, 0.005995555607295189, 0.010995639622978393, 0.02891508983016861,
      0.06363606791772038, 0.029512883150628392, 0.03364401444065438, -1.2297585323065077,
      0.30019143529481035, 0.0005507879189314305, 0.010940405502457066, 0.03254227217926133,  },
        { 0.12208371585808706, 3.370446316512181E-05, 0.06124291604502224, 3.7278870763639942E-05,
      0.01642736155572521, 0.007640356339217293, 0.009656512687265782, 0.00012608061732525926,
      0.004307059452945267, 0.1747276205554598, 0.06597383773787036, 0.014055946445220851,
      0.16420378239483205, 0.011477232336355487, 0.01817550205414662, 0.5554829116332598,
      -1.5326952365300517, 5.507879189314304E-05, 0.016775288437100834, 0.2902130502543961,  },
        { 0.00010006861955580907, 3.370446316512181E-05, 0.002951465833495047, 3.7278870763639942E-05,
      0.0019748740668084955, 0.0012155112357845695, 0.0024702706874400837, 0.0012608061732525923,
      4.5337467925739656E-05, 0.00017130158877986254, 0.028863554010318277, 0.003212787758907623,
      0.04034072162641202, 0.04426932472594259, 7.734256193253882E-05, 0.0016832815504038176,
      9.096710160448799E-05, -0.1427631412866295, 0.01385784696977895, 0.00010669597435823387,  },
        { 0.0005003430977790454, 0.0006740892633024363, 0.09260224052590711, 3.7278870763639942E-05,
      0.006463224218645984, 0.015107068216179647, 0.0017965604999564246, 0.005673627779636666,
      0.04329728186908136, 0.010278095326791751, 0.027489099057445978, 0.023493510487011995,
      0.04034072162641202, 0.6492834293138247, 0.006960830573928494, 0.025249223256057267,
      0.020922433369032236, 0.01046497045969718, -0.9817009875550362, 0.0010669597435823386,  },
        { 0.1701166532448754, 3.3704463165121 81E-05, 0.008485464271298261, 3.7278870763639942E-05,
      0.031418451062862424, 3.4728892450987694E-05, 0.0031439808749237426, 0.0018912092598788886,
      4.5337467925739656E-05, 1.588822235933225, 0.11682867099414541, 0.005220780108224888,
      0.1596583489721377, 0.04262972010646324, 0.01237480990920621, 0.05134008728731644, 0.2474305163642073,
      5.5078791893143 04E-05, 0.000729360366830471, -2.4402964172417945,  },
                }, new double[20] {
0.054116, 0.018227, 0.039903, 0.02016, 0.009709, 0.018781, 0.024289, 0.068183, 0.024518, 0.092638, 0.148658,
      0.021718, 0.061453, 0.088668, 0.041826, 0.09103, 0.049194, 0.029786, 0.039443, 0.0577,  })
```

mtArt protein sequence evolution matrix.

Citation: Abascal, F., D. Posada, and R. Zardoya. 2007. MtArt: A new Model of amino acid replacement for Arthropoda.Mol.Biol.Evol. 24:1-5.

Definition at line 318 of file RateMatrices.cs.

7.22.3.9 mtmamMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.mtmamMatrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
        { -1.2307900321918033, 0.005840880738946804, 0.0007935979264873374, 0.0020292666969913653, 0, 0, 0,
      0.043098319392711075, 0.002198266256369925, 0.06733182407541004, 0.034893508830240116, 0,
      0.042294801492142646, 0, 0.028180662369565356, 0.24596575734066914, 0.587728704382441,
      0.0014532762028799367, 0, 0.16898080648694874,  },
        { 0.0219667906051695, -0.19984680584026565, 0.0015871958529746748, 0, 0.011993248664039885,
      0.058079464249975794, 0, 0.009945766013702556, 0.06374972143472782, 0, 0.009969573951497179,
      0.010961571359606348, 0, 0, 0.004785395496718645, 0.0021575943626374484, 0, 0.004650483849215797, 0,
      0,  },
        { 0.0013729244128230938, 0.0007301100923683505, -0.9246596320493131, 0.15941795147277635, 0,
      0.0018887630650398634, 0, 0.025969500146890001, 0.1258507431771782, 0.01705739543243721, 0,
      0.0894464222943878, 0.011686721464934153, 0.003636662498128224, 0.017546450154635032,
      0.3207623619121007, 0.09493415195604774, 0.001743931443455924, 0.052615542526110476, 0,  },
        { 0.007551084270527016, 0, 0.3428343042425298, -0.5616838124325348, 0, 0.011568673773369164,
      0.13320937995053203, 0.04365086194902788, 0.0030226161025086646, 0, 0, 0, 0, 0.00303055208177352,
      0.0010634212214930322, 0.011507169934066393, 0, 0, 0, 0.004245748906707255,  },
        { 0, 0.0339501192951283, 0, 0, -0.7492715143033871, 0, 0, 0, 0.08380890102410338,
      0.03680806382789082, 0.0448630827817373, 0, 0, 0.004242772914482928, 0, 0.2495617479450649,
      0.09838630293626766, 0.01889259063743918, 0.17875793294127276, 0,  },
        { 0, 0.044901770680655355, 0.0031743917059493497, 0.009041064377506 99, 0, -0.4518409314300142,
      0.06414652039797149, 0, 0.1511308051254323, 0, 0.03323191317165726, 0.05305400538049473,
      0.012243232010883398, 0, 0.027117241148072322, 0.021575943626374484, 0, 0, 0.018213072412884397,
      0.014010971392133942,  },
        { 0, 0, 0, 0.10498705368982608, 0, 0.06469013497761532, -0.26205992328983235, 0.012155936238969793,
      0.006045232205017292, 0, 0, 0.0471347568463073, 0, 0, 0, 0.01510316053846214, 0.0034521509802199176,
      0, 0, 0.008491497813414 51,  },
        { 0.05354405210010066, 0.003285495415657577, 0.018649551272452432, 0.014576409914756169, 0, 0,
      0.005150450542902 82, -0.17821630235644514, 0, 0, 0, 0, 0, 0, 0.08055018953846474, 0, 0,
      0.00033727911875711846, 0.0021228744533536275,  },
        { 0.005491697651292375, 0.042346385357364326, 0.18173392516560027, 0.0020292666969913653,
      0.019666348615764332, 0.12985246072149062, 0.005150450542902 82, 0, -0.9872507005113695, 0,
      0.043201487123154435, 0, 0, 0, 0.028180662369565356, 0.014383962417582991, 0.0008630377450549794, 0,
      0.5143506561046056, 0,  },
```

```
      { 0.05148466548086601, 0, 0.007539180301629706, 0, 0.0026436730926109432, 0, 0, 0, 0,
    -1.954687244793166, 0.38549019279122415, 0.0013153885631527619, 0.21036098636881476,
    0.034548293373221813, 0.0026585530537325806, 0, 0.3106935882197926, 0, 0.005396465900113895,
    0.9425562572890106,  },
      { 0.014415706334642486, 0.0010951651385525257, 0, 0, 0.0017409554512315963, 0.004721907662599658, 0,
    0, 0.007144365333202255, 0.2082797758066017, -0.8860977168038853, 0.0008769257087685079,
    0.3389149224830904, 0.1491031624232572, 0.022863556262100192, 0.05322066094505706, 0.0293432833318693,
    0.003487862886911848, 0.00843197796892796, 0.04245748906707255,  },
      { 0, 0.00912637615460438, 0.16189397700341684, 0, 0, 0.05713508271745587, 0.050333948487459375, 0,
    0, 0.005386545926032802, 0.006664638263433145 1, -0.4454246921943641, 0.032834122211100548, 0,
    0.009570790993437 29, 0.046747877857144716, 0.04315188725274897, 0, 0.022597700956726936, 0,  },
      { 0.052171127687277566, 0, 0.008332778228117043, 0, 0, 0.005194098428859624, 0, 0, 0,
    0.33935239334006656, 1.0119117560769635, 0.012934654204335492, -2.423750243225364,
    0.006667214579901744, 0, 0.03380231168132003, 0.5963590818329908, 0.0037785181274878356, 0,
    0.35324630903804366,  },
      { 0, 0, 0.0023807937794620123, 0.0009225575895415298, 0.0004513588206896732, 0, 0, 0, 0,
    0.05117218629731163, 0.4087525320113843, 0, 0.006121616005441699, -0.7830440660624641,
    0.009039080382690775, 0.06472783087912345, 0.006904301960439835, 0, 0.2300243589923548,
    0.002547449344024353,  },
      { 0.036382496939811985, 0.0016427477078287886, 0.013094365787041068, 0.0003690230358166119, 0,
    0.012040864539629128, 0, 0, 0.01456351394845075, 0.004488788271694003, 0.0714486133190631,
    0.003946165689458286, 0, 0.010303877078029967, -0.3856082404827892, 0.14527802041758822,
    0.0673169441142884, 0.0020345866840319115, 0.0026982329500569477, 0,  },
      { 0.23477007459274904, 0.0005475825692762629, 0.17697233760667624, 0.002952184286532895,
    0.02237450153990237, 0.0070828614938994875, 0.004916339154589056, 0.06188476630748258,
    0.005495665640924811, 0, 0.12295807873513186, 0.014250042767488254, 0.02615599565961453,
    0.05454993747192336, 0.10740554337079625, -1.413251091457548, 0.5299051754637574,
    0.004941139089791785, 0.036088865707011676, 0,  },
      { 0.4674807625662634, 0, 0.043647885956803564, 0, 0.007350700794088963, 0, 0.0009364455532550581, 0,
    0.0002747832820462406, 0.32319275556196814, 0.05649425239181734, 0.010961571359606348,
    0.38454878725092856, 0.004848883330837632, 0.04147342763822826, 0.4415876462197978,
    -1.8834221509946032, 0, 0, 0.10062424908896195,  },
      { 0.0034323110320577345, 0.002920440369473402, 0.0023807937794620123, 0, 0.004191189049261251, 0, 0,
    0, 0, 0, 0.019939147902994357, 0, 0.007234637097340189, 0, 0.003721974275225613, 0.012226368054945543,
    0, -0.06076876922335976, 0.004721907662599658, 0,  },
      { 0, 0, 0.061900638266012324, 0, 0.034174310709360965, 0.012749150689019077, 0,
    0.0005525425563168088, 0.4190445051205169, 0.014364122469420808, 0.04153989146457157,
    0.014688505621872508, 0, 0.41336730395390814, 0.004253684885972129, 0.076954198934069, 0,
    0.004069173368063823, -1.097658028039104, 0,  },
      { 0.27321195815179566, 0, 0, 0.0018451151790830596, 0, 0.0077911476432894365, 0.004682227766627529,
    0.002762712781584043, 0, 1.993021992632137, 0.16615956585828628, 0, 0.4630167742297721,
    0.003636662498128224, 0, 0, 0.20453994557803012, 0, 0, -3.120668102318381,  },
              }, new double[20] {
0.0692, 0.0184, 0.04, 0.0186, 0.0065, 0.0238, 0.0236, 0.0557, 0.0277, 0.0905, 0.1675, 0.0221, 0.0561,
      0.0611, 0.0536, 0.0725, 0.087, 0.0293, 0.034, 0.0428,  })
```

mtmam protein sequence evolution matrix.

Citation: Yang, Z., R. Nielsen, and M. Hasegawa. 1998. Models of amino acid substitution and applications to Mitochondrial protein evolution, Molecular Biology and Evolution 15:1600-1611.

Definition at line 347 of file RateMatrices.cs.

7.22.3.10 mtREV24Matrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.mtREV24Matrix
[static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
              new double[20, 20] {
    { -1.1143824850663313, 0.0044013246586137745, 0.010503638078279842, 0.003355108141402304,
    0.0035934524334597455, 0.00047468988984186527, 0.0023432691656867484, 0.06755346799489362,
    0.0038894590552937675, 0.08485576457755911, 0.04299930898143552, 0.0019215446740798703,
    0.07656518073307889, 0.003883163168333759, 0.029308253211259617, 0.27907688186384716,
    0.41314929369792014, 0.0005506402722165637, 0.002137003916711252, 0.08382104055241751,  },
    { 0.016678703969483774, -0.23671605576792737, 0.005160228874078854, 0.00036076431627981757,
    0.0061957523769296755, 0.055211143092429148, 0.00045570229424819064, 0.012888380150131723,
    0.046234195662315763, 0.0016709084122433656, 0.026313009973714274, 0.03250076757355188,
    0.0010253301620584288, 0.0028590322228391417, 0.012757265805821715, 0.004345960827251165,
    0.0017876321577876388, 0.006361344197449249, 0.0006265906545912622, 0.0032830552128599905,  },
    { 0.019391331836824323, 0.0025139576566025183, -1.3045719361998, 0.15083366187703234,
    0.0035340912135510996, 0.043361672253133754, 0.015122120869657061, 0.0298285133305263,
    0.13882570655416523, 0.023832430511471162, 0.025603673376220052, 0.13990959845842313,
```

0.0352983399474957, 0.00926594664971321, 0.03956155483184391, 0.3557284061895204, 0.20494171362790403,
0.003095177951196263, 0.06310757245399154, 0.0008164666105280082, },
{ 0.012714094009524519, 0.00036076431627981757, 0.30960593753706644, -0.6463999134005746,
0.00011392557356204766, 0.013810977426557007, 0.13996056516238506, 0.03177044468619096,
0.03189636242539112, 0.0038167065837558982, 0.003208903655331009, 0.0005309531336273326,
0.0010253301620584288, 0.003035816731287617, 0.007247465303391948, 0.04966195634054229,
0.024072873432251527, 0.005755639898011029, 0.006994730412568774, 0.0008164666105280082, },
{ 0.04312142920151694, 0.019619882526943973, 0.022971592888082145, 0.00036076431627981757,
-0.7729845169449252, 0.018797719637737863, 0.00045570229424819064, 0.01718637231483045,
0.03959133537651188, 0.055166360368434905, 0.043320199346968624, 0.00043671469865451596,
0.0033350212639584684, 0.04315980413155889, 0.01686937940312973, 0.19934576940230717,
0.15467315357550065, 0.009737638498145547, 0.08401921108958729, 0.0008164666105280082, },
{ 0.0013671068827445718, 0.04196068750246152, 0.06764420871488866, 0.010496342844183323,
0.0045114527130570865, -0.8026638277479705, 0.07520526914971719, 0.0037775321760047382,
0.1629655362448266, 0.007334408504268246, 0.0670491974298111, 0.10701348915766819,
0.025563099882477774, 0.01164948950500128, 0.0740881989205272, 0.03893376496068883,
0.08158650035518297, 0.0005506402722165637, 0.01280223642696463, 0.00816466610528008, },
{ 0.007029807497060245, 0.00036076431627981757, 0.024573446413192725, 0.11080211408688817,
0.00011392557356204766, 0.07833882203095541, -0.404691111933864726, 0.015826460735146874,
0.013744620776692794, 0.0029108983392239682, 0.003208903655331009, 0.0721406712208981,
0.0010253301620584288, 0.0016276366812325177, 0.006923676831116297, 0.0393654829236608,
0.012736879124236928, 0.0005506402722165637, 0.004326773362230189, 0.009084265340295838, },
{ 0.0868544588505775, 0.004372843265223263, 0.020773428926616534, 0.010779258018529076,
0.0018413970337318336, 0.001686398292859258, 0.006782768884915174, -0.25690566609739524,
0.0005316526766228891, 0.005258964371165961, 0.0040702409522882805, 0.005224486894956394,
0.0010253301620584288, 0.0011582433312141511, 0.0010253301620584288, 0.09061040512843364,
0.00959992846273458, 0.003164732511897303, 0.0010586084217041185, 0.0010871897498083477, },
{ 0.010001466142183972, 0.03137320419942855, 0.19336437698615874, 0.021643960217229686,
0.008483857580681117, 0.14550494307573805, 0.011781103522879538, 0.0010633053532457782,
-0.8695592256534729, 0.010781756386370347, 0.019405422631449103, 0.029344929251169504,
0.006459580020968103, 0.02935842043751238, 0.032902030525300127, 0.055734789019681326,
0.03848565770467811, 0.0020518595406806687, 0.22100182171988864, 0.0008164666105280082, },
{ 0.06942744374527565, 0.00036076431627981757, 0.010562099885765628, 0.0008240616487654779,
0.003761342752393289, 0.002083638779621661, 0.0007938813652429005, 0.0033466136907419754,
0.003430558850208747, -1.8824460168416577, 0.5557990020699378, 0.004498161396141515,
0.2795265880752763, 0.0516149804494222, 0.011132926970139678, 0.034321578056271625,
0.31664294033350954, 0.0005506402722165637, 0.0082479117217513, 0.5255208824626959, },
{ 0.018319232228777262, 0.002958267393494504, 0.005908540009896934, 0.00036076431627981757,
0.0015379952430876432, 0.009918520329853712, 0.00045570229424819064, 0.001348718895432803,
0.003215099607577366, 0.2894101312553522, -1.0052316921878106, 0.0034201656399890515,
0.2900766957954038, 0.13171055481164712, 0.021639862893969998, 0.05296459875727785,
0.10863303112709499, 0.009401458121423855, 0.014559988105370642, 0.0393923653616329, },
{ 0.006015270284076116, 0.026848460169455895, 0.23723801477732614, 0.00043861345821388347,
0.00011392557356204766, 0.11631900995398717, 0.07527722214354586, 0.012720489831198178,
0.03572426169707592, 0.017210356646106664, 0.02513078231122391, -0.8512394937511274,
0.049307587845936134, 0.0039258352910627025, 0.027036337431119627, 0.07611907217134119,
0.11716725580345616, 0.006955456070103963, 0.016875075681807836, 0.0008164666105280082, },
{ 0.10208690764410518, 0.00036076431627981757, 0.025493245517635783, 0.00036076431627981757,
0.00037055791821760767, 0.011834768464110083, 0.00045570229424819064, 0.0010633053532457782,
0.003349411862724201, 0.45552481019674657, 0.9078326220263564, 0.02100138000845428,
-2.3151812152638804, 0.05536403123203642, 0.010166958027989895, 0.07998294793994032,
0.4539296522974506, 0.0062917896367482105, 0.013178190819719387, 0.166533405391591773, },
{ 0.004583405706885749, 0.0008905182333433393, 0.0059241298252264785, 0.0009455822605649956,
0.004245226635891039, 0.004774380944672655, 0.0006403816450750889, 0.0010633053532457782,
0.013475996266399124, 0.07446095540244513, 0.3649030125109568, 0.0014802329785974123,
0.0490107817463929, -0.7696317073019369, 0.009341297423806, 0.046258579732244659, 0.029091994490919027,
0.0022721156495672944, 0.15354109313926306, 0.002728717356238343, },
{ 0.03907767094834616, 0.004488667598344678, 0.0285722340452206, 0.0025500340882305002,
0.0018743754892366368, 0.03430009209283667, 0.003077189702739098, 0.0010633053532457782,
0.017060454557563029, 0.018142547655042435, 0.0677247560940913, 0.011515477053995396,
0.010166958027989895, 0.010552206349114187, -0.49271381499180406, 0.12224813651489619,
0.11019720926515916, 0.00122010291896407, 0.00534580763732861, 0.0035365895813923723, },
{ 0.27907688186384716, 0.0011468507738579465, 0.19268622001932356, 0.013105238478754215,
0.016612147450192266, 0.013518668389128807, 0.013121827641220269, 0.07047475954433728,
0.021674640174320515, 0.04194859540210976, 0.1243196831941661, 0.024315814721400066,
0.05998721095495524, 0.039191296717767256, 0.09168610238617214, -1.5495382021878925,
0.5132652889194018, 0.011180895632692119, 0.02140961331371828, 0.0008164666105280082, },
{ 0.34589243193314245, 0.00039494198834843185, 0.09293868408707276, 0.005318425525788258,
0.010791150249453535, 0.023717005917204353, 0.0035544778951358868, 0.006251116208292285,
0.012530214136406828, 0.3240067296435911, 0.21347653791254714, 0.03133542887766851,
0.2850255956286318, 0.020635019348210007, 0.0691935965153325, 0.42971047444441504, -1.978334373936851,
0.0028952085891807743, 0.012772555817010306, 0.08789477921968357, },
{ 0.0013671068827445718, 0.0041677772328115766, 0.004162480692988078, 0.0037709364849037768,
0.002014683827202527, 0.00047468988984186527, 0.00045570229424819064, 0.0061112076091809995,
0.0019811057634158183, 0.0016709084122433656, 0.05478780767312522, 0.005516396193530729 4,
0.011715746220151837, 0.00477927774564155, 0.0022719157801399923, 0.027759465019097672,
0.0085857909886605053, -0.15255743575279707, 0.008656844570010858, 0.002307592472913337, },
{ 0.0046625540000972765, 0.00036076431627981757, 0.07458167653653545, 0.004027269025418384,
0.015276220198106781, 0.009698663959821689, 0.0031467442634401373, 0.00179642641258892,
0.1875166972168752, 0.02199443125800346, 0.07456478756992845, 0.011761416384290308,
0.021564312250449906, 0.28381838428772865, 0.008747685224719543, 0.046711883593567716,
0.03328605455342079, 0.00760753007667621, -0.8119399677384762, 0.0008164666105280082, },
{ 0.14035150976218747, 0.00145065230335674, 0.0007405162281533098, 0.00036076431627981757,
0.00011392557356204766, 0.004746898898418653, 0.0050702876317930265, 0.0014158750230062204,
0.0005316526766228891, 1.0754845966678428, 0.1548211568864177, 0.004671188359356716,
0.2091349742126966, 0.0038709711332683473, 0.004441298544074142, 0.0013671068827445718,
0.17578955843936714, 0.0015562832956857616, 0.0006265906545912622, -1.7823113338287229, },

```
                }, new double[20] {
0.072, 0.019, 0.039, 0.019, 0.006, 0.025, 0.024, 0.056, 0.028, 0.088, 0.169, 0.023, 0.054, 0.061, 0.054,
      0.072, 0.086, 0.029, 0.033, 0.043,  })
```

mtREV24 protein sequence evolution matrix.

Citation: Adachi, J. and Hasegawa, M. (1996) MOLPHY version 2.3: programs for molecular phylogenetics based on maximum likelihood.Computer Science Monographs of Institute of Statistical Mathematics 28:1-150.

Definition at line 376 of file RateMatrices.cs.

7.22.3.11 MtZoaMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.MtZoaMatrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
  { -1.3402360111467704, 0.0006819733423697274, 0.0005075154567039491, 0.003273270856518464,
      0.026678231787002286, 0.0013354946587779033, 0.004197731337389084, 0.20453023598839784,
      0.0006058416032545011, 0.027745616471483454, 0.02401653557325466, 3.7871853912569246E-05,
      0.06792019148365738, 0.021271659203919963, 0.027199769417465006, 0.5102073279990578,
      0.20943420686797376, 0.0008526255743195561, 0.0007716262528185448, 0.20896828541849408,  },
  { 0.002232938338281448, -0.17538894562565346, 0.010030893732501583, 0.0006505879963266512,
      0.0059817980263700546, 0.0422601734489994, 0.0015710807344614117, 0.005090278946133648,
      0.016252359530783793, 0.0001671422679005027 3, 0.006312038195534879, 0.05534971449321995,
      0.0018908185174937584, 0.0001605408241805 2806, 0.0033009428904690542, 0.009335472895625497,
      0.0017725190742594974, 0.004648184582580804, 0.005070686804236152, 0.0033107743262948738,  },
  { 0.0011503015682055944, 0.006943728576855407, -1.0116367447720385, 0.12544149804173244,
      0.009261507255230885, 0.03481433336512808, 0.04271375746816963, 0.07020343046542657,
      0.060294409993458836, 0.02030778554991108, 0.012162219937737938, 0.07828112203728063,
      0.03667192756297106, 0.010354883159644057, 0.009946841243280085, 0.33243777209676556,
      0.10225219409634474, 0.0017602592502081156, 0.052029655904336156, 0.004609117199351687,  },
  { 0.010894032498888278, 0.0006613074835100387, 0.1841982569331392, -0.503506159255173,
      0.0009300667962441165, 0.00353082834444021, 0.21690733390157868, 0.018381562861038175,
      0.014645562235195766, 0.0012535670092537704, 0.0007697607555530341, 3.7871853912569246E-05,
      9.951676407861888E-05, 0.0016054082418052802, 8.802514374584145E-05, 0.04066468701992801,
      0.0012739980846240137, 5.5008101569003614E-05, 0.005107430911513226, 0.0024019343151555105,  },
  { 0.18438657490354382, 0.012626839763269802, 0.02824174247305505, 0.0019314331140947456,
      -1.5337206152318523, 0.008982573663834939, 0.0008346366401826249, 0.05825541460575176,
      0.009877852226975563, 0.04078271336772266, 0.09190943421303227, 3.7871853912569246E-05,
      0.1317601956400914, 0.13469375148746301, 0.003565018321706579, 0.5192263441863569,
      0.11022852993051248, 0.009928962333205152, 0.028182730281515425, 0.15826799622562557,  },
  { 0.004939530263471082, 0.04773813396588092, 0.05681187730044796, 0.0039238588528451145,
      0.004806976810061697, -0.6173871314482275, 0.08577118884700269, 0.00629215036397076,
      0.11110608184902114, 0.0001671422679005027 3, 0.035408994755439564, 0.06324599603399064,
      0.04174728253098062, 0.007625689148575082, 0.02297456251766462, 0.07626606670663541,
      0.02182414110182006, 0.00167774709785461, 0.019217168105909472, 0.00584254292875566,  },
  { 0.011570680480185686, 0.0013226149670200775, 0.05194569968616892, 0.17964361048569658,
      0.000332866011287368, 0.06392079914753415, -0.4557107027529902, 0.03047097535810559,
      0.00392479995151829, 0.0061006927783683 49, 0.0015395215111060681, 0.0309034327926565,
      9.951676407861888E-05, 0.0001605408241805 2806, 0.009066589805821671, 0.030379843999323648,
      0.019109971269360204, 0.0009626417774575633, 0.003674410727707356, 0.01058149441541303,  },
  { 0.1957542609893403, 0.0014879418378975873, 0.0296448734415895, 0.005286027470154041,
      0.008067105685317 39, 0.0016282058168666211, 0.010580246821138568, -0.5223731374718185,
      0.0019492295061231777, 0.002841418554308546, 0.005388325288871239, 0.001912528622584747,
      0.025973875424519528, 0.004655683901235313, 0.0005721634343479695, 0.2034025492767216,
      0.002880343495671683, 0.003382998246493722, 0.001579996612914163, 0.015385363045723237,  },
  { 0.0015562903569840394, 0.012750834916427934, 0.06833546355266704, 0.011303966436175565,
      0.0036713163009636177, 0.07716597905103008, 0.0036576723349179 74, 0.00523167558352625,
      -0.3489878469157433, 0.0013371381432040219, 0.010160841973300049, 0.004525686542552024,
      0.00353284512479097, 0.010515423983824585, 0.006865961212175634, 0.018275374905843134,
      0.010745896887698203, 0.0012376822285302581 2, 0.09792304589340105, 0.000194751403958522,  },
  { 0.022464712979073962, 4.133171771937742E-05, 0.007254485645827038, 0.0003049631232781 1775,
      0.004777606279653988, 3.6588894761038444E-05, 0.0017920139627450476, 0.0024037428356742225,
      0.0004214550283509574, -2.2398689533654137, 0.6546045465223002, 0.0015906178643279084,
      0.22376344403077453, 0.0724841821175084, 0.0011443268686959 39, 0.0056962207498731845,
      0.12313468443996445, 0.0026678929260966 75, 0.004813478053296637, 1.1104726593254923,  },
  { 0.010555708508239572, 0.0008473002132472 37, 0.0023584541811536 46, 0.0001016543744260 3925,
      0.0058447355511340796, 0.004207722897519421, 0.00024548136475959555, 0.0024744411543705233,
      0.001738501991947699, 0.3553444615564687 6, -1.025767491650488, 0.0012687071060710699,
      0.3166125849161259, 0.1879933051153983, 0.0050174331935129 63, 0.012025354916398943,
      0.027695610535304647, 0.0074811018133844 91, 0.0020944141147931 93, 0.08186051814623208,  },
```

```
  { 0.0001353295962594817, 0.060406305446870104, 0.1234158175302427, 4.06617497704157E-05,
1.958035360513393E-05, 0.061103454250934205, 0.040062558728766, 0.007140530188326375,
0.006295484485992425, 0.007019975251821115, 0.010314794124410658, -0.5630458140068918,
0.04129945709262683, 0.013084077170713035, 0.010695054965119737, 0.11463644259119783,
0.04182037190831001, 0.001815267351777119, 0.0165348482746831, 0.007205802945465314,  },
  { 0.09236244944709626, 0.0007853026366681171, 0.022002287740635916, 4.06617497704157E-05,
0.025924388173204426, 0.015349041352255626, 4.9096272951919116E-05, 0.03690452235946895,
0.0018702066883073732, 0.3758193893742804, 0.9795975375167911, 0.01571681937371624,
-2.1963706830761636, 0.1730630084666092, 0.0023766788811377193, 0.07539581075873811,
0.16899861548642894, 0.013394472732052382, 0.015212060412708454, 0.1815083336533425,  },
  { 0.017931171504381324, 4.133171771937742E-05, 0.003851146700871144, 0.000406617497704157,
0.01642791667471187, 0.001737972501149326, 4.9096272951919116E-05, 0.004100502484385439,
0.003450663044623463, 0.07546473395707698, 0.3605559379010412, 0.003086556093874394,
0.10727907167675113, -0.9068412374055845, 0.004621320046656676, 0.025474765020266188,
0.010690505666627593, 0.01600735755658005, 0.2169739534711194, 0.03869061761709304,  },
  { 0.04181684524417985, 0.0015499394144766535, 0.006746970189123089, 4.06617497704157E-05,
0.0007930043210081414, 0.009549701532631034, 0.005056916114047669, 0.0009190781430519088,
0.004109186526421834, 0.0021728494827065355, 0.017550545226609177, 0.0046014302503771635,
0.0026869526301227095, 0.00842839326947772, -0.21411505189688756, 0.0630539991340129,
0.03151760478917668, 0.0003575526601985235, 0.0015432525056370897, 0.011620168713858478,  },
  { 0.43637028313869874, 0.0024385713454432682, 0.1254458793570585, 0.010450069690996835,
0.0642529303552646, 0.01763584727482053, 0.00942648440676847, 0.18176537736818904,
0.0060847569718169466, 0.006017121644418098, 0.023400726968812235, 0.02743815815965642,
0.04741973808346189, 0.025847072693065014, 0.03507801978271782, -1.4340226464349088,
0.36907170599346967, 0.0028329172308036864, 0.010912999861290847, 0.032133986108156126,  },
  { 0.25584060172855017, 0.0006613074835100387, 0.055110207827970006, 0.0004676101223597805,
0.0194824518371136, 0.007208012267924573, 0.008469107084206047, 0.0036763125722207635,
0.005110142218755357, 0.18577863077140877, 0.0769760755553034, 0.014296624851994889,
0.1518128236019331, 0.015492189533420957, 0.025043153395691894, 0.5271377618945142,
-1.6215405071253182, 0.000990145828242065, 0.010655791110351334, 0.25733155743986036,  },
  { 0.00209760876420219664, 0.003492530147287392, 0.0019106464252383968, 4.06617497704157E-05,
0.0035342538257276427, 0.0011159612902116726, 0.0008591847766585845, 0.008695893199644983,
0.0011853422672370676, 0.008106399993174382, 0.041874985102085045, 0.0012497711791147849,
0.024232332053143697, 0.04671737983653366, 0.0005721634343479695, 0.008148760239401916,
0.0019940839585419344, -0.19404340450768542, 0.0293217976071047, 0.00889364868043917,  },
  { 0.001420960760724558, 0.0028518885226370425, 0.04227305215839953, 0.002825991609043891,
0.0075090656075709195, 0.009567995980011552, 0.0024548136475959557, 0.003040027703940929,
0.07019860315970633, 0.010947818547482929, 0.008775272613304588, 0.00852116713032808,
0.020599970164274107, 0.473996783393009, 0.0018485280186626707, 0.023496910593226887,
0.016063454110476695, 0.02194823252603244, -0.7384676106562712, 0.010127074409843144,  },
  { 0.2178129851796358, 0.001053958801844124, 0.0021196233779988464, 0.0007522423707526905,
0.023868451044664802, 0.001646500264246 73, 0.004001346245581408, 0.01675550153102326,
7.90228178158045E-05, 1.4295678173529995, 0.19413366255047518, 0.002101887892147593,
0.13912443618190 92, 0.04784116560579735, 0.007878250365252 81, 0.03916151765537814, 0.2195708003238952,
0.0037680549574767475, 0.0057320807352234755, -2.3569693052541187,  },
        }, new double[20] {
0.06888, 0.021037, 0.03039, 0.020696, 0.009966, 0.018623, 0.024989, 0.071968, 0.026814, 0.085072, 0.156717,
0.019276, 0.050652, 0.081712, 0.044803, 0.080535, 0.056386, 0.027998, 0.037404, 0.066083,  })
```

MTZoa protein sequence evolution matrix.

Citation: Rota-Stabelli, O., Z. Yang, and M. Telford. 2009. MtZoa: a general mitochondrial amino acid substitutions model for animal evolutionary studies. Mol. Phyl. Evol.

Definition at line 405 of file RateMatrices.cs.

7.22.3.12 WAGMatrix

```
readonly ImmutableRateMatrix PhyloTree.SequenceSimulation.RateMatrix.Protein.WAGMatrix  [static]
```

Initial value:

```
= new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
                new double[20, 20] {
  { -1.1354111867880292, 0.025870671727558636, 0.021258361408223507, 0.04496676768561226,
      0.021151940281163652, 0.035559526431702 15, 0.09802540056627916, 0.12580784375147247,
      0.008259863595157933, 0.009994883181024 09, 0.03659089059741105, 0.05996215721773844,
      0.018587347341283 84, 0.008629023237601 8, 0.07022155771088084, 0.24995302845876177, 0.1380427166515966,
      0.0017360267088627171, 0.009057878082649915, 0.15169890215304813,  },
  { 0.05096702363658184, -0.9902438842087685, 0.026491061820913632, 0.008963197122538123,
      0.010878120120976895, 0.11892105714896123, 0.02719685430488389, 0.051919534535373714,
      0.05569441459136586, 0.009666295612820768, 0.0457641082002542, 0.3540715876461629,
      0.014211780897022 65, 0.004210550446840853, 0.03316866012819069, 0.09077693891014615,
      0.036081427491719 73, 0.017860360875955678, 0.01435553367191173, 0.019045377046147838,  },
```

```
  { 0.04711167749403789, 0.029800021754982527, -1.4761789236926435, 0.3303709452632036,
0.005462960616159395, 0.06047481490937984, 0.058659672973167695, 0.09995219705581013,
0.10310144608645852, 0.028652463727302693, 0.012094861109775404, 0.19928676177652271,
0.004123580382381817, 0.003942083838382985, 0.009522708073960827, 0.29469970668349693,
0.13211714496925678, 0.0011035622852153429, 0.040861759186482265, 0.014840555506666011,  },
  { 0.06828591157509498, 0.006909089542699483, 0.22638231903829553, -1.0058412779653094,
0.0006239249840549781, 0.024163560003791055, 0.3823637787284317, 0.07686575796612918,
0.024253540927980724, 0.002038783500194207, 0.007798347636671813, 0.03174914727118214,
0.0021583886621177524, 0.001915673166467582, 0.020696407441166527, 0.0794738496853742,
0.024396434423635458, 0.001991275559995653, 0.012255179048239709, 0.011519908803786914,  },
  { 0.09490196539650382, 0.024774065297941558, 0.011059978491039946, 0.0018433929866641786,
-0.49532302419935553, 0.0038713652226125303, 0.001322322616098299, 0.027233323927550304,
0.006488243590595672, 0.00879550753874639, 0.03533770673386853, 0.004898382207458443,
0.008123175217929567, 0.0163164927695339, 0.005340460394086955, 0.10438172655082653,
0.03338521102573777, 0.01100344437188255, 0.020462221992322477, 0.07578403786803438,  },
  { 0.08395752449304077, 0.14237591176657988, 0.06436282383022025, 0.03753019341518513,
0.0020351596038226377, -1.4023387548390867, 0.3387225496005603, 0.029309341610100084,
0.11190508042997918, 0.005889192889713301, 0.07995520871204236, 0.25770233446880264,
0.032145957399464874, 0.004096168561193013, 0.045561736306503266, 0.07629344230591827,
0.05583430930572718, 0.003310485836050631, 0.008567800353917012, 0.0227835339502657,  },
  { 0.14626068695265626, 0.02059804917927061, 0.039493883292954945, 0.37568710385386667,
0.000439745510285292, 0.21427611742531016, -1.257111487537639, 0.050414514957828435,
0.014854927673510668, 0.006585968101205492, 0.014185493274263147, 0.17099633898863866,
0.006555506956466206, 0.0033260155085525515, 0.0333085614068389, 0.052273098569976494,
0.053545886713018596, 0.0024023683050871133, 0.007386082793355458, 0.04452137716455361,  },
  { 0.13090971375655758, 0.027422899836273246, 0.04693077400841045, 0.05266930984980068,
0.006315966402268248, 0.012930368227352645, 0.03515853449997556, -0.5057491780818103,
0.006499657929126434, 0.0015741857002120755, 0.005637276754876061, 0.02471610789994531,
0.0036217925677535395, 0.0020468790524988626, 0.011889656120148239, 0.09949951573563934,
0.014697305104204883, 0.005171006588994236, 0.0038981967760187003, 0.01416003127175428,  },
  { 0.02928762028770396, 0.10024005265424024, 0.16495944410047816, 0.05663004701308376,
0.005127590819911471, 0.16822931995187806, 0.03530146821100753, 0.0221481551118462,
-1.0085628865968066, 0.007144039655446344, 0.045928802381322936, 0.05891458191114661,
0.008407322631387035, 0.027850238704916878, 0.03398429532181699, 0.054885496610970495,
0.03080301544484597, 0.004029123217092933, 0.1457417794689575, 0.008950493098753481,  },
  { 0.017864807096055722, 0.008769990316654042, 0.02310914075067111, 0.002399674176679085,
0.0035039388531466916, 0.004462899050317317, 0.007889532113049963, 0.0027040356760804616,
0.0036012498665890782, -1.2533192612174784, 0.2915914613866592, 0.02142603465447157,
0.0885677023866053, 0.04343207525887664, 0.004877936808716465, 0.023687324161655532,
0.09489765628029291, 0.0032605531823541914, 0.01580928670109047, 0.5914639624975127,  },
  { 0.03676869017832784, 0.02334256708442944, 0.005484124208196995, 0.005160221331516516,
0.007914410027679094, 0.034063762496917525, 0.009553458866952599, 0.005443903037288344,
0.013016046464044532, 0.16393035260316008, -0.7378782205897432, 0.01704088031890741,
0.10097790671870785, 0.08670960256101458, 0.020299060473896631, 0.02556331218433812,
0.021256694937168644, 0.01020917144994604, 0.014998372673478071, 0.13614568297377316,  },
  { 0.08374194741204094, 0.25100092299321725, 0.12558722824289453, 0.029198358192822254,
0.0015247318815057268, 0.15258956530702983, 0.16005293362321688, 0.03317276984592055,
0.023204776910099468, 0.016741216077158262, 0.023683869237943283, -1.1429624377918817,
0.019435691401668617, 0.0036417565742282135, 0.027184390256132007, 0.07171525737685298,
0.09026523242143569, 0.0021100152263456985, 0.005014181838146751, 0.023097592973222252,  },
  { 0.08256204867767036, 0.032042764827633095, 0.008264921421089172, 0.006313253911990309,
0.008041996363208195, 0.06053828126173739, 0.0195155297899663, 0.015460461909997288,
0.010531968466120388, 0.22009887163670735, 0.44635893288175915, 0.0618154782505777,
-1.3437091202174194, 0.04880886836387656, 0.00836327500681068, 0.03662436714269495,
0.09866971706786477, 0.007913512325499689, 0.016120340258359945, 0.15566453065385616,  },
  { 0.019450356660083032, 0.004817516808913584 5, 0.004009525732323614, 0.0028434651252855015,
0.008197241851056197, 0.0039145732720045874, 0.005024596801342159, 0.004433982329856833,
0.017704499045621883, 0.05477165998810143, 0.1945037358856186, 0.00587774900122482,
0.02476860933339688, -0.7253348974230803, 0.007880747393608459, 0.04048223317961673,
0.011187518384505948, 0.023472336939219917, 0.2428482827643911, 0.049146266926909016,  },
  { 0.1329268794994748, 0.031870487863733026, 0.008133997587276306, 0.02579870314996337,
0.0022531808639589773, 0.036566491501643955, 0.04225802963224779, 0.021629550301080065,
0.018143012914245478, 0.005166041753536194, 0.03823957958254853, 0.0368464913749617,
0.0035641476096533384, 0.006618260033879795, -0.6154883210172385, 0.11962899550311683,
0.051763919901001605, 0.0021391707401819723, 0.008128931514919656, 0.023812449689815537,  },
  { 0.3114723827103921, 0.057418931782417866, 0.16570746116372745, 0.06521476774596385,
0.028990828260031572, 0.04030779379636031, 0.043656649619226245, 0.11915655944900953,
0.01928890305017274, 0.016514161860740863, 0.03170100909405498, 0.06398923174787881,
0.010274678105550326, 0.022379978931120008, 0.07875084955254238, -1.4149959448154485,
0.2849233535059582, 0.008036824804019845, 0.029611342953450494, 0.01760023668286712,  },
  { 0.19599772922396227, 0.026003971790559993, 0.08464434333444128, 0.022809956637548038,
0.010564931193714416, 0.033610839966296824, 0.05095357658458772, 0.020054465792765177,
0.012334443668902279, 0.07538282700619177, 0.030035032277551553, 0.09176820556664868,
0.031539759608400324, 0.007047017879908512, 0.03882597299941695, 0.3246418073071975,
-1.1738518192966378, 0.0017012088873392932, 0.01095471405490418, 0.10498101551633642,  },
  { 0.010453871369374774, 0.054592051136009775, 0.0029986019360413057, 0.0078961005879026,
0.014768092593680887, 0.008451876826270943, 0.009695525562406397, 0.029924829611411037,
0.006842583227583025, 0.010984781663710872, 0.06117952033090722, 0.009097886852328098,
0.010728203089867357, 0.06270629616599628, 0.006804932920430533 5, 0.03883685991445591,
0.007215075002646070 5, -0.47432220705100175, 0.09351510834667694, 0.027630009910597178,  },
  { 0.0222447272158118, 0.017895275488070673, 0.04528130048432225, 0.019816295782384889,
0.01120026789504408, 0.008920940182306032, 0.012156982788869633, 0.009200262468255 94,
0.1009423640150575, 0.021721623431716408, 0.036655536052068394, 0.008817285142275928,
0.008912750961232759, 0.2645877417027 97, 0.010546096178238477, 0.05835762053579319,
0.018948032335748287, 0.038138327649229746, -0.7381466702685061, 0.023805770034191447,  },
  { 0.18536210040995543, 0.011812627574535127, 0.008182572831350187, 0.0092693248904432,
0.020639123533032152, 0.01180321364483485, 0.03645992488282914, 0.01662794434957646,
```

```
      0.003084425296401694, 0.40433951340286917, 0.1655530552458264, 0.020208748575353482,
      0.042821820281977445, 0.026641772069187292, 0.015370932983147012, 0.017258214807348946,
      0.090346582938766, 0.005606589965708167, 0.0118420087189333, -1.1032304964020754,  },
               }, new double[20] {
0.0866279, 0.043972, 0.0390894, 0.0570451, 0.0193078, 0.0367281, 0.0580589, 0.0832518, 0.0244313, 0.048466,
      0.086209, 0.0620286, 0.0195027, 0.0384319, 0.0457631, 0.0695179, 0.0610127, 0.0143859, 0.0352742,
      0.0708956,  })
```

WAG protein sequence evolution matrix.

Citation: Whelan, S. and N. Goldman. 2001. A general empirical model of protein evolution derived from multiple protein families using a maximum likelihood approach.Molecular Biology and Evolution 18, 691-699.

Definition at line 434 of file RateMatrices.cs.

The documentation for this class was generated from the following file:

- SequenceSimulation/RateMatrices.cs

7.23 PhyloTree.TreeBuilding.RandomTree Class Reference

Contains methods to generate random trees.

Static Public Member Functions

- static TreeNode UnlabelledTopology (int leafCount, TreeNode.NullHypothesis model=TreeNode.Null↩ Hypothesis.PDA, bool rooted=false)

  Samples a random unlabelled topology according to the specified model.
- static TreeNode UnlabelledTree (int leafCount, IContinuousDistribution branchLengthDistribution, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.PDA, bool rooted=false)

  Samples a random unlabelled tree according to the specified model, using branch lengths drawn from the supplied distribution.
- static TreeNode UnlabelledTree (int leafCount, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.↩ PDA, bool rooted=false)

  Samples a random unlabelled tree according to the specified model, using branch lengths drawn from a Uniform(0, 1) distribution.
- static TreeNode ResolvePolytomies (TreeNode tree, bool rooted=false)

  Randomly resolve all the polytomies in a tree.
- static TreeNode LabelledTopology (IReadOnlyList< string > leafNames, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.PDA, bool rooted=false, TreeNode constraint=null)

  Samples a random labelled topology according to the specified model.
- static TreeNode LabelledTopology (int leafCount, TreeNode.NullHypothesis model=TreeNode.Null↩ Hypothesis.PDA, bool rooted=false, TreeNode constraint=null)

  Samples a random labelled topology according to the specified model.
- static TreeNode LabelledTree (IReadOnlyList< string > leafNames, IContinuousDistribution branchLength↩ Distribution, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.PDA, bool rooted=false, TreeNode constraint=null)

  Samples a random labelled tree according to the specified model, using branch lengths drawn from the supplied distribution.
- static TreeNode LabelledTree (int leafCount, IContinuousDistribution branchLengthDistribution, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.PDA, bool rooted=false, TreeNode constraint=null)

  Samples a random labelled tree according to the specified model, using branch lengths drawn from the supplied distribution.

- static TreeNode LabelledTree (IReadOnlyList< string > leafNames, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.PDA, bool rooted=false, TreeNode constraint=null)

  Samples a random labelled tree according to the specified model, using branch lengths drawn from a Uniform(0, 1) distribution.
- static TreeNode LabelledTree (int leafCount, TreeNode.NullHypothesis model=TreeNode.NullHypothesis.PDA, bool rooted=false, TreeNode constraint=null)

  Samples a random labelled tree according to the specified model, using branch lengths drawn from a Uniform(0, 1) distribution.

Static Public Attributes

- static Random RandomNumberGenerator = new ThreadSafeRandom()

  Random number generator used for sampling.

7.23.1 Detailed Description

Contains methods to generate random trees.

Definition at line 12 of file RandomTree.cs.

7.23.2 Member Function Documentation

7.23.2.1 LabelledTopology() [1/2]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.LabelledTopology (
            int leafCount,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false,
            TreeNode constraint = null )  [static]
```

Samples a random labelled topology according to the specified model.

Parameters

leafCount	The number of terminal nodes in the topology. Their names will be in the form `t1, t2, ..., tN`, where `N` is *leafCount* .
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.
constraint	A tree to constrain the sampling. The topology produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the topology, using a constraint with TreeNode.NullHypothesis.YHK will bias the sampled topology distribution.

Returns

A TreeNode object containing a random labelled topology (branch lengths will all be set to double.NaN).

Exceptions

ArgumentException	Thrown if *model* is neither TreeNode.NullHypothesis.PDA nor TreeNode.NullHypothesis.YHK.

Definition at line 535 of file RandomTree.cs.

7.23.2.2 LabelledTopology() [2/2]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.LabelledTopology (
            IReadOnlyList< string > leafNames,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false,
            TreeNode constraint = null )  [static]
```

Samples a random labelled topology according to the specified model.

Parameters

leafNames	The labels for the terminal nodes of the topology.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.
constraint	A tree to constrain the sampling. The topology produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the topology, using a constraint with TreeNode.NullHypothesis.YHK will bias the sampled topology distribution.

Returns

A TreeNode object containing a random labelled topology (branch lengths will all be set to double.NaN).

Exceptions

ArgumentException	Thrown if *model* is neither TreeNode.NullHypothesis.PDA nor TreeNode.NullHypothesis.YHK.

Definition at line 292 of file RandomTree.cs.

7.23.2.3 LabelledTree() [1/4]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.LabelledTree (
            int leafCount,
            IContinuousDistribution branchLengthDistribution,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
```

```
            bool rooted = false,
            TreeNode constraint = null )  [static]
```

Samples a random labelled tree according to the specified model, using branch lengths drawn from the supplied distribution.

Parameters

leafCount	The number of terminal nodes in the topology. Their names will be in the form `t1, t2, ..., tN`, where `N` is *leafCount* .
branchLengthDistribution	The continuous univariate distribution from which the branch lengths will be drawn.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.
constraint	A tree to constrain the sampling. The topology produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the topology, using a constraint with TreeNode.NullHypothesis.YHK will bias the sampled topology distribution.

Returns

A TreeNode object containing a random unlabelled tree, with branch lengths.

Definition at line 585 of file RandomTree.cs.

7.23.2.4 LabelledTree() [2/4]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.LabelledTree (
            int leafCount,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false,
            TreeNode constraint = null )  [static]
```

Samples a random labelled tree according to the specified model, using branch lengths drawn from a Uniform(0, 1) distribution.

Parameters

leafCount	The number of terminal nodes in the topology. Their names will be in the form `t1, t2, ..., tN`, where `N` is *leafCount* .
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.
constraint	A tree to constrain the sampling. The topology produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the topology, using a constraint with TreeNode.NullHypothesis.YHK will bias the sampled topology distribution.

Returns

A TreeNode object containing a random unlabelled tree, with branch lengths.

Definition at line 620 of file RandomTree.cs.

7.23.2.5 LabelledTree() [3/4]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.LabelledTree (
            IReadOnlyList< string > leafNames,
            IContinuousDistribution branchLengthDistribution,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false,
            TreeNode constraint = null )  [static]
```

Samples a random labelled tree according to the specified model, using branch lengths drawn from the supplied distribution.

Parameters

leafNames	The labels for the terminal nodes of the topology.
branchLengthDistribution	The continuous univariate distribution from which the branch lengths will be drawn.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.
constraint	A tree to constrain the sampling. The topology produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the topology, using a constraint with TreeNode.NullHypothesis.YHK will bias the sampled topology distribution.

Returns

A TreeNode object containing a random unlabelled tree, with branch lengths.

Definition at line 557 of file RandomTree.cs.

7.23.2.6 LabelledTree() [4/4]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.LabelledTree (
            IReadOnlyList< string > leafNames,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false,
            TreeNode constraint = null )  [static]
```

Samples a random labelled tree according to the specified model, using branch lengths drawn from a Uniform(0, 1) distribution.

Parameters

leafNames	The labels for the terminal nodes of the topology.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.
constraint	A tree to constrain the sampling. The topology produced by this method will be compatible with this tree. The constraint tree can be multifurcating. Please note that, as the constraint is applied at every step while growing the topology, using a constraint with TreeNode.NullHypothesis.YHK will bias the sampled topology distribution.

Returns

A TreeNode object containing a random unlabelled tree, with branch lengths.

Definition at line 606 of file RandomTree.cs.

7.23.2.7 ResolvePolytomies()

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.ResolvePolytomies (
            TreeNode tree,
            bool rooted = false )  [static]
```

Randomly resolve all the polytomies in a tree.

Parameters

tree	The tree containing the polytomies to be resolved.
rooted	A bool indicating whether the tree is supposed to be rooted or not. If this is `true`, a trichotomy at the root node will be resolved.

Returns

The tree, where all the polytomies have been randomly resolved. This is performed in-place, but the return value of this method should be used in case the root node has changed.

Definition at line 217 of file RandomTree.cs.

7.23.2.8 UnlabelledTopology()

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.UnlabelledTopology (
            int leafCount,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false )  [static]
```

Samples a random unlabelled topology according to the specified model.

Parameters

leafCount	The number of terminal nodes in the topology.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.

Returns

A TreeNode object containing a random unlabelled topology (branch lengths will all be set to double.NaN).

Exceptions

ArgumentException	Thrown if *model* is neither TreeNode.NullHypothesis.PDA nor TreeNode.NullHypothesis.YHK.

Definition at line 27 of file RandomTree.cs.

7.23.2.9 UnlabelledTree() [1/2]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.UnlabelledTree (
            int leafCount,
            IContinuousDistribution branchLengthDistribution,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false )  [static]
```

Samples a random unlabelled tree according to the specified model, using branch lengths drawn from the supplied distribution.

Parameters

leafCount	The number of terminal nodes in the topology.
branchLengthDistribution	The continuous univariate distribution from which the branch lengths will be drawn.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.

Returns

A TreeNode object containing a random unlabelled tree, with branch lengths.

Definition at line 150 of file RandomTree.cs.

7.23.2.10 UnlabelledTree() [2/2]

```
static TreeNode PhyloTree.TreeBuilding.RandomTree.UnlabelledTree (
            int leafCount,
            TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA,
            bool rooted = false )  [static]
```

Samples a random unlabelled tree according to the specified model, using branch lengths drawn from a Uniform(0, 1) distribution.

Parameters

leafCount	The number of terminal nodes in the topology.
model	The model to use for growing the tree.
rooted	A bool indicating whether the tree should be rooted or not.

Returns

A TreeNode object containing a random unlabelled tree, with branch lengths.

Definition at line 175 of file RandomTree.cs.

7.23.3 Member Data Documentation

7.23.3.1 RandomNumberGenerator

```
Random PhyloTree.TreeBuilding.RandomTree.RandomNumberGenerator = new ThreadSafeRandom()  [static]
```

Random number generator used for sampling.

Definition at line 17 of file RandomTree.cs.

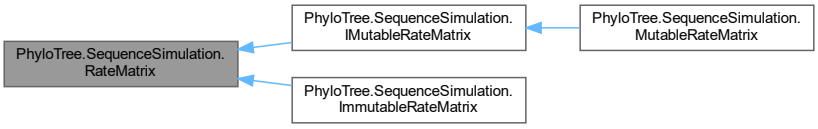
The documentation for this class was generated from the following file:

- TreeBuilding/RandomTree.cs

7.24 PhyloTree.SequenceSimulation.RateMatrix Class Reference

Represents a rate matrix for a continuous-type Markov chain model. This type cannot be instantiated directly, please use MutableRateMatrix or ImmutableRateMatrix instead, or access the static members for some pre-baked common rate matrices for DNA and protein evolution.

Inheritance diagram for PhyloTree.SequenceSimulation.RateMatrix:



Classes

- class DNA

  Contains rate matrices for DNA sequence evolution.
- class Protein

  Contains rate matrices for protein sequence evolution.

Properties

- abstract ImmutableArray< char > States [get]

  Gets the states for the character to which the rate matrix applies.
- abstract ImmutableArray< double > EquilibriumFrequencies [get]

  Gets the equilibrium frequences of the rate matrix.
- abstract double this[int from, int to] [get]

  Gets the rate of going from state number from to state number to . If `from == to` , the negative sum of the elements on the row is returned.
- abstract double this[char from, char to] [get]

  Gets the rate of going from state from to state to . If `from == to` , the negative sum of the elements on the row is returned.

7.24.1 Detailed Description

Represents a rate matrix for a continuous-type Markov chain model. This type cannot be instantiated directly, please use MutableRateMatrix or ImmutableRateMatrix instead, or access the static members for some pre-baked common rate matrices for DNA and protein evolution.

Definition at line 14 of file RateMatix.cs.

7.24.2 Property Documentation

7.24.2.1 EquilibriumFrequencies

```
abstract ImmutableArray<double> PhyloTree.SequenceSimulation.RateMatrix.EquilibriumFrequencies
[get]
```

Gets the equilibrium frequences of the rate matrix.

Definition at line 30 of file RateMatix.cs.

7.24.2.2 States

```
abstract ImmutableArray<char> PhyloTree.SequenceSimulation.RateMatrix.States [get]
```

Gets the states for the character to which the rate matrix applies.

Definition at line 25 of file RateMatix.cs.

7.24.2.3 this[char from, char to]

```
abstract double PhyloTree.SequenceSimulation.RateMatrix.this[char from, char to] [get]
```

Gets the rate of going from state *from* to state *to* . If `from == to` , the negative sum of the elements on the row is returned.

Parameters

from	The row state.
to	The column state.

Returns

The rate of going from state *from* to state *to* .

Exceptions

ArgumentOutOfRangeException	Thrown if the state is not part of the rate matrix.

Definition at line 53 of file RateMatix.cs.

7.24.2.4 this[int from, int to]

```
abstract double PhyloTree.SequenceSimulation.RateMatrix.this[int from, int to]  [get]
```

Gets the rate of going from state number *from* to state number *to* . If `from == to` , the negative sum of the elements on the row is returned.

Parameters

from	The row number.
to	The column number.

Returns

The rate of going from state number *from* to state number *to* .

Definition at line 41 of file RateMatix.cs.

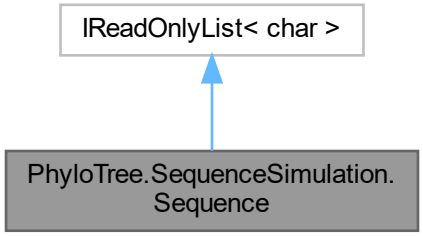
The documentation for this class was generated from the following file:

- SequenceSimulation/RateMatix.cs

7.25 PhyloTree.SequenceSimulation.Sequence Class Reference

Represents a sequence of characters.

Inheritance diagram for PhyloTree.SequenceSimulation.Sequence:



Public Member Functions

- Sequence (ReadOnlySpan< char > sequence)

  Creates a new Sequence.
- Sequence (ReadOnlySpan< char > sequence, IReadOnlyList< char > states)

  Creates a new Sequence.
- Sequence (ReadOnlySpan< char > sequence, IReadOnlyList< char > states, IReadOnlyList< double > conservation)

  Creates a new Sequence.
- Sequence (ReadOnlySpan< char > sequence, IReadOnlyList< char > states, IReadOnlyList< double > conservation, IReadOnlyList< double > indelProfile)

  Creates a new Sequence.
- override string ToString ()
- IEnumerator< char > GetEnumerator ()
- Sequence Evolve (RateMatrix rateMatrix, double time)

  Simulates the evolution of the Sequence over the specified amount of time , using the specified rate matrix. No indels are allowed to happen.
- Sequence Evolve (RateMatrix rateMatrix, IndelModel indelModel, double time)

  Simulates the evolution of the Sequence over the specified amount of time , using the specified rate matrix. Insertions and deletions happen according to the specified indelModel .
- Sequence Evolve (RateMatrix rateMatrix, IndelModel indelModel, double time, out Insertion[ ] insertions)

  Simulates the evolution of the Sequence over the specified amount of time , using the specified rate matrix. Insertions and deletions happen according to the specified indelModel .
- Dictionary< string, Sequence > Evolve (TreeNode tree, RateMatrix rateMatrix, double scale=1, IndelModel indelModel=null)

  Simulates the evolution of the sequence over a phylogenetic tree , with the specified rate matrix, scale factor, and insertion/deletion model.
- Dictionary< string, Sequence > EvolveAll (TreeNode tree, RateMatrix rateMatrix, double scale=1, IndelModel indelModel=null)

  Simulates the evolution of the sequence over a phylogenetic tree , with the specified rate matrix, scale factor, and insertion/deletion model.

Static Public Member Functions

- static implicit operator string (Sequence sequence)

  Converts a Sequence to a `string`
- static Sequence RandomSequence (int length, RateMatrix rateMatrix)

  Create a random sequence with the specified length , using the states and equilibrium frequencies from the rateMatrix .
- static Sequence RandomSequence (int length, IReadOnlyList< char > states)

  Create a random sequence with the specified length containing all states with equal probability.
- static Sequence RandomSequence (int length, char[ ] states)

  Create a random sequence with the specified length containing all states with equal probability.
- static Sequence RandomSequence (int length, IReadOnlyList< char > states, IReadOnlyList< double > stateFrequencies)

  Create a random sequence with the specified length containing each state with probabilities given by stateFrequencies .
- static Sequence RandomSequence (int length, char[ ] states, double[ ] stateFrequencies)

  Create a random sequence with the specified length containing each state with probabilities given by stateFrequencies .

Properties

- ImmutableArray< char > States `[get]`

  The possible states that make up the sequence. Note that some states may not actually be present in the sequence.
- ImmutableArray< double >? Conservation `[get]`

  The conservation profile of the sequence. If this is `null`*, all positions in the sequence are equally conserved.*
- ImmutableArray< double >? IndelProfile `[get]`

  The indel profile of the sequence. If this is `null`*, all positions in the sequence have equal probability of being affected by an indel event.*
- string StringSequence `[get]`

  Returns the sequence as a `string`*.*
- int Length `[get]`

  The length of the sequence.
- int Count `[get]`
- char this[int index] `[get]`

7.25.1 Detailed Description

Represents a sequence of characters.

Definition at line 14 of file Sequence.cs.

7.25.2 Constructor & Destructor Documentation

7.25.2.1 Sequence() [1/4]

```
PhyloTree.SequenceSimulation.Sequence.Sequence (
            ReadOnlySpan< char > sequence )
```

Creates a new Sequence.

Parameters

sequence	The sequence.

Definition at line 64 of file Sequence.cs.

7.25.2.2 Sequence() [2/4]

```
PhyloTree.SequenceSimulation.Sequence.Sequence (
            ReadOnlySpan< char > sequence,
            IReadOnlyList< char > states )
```

Creates a new Sequence.

Parameters

sequence	The sequence.
states	The possible states for the sequence.

Definition at line 95 of file Sequence.cs.

7.25.2.3 Sequence() [3/4]

```
PhyloTree.SequenceSimulation.Sequence.Sequence (
            ReadOnlySpan< char > sequence,
            IReadOnlyList< char > states,
            IReadOnlyList< double > conservation )
```

Creates a new Sequence.

Parameters

sequence	The sequence.
states	The possible states for the sequence.
conservation	The conservation profile for the sequence.

Definition at line 125 of file Sequence.cs.

7.25.2.4 Sequence() [4/4]

```
PhyloTree.SequenceSimulation.Sequence.Sequence (
            ReadOnlySpan< char > sequence,
```

```
            IReadOnlyList< char > states,
            IReadOnlyList< double > conservation,
            IReadOnlyList< double > indelProfile )
```

Creates a new Sequence.

Parameters

sequence	The sequence.
states	The possible states for the sequence.
conservation	The conservation profile for the sequence.
indelProfile	The indel profile for the sequence.

Definition at line 161 of file Sequence.cs.

7.25.3 Member Function Documentation

7.25.3.1 Evolve() [1/4]

```
Sequence PhyloTree.SequenceSimulation.Sequence.Evolve (
            RateMatrix rateMatrix,
            double time )
```

Simulates the evolution of the Sequence over the specified amount of *time* , using the specified rate matrix. No indels are allowed to happen.

Parameters

rateMatrix	The rate matrix to simulate sequence evolution.
time	The length of time over which the sequence evolves.

Returns

The evolved sequence.

Definition at line 288 of file Sequence.cs.

7.25.3.2 Evolve() [2/4]

```
Sequence PhyloTree.SequenceSimulation.Sequence.Evolve (
            RateMatrix rateMatrix,
            IndelModel indelModel,
            double time )
```

Simulates the evolution of the Sequence over the specified amount of *time* , using the specified rate matrix. Insertions and deletions happen according to the specified *indelModel* .

Parameters

rateMatrix	The rate matrix to simulate sequence evolution.
indelModel	The insertion/deletion model.
time	The length of time over which the sequence evolves.

Returns

The evolved sequence.

Definition at line 303 of file Sequence.cs.

7.25.3.3 Evolve() [3/4]

```
Sequence PhyloTree.SequenceSimulation.Sequence.Evolve (
            RateMatrix rateMatrix,
            IndelModel indelModel,
            double time,
            out Insertion[] insertions )
```

Simulates the evolution of the Sequence over the specified amount of *time* , using the specified rate matrix. Insertions and deletions happen according to the specified *indelModel* .

Parameters

rateMatrix	The rate matrix to simulate sequence evolution.
indelModel	The insertion/deletion model.
time	The length of time over which the sequence evolves.
insertions	An array containing all the insertion events that have occurred during the evolution of the sequence (useful to map the evolved sequence back onto the ancestral sequence).

Returns

The evolved sequence and an array containing all the insertion events that happened during its evolution.

Definition at line 320 of file Sequence.cs.

7.25.3.4 Evolve() [4/4]

```
Dictionary< string, Sequence > PhyloTree.SequenceSimulation.Sequence.Evolve (
            TreeNode tree,
            RateMatrix rateMatrix,
            double scale = 1,
            IndelModel indelModel = null )
```

Simulates the evolution of the sequence over a phylogenetic *tree* , with the specified rate matrix, *scale* factor, and insertion/deletion model.

Parameters

tree	The tree over which the sequence evolves. This is assumed to be rooted (i.e., the ancestral sequence is placed at the root of the tree).
rateMatrix	The rate matrix that governs the evolution of the sequence.
scale	A scaling factor. If this is different from 1, the effect is the same as multiplying the branch lengths of the tree or the rate matrix by the supplied value.
indelModel	The model for insertions/deletions. If this is null, no insertions/deletions are allowed to happen.

Returns

A Dictionary<String, Sequence> containing entries for all the nodes in the tree that have names. For each entry, the key is a `string` containing the TreeNode.Name of the node, and the value is a Sequence containing the sequence. The sequences are all aligned.

Definition at line 351 of file Sequence.cs.

7.25.3.5 EvolveAll()

```
Dictionary< string, Sequence > PhyloTree.SequenceSimulation.Sequence.EvolveAll (
            TreeNode tree,
            RateMatrix rateMatrix,
            double scale = 1,
            IndelModel indelModel = null )
```

Simulates the evolution of the sequence over a phylogenetic *tree* , with the specified rate matrix, *scale* factor, and insertion/deletion model.

Parameters

tree	The tree over which the sequence evolves. This is assumed to be rooted (i.e., the ancestral sequence is placed at the root of the tree).
rateMatrix	The rate matrix that governs the evolution of the sequence.
scale	A scaling factor. If this is different from 1, the effect is the same as multiplying the branch lengths of the tree or the rate matrix by the supplied value.
indelModel	The model for insertions/deletions. If this is null, no insertions/deletions are allowed to happen.

Returns

A Dictionary<String, Sequence> containing entries for all the nodes in the tree. For each entry, the key is a `string` containing the TreeNode.Id of the node, and the value is a Sequence containing the sequence. The sequences are all aligned.

Definition at line 367 of file Sequence.cs.

7.25.3.6 GetEnumerator()

```
IEnumerator< char > PhyloTree.SequenceSimulation.Sequence.GetEnumerator ( )
```

Definition at line 228 of file Sequence.cs.

7.25.3.7 operator string()

```
static implicit PhyloTree.SequenceSimulation.Sequence.operator string (
            Sequence sequence )  [static]
```

Converts a Sequence to a `string`

Parameters

sequence	The Sequence to convert.

Definition at line 216 of file Sequence.cs.

7.25.3.8 RandomSequence() [1/5]

```
static Sequence PhyloTree.SequenceSimulation.Sequence.RandomSequence (
            int length,
            char[] states )  [static]
```

Create a random sequence with the specified *length* containing all *states* with equal probability.

Parameters

length	The length of the sequence.
states	The character states to use for the sequence.

Returns

A random sequence

7.25.3.9 RandomSequence() [2/5]

```
static Sequence PhyloTree.SequenceSimulation.Sequence.RandomSequence (
            int length,
            char[] states,
            double[] stateFrequencies )  [static]
```

Create a random sequence with the specified *length* containing each state with probabilities given by *state*↩ *Frequencies* .

Parameters

length	The length of the sequence.
states	The character states to use for the sequence.
stateFrequencies	The frequency for each state.

Returns

A random sequence

7.25.3.10 RandomSequence() [3/5]

```
static Sequence PhyloTree.SequenceSimulation.Sequence.RandomSequence (
            int length,
            IReadOnlyList< char > states )  [static]
```

Create a random sequence with the specified *length* containing all *states* with equal probability.

Parameters

length	The length of the sequence.
states	The character states to use for the sequence.

Returns

A random sequence

7.25.3.11 RandomSequence() [4/5]

```
static Sequence PhyloTree.SequenceSimulation.Sequence.RandomSequence (
            int length,
            IReadOnlyList< char > states,
            IReadOnlyList< double > stateFrequencies )  [static]
```

Create a random sequence with the specified *length* containing each state with probabilities given by *state*↩ *Frequencies* .

Parameters

length	The length of the sequence.
states	The character states to use for the sequence.
stateFrequencies	The frequency for each state.

Returns

A random sequence

7.25.3.12 RandomSequence() [5/5]

```
static Sequence PhyloTree.SequenceSimulation.Sequence.RandomSequence (
            int length,
            RateMatrix rateMatrix )  [static]
```

Create a random sequence with the specified *length* , using the states and equilibrium frequencies from the *rate*↩ *Matrix* .

Parameters

length	The length of the sequence.
rateMatrix	The rate matrix.

Returns

A random sequence.

7.25.3.13 ToString()

```
override string PhyloTree.SequenceSimulation.Sequence.ToString ( )
```

Definition at line 222 of file Sequence.cs.

7.25.4 Property Documentation

7.25.4.1 Conservation

```
ImmutableArray<double>?  PhyloTree.SequenceSimulation.Sequence.Conservation  [get]
```

The conservation profile of the sequence. If this is `null`, all positions in the sequence are equally conserved.

Definition at line 27 of file Sequence.cs.

7.25.4.2 Count

```
int PhyloTree.SequenceSimulation.Sequence.Count  [get]
```

Definition at line 55 of file Sequence.cs.

7.25.4.3 IndelProfile

```
ImmutableArray<double>?  PhyloTree.SequenceSimulation.Sequence.IndelProfile  [get]
```

The indel profile of the sequence. If this is `null`, all positions in the sequence have equal probability of being affected by an indel event.

Definition at line 33 of file Sequence.cs.

7.25.4.4 Length

```
int PhyloTree.SequenceSimulation.Sequence.Length  [get]
```

The length of the sequence.

Definition at line 49 of file Sequence.cs.

7.25.4.5 States

```
ImmutableArray<char> PhyloTree.SequenceSimulation.Sequence.States  [get]
```

The possible states that make up the sequence. Note that some states may not actually be present in the sequence.

Definition at line 22 of file Sequence.cs.

7.25.4.6 StringSequence

```
string PhyloTree.SequenceSimulation.Sequence.StringSequence  [get]
```

Returns the sequence as a `string`.

Definition at line 38 of file Sequence.cs.

7.25.4.7 this[int index]

```
char PhyloTree.SequenceSimulation.Sequence.this[int index]  [get]
```

Definition at line 58 of file Sequence.cs.

The documentation for this class was generated from the following file:

- SequenceSimulation/Sequence.cs

7.26 PhyloTree.SequenceSimulation.SequenceSimulation Class Reference

Contains methods to simulate sequence evolution.

Public Member Functions

- delegate double GetScale (double conservation)

  Represents a function that can be evaluated to return the scaling factor to use in order to obtain the specified average conservation .

Static Public Member Functions

- static Dictionary< string, Sequence > SimulateSequences (this TreeNode tree, Sequence ancestral↩ Sequence, RateMatrix rateMatrix, double scale=1, IndelModel indelModel=null)

  Simulates the evolution of the specified ancestral sequence over a phylogenetic tree , with the specified rate matrix, scale factor, and insertion/deletion model.
- static Dictionary< string, Sequence > SimulateSequences (this TreeNode tree, int ancestralSequence↩ Length, RateMatrix rateMatrix, double scale=1, IndelModel indelModel=null)

  Simulates the evolution of a random ancestral sequence with the specified length over a phylogenetic tree , with the specified rate matrix, scale factor, and insertion/deletion model.
- static Dictionary< string, Sequence > SimulateAllSequences (this TreeNode tree, Sequence ancestral↩ Sequence, RateMatrix rateMatrix, double scale=1, IndelModel indelModel=null)

  Simulates the evolution of the specified ancestral sequence over a phylogenetic tree , with the specified rate matrix, scale factor, and insertion/deletion model.
- static Dictionary< string, Sequence > SimulateAllSequences (this TreeNode tree, int ancestralSequence↩ Length, RateMatrix rateMatrix, double scale=1, IndelModel indelModel=null)

  Simulates the evolution of a random ancestral sequence with the specified length over a phylogenetic tree , with the specified rate matrix, scale factor, and insertion/deletion model.
- static GetScale ConservationToScale (TreeNode tree, RateMatrix rateMatrix, double minRate=1e-5, double maxRate=1e4)

  Returns a method that can be evaluated to determine the scaling factor that, when applied to a sequence simulation done using the specified tree and rate matrix, will produce at the tips a sequence alignment with the specified (average) percent identity.
- static Dictionary< string, string > ToStringAlignment (this Dictionary< string, Sequence > alignment)

  Converts a sequence alignment where the sequences are stored as Sequences into an alignment where the sequences are stored as `string`s.

Properties

- static Random RandomNumberGenerator = new ThreadSafeRandom() `[get, set]`

  The random number generator used to simulate sequence evolution. If you change this, please ensure that it is thread-safe.

7.26.1 Detailed Description

Contains methods to simulate sequence evolution.

Definition at line 16 of file SequenceSimulation.cs.

7.26.2 Member Function Documentation

7.26.2.1 ConservationToScale()

```
static GetScale PhyloTree.SequenceSimulation.SequenceSimulation.ConservationToScale (
            TreeNode tree,
            RateMatrix rateMatrix,
            double minRate = 1e-5,
            double maxRate = 1e4 )  [static]
```

Returns a method that can be evaluated to determine the scaling factor that, when applied to a sequence simulation done using the specified *tree* and rate matrix, will produce at the tips a sequence alignment with the specified (average) percent identity.

Parameters

tree	The TreeNode on which the sequence simulations will be performed.
rateMatrix	The rate matrix that will be used for the sequence simulations.
minRate	Minimum rate value to test.
maxRate	Maximum rate value to test.

Returns

A method that can be evaluated to determine the scaling factor that, when applied to a sequence simulation done using the specified *tree* and rate matrix, will produce at the tips a sequence alignment with the specified (average) percent identity.

Definition at line 211 of file SequenceSimulation.public.cs.

7.26.2.2 GetScale()

```
delegate double PhyloTree.SequenceSimulation.SequenceSimulation.GetScale (
            double conservation )
```

Represents a function that can be evaluated to return the scaling factor to use in order to obtain the specified average *conservation* .

Parameters

conservation	The average conservation whose corresponding scaling factor will be returned.

Returns

The scaling factor that will produce the speicfied average *conservation* .

7.26.2.3 SimulateAllSequences() [1/2]

```
static Dictionary< string, Sequence > PhyloTree.SequenceSimulation.SequenceSimulation.Simulate↩
AllSequences (
            this TreeNode tree,
            int ancestralSequenceLength,
            RateMatrix rateMatrix,
            double scale = 1,
            IndelModel indelModel = null )  [static]
```

Simulates the evolution of a random ancestral sequence with the specified length over a phylogenetic *tree* , with the specified rate matrix, *scale* factor, and insertion/deletion model.

Parameters

tree	The tree over which the sequence evolves. This is assumed to be rooted (i.e., the ancestral sequence is placed at the root of the tree).
ancestralSequenceLength	The length of the ancestral sequence whose evolution is being simulated. Note that if insertions/deletions are allowed to happen, the final length of the (aligned) sequences may differ from this.
rateMatrix	The rate matrix that governs the evolution of the sequence.
scale	A scaling factor. If this is different from 1, the effect is the same as multiplying the branch lengths of the tree or the rate matrix by the supplied value.
indelModel	The model for insertions/deletions. If this is null, no insertions/deletions are allowed to happen.

Returns

A Dictionary<String, Sequence> containing entries for all the nodes in the tree. For each entry, the key is a `string` containing the TreeNode.Id of the node, and the value is a Sequence containing the sequence. The sequences are all aligned.

Definition at line 187 of file SequenceSimulation.public.cs.

7.26.2.4 SimulateAllSequences() [2/2]

```
static Dictionary< string, Sequence > PhyloTree.SequenceSimulation.SequenceSimulation.Simulate↩
AllSequences (
```

```
            this TreeNode tree,
            Sequence ancestralSequence,
            RateMatrix rateMatrix,
            double scale = 1,
            IndelModel indelModel = null )  [static]
```

Simulates the evolution of the specified ancestral sequence over a phylogenetic *tree* , with the specified rate matrix, *scale* factor, and insertion/deletion model.

Parameters

tree	The tree over which the sequence evolves. This is assumed to be rooted (i.e., the ancestral sequence is placed at the root of the tree).
ancestralSequence	The ancestral sequence whose evolution is being simulated.
rateMatrix	The rate matrix that governs the evolution of the sequence.
scale	A scaling factor. If this is different from 1, the effect is the same as multiplying the branch lengths of the tree or the rate matrix by the supplied value.
indelModel	The model for insertions/deletions. If this is null, no insertions/deletions are allowed to happen.

Returns

A Dictionary<String, Sequence> containing entries for all the nodes in the tree. For each entry, the key is a `string` containing the TreeNode.Id of the node, and the value is a Sequence containing the sequence. The sequences are all aligned.

Definition at line 116 of file SequenceSimulation.public.cs.

7.26.2.5 SimulateSequences() [1/2]

```
static Dictionary< string, Sequence > PhyloTree.SequenceSimulation.SequenceSimulation.Simulate↩
Sequences (
            this TreeNode tree,
            int ancestralSequenceLength,
            RateMatrix rateMatrix,
            double scale = 1,
            IndelModel indelModel = null )  [static]
```

Simulates the evolution of a random ancestral sequence with the specified length over a phylogenetic *tree* , with the specified rate matrix, *scale* factor, and insertion/deletion model.

Parameters

tree	The tree over which the sequence evolves. This is assumed to be rooted (i.e., the ancestral sequence is placed at the root of the tree).
ancestralSequenceLength	The length of the ancestral sequence whose evolution is being simulated. Note that if insertions/deletions are allowed to happen, the final length of the (aligned) sequences may differ from this.
rateMatrix	The rate matrix that governs the evolution of the sequence.
scale	A scaling factor. If this is different from 1, the effect is the same as multiplying the branch lengths of the tree or the rate matrix by the supplied value.
indelModel	The model for insertions/deletions. If this is null, no insertions/deletions are
	allowed to happen.

Returns

A Dictionary<String, Sequence> containing entries for all the nodes in the tree that have names. For each entry, the key is a `string` containing the TreeNode.Name of the node, and the value is a Sequence containing the sequence. The sequences are all aligned.

Definition at line 97 of file SequenceSimulation.public.cs.

7.26.2.6 SimulateSequences() [2/2]

```
static Dictionary< string, Sequence > PhyloTree.SequenceSimulation.SequenceSimulation.Simulate↩
Sequences (
            this TreeNode tree,
            Sequence ancestralSequence,
            RateMatrix rateMatrix,
            double scale = 1,
            IndelModel indelModel = null )  [static]
```

Simulates the evolution of the specified ancestral sequence over a phylogenetic *tree* , with the specified rate matrix, *scale* factor, and insertion/deletion model.

Parameters

tree	The tree over which the sequence evolves. This is assumed to be rooted (i.e., the ancestral sequence is placed at the root of the tree).
ancestralSequence	The ancestral sequence whose evolution is being simulated.
rateMatrix	The rate matrix that governs the evolution of the sequence.
scale	A scaling factor. If this is different from 1, the effect is the same as multiplying the branch lengths of the tree or the rate matrix by the supplied value.
indelModel	The model for insertions/deletions. If this is null, no insertions/deletions are allowed to happen.

Returns

A Dictionary<String, Sequence> containing entries for all the nodes in the tree that have names. For each entry, the key is a `string` containing the TreeNode.Name of the node, and the value is a Sequence containing the sequence. The sequences are all aligned.

Definition at line 23 of file SequenceSimulation.public.cs.

7.26.2.7 ToStringAlignment()

```
static Dictionary< string, string > PhyloTree.SequenceSimulation.SequenceSimulation.ToString↩
Alignment (
            this Dictionary< string, Sequence > alignment )  [static]
```

Converts a sequence alignment where the sequences are stored as Sequences into an alignment where the sequences are stored as `string`s.

Parameters

alignment	The alignment to convert.

Returns

A Dictionary<String, String> where both keys and values are string.

Definition at line 310 of file SequenceSimulation.public.cs.

7.26.3 Property Documentation

7.26.3.1 RandomNumberGenerator

```
Random PhyloTree.SequenceSimulation.SequenceSimulation.RandomNumberGenerator = new ThreadSafeRandom()
[static], [get], [set]
```

The random number generator used to simulate sequence evolution. If you change this, please ensure that it is thread-safe.

Definition at line 21 of file SequenceSimulation.cs.

The documentation for this class was generated from the following files:

- SequenceSimulation/SequenceSimulation.cs
- SequenceSimulation/SequenceSimulation.public.cs

7.27 PhyloTree.TreeBuilding.ThreadSafeRandom Class Reference

Represents a thread-safe random number generator.

Inheritance diagram for PhyloTree.TreeBuilding.ThreadSafeRandom:

Public Member Functions

- ThreadSafeRandom (int seed)

  Initialise a new thread-safe random number generator with the specified seed.
- ThreadSafeRandom ()

  Initialise a new thread-safe random number generator.
- override int Next ()
- override int Next (int maxValue)
- override int Next (int minValue, int maxValue)
- override double NextDouble ()
- override void NextBytes (byte[ ] buffer)

7.27.1 Detailed Description

Represents a thread-safe random number generator.

Adapted from https://stackoverflow.com/questions/3049467/is-c-sharp-random-number-generator

Definition at line 10 of file ThreadSafeRandom.cs.

7.27.2 Constructor & Destructor Documentation

7.27.2.1 ThreadSafeRandom() [1/2]

```
PhyloTree.TreeBuilding.ThreadSafeRandom.ThreadSafeRandom (
            int seed )
```

Initialise a new thread-safe random number generator with the specified seed.

Parameters

seed	A number used to generate a starting number for the pseudo-random sequence.

Definition at line 22 of file ThreadSafeRandom.cs.

7.27.2.2 ThreadSafeRandom() [2/2]

```
PhyloTree.TreeBuilding.ThreadSafeRandom.ThreadSafeRandom ( )
```

Initialise a new thread-safe random number generator.

Definition at line 34 of file ThreadSafeRandom.cs.

7.27.3 Member Function Documentation

7.27.3.1 Next() [1/3]

```
override int PhyloTree.TreeBuilding.ThreadSafeRandom.Next ( )
```

Definition at line 60 of file ThreadSafeRandom.cs.

7.27.3.2 Next() [2/3]

```
override int PhyloTree.TreeBuilding.ThreadSafeRandom.Next (
            int maxValue )
```

Definition at line 67 of file ThreadSafeRandom.cs.

7.27.3.3 Next() [3/3]

```
override int PhyloTree.TreeBuilding.ThreadSafeRandom.Next (
            int minValue,
            int maxValue )
```

Definition at line 74 of file ThreadSafeRandom.cs.

7.27.3.4 NextBytes()

```
override void PhyloTree.TreeBuilding.ThreadSafeRandom.NextBytes (
            byte[] buffer )
```

Definition at line 88 of file ThreadSafeRandom.cs.

7.27.3.5 NextDouble()

```
override double PhyloTree.TreeBuilding.ThreadSafeRandom.NextDouble ( )
```

Definition at line 81 of file ThreadSafeRandom.cs.

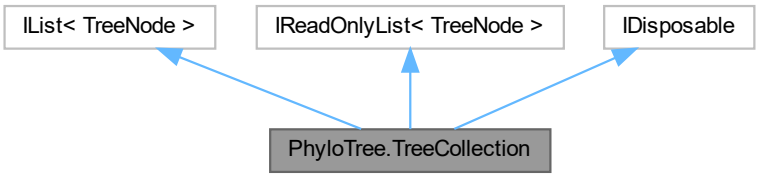
The documentation for this class was generated from the following file:

- TreeBuilding/ThreadSafeRandom.cs

7.28 PhyloTree.TreeCollection Class Reference

Represents a collection of TreeNode objects. If the full representations of the TreeNode objects reside in memory, this offers the best performance at the expense of memory usage. Alternatively, the trees may be read on demand from a stream in binary format. In this case, accessing any of the trees will require the tree to be parsed. This reduces memory usage, but worsens performance. The internal storage model of the collection is transparent to consumers (except for the difference in performance/memory usage).

Inheritance diagram for PhyloTree.TreeCollection:



Public Member Functions

- void Add (TreeNode item)

  Adds an element to the collection. This is stored in memory, even if the internal storage model of the collection is a Stream.
- void AddRange (IEnumerable< TreeNode > items)

  Adds multiple elements to the collection. These are stored in memory, even if the internal storage model of the collection is a Stream.
- IEnumerator< TreeNode > GetEnumerator ()

  Get an IEnumerator over the collection.
- int IndexOf (TreeNode item)

  Finds the index of the first occurrence of an element in the collection.
- void Insert (int index, TreeNode item)

  Inserts an element in the collection at the specified index.
- void RemoveAt (int index)

  Removes from the collection the element at the specified index.
- void Clear ()

  Removes all elements from the collection. If the internal storage model is a Stream, it is disposed and the internal storage model is converted to a List<TreeNode>.
- bool Contains (TreeNode item)

  Determines whether the collection contains the specified element.
- void CopyTo (TreeNode[] array, int arrayIndex)

  Copies the collection to an array.
- bool Remove (TreeNode item)

  Removes the specified element from the collection.
- TreeCollection (List< TreeNode > internalStorage)

  Constructs a TreeCollection object from a List<TreeNode>.
- TreeCollection (Stream binaryTreeStream)

  Constructs a TreeCollection object from a stream of trees in binary format.
- void Dispose ()

  Disposes the TreeCollection, the underlying Stream and StreamReader, and deletes the TemporaryFile (if applicable).

Properties

- Stream UnderlyingStream = null [get]

  A stream containing the tree data in binary format, if this is the chosen storage model. This can be either a MemoryStream or a FileStream.
- string TemporaryFile = null [get, set]

  If the trees are stored on disk in a temporary file, you should assign this property to the full path of the file. The file will be deleted when the TreeCollection is Dispose()d.
- int Count [get]

  The number of trees in the collection.
- bool IsReadOnly [get]

  Determine whether the collection is read-only. This is always `false` in the current implementation.
- TreeNode this[int index] [get, set]

  Obtains an element from the collection.

7.28.1 Detailed Description

Represents a collection of TreeNode objects. If the full representations of the TreeNode objects reside in memory, this offers the best performance at the expense of memory usage. Alternatively, the trees may be read on demand from a stream in binary format. In this case, accessing any of the trees will require the tree to be parsed. This reduces memory usage, but worsens performance. The internal storage model of the collection is transparent to consumers (except for the difference in performance/memory usage).

Definition at line 16 of file TreeCollection.cs.

7.28.2 Constructor & Destructor Documentation

7.28.2.1 TreeCollection() [1/2]

```
PhyloTree.TreeCollection.TreeCollection (
            List< TreeNode > internalStorage )
```

Constructs a TreeCollection object from a List<TreeNode>.

Parameters

internalStorage	The List<TreeNode> containing the trees to store in the collection. Note that this list is not copied, but used as-is.

Definition at line 429 of file TreeCollection.cs.

7.28.2.2 TreeCollection() [2/2]

```
PhyloTree.TreeCollection.TreeCollection (
            Stream binaryTreeStream )
```

Constructs a TreeCollection object from a stream of trees in binary format.

Parameters

binaryTreeStream	The stream of trees in binary format to use. The stream will be disposed when the TreeCollection is disposed. It should not be disposed earlier by external code.

Definition at line 439 of file TreeCollection.cs.

7.28.3 Member Function Documentation

7.28.3.1 Add()

```
void PhyloTree.TreeCollection.Add (
            TreeNode item )
```

Adds an element to the collection. This is stored in memory, even if the internal storage model of the collection is a Stream.

Parameters

item	The element to add.

Definition at line 161 of file TreeCollection.cs.

7.28.3.2 AddRange()

```
void PhyloTree.TreeCollection.AddRange (
            IEnumerable< TreeNode > items )
```

Adds multiple elements to the collection. These are stored in memory, even if the internal storage model of the collection is a Stream.

Parameters

items	The elements to add.

Definition at line 174 of file TreeCollection.cs.

7.28.3.3 Clear()

```
void PhyloTree.TreeCollection.Clear ( )
```

Removes all elements from the collection. If the internal storage model is a Stream, it is disposed and the internal storage model is converted to a List<TreeNode>.

Definition at line 329 of file TreeCollection.cs.

7.28.3.4 Contains()

```
bool PhyloTree.TreeCollection.Contains (
            TreeNode item )
```

Determines whether the collection contains the specified element.

Parameters

item	The element to search for.

Returns

`true` if the collection contains the specified element, `false` otherwise.

Definition at line 356 of file TreeCollection.cs.

7.28.3.5 CopyTo()

```
void PhyloTree.TreeCollection.CopyTo (
            TreeNode[] array,
            int arrayIndex )
```

Copies the collection to an array.

Parameters

array	The array in which to copy the collection.
arrayIndex	The index at which to start the copy.

Definition at line 383 of file TreeCollection.cs.

7.28.3.6 Dispose()

```
void PhyloTree.TreeCollection.Dispose ( )
```

Disposes the TreeCollection, the underlying Stream and StreamReader, and deletes the TemporaryFile (if applicable).

Definition at line 513 of file TreeCollection.cs.

7.28.3.7 GetEnumerator()

```
IEnumerator< TreeNode > PhyloTree.TreeCollection.GetEnumerator ( )
```

Get an IEnumerator over the collection.

Returns

An IEnumerator over the collection.

Definition at line 191 of file TreeCollection.cs.

7.28.3.8 IndexOf()

```
int PhyloTree.TreeCollection.IndexOf (
            TreeNode item )
```

Finds the index of the first occurrence of an element in the collection.

Parameters

item	The item to search for.

Returns

The index of the item in the collection.

Definition at line 244 of file TreeCollection.cs.

7.28.3.9 Insert()

```
void PhyloTree.TreeCollection.Insert (
            int index,
            TreeNode item )
```

Inserts an element in the collection at the specified index.

Parameters

index	The index at which to insert the element.
item	The element to insert.

Definition at line 271 of file TreeCollection.cs.

7.28.3.10 Remove()

```
bool PhyloTree.TreeCollection.Remove (
            TreeNode item )
```

Removes the specified element from the collection.

Parameters

item	The element to remove.

Returns

`true` if the removal was successful (i.e. the list contained the element in the first place), `false` otherwise.

Definition at line 404 of file TreeCollection.cs.

7.28.3.11 RemoveAt()

```
void PhyloTree.TreeCollection.RemoveAt (
            int index )
```

Removes from the collection the element at the specified index.

Parameters

index	

Definition at line 298 of file TreeCollection.cs.

7.28.4 Property Documentation

7.28.4.1 Count

```
int PhyloTree.TreeCollection.Count  [get]
```

The number of trees in the collection.

Definition at line 87 of file TreeCollection.cs.

7.28.4.2 IsReadOnly

```
bool PhyloTree.TreeCollection.IsReadOnly  [get]
```

Determine whether the collection is read-only. This is always `false` in the current implementation.

Definition at line 105 of file TreeCollection.cs.

7.28.4.3 TemporaryFile

```
string PhyloTree.TreeCollection.TemporaryFile = null  [get], [set]
```

If the trees are stored on disk in a temporary file, you should assign this property to the full path of the file. The file will be deleted when the TreeCollection is Dispose()d.

Definition at line 82 of file TreeCollection.cs.

7.28.4.4 this[int index]

```
TreeNode PhyloTree.TreeCollection.this[int index]  [get], [set]
```

Obtains an element from the collection.

Parameters

index	The index of the element to extract.

Returns

The requested element from the collection.

Definition at line 112 of file TreeCollection.cs.

7.28.4.5 UnderlyingStream

```
Stream PhyloTree.TreeCollection.UnderlyingStream = null  [get]
```

A stream containing the tree data in binary format, if this is the chosen storage model. This can be either a MemoryStream or a FileStream.

Definition at line 26 of file TreeCollection.cs.

The documentation for this class was generated from the following file:

- TreeCollection.cs

7.29 PhyloTree.TreeNode Class Reference

Represents a node in a tree (or a whole tree).

Public Types

- enum TreeComparisonPruningMode

  Defines ways of pruning trees during comparisons.
- enum NodeRelationship

  Describes the relationship between two nodes.
- enum NullHypothesis

  Null hypothesis for normalising tree shape indices.

Public Member Functions

- double RobinsonFouldsDistance (TreeNode otherTree, bool weighted)

  Computes the Robinson-Foulds distance between the current tree and another tree.
- double EdgeLengthDistance (TreeNode otherTree)

  Computes the edge-length distance between the current tree and another tree.
- TreeNode (TreeNode parent)

  Creates a new TreeNode object.
- bool IsRooted ()

  Checks whether the node belongs to a rooted tree.
- TreeNode GetUnrootedTree ()

  Get an unrooted version of the tree.
- TreeNode GetRootedTree (TreeNode outgroup, double position=0.5)

  Get a version of the tree that is rooted at the specified point of the branch leading to the outgroup .
- TreeNode Clone ()

  Recursively clone a TreeNode object.
- List< TreeNode > GetChildrenRecursive ()

  Recursively get all the nodes that descend from this node.
- IEnumerable< TreeNode > GetChildrenRecursiveLazy ()

  Lazily recursively get all the nodes that descend from this node.
- List< TreeNode > GetLeaves ()

  Get all the leaves that descend (directly or indirectly) from this node.
- List< string > GetLeafNames ()

  Get the names of all the leaves that descend (directly or indirectly) from this node.
- List< string > GetNodeNames ()

  Get the names of all the named nodes that descend (directly or indirectly) from this node.
- TreeNode GetNodeFromName (string nodeName)

  Get the child node with the specified name.
- TreeNode GetNodeFromId (string nodeId)

  Get the child node with the specified Id.
- double UpstreamLength ()

  Get the sum of the branch lengths from this node up to the root.
- double LongestDownstreamLength ()

  Get the sum of the branch lengths from this node down to the leaves of the tree. If the tree is not clock-like, the length of the longest path is returned.
- double ShortestDownstreamLength ()

Get the sum of the branch lengths from this node down to the leaves of the tree. If the tree is not clock-like, the length of the shortest path is returned.

- TreeNode GetRootNode ()

  Get the node of the tree from which all other nodes descend.

- double PathLengthTo (TreeNode otherNode, NodeRelationship nodeRelationship=NodeRelationship.↩ Unknown)

  Get the sum of the branch lengths from this node to the specified node.

- double TotalLength ()

  Get the sum of the branch lengths of this node and all its descendants.

- void SortNodes (bool descending)

  Sort (in place) the nodes in the tree in an aesthetically pleasing way.

- override string ToString ()

  Convert the tree to a Newick string.

- bool IsClockLike (double tolerance=0.001)

  Determines whether the tree is clock-like (i.e. all tips are contemporaneous) or not.

- TreeNode GetLastCommonAncestor (params string[ ] monophyleticConstraint)

  Gets the last common ancestor of all the nodes with the specified names, or `null` if the tree doesn't contain all the named nodes.

- TreeNode GetLastCommonAncestor (IEnumerable< string > monophyleticConstraint)

  Gets the last common ancestor of all the nodes with the specified names, or `null` if the tree doesn't contain all the named nodes.

- bool IsLastCommonAncestor (IEnumerable< string > monophyleticConstraint)

  Checks whether this node is the last common ancestor of all the nodes with the specified names.

- List< TreeNode > List< TreeNode > side2 GetSplit ()
- IEnumerable<(List< TreeNode > side1, List< TreeNode > side2, double branchLength)> GetSplits ()

  Gets all the splits in the tree.

- TreeNode Prune (bool leaveParent)

  Prunes the current node from the tree.

- TreeNode Prune (TreeNode nodeToPrune, bool leaveParent)

  Prunes a node from the tree.

- double[ ][ ] CreateDistanceMatrixDouble (int maxDegreeOfParallelism=0, Action< double > progress↩ Callback=null)

  Creates a lower triangular distance matrix, where each entry is the path length distance between two leaves in the tree. Entries are in the same order as returned by the GetLeaves method.

- float[ ][ ] CreateDistanceMatrixFloat (int maxDegreeOfParallelism=0, Action< double > progress↩ Callback=null)

  Creates a lower triangular distance matrix, where each entry is the path length distance between two leaves in the tree. Entries are in the same order as returned by the GetLeaves method.

- int GetDepth ()

  Compute the depth of the node (number of branches from this node until the root node).

- double SackinIndex (NullHypothesis model=NullHypothesis.None)

  Computes the Sackin index of the tree (sum of the leaf depths).

- double CollessIndex (NullHypothesis model=NullHypothesis.None, double yhkExpectation=double.NaN)

  Compute the Colless index of the tree.

- double NumberOfCherries (NullHypothesis model=NullHypothesis.None)

  Computes the number of cherries in the tree.

Static Public Member Functions

- static double RobinsonFouldsDistance (TreeNode tree1, TreeNode tree2, bool weighted)

  Computes the Robinson-Foulds distance between two trees.
- static double EdgeLengthDistance (TreeNode tree1, TreeNode tree2)

  Computes the edge-length distance between two trees.
- static double[,] RobinsonFouldsDistances (IReadOnlyList< TreeNode > trees, bool weighted, TreeComparisonPruningMode pruningMode=TreeComparisonPruningMode.Pairwise, int maxThreadCount=-1, IProgress< double > progress=null)

  Computes a distance matrix containing the Robinson-Foulds distances between each pair of trees in a list.
- static double[,] EdgeLengthDistances (IReadOnlyList< TreeNode > trees, TreeComparisonPruningMode pruningMode=TreeComparisonPruningMode.Pairwise, int maxThreadCount=-1, IProgress< double > progress=null)

  Computes a distance matrix containing the edge-length distances between each pair of trees in a list.
- static void RobinsonFouldsDistances (IReadOnlyList< TreeNode > trees, out double[,] RFDistances, out double[,] weightedRFDistances, TreeComparisonPruningMode pruningMode=TreeComparisonPruning↩ Mode.Pairwise, int maxThreadCount=-1, IProgress< double > progress=null)

  Computes two distance matrices containing the unweighted and weighted Robinson-Foulds distances between each pair of trees in a list. Much faster than computing the two distance matrices separately.
- static void TreeDistances (IReadOnlyList< TreeNode > trees, out double[,] RFDistances, out double[,] weightedRFDistances, out double[,] ELDistances, TreeComparisonPruningMode pruningMode=Tree↩ ComparisonPruningMode.Pairwise, int maxThreadCount=-1, IProgress< double > progress=null)

  Computes three distance matrices containing the unweighted and weighted Robinson-Foulds distances and the edge-length distances between each pair of trees in a list. Much faster than computing the three distance matrices separately.
- static TreeNode GetLastCommonAncestor (IEnumerable< TreeNode > monophyleticConstraint)

  Gets the last common ancestor of all the specified nodes, or `null` if the tree doesn't contain all the nodes.
- static double GetCollessExpectationYHK (int numberOfLeaves)

  Computes the expected value of the Colless index under the YHK model.

Public Attributes

- List< TreeNode > side1

  Gets the split corresponding to the branch underlying this node. If this is an internal node, `side1` will contain all the leaves in the tree except those descending from this node, and `side2` will contain all the leaves descending from this node. If this is the root `side1` will be empty and `side2` will contain all the leaves in the tree. If the tree is rooted (the root node has exactly 2 children), `side1` will contain in all cases an additional `null` element.

Properties

- TreeNode Parent `[get, set]`

  The parent node of this node. This will be `null` for the root node.
- List< TreeNode > Children `[get]`

  The child nodes of this node. This will be empty (but initialised) for leaf nodes.
- AttributeDictionary Attributes = new AttributeDictionary() `[get]`

  The attributes of this node. Attributes Name, Length and Support are always included. See the respective properties for default values.
- double Length `[get, set]`

  The length of the branch leading to this node. This is `double.NaN` for branches whose length is not specified (e.g. the root node).
- double Support `[get, set]`

  The support value of this node. This is `double.NaN` for branches whose support is not specified. The interpretation of the support value depends on how the tree was built.
- string Name `[get, set]`

  The name of this node (e.g. the species name for leaf nodes). Default is `""`.
- string Id `[get]`

  A univocal identifier for the node.

7.29.1 Detailed Description

Represents a node in a tree (or a whole tree).

Definition at line 16 of file TreeNode.Comparisons.cs.

7.29.2 Member Enumeration Documentation

7.29.2.1 NodeRelationship

```
enum PhyloTree.TreeNode.NodeRelationship
```

Describes the relationship between two nodes.

Definition at line 807 of file TreeNode.cs.

7.29.2.2 NullHypothesis

```
enum PhyloTree.TreeNode.NullHypothesis
```

Null hypothesis for normalising tree shape indices.

Definition at line 13 of file TreeNode.ShapeIndices.cs.

7.29.2.3 TreeComparisonPruningMode

```
enum PhyloTree.TreeNode.TreeComparisonPruningMode
```

Defines ways of pruning trees during comparisons.

Definition at line 99 of file TreeNode.Comparisons.cs.

7.29.3 Constructor & Destructor Documentation

7.29.3.1 TreeNode()

```
PhyloTree.TreeNode.TreeNode (
            TreeNode parent )
```

Creates a new TreeNode object.

Parameters

parent	The parent node of this node. For the root node, this should be `null`.

Definition at line 378 of file TreeNode.cs.

7.29.4 Member Function Documentation

7.29.4.1 Clone()

```
TreeNode PhyloTree.TreeNode.Clone ( )
```

Recursively clone a TreeNode object.

Returns

The cloned TreeNode

Definition at line 554 of file TreeNode.cs.

7.29.4.2 CollessIndex()

```
double PhyloTree.TreeNode.CollessIndex (
            NullHypothesis model = NullHypothesis.None,
            double yhkExpectation = double.NaN )
```

Compute the Colless index of the tree.

Parameters

model	If this is NullHypothesis.None, the raw Colless index is returned. If this is NullHypothesis.YHK or NullHypothesis.PDA, the Colless index is normalised with respect to the corresponding null tree model (which makes scores comparable across trees of different sizes).
yhkExpectation	If *model* is NullHypothesis.YHK, you can optionally use this parameter to provide a pre-computed value for the expected value of the Colless index under the YHK model. This is useful to save time if you need to compute the Colless index of many trees with the same number of leaves. If this is double.NaN, the expected value under the YHK model is computed by this method.

Returns

The Colless index of the tree.

Definition at line 139 of file TreeNode.ShapeIndices.cs.

7.29.4.3 CreateDistanceMatrixDouble()

```
double[][] PhyloTree.TreeNode.CreateDistanceMatrixDouble (
            int maxDegreeOfParallelism = 0,
            Action< double > progressCallback = null )
```

Creates a lower triangular distance matrix, where each entry is the path length distance between two leaves in the tree. Entries are in the same order as returned by the GetLeaves method.

Parameters

maxDegreeOfParallelism	Maximum number of threads to use, or -1 to let the runtime decide. If this argument is set to 0 (the default), the value used is 1 for trees with 1500 or fewer leaves, or -1 for larger trees.
progressCallback	A method used to report progress.

Returns

A T:double[ ][ ] jagged array containing the distance matrix.

Definition at line 1325 of file TreeNode.cs.

7.29.4.4 CreateDistanceMatrixFloat()

```
float[][] PhyloTree.TreeNode.CreateDistanceMatrixFloat (
            int maxDegreeOfParallelism = 0,
            Action< double > progressCallback = null )
```

Creates a lower triangular distance matrix, where each entry is the path length distance between two leaves in the tree. Entries are in the same order as returned by the GetLeaves method.

Parameters

maxDegreeOfParallelism	Maximum number of threads to use, or -1 to let the runtime decide. If this argument is set to 0 (the default), the value used is 1 for trees with 1500 or fewer leaves, or -1 for larger trees.
progressCallback	A method used to report progress.

Returns

A T:float[ ][ ] jagged array containing the distance matrix.

Definition at line 1445 of file TreeNode.cs.

7.29.4.5 EdgeLengthDistance() [1/2]

```
double PhyloTree.TreeNode.EdgeLengthDistance (
            TreeNode otherTree )
```

Computes the edge-length distance between the current tree and another tree.

Parameters

otherTree	The other tree whose distance to the current tree is computed.

Returns

The edge-length distance between this tree and the *otherTree* .

Definition at line 67 of file TreeNode.Comparisons.cs.

7.29.4.6 EdgeLengthDistance() [2/2]

```
static double PhyloTree.TreeNode.EdgeLengthDistance (
            TreeNode tree1,
            TreeNode tree2 )  [static]
```

Computes the edge-length distance between two trees.

Parameters

tree1	The first tree.
tree2	The second tree.

Returns

The edge-length distance between *tree1* and *tree2* .

Definition at line 78 of file TreeNode.Comparisons.cs.

7.29.4.7 EdgeLengthDistances()

```
static double[,] PhyloTree.TreeNode.EdgeLengthDistances (
            IReadOnlyList< TreeNode > trees,
            TreeComparisonPruningMode pruningMode = TreeComparisonPruningMode.Pairwise,
            int maxThreadCount = -1,
            IProgress< double > progress = null )  [static]
```

Computes a distance matrix containing the edge-length distances between each pair of trees in a list.

Parameters

trees	The list of trees that should be compared.
pruningMode	If this is TreeComparisonPruningMode.Global, all trees are pruned so that they only contain the subset of leaves that are present in all trees. If this is TreeComparisonPruningMode.Pairwise, during each comparisons the two trees are pruned so that they contain the subset of leaves that are in common between both of them.
maxThreadCount	The maximum number of threads to use for parallelised steps.
progress	An IProgress<T> for progress reporting.

Returns

A square `double`[,] matrix containing the requested distances between the trees.

Definition at line 145 of file TreeNode.Comparisons.cs.

7.29.4.8 GetChildrenRecursive()

```
List< TreeNode > PhyloTree.TreeNode.GetChildrenRecursive ( )
```

Recursively get all the nodes that descend from this node.

Returns

A List<T> of TreeNode objects, containing the nodes that descend from this node.

Definition at line 584 of file TreeNode.cs.

7.29.4.9 GetChildrenRecursiveLazy()

```
IEnumerable< TreeNode > PhyloTree.TreeNode.GetChildrenRecursiveLazy ( )
```

Lazily recursively get all the nodes that descend from this node.

Returns

An IEnumerable<T> of TreeNode objects, containing the nodes that descend from this node.

Definition at line 602 of file TreeNode.cs.

7.29.4.10 GetCollessExpectationYHK()

```
static double PhyloTree.TreeNode.GetCollessExpectationYHK (
            int numberOfLeaves )  [static]
```

Computes the expected value of the Colless index under the YHK model.

Parameters

numberOfLeaves	The number of leaves in the tree.

Returns

The expected value of the Colless index for a tree with the specified *numberOfLeaves* .

Proof in DOI: 10.1214/105051606000000547

Definition at line 106 of file TreeNode.ShapeIndices.cs.

7.29.4.11 GetDepth()

```
int PhyloTree.TreeNode.GetDepth ( )
```

Compute the depth of the node (number of branches from this node until the root node).

Returns

The depth of the node.

Definition at line 35 of file TreeNode.ShapeIndices.cs.

7.29.4.12 GetLastCommonAncestor() [1/3]

```
TreeNode PhyloTree.TreeNode.GetLastCommonAncestor (
            IEnumerable< string > monophyleticConstraint )
```

Gets the last common ancestor of all the nodes with the specified names, or `null` if the tree doesn't contain all the named nodes.

Parameters

monophyleticConstraint	The collection of names representing nodes whose last common ancestor is to be determined.

Returns

The last common ancestor of all the nodes with the specified names, or `null` if the tree doesn't contain all the named nodes.

Definition at line 1056 of file TreeNode.cs.

7.29.4.13 GetLastCommonAncestor() [2/3]

```
static TreeNode PhyloTree.TreeNode.GetLastCommonAncestor (
            IEnumerable< TreeNode > monophyleticConstraint )  [static]
```

Gets the last common ancestor of all the specified nodes, or `null` if the tree doesn't contain all the nodes.

Parameters

monophyleticConstraint	The collection of nodes whose last common ancestor is to be determined.

Returns

The last common ancestor of all the specified nodes, or `null` if the tree doesn't contain all the nodes.

Definition at line 1022 of file TreeNode.cs.

7.29.4.14 GetLastCommonAncestor() [3/3]

```
TreeNode PhyloTree.TreeNode.GetLastCommonAncestor (
            params string[] monophyleticConstraint )
```

Gets the last common ancestor of all the nodes with the specified names, or `null` if the tree doesn't contain all the named nodes.

Parameters

monophyleticConstraint	The collection of names representing nodes whose last common ancestor is to be determined.

Returns

The last common ancestor of all the nodes with the specified names, or `null` if the tree doesn't contain all the named nodes.

Definition at line 1046 of file TreeNode.cs.

7.29.4.15 GetLeafNames()

```
List< string > PhyloTree.TreeNode.GetLeafNames ( )
```

Get the names of all the leaves that descend (directly or indirectly) from this node.

Returns

A List<T> of strings, containing the names of the leaves that descend from this node.

Definition at line 639 of file TreeNode.cs.

7.29.4.16 GetLeaves()

```
List< TreeNode > PhyloTree.TreeNode.GetLeaves ( )
```

Get all the leaves that descend (directly or indirectly) from this node.

Returns

A List<T> of TreeNode objects, containing the leaves that descend from this node.

Definition at line 619 of file TreeNode.cs.

7.29.4.17 GetNodeFromId()

```
TreeNode PhyloTree.TreeNode.GetNodeFromId (
            string nodeId )
```

Get the child node with the specified Id.

Parameters

nodeId	The Id of the node to search.

Returns

The TreeNode object with the specified Id, or `null` if no node with such Id exists.

Definition at line 704 of file TreeNode.cs.

7.29.4.18 GetNodeFromName()

```
TreeNode PhyloTree.TreeNode.GetNodeFromName (
            string nodeName )
```

Get the child node with the specified name.

Parameters

nodeName	The name of the node to search.

Returns

The TreeNode object with the specified name, or `null` if no node with such name exists.

Definition at line 680 of file TreeNode.cs.

7.29.4.19 GetNodeNames()

```
List< string > PhyloTree.TreeNode.GetNodeNames ( )
```

Get the names of all the named nodes that descend (directly or indirectly) from this node.

Returns

A List<T> of strings, containing the names of the named nodes that descend from this node.

Definition at line 659 of file TreeNode.cs.

7.29.4.20 GetRootedTree()

```
TreeNode PhyloTree.TreeNode.GetRootedTree (
            TreeNode outgroup,
            double position = 0.5 )
```

Get a version of the tree that is rooted at the specified point of the branch leading to the *outgroup* .

Parameters

outgroup	The outgroup to be used when rooting the tree.
position	The (relative) position on the branch connecting the outgroup to the rest of the tree on which to place the root.

Returns

A TreeNode containing the rooted tree.

Definition at line 445 of file TreeNode.cs.

7.29.4.21 GetRootNode()

```
TreeNode PhyloTree.TreeNode.GetRootNode ( )
```

Get the node of the tree from which all other nodes descend.

Returns

The node of the tree from which all other nodes descend

Definition at line 794 of file TreeNode.cs.

7.29.4.22 GetSplit()

```
List< TreeNode > List< TreeNode > side2 PhyloTree.TreeNode.GetSplit ( )
```

Definition at line 1158 of file TreeNode.cs.

7.29.4.23 GetSplits()

```
IEnumerable<(List< TreeNode > side1, List< TreeNode > side2, double branchLength)> Phylo↩
Tree.TreeNode.GetSplits ( )
```

Gets all the splits in the tree.

Returns

An IEnumerable<T> that enumerates all the splits in the tree.

Definition at line 1198 of file TreeNode.cs.

7.29.4.24 GetUnrootedTree()

```
TreeNode PhyloTree.TreeNode.GetUnrootedTree ( )
```

Get an unrooted version of the tree.

Returns

A TreeNode containing the unrooted tree, having at least 3 children.

Definition at line 398 of file TreeNode.cs.

7.29.4.25 IsClockLike()

```
bool PhyloTree.TreeNode.IsClockLike (
            double tolerance = 0.001 )
```

Determines whether the tree is clock-like (i.e. all tips are contemporaneous) or not.

Parameters

tolerance	The (relative) tolerance when comparing branch lengths.

Returns

A boolean value determining whether the tree is clock-like or not

Definition at line 1000 of file TreeNode.cs.

7.29.4.26 IsLastCommonAncestor()

```
bool PhyloTree.TreeNode.IsLastCommonAncestor (
            IEnumerable< string > monophyleticConstraint )
```

Checks whether this node is the last common ancestor of all the nodes with the specified names.

Parameters

monophyleticConstraint	The collection of names representing nodes whose last common ancestor is to be determined.

Returns

`true` if this node is the last common ancestor of all the nodes with the specified names, `false` otherwise.

Definition at line 1080 of file TreeNode.cs.

7.29.4.27 IsRooted()

```
bool PhyloTree.TreeNode.IsRooted ( )
```

Checks whether the node belongs to a rooted tree.

Returns

`true` if the node belongs to a rooted tree, `false` otherwise.

Definition at line 389 of file TreeNode.cs.

7.29.4.28 LongestDownstreamLength()

```
double PhyloTree.TreeNode.LongestDownstreamLength ( )
```

Get the sum of the branch lengths from this node down to the leaves of the tree. If the tree is not clock-like, the length of the longest path is returned.

Returns

The sum of the branch lengths from this node down to the leaves of the tree. If the tree is not clock-like, the length of the longest path is returned.

Definition at line 746 of file TreeNode.cs.

7.29.4.29 NumberOfCherries()

```
double PhyloTree.TreeNode.NumberOfCherries (
            NullHypothesis model = NullHypothesis.None )
```

Computes the number of cherries in the tree.

Parameters

model	If this is NullHypothesis.None, the raw number of cherries is returned. If this is NullHypothesis.YHK or NullHypothesis.PDA, the number of cherries is normalised with respect to the corresponding null tree model (which makes scores comparable across trees of different sizes).

Returns

The number of cherries in the tree.

Proofs in DOI: 10.1016/S0025-5564(99)00060-7

Definition at line 167 of file TreeNode.ShapeIndices.cs.

7.29.4.30 PathLengthTo()

```
double PhyloTree.TreeNode.PathLengthTo (
            TreeNode otherNode,
            NodeRelationship nodeRelationship = NodeRelationship.Unknown )
```

Get the sum of the branch lengths from this node to the specified node.

Parameters

otherNode	The node that should be reached
nodeRelationship	A value indicating how this node is related to *otherNode* .

Returns

The sum of the branch lengths from this node to the specified node.

Definition at line 836 of file TreeNode.cs.

7.29.4.31 Prune() [1/2]

```
TreeNode PhyloTree.TreeNode.Prune (
            bool leaveParent )
```

Prunes the current node from the tree.

Parameters

leaveParent	This value determines what happens to the parent node of the current node if it only has two children (i.e., the current node and another node). If this is `false`, the parent node is also pruned; if it is `true`, the parent node is left untouched.

Note that the node is pruned in-place; however, the return value of this method should be used, because pruning the node may have caused the root of the tree to move.

Returns

The TreeNode corresponding to the root of the tree after the current node has been pruned.

Definition at line 1223 of file TreeNode.cs.

7.29.4.32 Prune() [2/2]

```
TreeNode PhyloTree.TreeNode.Prune (
            TreeNode nodeToPrune,
            bool leaveParent )
```

Prunes a node from the tree.

Parameters

nodeToPrune	The node that should be pruned.
leaveParent	This value determines what happens to the parent node of the pruned node if it only has two children (i.e., the pruned node and another node). If this is `false`, the parent node is also pruned; if it is `true`, the parent node is left untouched.

Note that the node is pruned in-place; however, the return value of this method should be used, because pruning the node may have caused the root of the tree to move.

Returns

The TreeNode corresponding to the root of the tree after the *nodeToPrune* has been pruned.

Definition at line 1295 of file TreeNode.cs.

7.29.4.33 RobinsonFouldsDistance() [1/2]

```
double PhyloTree.TreeNode.RobinsonFouldsDistance (
            TreeNode otherTree,
            bool weighted )
```

Computes the Robinson-Foulds distance between the current tree and another tree.

Parameters

otherTree	The other tree whose distance to the current tree is computed.
weighted	If this is `true`, the distance is weighted based on the branch lengths; otherwise, it is unweighted.

Returns

The Robinson-Foulds distance between this tree and the *otherTree* .

Definition at line 24 of file TreeNode.Comparisons.cs.

7.29.4.34 RobinsonFouldsDistance() [2/2]

```
static double PhyloTree.TreeNode.RobinsonFouldsDistance (
            TreeNode tree1,
            TreeNode tree2,
            bool weighted )  [static]
```

Computes the Robinson-Foulds distance between two trees.

Parameters

tree1	The first tree.
tree2	The second tree.
weighted	If this is `true`, the distance is weighted based on the branch lengths; otherwise, it is unweighted.

Returns

The Robinson-Foulds distance between *tree1* and *tree2* .

Definition at line 36 of file TreeNode.Comparisons.cs.

7.29.4.35 RobinsonFouldsDistances() [1/2]

```
static double[,] PhyloTree.TreeNode.RobinsonFouldsDistances (
            IReadOnlyList< TreeNode > trees,
            bool weighted,
            TreeComparisonPruningMode pruningMode = TreeComparisonPruningMode.Pairwise,
            int maxThreadCount = -1,
            IProgress< double > progress = null )  [static]
```

Computes a distance matrix containing the Robinson-Foulds distances between each pair of trees in a list.

Parameters

trees	The list of trees that should be compared.

Parameters

weighted	If this is `true`, the distance is weighted based on the branch lengths; otherwise, it is unweighted.
pruningMode	If this is TreeComparisonPruningMode.Global, all trees are pruned so that they only contain the subset of leaves that are present in all trees. If this is TreeComparisonPruningMode.Pairwise, during each comparisons the two trees are pruned so that they contain the subset of leaves that are in common between both of them.
maxThreadCount	The maximum number of threads to use for parallelised steps.
progress	An IProgress<T> for progress reporting.

Returns

A square `double`[,] matrix containing the requested distances between the trees.

Definition at line 121 of file TreeNode.Comparisons.cs.

7.29.4.36 RobinsonFouldsDistances() [2/2]

```
static void PhyloTree.TreeNode.RobinsonFouldsDistances (
            IReadOnlyList< TreeNode > trees,
            out double RFDistances[,],
            out double weightedRFDistances[,],
            TreeComparisonPruningMode pruningMode = TreeComparisonPruningMode.Pairwise,
            int maxThreadCount = -1,
            IProgress< double > progress = null )  [static]
```

Computes two distance matrices containing the unweighted and weighted Robinson-Foulds distances between each pair of trees in a list. Much faster than computing the two distance matrices separately.

Parameters

trees	The list of trees that should be compared.
RFDistances	When this method returns, this variable will contain the computed Robinson-Foulds distances between the trees.
weightedRFDistances	When this method returns, this variable will contain the computed weighted Robinson-Foulds distances between the trees.
pruningMode	If this is TreeComparisonPruningMode.Global, all trees are pruned so that they only contain the subset of leaves that are present in all trees. If this is TreeComparisonPruningMode.Pairwise, during each comparisons the two trees are pruned so that they contain the subset of leaves that are in common between both of them.
maxThreadCount	The maximum number of threads to use for parallelised steps.
progress	An IProgress<T> for progress reporting.

Returns

A square `double`[,] matrix containing the requested distances between the trees.

Definition at line 164 of file TreeNode.Comparisons.cs.

7.29.4.37 SackinIndex()

```
double PhyloTree.TreeNode.SackinIndex (
            NullHypothesis model = NullHypothesis.None )
```

Computes the Sackin index of the tree (sum of the leaf depths).

Parameters

model	If this is NullHypothesis.None, the raw Sackin index is returned. If this is NullHypothesis.YHK or NullHypothesis.PDA, the Sackin index is normalised with respect to the corresponding null tree model (which makes scores comparable across trees of different sizes).

Returns

The Sackin index of the tree, either as a raw value, or normalised according to the selected null tree model.

Definition at line 56 of file TreeNode.ShapeIndices.cs.

7.29.4.38 ShortestDownstreamLength()

```
double PhyloTree.TreeNode.ShortestDownstreamLength ( )
```

Get the sum of the branch lengths from this node down to the leaves of the tree. If the tree is not clock-like, the length of the shortest path is returned.

Returns

The sum of the branch lengths from this node down to the leaves of the tree. If the tree is not clock-like, the length of the shortest path is returned.

Definition at line 770 of file TreeNode.cs.

7.29.4.39 SortNodes()

```
void PhyloTree.TreeNode.SortNodes (
            bool descending )
```

Sort (in place) the nodes in the tree in an aesthetically pleasing way.

Parameters

descending	The way the nodes should be sorted.

Definition at line 921 of file TreeNode.cs.

7.29.4.40 ToString()

```
override string PhyloTree.TreeNode.ToString ( )
```

Convert the tree to a Newick string.

Returns

Definition at line 990 of file TreeNode.cs.

7.29.4.41 TotalLength()

```
double PhyloTree.TreeNode.TotalLength ( )
```

Get the sum of the branch lengths of this node and all its descendants.

Returns

The sum of the branch lengths of this node and all its descendants.

Definition at line 906 of file TreeNode.cs.

7.29.4.42 TreeDistances()

```
static void PhyloTree.TreeNode.TreeDistances (
            IReadOnlyList< TreeNode > trees,
            out double RFDistances[,],
            out double weightedRFDistances[,],
            out double ELDistances[,],
            TreeComparisonPruningMode pruningMode = TreeComparisonPruningMode.Pairwise,
            int maxThreadCount = -1,
            IProgress< double > progress = null )  [static]
```

Computes three distance matrices containing the unweighted and weighted Robinson-Foulds distances and the edge-length distances between each pair of trees in a list. Much faster than computing the three distance matrices separately.

Parameters

trees	The list of trees that should be compared.
RFDistances	When this method returns, this variable will contain the computed Robinson-Foulds distances between the trees.
weightedRFDistances	When this method returns, this variable will contain the computed weighted Robinson-Foulds distances between the trees.
ELDistances	When this method returns, this variable will contain the computed edge-length distances between the trees.

Returns

A square `double`[,] matrix containing the requested distances between the trees.

Definition at line 183 of file TreeNode.Comparisons.cs.

7.29.4.43 UpstreamLength()

```
double PhyloTree.TreeNode.UpstreamLength ( )
```

Get the sum of the branch lengths from this node up to the root.

Returns

The sum of the branch lengths from this node up to the root.

Definition at line 727 of file TreeNode.cs.

7.29.5 Member Data Documentation

7.29.5.1 side1

```
List<TreeNode> PhyloTree.TreeNode.side1
```

Gets the split corresponding to the branch underlying this node. If this is an internal node, `side1` will contain all the leaves in the tree except those descending from this node, and `side2` will contain all the leaves descending from this node. If this is the root `side1` will be empty and `side2` will contain all the leaves in the tree. If the tree is rooted (the root node has exactly 2 children), `side1` will contain in all cases an additional `null` element.

Returns

The leaves on the two sides of the split.

Definition at line 1158 of file TreeNode.cs.

7.29.6 Property Documentation

7.29.6.1 Attributes

```
AttributeDictionary PhyloTree.TreeNode.Attributes = new AttributeDictionary()  [get]
```

The attributes of this node. Attributes Name, Length and Support are always included. See the respective properties for default values.

Definition at line 321 of file TreeNode.cs.

7.29.6.2 Children

```
List<TreeNode> PhyloTree.TreeNode.Children  [get]
```

The child nodes of this node. This will be empty (but initialised) for leaf nodes.

Definition at line 316 of file TreeNode.cs.

7.29.6.3 Id

```
string PhyloTree.TreeNode.Id  [get]
```

A univocal identifier for the node.

Definition at line 372 of file TreeNode.cs.

7.29.6.4 Length

```
double PhyloTree.TreeNode.Length  [get], [set]
```

The length of the branch leading to this node. This is `double.NaN` for branches whose length is not specified (e.g. the root node).

Definition at line 326 of file TreeNode.cs.

7.29.6.5 Name

```
string PhyloTree.TreeNode.Name  [get], [set]
```

The name of this node (e.g. the species name for leaf nodes). Default is `""`.

Definition at line 356 of file TreeNode.cs.

7.29.6.6 Parent

```
TreeNode PhyloTree.TreeNode.Parent  [get], [set]
```

The parent node of this node. This will be `null` for the root node.

Definition at line 311 of file TreeNode.cs.

7.29.6.7 Support

```
double PhyloTree.TreeNode.Support  [get], [set]
```

The support value of this node. This is `double.NaN` for branches whose support is not specified. The interpretation of the support value depends on how the tree was built.

Definition at line 341 of file TreeNode.cs.

The documentation for this class was generated from the following files:

- TreeNode.Comparisons.cs
- TreeNode.cs
- TreeNode.ShapeIndices.cs

7.30 PhyloTree.Extensions.TypeExtensions Class Reference

Useful extension methods

Static Public Member Functions

- static bool ContainsAll< T > (this IEnumerable< T > haystack, IEnumerable< T > needle)

  Determines whether haystack contains all of the elements in needle .
- static double Median (this IEnumerable< double > array)

  Compute the median of a list of values.
- static bool ContainsAny< T > (this IEnumerable< T > haystack, IEnumerable< T > needle)

  Determines whether haystack contains at least one of the elements in needle .
- static IEnumerable< T > Intersection< T > (this IEnumerable< T > set1, IEnumerable< T > set2)

  Computes the intersection between two sets.
- static TreeNode GetConsensus (this IEnumerable< TreeNode > trees, bool rooted, bool clockLike, double threshold, bool useMedian, Action< double > progressAction=null, bool useParallelOptimisation=false)

  Constructs a consensus tree.
- static char NextToken (this TextReader reader, ref bool escaping, out bool escaped, ref bool openQuotes, ref bool openApostrophe, out bool eof)

  Reads the next non-whitespace character, taking into account quoting and escaping.
- static string NextWord (this TextReader reader, out bool eof)

  Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.
- static string NextWord (this TextReader reader, out bool eof, out string headingTrivia)

  Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.

7.30.1 Detailed Description

Useful extension methods

Definition at line 19 of file Extensions.cs.

7.30.2 Member Function Documentation

7.30.2.1 ContainsAll< T >()

```
static bool PhyloTree.Extensions.TypeExtensions.ContainsAll< T > (
            this IEnumerable< T > haystack,
            IEnumerable< T > needle )  [static]
```

Determines whether *haystack* contains all of the elements in *needle* .

Template Parameters

T	The type of the elements in the collections.

Parameters

haystack	The collection in which to search.
needle	The items to be searched.

Returns

`true` if haystack contains all of the elements that are in needle or needle is empty.

Definition at line 28 of file Extensions.cs.

7.30.2.2 ContainsAny< T >()

```
static bool PhyloTree.Extensions.TypeExtensions.ContainsAny< T > (
            this IEnumerable< T > haystack,
            IEnumerable< T > needle )  [static]
```

Determines whether *haystack* contains at least one of the elements in *needle* .

Template Parameters

T	The type of the elements in the collections.

Parameters

haystack	The collection in which to search.
needle	The items to be searched.

Returns

`true` if haystack contains at least one of the elements that are in needle. Returns `false` if needle is empty.

Definition at line 69 of file Extensions.cs.

7.30.2.3 GetConsensus()

```
static TreeNode PhyloTree.Extensions.TypeExtensions.GetConsensus (
            this IEnumerable< TreeNode > trees,
            bool rooted,
            bool clockLike,
            double threshold,
            bool useMedian,
            Action< double > progressAction = null,
            bool useParallelOptimisation = false )  [static]
```

Constructs a consensus tree.

Parameters

trees	The collection of trees whose consensus is to be computed.
rooted	Whether the consensus tree should be rooted or not.
clockLike	Whether the trees are to be treated as clock-like trees or not. This has an effect on how the branch lengths of the consensus tree are computed.
threshold	The (inclusive) threshold for splits to be included in the consensus tree. Use `0` to get all compatible splits, `0.5` for a majority-rule consensus or `1` for a strict consensus.
useMedian	If this is `true`, the lengths of the branches in the tree will be computed based on the median length/age of the splits used to build the tree. Otherwise, the mean will be used.
progressAction	An Action that will be invoked as the trees are processed.
useParallelOptimisation	If this is `true`, parts of the consensus computation will be parallelised. This will, however, increase the number of computations that need to be performed. The advantages will differ based on the number of trees, how discordant they are, and the characteristics of the processor.

Returns

A rooted consensus tree.

Definition at line 115 of file Extensions.cs.

7.30.2.4 Intersection< T >()

```
static IEnumerable< T > PhyloTree.Extensions.TypeExtensions.Intersection< T > (
            this IEnumerable< T > set1,
            IEnumerable< T > set2 )  [static]
```

Computes the intersection between two sets.

Template Parameters

T	The type of the elements in the collections.

Parameters

set1	The first set.
set2	The second set.

Returns

The intersection between the two sets.

Definition at line 90 of file Extensions.cs.

7.30.2.5 Median()

```
static double PhyloTree.Extensions.TypeExtensions.Median (
            this IEnumerable< double > array )  [static]
```

Compute the median of a list of values.

Parameters

array	The list of values whose median is to be computed.

Returns

The median of the list of values.

Definition at line 47 of file Extensions.cs.

7.30.2.6 NextToken()

```
static char PhyloTree.Extensions.TypeExtensions.NextToken (
            this TextReader reader,
```

```
            ref bool escaping,
            out bool escaped,
            ref bool openQuotes,
            ref bool openApostrophe,
            out bool eof )  [static]
```

Reads the next non-whitespace character, taking into account quoting and escaping.

Parameters

reader	The TextReader to read from.
escaping	A bool indicating whether the next character will be escaped.
escaped	A bool indicating whether the current character will be escaped.
openQuotes	A bool indicating whether double quotes have been opened.
openApostrophe	A bool indicating whether single quotes have been opened.
eof	A bool indicating whether we have arrived at the end of the file.

Returns

The next non-whitespace character.

Definition at line 333 of file Extensions.cs.

7.30.2.7 NextWord() [1/2]

```
static string PhyloTree.Extensions.TypeExtensions.NextWord (
            this TextReader reader,
            out bool eof )  [static]
```

Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.

Parameters

reader	The TextReader to read from.
eof	A bool indicating whether we have arrived at the end of the file.

Returns

The next word.

Definition at line 421 of file Extensions.cs.

7.30.2.8 NextWord() [2/2]

```
static string PhyloTree.Extensions.TypeExtensions.NextWord (
            this TextReader reader,
```

```
            out bool eof,
            out string headingTrivia )  [static]
```

Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.

Parameters

reader	The TextReader to read from.
eof	A bool indicating whether we have arrived at the end of the file.
headingTrivia	A string containing any whitespace that was discarding before the start of the word.

Returns

The next word.

Definition at line 477 of file Extensions.cs.

The documentation for this class was generated from the following file:

- Extensions.cs

7.31 PhyloTree.TreeBuilding.UPGMA Class Reference

Contains methods to compute UPGMA trees.

Static Public Member Functions

- static TreeNode BuildTree (Dictionary< string, string > alignment, EvolutionModel evolutionModel=EvolutionModel.Kimura, int bootstrapReplicates=0, AlignmentType alignmentType=AlignmentType.Autodetect, TreeNode constraint=null, int numCores=-1, Action< double > progressCallback=null)

  Builds a UPGMA tree using data from a sequence alignment. This method first computes a distance matrix from the sequence alignment, and then uses the distance matrix to compute the tree.
- static TreeNode BuildTree (float[][] distanceMatrix, IReadOnlyList< string > sequenceNames, TreeNode constraint=null, bool copyMatrix=true, int numCores=-1, Action< double > progressCallback=null)

  Builds a UPGMA tree using data from a distance matrix.

7.31.1 Detailed Description

Contains methods to compute UPGMA trees.

Definition at line 13 of file UPGMA.cs.

7.31.2 Member Function Documentation

7.31.2.1 BuildTree() [1/2]

```
static TreeNode PhyloTree.TreeBuilding.UPGMA.BuildTree (
            Dictionary< string, string > alignment,
            EvolutionModel evolutionModel = EvolutionModel.Kimura,
            int bootstrapReplicates = 0,
            AlignmentType alignmentType = AlignmentType.Autodetect,
            TreeNode constraint = null,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Builds a UPGMA tree using data from a sequence alignment. This method first computes a distance matrix from the sequence alignment, and then uses the distance matrix to compute the tree.

Parameters

alignment	The sequence alignment.
evolutionModel	The evolutionary model to use when computing the distance matrix.
bootstrapReplicates	The number of bootstrap replicates to perform.
alignmentType	The type of sequence alignment (DNA, protein, or autodetect).
constraint	An optional tree to constrain the search. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating.
numCores	Maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

The UPGMA tree built from the supplied *alignment* .

Definition at line 26 of file UPGMA.cs.

7.31.2.2 BuildTree() [2/2]

```
static TreeNode PhyloTree.TreeBuilding.UPGMA.BuildTree (
            float distanceMatrix[][],
            IReadOnlyList< string > sequenceNames,
            TreeNode constraint = null,
            bool copyMatrix = true,
            int numCores = -1,
            Action< double > progressCallback = null )  [static]
```

Builds a UPGMA tree using data from a distance matrix.

Parameters

distanceMatrix	The distance matrix containg distances between the taxa. This can be a lower triangular matrix or a full matrix; values above the diagonal will not be used.
sequenceNames	The names of the taxa. The indices of this list should correspond to the rows and columns of the *distanceMatrix* .
constraint	An optional tree to constrain the search. The tree produced by this method will be compatible with this tree. The constraint tree can be multifurcating.
copyMatrix	If this is `true`, the matrix is copied before using it to compute the tree. If this is `false`, the matrix is not copied. Copying the matrix increases the memory used by the method, but note that if the matrix is not copied, it will be modified in-place!
numCores	Maximum number of threads to use, or -1 to let the runtime decide.
progressCallback	A method used to report progress.

Returns

The UPGMA tree built from the supplied *distanceMatrix* .

Definition at line 127 of file UPGMA.cs.

The documentation for this class was generated from the following file:

- TreeBuilding/UPGMA.cs

Chapter 8

File Documentation

8.1 AttributeDictionary.cs

```
00001 using System;
00002 using System.Collections;
00003 using System.Collections.Generic;
00004 using System.Diagnostics.Contracts;
00005 using System.Linq;
00006
00007 namespace PhyloTree
00008 {
00009 /// <summary>
00010 /// Represents the attributes of a node.  Attributes <see cref="Name"/>, <see cref="Length"/> and <see
      cref="Support"/> are always included.  See the respective properties for default values.
00011 /// </summary>
00012     [Serializable]
00013     public class AttributeDictionary :  IDictionary<string, object>
00014     {
00015         private readonly Dictionary<string, object> InternalStorage;
00016
00017         private string _name;
00018
00019 /// <summary>
00020 /// The name of this node (e.g.  the species name for leaf nodes).  Default is <c>""</c>.  Getting the
      value of this property does not require a dictionary lookup.
00021 /// </summary>
00022         public string Name
00023         {
00024             get
00025             {
00026                 return _name;
00027             }
00028             set
00029             {
00030                 _name = value;
00031                 InternalStorage["Name"] = value;
00032             }
00033         }
00034
00035         private double _length;
00036
00037 /// <summary>
00038 /// The length of the branch leading to this node.  This is <c>double.NaN</c> for branches whose
      length is not specified (e.g.  the root node).  Getting the value of this property does not require a
      dictionary lookup.
00039 /// </summary>
00040         public double Length
00041         {
00042             get
00043             {
00044                 return _length;
00045             }
00046             set
00047             {
00048                 _length = value;
00049                 InternalStorage["Length"] = value;
00050             }
00051         }
00052
00053         private double _support;
00054
```

```
00055 /// <summary>
00056 /// The support value of this node.  This is <c>double.NaN</c> for branches whose support is not
      specified.  The interpretation of the support value depends on how the tree was built.  Getting the
      value of this property does not require a dictionary lookup.
00057 /// </summary>
00058         public double Support
00059         {
00060             get
00061             {
00062                 return _support;
00063             }
00064             set
00065             {
00066                 _support = value;
00067                 InternalStorage["Support"] = value;
00068             }
00069         }
00070 
00071 /// <summary>
00072 /// Gets or sets the value of the attribute with the specified <paramref name="name"/>.  Getting the
      value of attributes <c>"Name"</c>, <c>"Length"</c> and <c>"Support"</c> does not require a dictionary
      lookup.
00073 /// </summary>
00074 /// <param name="name">The name of the attribute to get/set.</param>
00075 /// <returns>The value of the attribute, boxed into an <c>object</c>.</returns>
00076         public object this[string name]
00077         {
00078             get
00079             {
00080                 Contract.Requires(name != null);
00081                 if (name.Equals("Name", StringComparison.OrdinalIgnoreCase))
00082                 {
00083                     return _name;
00084                 }
00085                 else if (name.Equals("Length", StringComparison.OrdinalIgnoreCase))
00086                 {
00087                     return _length;
00088                 }
00089                 else if (name.Equals("Support", StringComparison.OrdinalIgnoreCase))
00090                 {
00091                     return _support;
00092                 }
00093                 else
00094                 {
00095                     return InternalStorage[name];
00096                 }
00097             }
00098 
00099             set
00100             {
00101                 Contract.Requires(name != null);
00102                 InternalStorage[name] = value;
00103 
00104                 if (name.Equals("Name", StringComparison.OrdinalIgnoreCase))
00105                 {
00106                     _name = Convert.ToString(value,
      System.Globalization.CultureInfo.InvariantCulture);
00107                 }
00108                 else if (name.Equals("Length", StringComparison.OrdinalIgnoreCase))
00109                 {
00110                     _length = Convert.ToDouble(value,
      System.Globalization.CultureInfo.InvariantCulture);
00111                 }
00112                 else if (name.Equals("Support", StringComparison.OrdinalIgnoreCase))
00113                 {
00114                     _support = Convert.ToDouble(value,
      System.Globalization.CultureInfo.InvariantCulture);
00115                 }
00116             }
00117         }
00118 
00119 /// <summary>
00120 /// Gets a collection containing the names of the attributes in the <see cref="AttributeDictionary"/>.
00121 /// </summary>
00122         public ICollection<string> Keys => InternalStorage.Keys;
00123 
00124 /// <summary>
00125 /// Gets a collection containing the values of the attributes in the <see
      cref="AttributeDictionary"/>.
00126 /// </summary>
00127         public ICollection<object> Values => InternalStorage.Values;
00128 
00129 /// <summary>
00130 /// Gets the number of attributes contained in the <see cref="AttributeDictionary"/>.
00131 /// </summary>
00132         public int Count => InternalStorage.Count;
00133 
```

```
00134 /// <summary>
00135 /// Determine whether the <see cref="AttributeDictionary"/> is read-only.  This is always <c>false</c>
      in the current implementation.
00136 /// </summary>
00137         public bool IsReadOnly => false;
00138 
00139 /// <summary>
00140 /// Adds an attribute with the specified <paramref name="name"/> and <paramref name="value"/> to the
      <see cref="AttributeDictionary"/>.  Throws an exception if the <see cref="AttributeDictionary"/>
      already contains an attribute with the same <paramref name="name"/>.
00141 /// </summary>
00142 /// <param name="name">The name of the attribute.</param>
00143 /// <param name="value">The value of the attribute.</param>
00144         public void Add(string name, object value)
00145         {
00146             InternalStorage.Add(name, value);
00147         }
00148 
00149 /// <summary>
00150 /// Adds an attribute with the specified name and value to the <see cref="AttributeDictionary"/>.
      Throws an exception if the <see cref="AttributeDictionary"/> already contains an attribute with the
      same name.
00151 /// </summary>
00152 /// <param name="item">The item to be added to the dictionary.</param>
00153         public void Add(KeyValuePair<string, object> item)
00154         {
00155             InternalStorage.Add(item.Key, item.Value);
00156         }
00157 
00158 /// <summary>
00159 /// Removes all attributes from the dictionary, except the <c>"Name"</c>, <c>"Length"</c> and
      <c>"Support"</c> attributes.
00160 /// </summary>
00161         public void Clear()
00162         {
00163             InternalStorage.Clear();
00164 
00165             _name = "";
00166             _length = double.NaN;
00167             _support = double.NaN;
00168 
00169             InternalStorage.Add("Name", _name);
00170             InternalStorage.Add("Length", _length);
00171             InternalStorage.Add("Support", _support);
00172         }
00173 
00174 /// <summary>
00175 /// Determines whether the <see cref="AttributeDictionary"/> contains the specified <paramref
      name="item"/>.
00176 /// </summary>
00177 /// <param name="item">The item to locate in the <see cref="AttributeDictionary"/></param>
00178 /// <returns><c>true</c> if the <see cref="AttributeDictionary"/> contains the specified <paramref
      name="item"/>, <c>false</c> otherwise.</returns>
00179         public bool Contains(KeyValuePair<string, object> item)
00180         {
00181             return InternalStorage.Contains(item);
00182         }
00183 
00184 /// <summary>
00185 /// Determines whether the <see cref="AttributeDictionary"/> contains an attribute with the specified
      name <paramref name="name"/>.
00186 /// </summary>
00187 /// <param name="name">The name of the attribute to locate.</param>
00188 /// <returns><c>true</c> if the <see cref="AttributeDictionary"/> contains an attribute with the
      specified <paramref name="name"/>, <c>false</c> otherwise.</returns>
00189         public bool ContainsKey(string name)
00190         {
00191             return InternalStorage.ContainsKey(name);
00192         }
00193 
00194 /// <summary>
00195 /// Copies the elements of the <see cref="AttributeDictionary"/> to an array, starting at a specific
      array index.
00196 /// </summary>
00197 /// <param name="array">The array to which the elements will be copied.</param>
00198 /// <param name="arrayIndex">The index at which to start copying.</param>
00199         public void CopyTo(KeyValuePair<string, object>[] array, int arrayIndex)
00200         {
00201             Contract.Requires(array != null);
00202 
00203             foreach (KeyValuePair<string, object> item in InternalStorage)
00204             {
00205                 array[arrayIndex] = item;
00206                 arrayIndex++;
00207             }
00208         }
00209 
```

```
00210 /// <summary>
00211 /// Returns an enumerator that iterates through the <see cref="AttributeDictionary"/>.
00212 /// </summary>
00213 /// <returns>An enumerator that iterates through the <see cref="AttributeDictionary"/>.</returns>
00214         public IEnumerator<KeyValuePair<string, object» GetEnumerator()
00215         {
00216             return InternalStorage.GetEnumerator();
00217         }
00218 
00219 /// <summary>
00220 /// Removes the attribute with the specified name from the <see cref="AttributeDictionary"/>.
      Attributes <c>"Name"</c>, <c>"Length"</c> and <c>"Support"</c> cannot be removed.
00221 /// </summary>
00222 /// <param name="name">The name of the attribute to remove.</param>
00223 /// <returns>A <c>bool</c> indicating whether the attribute was succesfully removed.</returns>
00224         public bool Remove(string name)
00225         {
00226             if (name == null || !name.Equals("Name", StringComparison.OrdinalIgnoreCase) &&
      !name.Equals("Length", StringComparison.OrdinalIgnoreCase) && !name.Equals("Support",
      StringComparison.OrdinalIgnoreCase))
00227             {
00228                 return InternalStorage.Remove(name);
00229             }
00230             else
00231             {
00232                 return false;
00233             }
00234         }
00235 
00236 /// <summary>
00237 /// Removes the attribute with the specified name from the <see cref="AttributeDictionary"/>.
      Attributes <c>"Name"</c>, <c>"Length"</c> and <c>"Support"</c> cannot be removed.
00238 /// </summary>
00239 /// <param name="item">The attribute to remove (only the name will be used).</param>
00240 /// <returns>A <c>bool</c> indicating whether the attribute was succesfully removed.</returns>
00241         public bool Remove(KeyValuePair<string, object> item)
00242         {
00243             if (!item.Key.Equals("Name", StringComparison.OrdinalIgnoreCase) &&
      !item.Key.Equals("Length", StringComparison.OrdinalIgnoreCase) && !item.Key.Equals("Support",
      StringComparison.OrdinalIgnoreCase))
00244             {
00245                 return InternalStorage.Remove(item.Key);
00246             }
00247             else
00248             {
00249                 return false;
00250             }
00251         }
00252 
00253 /// <summary>
00254 /// Gets the value of the attribute with the specified <paramref name="name"/>.  Getting the value of
      attributes <c>"Name"</c>, <c>"Length"</c> and <c>"Support"</c> does not require a dictionary lookup.
00255 /// </summary>
00256 /// <param name="name">The name of the attribute to get.</param>
00257 /// <param name="value">When this method returns, contains the value of the attribute with the
      specified <paramref name="name"/>, if this is found in the <see cref="AttributeDictionary"/>, or
      <c>null</c> otherwise.</param>
00258 /// <returns>A <c>bool</c> indicating whether an attribute with the specified <paramref name="name"/>
      was found in the <see cref="AttributeDictionary"/>.</returns>
00259         public bool TryGetValue(string name, out object value)
00260         {
00261             if (name?.Equals("Name", StringComparison.OrdinalIgnoreCase) == true)
00262             {
00263                 value = _name;
00264                 return true;
00265             }
00266             else if (name?.Equals("Length", StringComparison.OrdinalIgnoreCase) == true)
00267             {
00268                 value = _length;
00269                 return true;
00270             }
00271             else if (name?.Equals("Support", StringComparison.OrdinalIgnoreCase) == true)
00272             {
00273                 value = _support;
00274                 return true;
00275             }
00276             else
00277             {
00278                 return InternalStorage.TryGetValue(name, out value);
00279             }
00280         }
00281 
00282 /// <summary>
00283 /// Returns an enumerator that iterates through the <see cref="AttributeDictionary"/>.
00284 /// </summary>
00285 /// <returns>An enumerator that iterates through the <see cref="AttributeDictionary"/>.</returns>
00286         IEnumerator IEnumerable.GetEnumerator()
```

```
00287         {
00288             return InternalStorage.GetEnumerator();
00289         }
00290 
00291 /// <summary>
00292 /// Constructs an <see cref="AttributeDictionary"/> containing only the <c>"Name"</c>, <c>"Length"</c>
      and <c>"Support"</c> attributes.
00293 /// </summary>
00294         public AttributeDictionary()
00295         {
00296             this._name = "";
00297             this._length = double.NaN;
00298             this._support = double.NaN;
00299             this.InternalStorage = new Dictionary<string, object>(StringComparer.OrdinalIgnoreCase) {
      { "Name", _name }, { "Length", _length }, { "Support", _support } };
00300         }
00301     }
00302 }
```

8.2 Binary.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.IO;
00005 using System.Linq;
00006 using System.Text;
00007 
00008 /// <summary>
00009 /// Contains classes and methods to read and write phylogenetic trees in multiple formats
00010 /// </summary>
00011 namespace PhyloTree.Formats
00012 {
00013 /// <summary>
00014 /// Contains methods to read and write tree files in binary format.
00015 /// </summary>
00016     public static class BinaryTree
00017     {
00018 /// <summary>
00019 /// Determines whether the tree file stream has a valid trailer.
00020 /// </summary>
00021 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.  Its <see
      cref="Stream.CanSeek"/> must be <c>true</c>.  It does not have to be a <see
      cref="FileStream"/>.</param>
00022 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00023 /// <returns><c>true</c> if the <paramref name="inputStream"/> has a valid trailer, <c>false</c>
      otherwise.</returns>
00024         public static bool HasValidTrailer(Stream inputStream, bool keepOpen = false)
00025         {
00026             Contract.Requires(inputStream != null);
00027 
00028             BinaryReader reader = new BinaryReader(inputStream, Encoding.UTF8, true);
00029 
00030             try
00031             {
00032                 long position = inputStream.Position;
00033                 inputStream.Seek(-4, SeekOrigin.End);
00034 
00035                 if (reader.ReadByte() != (byte)'E' || reader.ReadByte() != (byte)'N' ||
      reader.ReadByte() != (byte)'D' || reader.ReadByte() != (byte)255)
00036                 {
00037                     inputStream.Position = position;
00038                     return false;
00039                 }
00040 
00041                 inputStream.Position = position;
00042                 return true;
00043             }
00044             finally
00045             {
00046                 reader.Dispose();
00047 
00048                 if (!keepOpen)
00049                 {
00050                     inputStream.Dispose();
00051                 }
00052             }
00053         }
00054 
00055 /// <summary>
00056 /// Determines whether the tree file stream is valid (i.e.  it has a valid header).
00057 /// </summary>
```

```
00058 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.  Its <see
      cref="Stream.CanSeek"/> must be <c>true</c>.  It does not have to be a <see
      cref="FileStream"/>.</param>
00059 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00060 /// <returns><c>true</c> if the <paramref name="inputStream"/> has a valid header, <c>false</c>
      otherwise.</returns>
00061         public static bool IsValidStream(Stream inputStream, bool keepOpen = false)
00062         {
00063             Contract.Requires(inputStream != null);
00064 
00065             BinaryReader reader = new BinaryReader(inputStream, Encoding.UTF8, true);
00066 
00067             try
00068             {
00069                 long position = inputStream.Position;
00070 
00071                 if (reader.ReadByte() != (byte)'#' || reader.ReadByte() != (byte)'T' ||
      reader.ReadByte() != (byte)'R' || reader.ReadByte() != (byte)'E')
00072                 {
00073                     inputStream.Position = position;
00074                     return false;
00075                 }
00076 
00077                 byte headerByte = reader.ReadByte();
00078 
00079                 if ((headerByte & 0b11111100) != 0)
00080                 {
00081                     inputStream.Position = position;
00082                     return false;
00083                 }
00084 
00085                 return true;
00086             }
00087             finally
00088             {
00089                 reader.Dispose();
00090 
00091                 if (!keepOpen)
00092                 {
00093                     inputStream.Dispose();
00094                 }
00095             }
00096         }
00097 
00098 /// <summary>
00099 /// Reads the metadata from a file containing trees in binary format.
00100 /// </summary>
00101 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.  Its <see
      cref="Stream.CanSeek"/> must be <c>true</c>.  It does not have to be a <see
      cref="FileStream"/>.</param>
00102 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00103 /// <param name="reader">A <see cref="BinaryReader"/> to read from the <paramref name="inputStream"/>.
      If this is <c>null</c>, a new <see cref="BinaryReader"/> will be initialised and disposed within this
      method.</param>
00104 /// <param name="progressAction">An <see cref="Action"/> that may be invoked while parsing the tree
      file, with an argument ranging from 0 to 1 describing the progress made in reading the file
      (determined by the position in the stream).</param>
00105 /// <returns>A <see cref="BinaryTreeMetadata"/> object containing metadata information about the tree
      file.</returns>
00106         [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1031")]
00107         [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA2000")]
00108         public static BinaryTreeMetadata ParseMetadata(Stream inputStream, bool keepOpen = false,
      BinaryReader reader = null, Action<double> progressAction = null)
00109         {
00110             Contract.Requires(inputStream != null);
00111 
00112             bool wasExternalReader = true;
00113 
00114             try
00115             {
00116                 if (reader == null)
00117                 {
00118                     wasExternalReader = false;
00119                     reader = new BinaryReader(inputStream, Encoding.UTF8, true);
00120                 }
00121 
00122                 BinaryTreeMetadata tbr = new BinaryTreeMetadata();
00123 
00124                 if (reader.ReadByte() != (byte)'#' || reader.ReadByte() != (byte)'T' ||
      reader.ReadByte() != (byte)'R' || reader.ReadByte() != (byte)'E')
00125                 {
00126                     throw new FormatException("Invalid file header!");
00127                 }
00128 
00129                 byte headerByte = reader.ReadByte();
```

```
00130
00131                 if ((headerByte & 0b11111100) != 0)
00132                 {
00133                     throw new FormatException("Invalid file header!");
00134                 }
00135
00136                 bool globalNames = (headerByte & 0b1) != 0;
00137                 bool globalAttributes = (headerByte & 0b10) != 0;
00138
00139                 tbr.GlobalNames = globalNames;
00140
00141                 inputStream.Seek(-4, SeekOrigin.End);
00142
00143                 bool validTrailer = true;
00144
00145                 if (reader.ReadByte() != (byte)'E' || reader.ReadByte() != (byte)'N' ||
      reader.ReadByte() != (byte)'D' || reader.ReadByte() != (byte)255)
00146                 {
00147                     validTrailer = false;
00148                 }
00149
00150                 if (validTrailer)
00151                 {
00152                     inputStream.Seek(-12, SeekOrigin.End);
00153                     long labelAddress = reader.ReadInt64();
00154
00155                     IEnumerable<long> getEnumerable()
00156                     {
00157                         inputStream.Seek(labelAddress, SeekOrigin.Begin);
00158                         int numOfTrees = reader.ReadInt();
00159
00160                         for (int i = 0; i < numOfTrees; i++)
00161                         {
00162                             yield return reader.ReadInt64();
00163                         }
00164                     };
00165
00166                     tbr.TreeAddresses = getEnumerable();
00167                 }
00168
00169                 inputStream.Seek(5, SeekOrigin.Begin);
00170
00171                 string[] allNames = null;
00172
00173                 if (globalNames)
00174                 {
00175                     allNames = new string[reader.ReadInt()];
00176                     for (int i = 0; i < allNames.Length; i++)
00177                     {
00178                         allNames[i] = reader.ReadMyString();
00179                     }
00180                     tbr.Names = allNames;
00181                 }
00182
00183                 Attribute[] allAttributes = null;
00184
00185                 if (globalAttributes)
00186                 {
00187                     allAttributes = new Attribute[reader.ReadInt()];
00188
00189                     for (int i = 0; i < allAttributes.Length; i++)
00190                     {
00191                         allAttributes[i] = new Attribute(reader.ReadMyString(), reader.ReadInt() ==
      2);
00192                     }
00193                     tbr.AllAttributes = allAttributes;
00194                 }
00195
00196                 if (!validTrailer)
00197                 {
00198                     IEnumerable<long> getEnumerable()
00199                     {
00200                         bool error = false;
00201
00202                         while (!error)
00203                         {
00204                             long position = inputStream.Position;
00205
00206                             TreeNode tree = null;
00207                             try
00208                             {
00209                                 tree = reader.ReadTree(globalNames, allNames, allAttributes);
00210                             }
00211                             catch
00212                             {
00213                                 error = true;
00214                             }
```

```
00215
00216                           if (!error)
00217                           {
00218                               yield return position;
00219                               double progress = Math.Max(0, Math.Min(1, (double)position /
      inputStream.Length));
00220                               progressAction?.Invoke(progress);
00221                           }
00222                       }
00223                   }
00224
00225                   tbr.TreeAddresses = getEnumerable();
00226               }
00227
00228               return tbr;
00229           }
00230           finally
00231           {
00232               if (!wasExternalReader || !keepOpen)
00233               {
00234                   reader.Dispose();
00235               }
00236               if (!keepOpen)
00237               {
00238                   inputStream.Dispose();
00239               }
00240           }
00241       }
00242
00243
00244 /// <summary>
00245 /// Lazily parses trees from a file in binary format.  Each tree in the file is not read and parsed
      until it is requested.
00246 /// </summary>
00247 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.  Its <see
      cref="Stream.CanSeek"/> must be <c>true</c>.  It does not have to be a <see
      cref="FileStream"/>.</param>
00248 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00249 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00250 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the file.</returns>
00251       [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1031")]
00252       public static IEnumerable<TreeNode> ParseTrees(Stream inputStream, bool keepOpen = false,
      Action<double> progressAction = null)
00253       {
00254           Contract.Requires(inputStream != null);
00255
00256           BinaryReader reader = new BinaryReader(inputStream, Encoding.UTF8, true);
00257
00258           try
00259           {
00260               if (reader.ReadByte() != (byte)'#' || reader.ReadByte() != (byte)'T' ||
      reader.ReadByte() != (byte)'R' || reader.ReadByte() != (byte)'E')
00261               {
00262                   throw new FormatException("Invalid file header!");
00263               }
00264
00265               byte headerByte = reader.ReadByte();
00266
00267               if ((headerByte & 0b11111100) != 0)
00268               {
00269                   throw new FormatException("Invalid file header!");
00270               }
00271
00272               bool globalNames = (headerByte & 0b1) != 0;
00273               bool globalAttributes = (headerByte & 0b10) != 0;
00274
00275
00276               inputStream.Seek(-4, SeekOrigin.End);
00277
00278               bool validTrailer = true;
00279
00280               if (reader.ReadByte() != (byte)'E' || reader.ReadByte() != (byte)'N' ||
      reader.ReadByte() != (byte)'D' || reader.ReadByte() != (byte)255)
00281               {
00282                   validTrailer = false;
00283               }
00284
00285               List<long> treeAddresses;
00286
00287               if (validTrailer)
00288               {
00289                   inputStream.Seek(-12, SeekOrigin.End);
00290                   long labelAddress = reader.ReadInt64();
00291
```

```
00292                     inputStream.Seek(labelAddress, SeekOrigin.Begin);
00293                     int numOfTrees = reader.ReadInt();
00294                     treeAddresses = new List<long>(numOfTrees);
00295 
00296                     for (int i = 0; i < numOfTrees; i++)
00297                     {
00298                         treeAddresses.Add(reader.ReadInt64());
00299                     }
00300                 }
00301                 else
00302                 {
00303                     treeAddresses = new List<long>();
00304                 }
00305 
00306                 inputStream.Seek(5, SeekOrigin.Begin);
00307 
00308                 string[] allNames = null;
00309 
00310                 if (globalNames)
00311                 {
00312                     allNames = new string[reader.ReadInt()];
00313                     for (int i = 0; i < allNames.Length; i++)
00314                     {
00315                         allNames[i] = reader.ReadMyString();
00316                     }
00317                 }
00318 
00319                 Attribute[] allAttributes = null;
00320 
00321                 if (globalAttributes)
00322                 {
00323                     allAttributes = new Attribute[reader.ReadInt()];
00324 
00325                     for (int i = 0; i < allAttributes.Length; i++)
00326                     {
00327                         allAttributes[i] = new Attribute(reader.ReadMyString(), reader.ReadInt() ==
      2);
00328                     }
00329                 }
00330 
00331                 if (validTrailer)
00332                 {
00333                     for (int i = 0; i < treeAddresses.Count; i++)
00334                     {
00335                         inputStream.Seek(treeAddresses[i], SeekOrigin.Begin);
00336                         yield return reader.ReadTree(globalNames, allNames, allAttributes);
00337                         double progress = Math.Max(0, Math.Min(1, (double)(i + 1) /
      treeAddresses.Count));
00338                         progressAction?.Invoke(progress);
00339                     }
00340                 }
00341                 else
00342                 {
00343                     bool error = false;
00344 
00345                     while (!error)
00346                     {
00347                         TreeNode tree = null;
00348                         try
00349                         {
00350                             tree = reader.ReadTree(globalNames, allNames, allAttributes);
00351                         }
00352                         catch
00353                         {
00354                             error = true;
00355                         }
00356 
00357                         if (!error)
00358                         {
00359                             yield return tree;
00360                             double progress = Math.Max(0, Math.Min(1, (double)(inputStream.Position) /
      inputStream.Length));
00361                             progressAction?.Invoke(progress);
00362                         }
00363                     }
00364                 }
00365             }
00366             finally
00367             {
00368                 reader.Dispose();
00369 
00370                 if (!keepOpen)
00371                 {
00372                     inputStream.Dispose();
00373                 }
00374             }
00375         }
```

```
00376
00377 /// <summary>
00378 /// Parses trees from a file in binary format and completely loads them in memory.
00379 /// </summary>
00380 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.  Its <see
      cref="Stream.CanSeek"/> must be <c>true</c>.  It does not have to be a <see
      cref="FileStream"/>.</param>
00381 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00382 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00383 /// <returns>A <see cref="List{T}"/> containing the trees defined in the file.</returns>
00384         public static List<TreeNode> ParseAllTrees(Stream inputStream, bool keepOpen = false,
      Action<double> progressAction = null)
00385         {
00386             return ParseTrees(inputStream, keepOpen, progressAction).ToList();
00387         }
00388
00389 /// <summary>
00390 /// Lazily parses trees from a file in binary format.  Each tree in the file is not read and parsed
      until it is requested.
00391 /// </summary>
00392 /// <param name="inputFile">The path to the input file.</param>
00393 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00394 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the file.</returns>
00395         public static IEnumerable<TreeNode> ParseTrees(string inputFile, Action<double> progressAction
      = null)
00396         {
00397             FileStream inputStream = File.OpenRead(inputFile);
00398             return ParseTrees(inputStream, false, progressAction);
00399         }
00400
00401 /// <summary>
00402 /// Parses trees from a file in binary format and completely loads them in memory.
00403 /// </summary>
00404 /// <param name="inputFile">The path to the input file.</param>
00405 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00406 /// <returns>A <see cref="List{T}"/> containing the trees defined in the file.</returns>
00407         public static List<TreeNode> ParseAllTrees(string inputFile, Action<double> progressAction =
      null)
00408         {
00409             using FileStream inputStream = File.OpenRead(inputFile);
00410             return ParseAllTrees(inputStream, false, progressAction);
00411         }
00412
00413 /// <summary>
00414 /// Writes a single tree in Binary format.
00415 /// </summary>
00416 /// <param name="tree">The tree to be written.</param>
00417 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00418 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00419 /// <param name="additionalDataToCopy">A stream containing additional data that will be copied into
      the binary file.</param>
00420         public static void WriteTree(TreeNode tree, Stream outputStream, bool keepOpen = false, Stream
      additionalDataToCopy = null)
00421         {
00422             WriteAllTrees(new List<TreeNode> { tree }, outputStream, keepOpen, null,
      additionalDataToCopy);
00423         }
00424
00425 /// <summary>
00426 /// Writes a single tree in Binary format.
00427 /// </summary>
00428 /// <param name="tree">The tree to be written.</param>
00429 /// <param name="outputFile">The file on which the trees should be written.</param>
00430 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00431 /// <param name="additionalDataToCopy">A stream containing additional data that will be copied into
      the binary file.</param>
00432         public static void WriteTree(TreeNode tree, string outputFile, bool append = false, Stream
      additionalDataToCopy = null)
00433         {
00434             using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
      File.Create(outputFile);
00435             WriteAllTrees(new List<TreeNode>() { tree }, outputStream, false, null,
      additionalDataToCopy);
00436         }
00437
00438 /// <summary>
00439 /// Writes trees in binary format.
00440 /// </summary>
00441 /// <param name="trees">An <see cref="IEnumerable{T}"/> containing the trees to be written.  It will
```

```
ony be enumerated once.</param>
00442 /// <param name="outputFile">The file on which the trees should be written.</param>
00443 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00444 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
written, with the number of trees written so far.</param>
00445 /// <param name="additionalDataToCopy">A stream containing additional data that will be copied into
the binary file.</param>
00446         public static void WriteAllTrees(IEnumerable<TreeNode> trees, string outputFile, bool append =
false, Action<int> progressAction = null, Stream additionalDataToCopy = null)
00447         {
00448             using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
File.Create(outputFile);
00449             WriteAllTrees(trees, outputStream, false, progressAction, additionalDataToCopy);
00450         }
00451
00452 /// <summary>
00453 /// Writes trees in binary format.
00454 /// </summary>
00455 /// <param name="trees">An <see cref="IEnumerable{T}"/> containing the trees to be written.  It will
ony be enumerated once.</param>
00456 /// <param name="outputStream">The <see cref="Stream"/> on which the trees should be written.</param>
00457 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
after the end of this method.</param>
00458 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
written, with the number of trees written so far.</param>
00459 /// <param name="additionalDataToCopy">A stream containing additional data that will be copied into
the binary file.</param>
00460         public static void WriteAllTrees(IEnumerable<TreeNode> trees, Stream outputStream, bool
keepOpen = false, Action<int> progressAction = null, Stream additionalDataToCopy = null)
00461         {
00462             Contract.Requires(trees != null);
00463
00464             using BinaryWriter writer = new BinaryWriter(outputStream, Encoding.UTF8, keepOpen);
00465
00466             writer.Write(new byte[] { (byte)'#', (byte)'T', (byte)'R', (byte)'E', 0 });
00467
00468
00469             List<long> addresses = new List<long>();
00470
00471             foreach (TreeNode tree in trees)
00472             {
00473                 writer.Flush();
00474                 addresses.Add(outputStream.Position);
00475                 writer.WriteTree(tree);
00476                 progressAction?.Invoke(addresses.Count);
00477             }
00478
00479             if (additionalDataToCopy != null)
00480             {
00481                 additionalDataToCopy.CopyTo(outputStream);
00482             }
00483
00484             writer.Flush();
00485
00486             long labelAddress = outputStream.Position;
00487
00488             writer.WriteInt(addresses.Count);
00489
00490             for (int i = 0; i < addresses.Count; i++)
00491             {
00492                 writer.Write(addresses[i]);
00493             }
00494
00495             writer.Write(labelAddress);
00496
00497             writer.Write(new byte[] { (byte)'E', (byte)'N', (byte)'D', (byte)255 });
00498
00499         }
00500
00501 /// <summary>
00502 /// Writes trees in binary format.
00503 /// </summary>
00504 /// <param name="trees">A collection of trees to be written.  Each tree will be accessed
twice.</param>
00505 /// <param name="outputFile">The file on which the trees should be written.</param>
00506 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00507 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
written, with a value between 0 and 1 depending on how many trees have been written so far.</param>
00508 /// <param name="additionalDataToCopy">A stream containing additional data that will be copied into
the binary file.</param>
00509         public static void WriteAllTrees(IList<TreeNode> trees, string outputFile, bool append =
false, Action<double> progressAction = null, Stream additionalDataToCopy = null)
00510         {
00511             using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
File.Create(outputFile);
00512             WriteAllTrees(trees, outputStream, false, progressAction, additionalDataToCopy);
00513         }
```

```
00514
00515
00516 /// <summary>
00517 /// Writes trees in binary format.
00518 /// </summary>
00519 /// <param name="trees">A collection of trees to be written.  Each tree will be accessed
      twice.</param>
00520 /// <param name="outputStream">The <see cref="Stream"/> on which the trees should be written.</param>
00521 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00522 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
      written, with a value between 0 and 1 depending on how many trees have been written so far.</param>
00523 /// <param name="additionalDataToCopy">A stream containing additional data that will be copied into
      the binary file.</param>
00524         public static void WriteAllTrees(IList<TreeNode> trees, Stream outputStream, bool keepOpen =
      false, Action<double> progressAction = null, Stream additionalDataToCopy = null)
00525         {
00526             Contract.Requires(trees != null);
00527
00528             Dictionary<string, int> allNamesLookup = new Dictionary<string, int>();
00529             List<string> allNamesLookupReverse = new List<string>();
00530
00531             Dictionary<(string, bool), int> allAttributesLookup = new Dictionary<(string, bool),
      int>();
00532             List<(string, bool)> allAttributesLookupReverse = new List<(string, bool)>();
00533
00534             bool includeNamesPerTree = false;
00535             bool includeAttributesPerTree = false;
00536
00537             for (int i = 0; i < trees.Count; i++)
00538             {
00539                 int prevNameCount = allNamesLookup.Count;
00540                 int prevAttributeCount = allAttributesLookup.Count;
00541
00542                 int count = 0;
00543
00544                 int maxAttributeCount = 0;
00545
00546                 foreach (TreeNode node in trees[i].GetChildrenRecursiveLazy())
00547                 {
00548                     if (!string.IsNullOrEmpty(node.Name))
00549                     {
00550                         count++;
00551                         if (allNamesLookup.TryAdd(node.Name, allNamesLookup.Count))
00552                         {
00553                             allNamesLookupReverse.Add(node.Name);
00554                         }
00555                     }
00556
00557                     maxAttributeCount = Math.Max(maxAttributeCount, node.Attributes.Count);
00558
00559                     foreach (KeyValuePair<string, object> kvp in node.Attributes)
00560                     {
00561                         bool isDouble = kvp.Value is double;
00562                         if (allAttributesLookup.TryAdd((kvp.Key, isDouble),
      allAttributesLookup.Count))
00563                         {
00564                             allAttributesLookupReverse.Add((kvp.Key, isDouble));
00565                         }
00566                     }
00567                 }
00568
00569                 if (prevNameCount != 0 && (allNamesLookup.Count - prevNameCount) * 2 > count)
00570                 {
00571                     includeNamesPerTree = true;
00572                 }
00573
00574                 if (prevAttributeCount != 0 && (allAttributesLookup.Count - prevAttributeCount) * 2 >
      maxAttributeCount)
00575                 {
00576                     includeAttributesPerTree = true;
00577                 }
00578
00579                 if (includeNamesPerTree && includeAttributesPerTree)
00580                 {
00581                     break;
00582                 }
00583             }
00584
00585             using BinaryWriter writer = new BinaryWriter(outputStream, Encoding.UTF8, keepOpen);
00586
00587             writer.Write(new byte[] { (byte)'#', (byte)'T', (byte)'R', (byte)'E' });
00588
00589             if (!includeNamesPerTree && !includeAttributesPerTree)
00590             {
00591                 writer.Write((byte)0b00000011);
00592             }
```

```
00593             else if (!includeNamesPerTree && includeAttributesPerTree)
00594             {
00595                 writer.Write((byte)0b00000001);
00596             }
00597             else if (includeNamesPerTree && !includeAttributesPerTree)
00598             {
00599                 writer.Write((byte)0b00000010);
00600             }
00601             else
00602             {
00603                 writer.Write((byte)0b00000000);
00604             }
00605 
00606             if (!includeNamesPerTree)
00607             {
00608                 writer.WriteInt(allNamesLookup.Count);
00609 
00610                 for (int i = 0; i < allNamesLookup.Count; i++)
00611                 {
00612                     writer.WriteMyString(allNamesLookupReverse[i]);
00613                 }
00614             }
00615 
00616             if (!includeAttributesPerTree)
00617             {
00618                 writer.WriteInt(allAttributesLookup.Count);
00619 
00620                 for (int i = 0; i < allAttributesLookup.Count; i++)
00621                 {
00622                     writer.WriteMyString(allAttributesLookupReverse[i].Item1);
00623                     writer.WriteInt(allAttributesLookupReverse[i].Item2 ?  2 :  1);
00624                 }
00625             }
00626 
00627             long[] addresses = new long[trees.Count];
00628 
00629             for (int i = 0; i < trees.Count; i++)
00630             {
00631                 writer.Flush();
00632                 addresses[i] = outputStream.Position;
00633                 writer.WriteTree(trees[i], !includeNamesPerTree, !includeAttributesPerTree,
      allNamesLookup, allAttributesLookup);
00634 
00635                 double progress = Math.Max(0, Math.Min(1, (double)(i + 1) / trees.Count));
00636 
00637                 progressAction?.Invoke(progress);
00638             }
00639 
00640             if (additionalDataToCopy != null)
00641             {
00642                 additionalDataToCopy.CopyTo(outputStream);
00643             }
00644 
00645             writer.Flush();
00646 
00647             long labelAddress = outputStream.Position;
00648 
00649             writer.WriteInt(addresses.Length);
00650 
00651             for (int i = 0; i < addresses.Length; i++)
00652             {
00653                 writer.Write(addresses[i]);
00654             }
00655 
00656             writer.Write(labelAddress);
00657 
00658             writer.Write(new byte[] { (byte)'E', (byte)'N', (byte)'D', (byte)255 });
00659 
00660         }
00661     }
00662 
00663 /// <summary>
00664 /// Holds metadata information about a file containing trees in binary format.
00665 /// </summary>
00666     public class BinaryTreeMetadata
00667     {
00668 /// <summary>
00669 /// The addresses of the trees (i.e.  byte offsets from the start of the file).
00670 /// </summary>
00671         public IEnumerable<long> TreeAddresses { get; set; }
00672 
00673 /// <summary>
00674 /// Determines whether there are any global names stored in the file's header that are used when
      parsing the trees.
00675 /// </summary>
00676         public bool GlobalNames { get; set; }
00677 
```

```
00678 /// <summary>
00679 /// Contains any global names stored in the file's header that are used when parsing the trees.
00680 /// </summary>
00681         public IReadOnlyList<string> Names { get; set; }
00682 
00683 /// <summary>
00684 /// Contains any global attributes stored in the file's header that are used when parsing the trees.
00685 /// </summary>
00686         public IReadOnlyList<Attribute> AllAttributes { get; set; }
00687     }
00688 
00689 /// <summary>
00690 /// Describes an attribute of a node.
00691 /// </summary>
00692     public struct Attribute :  IEquatable<Attribute>
00693     {
00694 /// <summary>
00695 /// The name of the attribute.
00696 /// </summary>
00697         public string AttributeName { get; }
00698 
00699 /// <summary>
00700 /// Whether the attribute is represented by a numeric value or a string.
00701 /// </summary>
00702         public bool IsNumeric { get; }
00703 
00704 /// <summary>
00705 /// Constructs a new <see cref="Attribute"/>.
00706 /// </summary>
00707 /// <param name="attributeName">The name of the attribute.</param>
00708 /// <param name="isNumeric">Whether the attribute is represented by a numeric value or a
      string.</param>
00709         public Attribute(string attributeName, bool isNumeric)
00710         {
00711             this.AttributeName = attributeName;
00712             this.IsNumeric = isNumeric;
00713         }
00714 
00715 /// <summary>
00716 /// Compares an <see cref="Attribute"/> and another <see cref="object"/>.
00717 /// </summary>
00718 /// <param name="obj">The <see cref="object"/> to compare to.</param>
00719 /// <returns><c>true</c> if <paramref name="obj"/> is an <see cref="Attribute"/> and it has the same
      <see cref="AttributeName"/> (case insensitive) and value for <see cref="IsNumeric"/> as the current
      instance.  <c>false</c> otherwise.</returns>
00720         public override bool Equals(object obj)
00721         {
00722             if (obj is Attribute attr)
00723             {
00724                 return this.AttributeName.Equals(attr.AttributeName,
      StringComparison.OrdinalIgnoreCase) && this.IsNumeric == attr.IsNumeric;
00725             }
00726             else
00727             {
00728                 return false;
00729             }
00730         }
00731 
00732 /// <summary>
00733 /// Returns the hash code for this <see cref="Attribute"/>.
00734 /// </summary>
00735 /// <returns>The hash code for this <see cref="Attribute"/>.</returns>
00736         public override int GetHashCode()
00737         {
00738             return this.AttributeName.GetHashCode(StringComparison.OrdinalIgnoreCase) +
      this.IsNumeric.GetHashCode();
00739         }
00740 
00741 /// <summary>
00742 /// Compares two <see cref="Attribute"/>s.
00743 /// </summary>
00744 /// <param name="left">The first <see cref="Attribute"/> to compare.</param>
00745 /// <param name="right">The second <see cref="Attribute"/> to compare.</param>
00746 /// <returns><c>true</c> if both <see cref="Attribute"/>s have the same <see cref="AttributeName"/>
      (case insensitive) and value for <see cref="IsNumeric"/>.  <c>false</c> otherwise.</returns>
00747         public static bool operator ==(Attribute left, Attribute right)
00748         {
00749             return left.Equals(right);
00750         }
00751 
00752 /// <summary>
00753 /// Compares two <see cref="Attribute"/>s (negated).
00754 /// </summary>
00755 /// <param name="left">The first <see cref="Attribute"/> to compare.</param>
00756 /// <param name="right">The second <see cref="Attribute"/> to compare.</param>
00757 /// <returns><c>false</c> if both <see cref="Attribute"/>s have the same <see cref="AttributeName"/>
      (case insensitive) and value for <see cref="IsNumeric"/>.  <c>true</c> otherwise.</returns>
```

```
00758         public static bool operator !=(Attribute left, Attribute right)
00759         {
00760             return !(left == right);
00761         }
00762 
00763 /// <summary>
00764 /// Compares two <see cref="Attribute"/>s.
00765 /// </summary>
00766 /// <param name="other">The <see cref="Attribute"/> to compare to.</param>
00767 /// <returns><c>true</c> if <paramref name="other"/> has the same <see cref="AttributeName"/> (case
      insensitive) and value for <see cref="IsNumeric"/> as the current instance.  <c>false</c>
      otherwise.</returns>
00768         public bool Equals(Attribute other)
00769         {
00770             return this.AttributeName.Equals(other.AttributeName, StringComparison.OrdinalIgnoreCase)
      && this.IsNumeric == other.IsNumeric;
00771         }
00772     }
00773 
00774 /// <summary>
00775 /// Extension methods for <see cref="BinaryReader"/> and <see cref="BinaryWriter"/>.
00776 /// </summary>
00777     internal static class BinaryExtensions
00778     {
00779 /// <summary>
00780 /// Writes a variable-width integer to the stream.  If the integer is smaller than 254, it is only
      1-byte wide; otherwise it is 40-bit (5-byte) wide.
00781 /// </summary>
00782 /// <param name="writer">The <see cref="BinaryWriter"/> on which to write.</param>
00783 /// <param name="value">The value to be written.</param>
00784         public static void WriteInt(this BinaryWriter writer, int value)
00785         {
00786             if (value < 254)
00787             {
00788                 writer.Write((byte)value);
00789             }
00790             else
00791             {
00792                 writer.Write((byte)254);
00793                 writer.Write(value);
00794             }
00795         }
00796 
00797 /// <summary>
00798 /// Reads a variable-width integer from the stream.  If the integer is smaller than 254, it is only
      1-byte wide; otherwise it is 40-bit (5-byte) wide.
00799 /// </summary>
00800 /// <param name="reader">The <see cref="BinaryReader"/> from which to read.</param>
00801 /// <returns>The value read.</returns>
00802         public static int ReadInt(this BinaryReader reader)
00803         {
00804             byte b = reader.ReadByte();
00805 
00806             if (b < 254)
00807             {
00808                 return b;
00809             }
00810             else
00811             {
00812                 return reader.ReadInt32();
00813             }
00814         }
00815 
00816 /// <summary>
00817 /// Writes a string to the stream.  The string is stored as an integer n representing its length
      followed by n integers that constitute the UTF-16 representation of the string.
00818 /// </summary>
00819 /// <param name="writer">The <see cref="BinaryWriter"/> on which to write.</param>
00820 /// <param name="value">The value to be written.</param>
00821         public static void WriteMyString(this BinaryWriter writer, string value)
00822         {
00823             writer.WriteInt(value.Length);
00824 
00825             foreach (char c in value)
00826             {
00827                 writer.WriteInt(c);
00828             }
00829         }
00830 
00831 /// <summary>
00832 /// Reads a string from the stream.  The string is stored as an integer n representing its length
      followed by n integers that constitute the UTF-16 representation of the string.
00833 /// </summary>
00834 /// <param name="reader">The <see cref="BinaryReader"/> from which to read.</param>
00835 /// <returns>The value read.</returns>
00836         public static string ReadMyString(this BinaryReader reader)
00837         {
```

```
00838             int length = reader.ReadInt();
00839             StringBuilder bld = new StringBuilder();
00840 
00841             for (int i = 0; i < length; i++)
00842             {
00843                 bld.Append((char)reader.ReadInt());
00844             }
00845 
00846             return bld.ToString();
00847         }
00848 
00849 /// <summary>
00850 /// Read a variable-width integer from the stream.  If the integer is equal to 0, 2 or 3, it is 2-bit
      wide; if it is 1, 4 or 5, it is 4-bit wide; if it is greater than 5, the current byte is padded and
      the integer is represented as an integer of the format read by <see cref="ReadInt(BinaryReader)"/> in
      the following byte(s).
00851 /// The initial value of currByte should be read using <see cref="BinaryReader.ReadByte"/> and the
      initial value of currIndex should be 0.  Successive reads should use the same variables, which will
      have been updated by this method.
00852 /// After the last read, if *currIndex is equal to 0, it means that currByte has not been processed
      (thus you should seek back by 1).
00853 /// </summary>
00854 /// <param name="reader">The <see cref="BinaryReader"/> from which to read.</param>
00855 /// <param name="currByte">The current byte that is being read</param>
00856 /// <param name="currIndex">The current index within the byte.</param>
00857 /// <returns>The value read.</returns>
00858         public static int ReadShortInt(this BinaryReader reader, ref byte currByte, ref int currIndex)
00859         {
00860             if (currIndex == 0)
00861             {
00862                 int twoBits = currByte & 0b00000011;
00863 
00864                 currIndex = 2;
00865 
00866                 if (twoBits == 0b00)
00867                 {
00868                     return 0;
00869                 }
00870                 else if (twoBits == 0b01)
00871                 {
00872                     return 2;
00873                 }
00874                 else if (twoBits == 0b10)
00875                 {
00876                     return 3;
00877                 }
00878                 else// if (twoBits == 0b11)
00879                 {
00880                     int fourBits = currByte & 0b00001111;
00881                     currIndex = 4;
00882 
00883                     if (fourBits == 0b0011)
00884                     {
00885                         return 1;
00886                     }
00887                     else if (fourBits == 0b0111)
00888                     {
00889                         return 4;
00890                     }
00891                     else if (fourBits == 0b1011)
00892                     {
00893                         return 5;
00894                     }
00895                     else// if (fourBits == 0b1111)
00896                     {
00897                         int tbr = reader.ReadInt();
00898                         currByte = reader.ReadByte();
00899                         currIndex = 0;
00900                         return tbr;
00901                     }
00902                 }
00903             }
00904             else if (currIndex == 2)
00905             {
00906                 int twoBits = (currByte & 0b00001100) » 2;
00907 
00908                 currIndex = 4;
00909 
00910                 if (twoBits == 0b00)
00911                 {
00912                     return 0;
00913                 }
00914                 else if (twoBits == 0b01)
00915                 {
00916                     return 2;
00917                 }
00918                 else if (twoBits == 0b10)
```

```
00919                 {
00920                     return 3;
00921                 }
00922                 else// if (twoBits == 0b11)
00923                 {
00924                     int fourBits = (currByte & 0b00111100) » 2;
00925                     currIndex = 6;
00926 
00927                     if (fourBits == 0b0011)
00928                     {
00929                         return 1;
00930                     }
00931                     else if (fourBits == 0b0111)
00932                     {
00933                         return 4;
00934                     }
00935                     else if (fourBits == 0b1011)
00936                     {
00937                         return 5;
00938                     }
00939                     else// if (fourBits == 0b1111)
00940                     {
00941                         int tbr = reader.ReadInt();
00942                         currByte = reader.ReadByte();
00943                         currIndex = 0;
00944                         return tbr;
00945                     }
00946                 }
00947             }
00948             else if (currIndex == 4)
00949             {
00950                 int twoBits = (currByte & 0b00110000) » 4;
00951 
00952                 currIndex = 6;
00953 
00954                 if (twoBits == 0b00)
00955                 {
00956                     return 0;
00957                 }
00958                 else if (twoBits == 0b01)
00959                 {
00960                     return 2;
00961                 }
00962                 else if (twoBits == 0b10)
00963                 {
00964                     return 3;
00965                 }
00966                 else// if (twoBits == 0b11)
00967                 {
00968                     int fourBits = (currByte & 0b11110000) » 4;
00969                     currIndex = 0;
00970 
00971                     if (fourBits == 0b0011)
00972                     {
00973                         currByte = reader.ReadByte();
00974                         return 1;
00975                     }
00976                     else if (fourBits == 0b0111)
00977                     {
00978                         currByte = reader.ReadByte();
00979                         return 4;
00980                     }
00981                     else if (fourBits == 0b1011)
00982                     {
00983                         currByte = reader.ReadByte();
00984                         return 5;
00985                     }
00986                     else// if (fourBits == 0b1111)
00987                     {
00988                         int tbr = reader.ReadInt();
00989                         currByte = reader.ReadByte();
00990                         currIndex = 0;
00991                         return tbr;
00992                     }
00993                 }
00994             }
00995             else //if (currIndex == 6)
00996             {
00997                 int twoBits = (currByte & 0b11000000) » 6;
00998 
00999                 currIndex = 0;
01000                 currByte = reader.ReadByte();
01001 
01002                 if (twoBits == 0b00)
01003                 {
01004                     return 0;
01005                 }
```

```
01006                 else if (twoBits == 0b01)
01007                 {
01008                     return 2;
01009                 }
01010                 else if (twoBits == 0b10)
01011                 {
01012                     return 3;
01013                 }
01014                 else// if (twoBits == 0b11)
01015                 {
01016                     int fourBits = twoBits | ((currByte & 0b00000011) << 2);
01017                     currIndex = 2;
01018 
01019                     if (fourBits == 0b0011)
01020                     {
01021                         return 1;
01022                     }
01023                     else if (fourBits == 0b0111)
01024                     {
01025                         return 4;
01026                     }
01027                     else if (fourBits == 0b1011)
01028                     {
01029                         return 5;
01030                     }
01031                     else// if (fourBits == 0b1111)
01032                     {
01033                         int tbr = reader.ReadInt();
01034                         currByte = reader.ReadByte();
01035                         currIndex = 0;
01036                         return tbr;
01037                     }
01038                 }
01039             }
01040         }
01041 
01042 /// <summary>
01043 /// Write a variable-width integer from the stream.  If the integer is equal to 0, 2 or 3, it is 2-bit wide; if it is 1, 4 or 5, it is 4-bit wide; if it is greater than 5, the current byte is padded and the integer is represented as an integer of the format written by readInt in the following byte(s).
01044 /// The initial value of currByte and currIndex should be 0.  Successive writes should use the same variables, which will have been updated by this method.
01045 /// After the last write, if *currIndex is not 0, it means that the current byte has not been written to the stream yet.
01046 /// </summary>
01047 /// <param name="writer">The <see cref="BinaryWriter"/> on which to write.</param>
01048 /// <param name="value">The value to be written.</param>
01049 /// <param name="currByte">The value of the byte that is being written.</param>
01050 /// <param name="currIndex">The current index within the byte.</param>
01051 /// <returns></returns>
01052         public static int WriteShortInt(this BinaryWriter writer, int value, ref byte currByte, int currIndex)
01053         {
01054             if (value == 0)
01055             {
01056                 //00
01057                 if (currIndex == 0)
01058                 {
01059                     return 2;
01060                 }
01061                 else if (currIndex == 2)
01062                 {
01063                     return 4;
01064                 }
01065                 else if (currIndex == 4)
01066                 {
01067                     return 6;
01068                 }
01069                 else if (currIndex == 6)
01070                 {
01071                     writer.Write(currByte);
01072                     currByte = 0;
01073                     return 0;
01074                 }
01075             }
01076             else if (value == 2)
01077             {
01078                 //01
01079                 if (currIndex == 0)
01080                 {
01081                     currByte = (byte)(currByte | 0b00000001);
01082                     return 2;
01083                 }
01084                 else if (currIndex == 2)
01085                 {
01086                     currByte = (byte)(currByte | 0b00000100);
01087                     return 4;
```

```
01088                 }
01089                 else if (currIndex == 4)
01090                 {
01091                     currByte = (byte)(currByte | 0b00010000);
01092                     return 6;
01093                 }
01094                 else if (currIndex == 6)
01095                 {
01096                     writer.Write((byte)(currByte | 0b01000000));
01097                     currByte = 0;
01098                     return 0;
01099                 }
01100             }
01101             else if (value == 3)
01102             {
01103                 //10
01104                 if (currIndex == 0)
01105                 {
01106                     currByte = (byte)(currByte | 0b00000010);
01107                     return 2;
01108                 }
01109                 else if (currIndex == 2)
01110                 {
01111                     currByte = (byte)(currByte | 0b00001000);
01112                     return 4;
01113                 }
01114                 else if (currIndex == 4)
01115                 {
01116                     currByte = (byte)(currByte | 0b00100000);
01117                     return 6;
01118                 }
01119                 else if (currIndex == 6)
01120                 {
01121                     writer.Write((byte)(currByte | 0b10000000));
01122                     currByte = 0;
01123                     return 0;
01124                 }
01125             }
01126             else if (value == 1)
01127             {
01128                 //0011
01129                 if (currIndex == 0)
01130                 {
01131                     currByte = (byte)(currByte | 0b00000011);
01132                     return 4;
01133                 }
01134                 else if (currIndex == 2)
01135                 {
01136                     currByte = (byte)(currByte | 0b00001100);
01137                     return 6;
01138                 }
01139                 else if (currIndex == 4)
01140                 {
01141                     writer.Write((byte)(currByte | 0b00110000));
01142                     currByte = 0;
01143                     return 0;
01144                 }
01145                 else if (currIndex == 6)
01146                 {
01147                     writer.Write((byte)(currByte | 0b11000000));
01148                     currByte = 0;
01149                     return 2;
01150                 }
01151             }
01152             else if (value == 4)
01153             {
01154                 //0111
01155                 if (currIndex == 0)
01156                 {
01157                     currByte = (byte)(currByte | 0b00000111);
01158                     return 4;
01159                 }
01160                 else if (currIndex == 2)
01161                 {
01162                     currByte = (byte)(currByte | 0b00011100);
01163                     return 6;
01164                 }
01165                 else if (currIndex == 4)
01166                 {
01167                     writer.Write((byte)(currByte | 0b01110000));
01168                     currByte = 0;
01169                     return 0;
01170                 }
01171                 else if (currIndex == 6)
01172                 {
01173                     writer.Write((byte)(currByte | 0b11000000));
01174                     currByte = 0b00000001;
```

```
01175                     return 2;
01176                 }
01177             }
01178             else if (value == 5)
01179             {
01180                 //1011
01181                 if (currIndex == 0)
01182                 {
01183                     currByte = (byte)(currByte | 0b00001011);
01184                     return 4;
01185                 }
01186                 else if (currIndex == 2)
01187                 {
01188                     currByte = (byte)(currByte | 0b00101100);
01189                     return 6;
01190                 }
01191                 else if (currIndex == 4)
01192                 {
01193                     writer.Write((byte)(currByte | 0b10110000));
01194                     currByte = 0;
01195                     return 0;
01196                 }
01197                 else if (currIndex == 6)
01198                 {
01199                     writer.Write((byte)(currByte | 0b11000000));
01200                     currByte = 0b00000010;
01201                     return 2;
01202                 }
01203             }
01204             else
01205             {
01206                 //1111
01207                 if (currIndex == 0)
01208                 {
01209                     writer.Write((byte)(currByte | 0b00001111));
01210                     writer.WriteInt(value);
01211                     currByte = 0;
01212                     return 0;
01213                 }
01214                 else if (currIndex == 2)
01215                 {
01216                     writer.Write((byte)(currByte | 0b00111100));
01217                     writer.WriteInt(value);
01218                     currByte = 0;
01219                     return 0;
01220                 }
01221                 else if (currIndex == 4)
01222                 {
01223                     writer.Write((byte)(currByte | 0b11110000));
01224                     writer.WriteInt(value);
01225                     currByte = 0;
01226                     return 0;
01227                 }
01228                 else if (currIndex == 6)
01229                 {
01230                     writer.Write((byte)(currByte | 0b11000000));
01231                     writer.Write((byte)0b00000011);
01232                     writer.WriteInt(value);
01233                     currByte = 0;
01234                     return 0;
01235                 }
01236             }
01237
01238             throw new IndexOutOfRangeException("Unexpected position!");
01239         }
01240
01241 /// <summary>
01242 /// Writes a tree in binary format to the stream.
01243 /// </summary>
01244 /// <param name="writer">The <see cref="BinaryWriter"/> on which to write.</param>
01245 /// <param name="tree">The <see cref="TreeNode"/> to be written.</param>
01246 /// <param name="globalNames">Specifies whether global names are stored in the file's header.</param>
01247 /// <param name="globalAttributes">Specified whether global attributes are stored in the file's
      header.</param>
01248 /// <param name="names">The global names specified in the file's header.</param>
01249 /// <param name="attributes">The global attributes specified in the file's header.</param>
01250         public static void WriteTree(this BinaryWriter writer, TreeNode tree, bool globalNames =
      false, bool globalAttributes = false, Dictionary<string, int> names = null, Dictionary<(string, bool),
      int> attributes = null)
01251         {
01252             List<TreeNode> nodes = tree.GetChildrenRecursive();
01253
01254
01255             if (!globalAttributes)
01256             {
01257                 attributes = new Dictionary<(string, bool), int>();
01258                 List<(string, bool)> attributesLookupReverse = new List<(string, bool)>();
```

```
01259
01260                 for (int i = 0; i < nodes.Count; i++)
01261                 {
01262                     foreach (KeyValuePair<string, object> kvp in nodes[i].Attributes)
01263                     {
01264                         bool isDouble = kvp.Value is double;
01265                         if (attributes.TryAdd((kvp.Key, isDouble), attributes.Count))
01266                         {
01267                             attributesLookupReverse.Add((kvp.Key, isDouble));
01268                         }
01269                     }
01270                 }
01271
01272                 writer.WriteInt(attributes.Count);
01273
01274                 for (int i = 0; i < attributes.Count; i++)
01275                 {
01276                     writer.WriteMyString(attributesLookupReverse[i].Item1);
01277                     writer.WriteInt(attributesLookupReverse[i].Item2 ?  2 :  1);
01278                 }
01279             }
01280             else
01281             {
01282                 writer.Write((byte)0);
01283             }
01284
01285             //Topology
01286             byte currByte = 0;
01287             int currPos = 0;
01288
01289             for (int i = 0; i < nodes.Count; i++)
01290             {
01291                 currPos = writer.WriteShortInt(nodes[i].Children.Count, ref currByte, currPos);
01292             }
01293
01294             if (currPos != 0)
01295             {
01296                 writer.Write(currByte);
01297             }
01298
01299
01300             //Attributes
01301             for (int i = 0; i < nodes.Count; i++)
01302             {
01303                 int attributeCount = 0;
01304
01305                 foreach (KeyValuePair<string, object> kvp in nodes[i].Attributes)
01306                 {
01307                     bool isDouble = kvp.Value is double;
01308
01309                     if (!isDouble && kvp.Key == "Name" && globalNames)
01310                     {
01311                         string value = (kvp.Value as string) ??  kvp.Value?.ToString() ??  "";
01312
01313                         if (!string.IsNullOrEmpty(value))
01314                         {
01315                             attributeCount++;
01316                         }
01317                     }
01318                     else
01319                     {
01320                         if (isDouble)
01321                         {
01322                             double value = (double)kvp.Value;
01323                             if (!double.IsNaN(value))
01324                             {
01325                                 attributeCount++;
01326                             }
01327                         }
01328                         else
01329                         {
01330                             string value = (kvp.Value as string) ??  kvp.Value?.ToString() ??  "";
01331                             if (!string.IsNullOrEmpty(value))
01332                             {
01333                                 attributeCount++;
01334                             }
01335                         }
01336                     }
01337                 }
01338
01339                 writer.WriteInt(attributeCount);
01340                 foreach (KeyValuePair<string, object> kvp in nodes[i].Attributes)
01341                 {
01342                     bool isDouble = kvp.Value is double;
01343                     int index = attributes[(kvp.Key, isDouble)];
01344
01345                     if (!isDouble && kvp.Key == "Name" && globalNames)
```

```
01346                     {
01347                         string value = (kvp.Value as string) ??  kvp.Value?.ToString() ??  "";
01348 
01349                         if (!string.IsNullOrEmpty(value))
01350                         {
01351                             writer.WriteInt(index);
01352                             if (names.TryGetValue(value, out int nameIndex))
01353                             {
01354                                 writer.WriteInt(nameIndex + 1);
01355                             }
01356                             else
01357                             {
01358                                 writer.Write((byte)255);
01359                                 writer.WriteMyString(value);
01360                             }
01361                         }
01362                     }
01363                     else
01364                     {
01365                         if (isDouble)
01366                         {
01367                             double value = (double)kvp.Value;
01368                             if (!double.IsNaN(value))
01369                             {
01370                                 writer.WriteInt(index);
01371                                 writer.Write(value);
01372                             }
01373                         }
01374                         else
01375                         {
01376                             string value = (kvp.Value as string) ??  kvp.Value?.ToString() ??  "";
01377                             if (!string.IsNullOrEmpty(value))
01378                             {
01379                                 writer.WriteInt(index);
01380                                 writer.WriteMyString(value);
01381                             }
01382                         }
01383                     }
01384                 }
01385             }
01386         }
01387 
01388 /// <summary>
01389 /// Reads a tree in binary format from the stream.
01390 /// </summary>
01391 /// <param name="reader">The <see cref="BinaryReader"/> from which to read.</param>
01392 /// <param name="globalNames">Specifies whether global names are stored in the file's header.</param>
01393 /// <param name="names">The global names specified in the file's header.</param>
01394 /// <param name="attributes">The global attributes specified in the file's header.</param>
01395 /// <returns>The <see cref="TreeNode"/> that has been read.</returns>
01396         public static TreeNode ReadTree(this BinaryReader reader, bool globalNames = false,
      IReadOnlyList<string> names = null, IReadOnlyList<Attribute> attributes = null)
01397         {
01398             int numAttributes = reader.ReadInt();
01399 
01400             if (numAttributes > 0)
01401             {
01402                 Attribute[] actualAttributes = new Attribute[numAttributes];
01403 
01404                 for (int i = 0; i < actualAttributes.Length; i++)
01405                 {
01406                     actualAttributes[i] = new Attribute(reader.ReadMyString(), reader.ReadInt() == 2);
01407                 }
01408 
01409                 attributes = actualAttributes;
01410             }
01411 
01412             //Topology
01413             TreeNode rootNode = new TreeNode(null);
01414 
01415             TreeNode currParent = rootNode;
01416 
01417             Dictionary<string, int> childCounts = new Dictionary<string, int>();
01418 
01419             byte currByte = reader.ReadByte();
01420             int currIndex = 0;
01421 
01422             while (currParent != null)
01423             {
01424                 int currCount = reader.ReadShortInt(ref currByte, ref currIndex);
01425 
01426                 childCounts.Add(currParent.Id, currCount);
01427 
01428                 while (currParent != null && currParent.Children.Count == childCounts[currParent.Id])
01429                 {
01430                     currParent = currParent.Parent;
01431                 }
```

```
01432
01433                 if (currParent != null)
01434                 {
01435                     TreeNode newNode = new TreeNode(currParent);
01436                     currParent.Children.Add(newNode);
01437                     currParent = newNode;
01438                 }
01439             }
01440
01441             if (currIndex == 0)
01442             {
01443                 reader.BaseStream.Seek(-1, SeekOrigin.Current);
01444             }
01445
01446             List<TreeNode> nodes = rootNode.GetChildrenRecursive();
01447
01448             //Attributes
01449             for (int i = 0; i < nodes.Count; i++)
01450             {
01451                 int attributeCount = reader.ReadInt();
01452
01453                 for (int j = 0; j < attributeCount; j++)
01454                 {
01455                     int attributeIndex = reader.ReadInt();
01456                     string attributeName = attributes[attributeIndex].AttributeName;
01457                     bool isDouble = attributes[attributeIndex].IsNumeric;
01458
01459                     if (isDouble)
01460                     {
01461                         if (string.Equals(attributeName, "Length",
      StringComparison.OrdinalIgnoreCase))
01462                         {
01463                             nodes[i].Length = reader.ReadDouble();
01464                         }
01465                         else if (string.Equals(attributeName, "Support",
      StringComparison.OrdinalIgnoreCase))
01466                         {
01467                             nodes[i].Support = reader.ReadDouble();
01468                         }
01469                         else
01470                         {
01471                             nodes[i].Attributes[attributeName] = reader.ReadDouble();
01472                         }
01473                     }
01474                     else if (!string.Equals(attributeName, "Name",
      StringComparison.OrdinalIgnoreCase))
01475                     {
01476                         nodes[i].Attributes[attributeName] = reader.ReadMyString();
01477                     }
01478                     else
01479                     {
01480                         if (!globalNames)
01481                         {
01482                             nodes[i].Name = reader.ReadMyString();
01483                         }
01484                         else
01485                         {
01486                             byte b = (byte)reader.BaseStream.ReadByte();
01487
01488                             if (b == 0)
01489                             {
01490                                 nodes[i].Name = "";
01491                             }
01492                             else if (b <= 254)
01493                             {
01494                                 reader.BaseStream.Position--;
01495                                 int index = reader.ReadInt();
01496                                 nodes[i].Name = names[index - 1];
01497                             }
01498                             else //if(b == 255)
01499                             {
01500                                 nodes[i].Name = reader.ReadMyString();
01501                             }
01502                         }
01503                     }
01504                 }
01505             }
01506
01507             return rootNode;
01508         }
01509     }
01510 }
```

8.3 Extensions.cs

```
00001 using System;
00002 using System.Collections.Concurrent;
00003 using System.Collections.Generic;
00004 using System.Diagnostics.Contracts;
00005 using System.IO;
00006 using System.Linq;
00007 using System.Text;
00008 using System.Threading;
00009 using System.Threading.Tasks;
00010
00011 /// <summary>
00012 /// Contains useful extension methods.
00013 /// </summary>
00014 namespace PhyloTree.Extensions
00015 {
00016 /// <summary>
00017 /// Useful extension methods
00018 /// </summary>
00019     public static class TypeExtensions
00020     {
00021 /// <summary>
00022 /// Determines whether <paramref name="haystack"/> contains all of the elements in <paramref
      name="needle"/>.
00023 /// </summary>
00024 /// <typeparam name="T">The type of the elements in the collections.</typeparam>
00025 /// <param name="haystack">The collection in which to search.</param>
00026 /// <param name="needle">The items to be searched.</param>
00027 /// <returns><c>true</c> if haystack contains all of the elements that are in needle or needle is
      empty.</returns>
00028         public static bool ContainsAll<T>(this IEnumerable<T> haystack, IEnumerable<T> needle)
00029         {
00030             Contract.Requires(needle != null);
00031
00032             foreach (T t in needle)
00033             {
00034                 if (!haystack.Contains(t))
00035                 {
00036                     return false;
00037                 }
00038             }
00039             return true;
00040         }
00041
00042 /// <summary>
00043 /// Compute the median of a list of values.
00044 /// </summary>
00045 /// <param name="array">The list of values whose median is to be computed.</param>
00046 /// <returns>The median of the list of values.</returns>
00047         public static double Median(this IEnumerable<double> array)
00048         {
00049             List<double> ordered = new List<double>(array);
00050             ordered.Sort();
00051
00052             if (ordered.Count % 2 == 0)
00053             {
00054                 return 0.5 * (ordered[ordered.Count / 2] + ordered[ordered.Count / 2 - 1]);
00055             }
00056             else
00057             {
00058                 return ordered[ordered.Count / 2];
00059             }
00060         }
00061
00062 /// <summary>
00063 /// Determines whether <paramref name="haystack"/> contains at least one of the elements in <paramref
      name="needle"/>.
00064 /// </summary>
00065 /// <typeparam name="T">The type of the elements in the collections.</typeparam>
00066 /// <param name="haystack">The collection in which to search.</param>
00067 /// <param name="needle">The items to be searched.</param>
00068 /// <returns><c>true</c> if haystack contains at least one of the elements that are in needle.
      Returns <c>false</c> if needle is empty.</returns>
00069         public static bool ContainsAny<T>(this IEnumerable<T> haystack, IEnumerable<T> needle)
00070         {
00071             Contract.Requires(needle != null);
00072
00073             foreach (T t in needle)
00074             {
00075                 if (haystack.Contains(t))
00076                 {
00077                     return true;
00078                 }
00079             }
00080             return false;
00081         }
```

```
00082
00083 /// <summary>
00084 /// Computes the intersection between two sets.
00085 /// </summary>
00086 /// <typeparam name="T">The type of the elements in the collections.</typeparam>
00087 /// <param name="set1">The first set.</param>
00088 /// <param name="set2">The second set.</param>
00089 /// <returns>The intersection between the two sets.</returns>
00090         public static IEnumerable<T> Intersection<T>(this IEnumerable<T> set1, IEnumerable<T> set2)
00091         {
00092             Contract.Requires(set1 != null);
00093             Contract.Requires(set2 != null);
00094
00095             foreach (T element in set1)
00096             {
00097                 if (set2.Contains(element))
00098                 {
00099                     yield return element;
00100                 }
00101             }
00102         }
00103
00104 /// <summary>
00105 /// Constructs a consensus tree.
00106 /// </summary>
00107 /// <param name="trees">The collection of trees whose consensus is to be computed.</param>
00108 /// <param name="rooted">Whether the consensus tree should be rooted or not.</param>
00109 /// <param name="clockLike">Whether the trees are to be treated as clock-like trees or not.  This has
      an effect on how the branch lengths of the consensus tree are computed.</param>
00110 /// <param name="threshold">The (inclusive) threshold for splits to be included in the consensus tree.
      Use <c>0</c> to get all compatible splits, <c>0.5</c> for a majority-rule consensus or <c>1</c> for a
      strict consensus.</param>
00111 /// <param name="useMedian">If this is <c>true</c>, the lengths of the branches in the tree will be
      computed based on the median length/age of the splits used to build the tree.  Otherwise, the mean
      will be used.</param>
00112 /// <param name="progressAction">An <see cref="Action"/> that will be invoked as the trees are
      processed.</param>
00113 /// <param name="useParallelOptimisation">If this is <c>true</c>, parts of the consensus computation
      will be parallelised.  This will, however, increase the number of computations that need to be
      performed.  The advantages will differ based on the number of trees, how discordant they are, and the
      characteristics of the processor.</param>
00114 /// <returns>A rooted consensus tree.</returns>
00115         public static TreeNode GetConsensus(this IEnumerable<TreeNode> trees, bool rooted, bool
      clockLike, double threshold, bool useMedian, Action<double> progressAction = null, bool
      useParallelOptimisation = false)
00116         {
00117             Contract.Requires(trees != null);
00118
00119             Dictionary<string, List<double» splits = new Dictionary<string, List<double»();
00120
00121             int totalTrees = 0;
00122
00123             Split.LengthTypes lengthType = clockLike ?  Split.LengthTypes.Age :
      Split.LengthTypes.Length;
00124
00125             int count = -1;
00126
00127             if (trees is IReadOnlyList<TreeNode> list)
00128             {
00129                 count = list.Count;
00130             }
00131
00132             foreach (TreeNode tree in trees)
00133             {
00134                 List<Split> treeSplits = tree.GetSplits(lengthType);
00135
00136                 for (int i = 0; i < treeSplits.Count; i++)
00137                 {
00138                     if (splits.TryGetValue(treeSplits[i].Name, out List<double> splitLengths))
00139                     {
00140                         splits[treeSplits[i].Name].Add(treeSplits[i].Length);
00141                     }
00142                     else
00143                     {
00144                         splits.Add(treeSplits[i].Name, new List<double>() { treeSplits[i].Length });
00145                     }
00146                 }
00147
00148                 totalTrees++;
00149
00150                 if (count > 0)
00151                 {
00152                     progressAction?.Invoke((double)totalTrees / count * 0.5);
00153                 }
00154                 else
00155                 {
00156                     progressAction?.Invoke(totalTrees * 0.5);
```

```
00157                 }
00158             }
00159
00160             List<Split> orderedSplits = new List<Split>(from el in splits orderby el.Value.Count
      descending where ((double)el.Value.Count / (double)totalTrees) >= threshold select new Split(el.Key,
      (useMedian ?  el.Value.Median() :  el.Value.Average()), lengthType, ((double)el.Value.Count /
      (double)totalTrees)));
00161             List<Split> finalSplits;
00162
00163             if (!useParallelOptimisation)
00164             {
00165                 finalSplits = new List<Split>();
00166
00167                 for (int i = 0; i < orderedSplits.Count; i++)
00168                 {
00169                     if (orderedSplits[i].IsCompatible(finalSplits))
00170                     {
00171                         finalSplits.Add(orderedSplits[i]);
00172                     }
00173
00174                     if (i % Math.Max(1, orderedSplits.Count / 100) == 0)
00175                     {
00176                         if (count > 0)
00177                         {
00178                             progressAction?.Invoke(0.5 + 0.5 * (i + 1) / orderedSplits.Count);
00179                         }
00180                         else
00181                         {
00182                             progressAction?.Invoke(totalTrees * (0.5 + 0.5 * (i + 1) /
      orderedSplits.Count));
00183                         }
00184                     }
00185                 }
00186             }
00187             else
00188             {
00189                 static int getIndex(int i, int j, int n)
00190                 {
00191                     return n * (n - 1) / 2 - (n - i) * (n - i - 1) / 2 + j - i - 1;
00192                 }
00193
00194                 static (int i, int j) getIndices(int index, int n)
00195                 {
00196                     int i = n - 2 - (int)Math.Floor(Math.Sqrt(-8 * index + 4 * n * (n - 1) - 7) / 2 -
      0.5);
00197                     int j = index + i + 1 - n * (n - 1) / 2 + (n - i) * ((n - i) - 1) / 2;
00198                     return (i, j);
00199                 }
00200
00201                 bool[] areCompatibles = new bool[orderedSplits.Count * (orderedSplits.Count - 1) / 2];
00202
00203                 int progressCount = 0;
00204                 object progressLock = new object();
00205
00206                 int threadCount = Environment.ProcessorCount / 2;
00207
00208                 int elementsByThread = (int)Math.Ceiling((double)areCompatibles.Length / threadCount);
00209
00210                 Thread[] threads = new Thread[threadCount];
00211
00212                 for (int p = 0; p < threadCount; p++)
00213                 {
00214                     int min = p * elementsByThread;
00215                     int max = Math.Min((p + 1) * elementsByThread, areCompatibles.Length);
00216
00217                     threads[p] = new Thread(() =>
00218                     {
00219                         int lastReported = min;
00220
00221                         for (int k = min; k < max; k++)
00222                         {
00223                             (int i, int j) = getIndices(k, orderedSplits.Count);
00224
00225                             areCompatibles[k] = Split.AreCompatible(orderedSplits[i],
      orderedSplits[j]);
00226
00227                             if ((k - min) % Math.Max(1, (max - min) / 50) == 0)
00228                             {
00229                                 lock (progressLock)
00230                                 {
00231                                     progressCount += k - lastReported + 1;
00232                                     lastReported = k + 1;
00233
00234                                     if (count > 0)
00235                                     {
00236                                         progressAction?.Invoke(Math.Min(1, 0.5 + 0.5 * progressCount /
      areCompatibles.Length));
```

```
00237                                   }
00238                                   else
00239                                   {
00240                                       progressAction?.Invoke(Math.Min(1, totalTrees * (0.5 + 0.5 *
    progressCount / areCompatibles.Length)));
00241                                   }
00242                               }
00243                           }
00244                       }
00245
00246                       lock (progressLock)
00247                       {
00248                           progressCount += max - lastReported;
00249
00250                           if (count > 0)
00251                           {
00252                               progressAction?.Invoke(Math.Min(1, 0.5 + 0.5 * progressCount /
    areCompatibles.Length));
00253                           }
00254                           else
00255                           {
00256                               progressAction?.Invoke(Math.Min(1, totalTrees * (0.5 + 0.5 *
    progressCount / areCompatibles.Length)));
00257                           }
00258                       }
00259                   });
00260               }
00261
00262               for (int p = 0; p < threadCount; p++)
00263               {
00264                   threads[p].Start();
00265               }
00266
00267               for (int p = 0; p < threadCount; p++)
00268               {
00269                   threads[p].Join();
00270               }
00271
00272               List<int> finalSplitIndices = new List<int>();
00273
00274               for (int i = 0; i < orderedSplits.Count; i++)
00275               {
00276                   bool isCompatible = true;
00277
00278                   for (int j = 0; j < finalSplitIndices.Count; j++)
00279                   {
00280                       if (finalSplitIndices[j] != i)
00281                       {
00282                           if (!areCompatibles[getIndex(Math.Max(i, finalSplitIndices[j]),
    Math.Min(i, finalSplitIndices[j]), orderedSplits.Count)])
00283                           {
00284                               isCompatible = false;
00285                               break;
00286                           }
00287                       }
00288                   }
00289
00290                   if (isCompatible)
00291                   {
00292                       finalSplitIndices.Add(i);
00293                   }
00294               }
00295
00296               finalSplits = new List<Split>(finalSplitIndices.Count);
00297
00298               for (int i = 0; i < finalSplitIndices.Count; i++)
00299               {
00300                   finalSplits.Add(orderedSplits[finalSplitIndices[i]]);
00301               }
00302           }
00303
00304           if (count > 0)
00305           {
00306               progressAction(1);
00307           }
00308           else
00309           {
00310               progressAction(totalTrees);
00311           }
00312
00313           if (finalSplits.Count > 0)
00314           {
00315               return Split.BuildTree(finalSplits, rooted);
00316           }
00317           else
00318           {
00319               return null;
```

```
00320             }
00321         }
00322
00323 /// <summary>
00324 /// Reads the next non-whitespace character, taking into account quoting and escaping.
00325 /// </summary>
00326 /// <param name="reader">The <see cref="TextReader"/> to read from.</param>
00327 /// <param name="escaping">A <see cref="bool"/> indicating whether the next character will be
      escaped.</param>
00328 /// <param name="escaped">A <see cref="bool"/> indicating whether the current character will be
      escaped.</param>
00329 /// <param name="openQuotes">A <see cref="bool"/> indicating whether double quotes have been
      opened.</param>
00330 /// <param name="openApostrophe">A <see cref="bool"/> indicating whether single quotes have been
      opened.</param>
00331 /// <param name="eof">A <see cref="bool"/> indicating whether we have arrived at the end of the
      file.</param>
00332 /// <returns>The next non-whitespace character.</returns>
00333         public static char NextToken(this TextReader reader, ref bool escaping, out bool escaped, ref
      bool openQuotes, ref bool openApostrophe, out bool eof)
00334         {
00335             Contract.Requires(reader != null);
00336
00337             int i = reader.Read();
00338
00339             if (i < 0)
00340             {
00341                 eof = true;
00342                 escaped = false;
00343                 return (char)i;
00344             }
00345
00346             eof = false;
00347             char c = (char)i;
00348
00349             if (!escaping)
00350             {
00351                 escaped = false;
00352                 if (!openQuotes && !openApostrophe)
00353                 {
00354                     while (Char.IsWhiteSpace(c))
00355                     {
00356                         i = reader.Read();
00357
00358                         if (i < 0)
00359                         {
00360                             eof = true;
00361                             escaped = false;
00362                             return (char)i;
00363                         }
00364
00365                         c = (char)i;
00366                     }
00367
00368                     switch (c)
00369                     {
00370                         case '\\':
00371                             escaping = true;
00372                             break;
00373                         case '"':
00374                             openQuotes = true;
00375                             break;
00376                         case '\'':
00377                             openApostrophe = true;
00378                             break;
00379                     }
00380                 }
00381                 else if (openQuotes)
00382                 {
00383                     switch (c)
00384                     {
00385                         case '"':
00386                             openQuotes = false;
00387                             break;
00388                         case '\\':
00389                             escaping = true;
00390                             break;
00391                     }
00392                 }
00393                 else if (openApostrophe)
00394                 {
00395                     switch (c)
00396                     {
00397                         case '\'':
00398                             openApostrophe = false;
00399                             break;
00400                         case '\\':
```

```
00401                             escaping = true;
00402                             break;
00403                         }
00404                     }
00405                 }
00406                 else
00407                 {
00408                     escaping = false;
00409                     escaped = true;
00410                 }
00411 
00412                 return c;
00413             }
00414 
00415 /// <summary>
00416 /// Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.
00417 /// </summary>
00418 /// <param name="reader">The <see cref="TextReader"/> to read from.</param>
00419 /// <param name="eof">A <see cref="bool"/> indicating whether we have arrived at the end of the
      file.</param>
00420 /// <returns>The next word.</returns>
00421         public static string NextWord(this TextReader reader, out bool eof)
00422         {
00423             Contract.Requires(reader != null);
00424 
00425             StringBuilder sb = new StringBuilder();
00426 
00427             int c = reader.Read();
00428 
00429             while (c >= 0 && Char.IsWhiteSpace((char)c))
00430             {
00431                 c = reader.Read();
00432             }
00433 
00434             if (c >= 0)
00435             {
00436                 sb.Append((char)c);
00437             }
00438 
00439             if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00440             {
00441                 eof = false;
00442                 return sb.ToString();
00443             }
00444 
00445             c = reader.Peek();
00446 
00447             while (c >= 0 && !Char.IsWhiteSpace((char)c))
00448             {
00449                 if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00450                 {
00451                     break;
00452                 }
00453                 c = reader.Read();
00454                 sb.Append((char)c);
00455                 c = reader.Peek();
00456             }
00457 
00458             if (c < 0)
00459             {
00460                 eof = true;
00461             }
00462             else
00463             {
00464                 eof = false;
00465             }
00466 
00467             return sb.ToString();
00468         }
00469 
00470 /// <summary>
00471 /// Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.
00472 /// </summary>
00473 /// <param name="reader">The <see cref="TextReader"/> to read from.</param>
00474 /// <param name="eof">A <see cref="bool"/> indicating whether we have arrived at the end of the
      file.</param>
00475 /// <param name="headingTrivia">A string containing any whitespace that was discarding before the
      start of the word.</param>
00476 /// <returns>The next word.</returns>
00477         public static string NextWord(this TextReader reader, out bool eof, out string headingTrivia)
00478         {
00479             Contract.Requires(reader != null);
00480 
00481             StringBuilder sb = new StringBuilder();
00482 
00483             StringBuilder headingTriviaBuilder = new StringBuilder();
00484             StringBuilder trailingTriviaBuilder = new StringBuilder();
```

```
00485
00486             int c = reader.Read();
00487
00488             while (c >= 0 && Char.IsWhiteSpace((char)c))
00489             {
00490                 headingTriviaBuilder.Append((char)c);
00491                 c = reader.Read();
00492             }
00493
00494             headingTrivia = headingTriviaBuilder.ToString();
00495
00496             if (c >= 0)
00497             {
00498                 sb.Append((char)c);
00499             }
00500
00501             if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00502             {
00503                 eof = false;
00504                 return sb.ToString();
00505             }
00506
00507             c = reader.Peek();
00508
00509             while (c >= 0 && !Char.IsWhiteSpace((char)c))
00510             {
00511                 if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00512                 {
00513                     break;
00514                 }
00515                 c = reader.Read();
00516                 sb.Append((char)c);
00517                 c = reader.Peek();
00518             }
00519
00520             if (c < 0)
00521             {
00522                 eof = true;
00523             }
00524             else
00525             {
00526                 eof = false;
00527             }
00528
00529             return sb.ToString();
00530         }
00531     }
00532 }
```

8.4 NcbiAsnBer.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.IO;
00005 using System.Text;
00006
00007 namespace PhyloTree.Formats
00008 {
00009 /// <summary>
00010 /// Contains methods to read and write trees in the NCBI ASN.1 binary format.<br/>
00011 /// <b>Note</b>:  this is a hackish reverse-engineering of the NCBI binary ASN format.  A lot of this
      is derived by assumptions and observations.
00012 /// </summary>
00013     public static class NcbiAsnBer
00014     {
00015 /// <summary>
00016 /// Tags indicating object types.
00017 /// </summary>
00018         internal enum ByteTags
00019         {
00020 /// <summary>
00021 /// The start of a generic object.  The object must be closed by two <see cref="EndOfContext"/> bytes.
00022 /// </summary>
00023             ObjectStart = 0x30,
00024
00025 /// <summary>
00026 /// The start of an array.  The array must be closed by two <see cref="EndOfContext"/> bytes.
00027 /// </summary>
00028             ArrayStart = 0x31,
00029
00030 /// <summary>
00031 /// A length value indicating that the object has an unspecified length.
```

```
00032 /// </summary>
00033             UndefinedLength = 0x80,
00034 
00035 /// <summary>
00036 /// Tag used to close objects with unspecified length.  Two of these are required to close each
      object.
00037 /// </summary>
00038             EndOfContext = 0x00,
00039 
00040 /// <summary>
00041 /// Indicates that the object is a string (UTF8-encoded, probably).
00042 /// </summary>
00043             String = 0x1A,
00044 
00045 /// <summary>
00046 /// Indicates that the object is an integer.
00047 /// </summary>
00048             Int = 0x02,
00049 
00050 /// <summary>
00051 /// Specifies the <c>treetype</c> property defined in the NCBI ASN.1 tree format.
00052 /// </summary>
00053             TreeType = 0xA0,
00054 
00055 /// <summary>
00056 /// Specifies the <c>fdict</c> property (feature dictionary) defined in the NCBI ASN.1 tree format.
00057 /// </summary>
00058             FDict = 0xA1,
00059 
00060 /// <summary>
00061 /// Specifies the <c>nodes</c> property (list of nodes) defined in the NCBI ASN.1 tree format.
00062 /// </summary>
00063             Nodes = 0xA2,
00064 
00065 /// <summary>
00066 /// Specifies the <c>label</c> property defined in the NCBI ASN.1 tree format.
00067 /// </summary>
00068             Label = 0xA3,
00069 
00070 /// <summary>
00071 /// Specifies the ID of a feature.
00072 /// </summary>
00073             FeatureId = 0xA0,
00074 
00075 /// <summary>
00076 /// Specifies the name of a feature.
00077 /// </summary>
00078             FeatureName = 0xA1,
00079 
00080 /// <summary>
00081 /// Specifies the ID of a node.
00082 /// </summary>
00083             NodeId = 0xA0,
00084 
00085 /// <summary>
00086 /// Specifies the parent of a node.
00087 /// </summary>
00088             NodeParent = 0xA1,
00089 
00090 /// <summary>
00091 /// Specifies the features of a node.
00092 /// </summary>
00093             NodeFeatures = 0xA2,
00094 
00095 /// <summary>
00096 /// Specifies the ID of a feature of a node.
00097 /// </summary>
00098             NodeFeatureId = 0xA0,
00099 
00100 /// <summary>
00101 /// Specifies the value of a fetuare of a node.
00102 /// </summary>
00103             NodeFeatureValue = 0xA1
00104         }
00105 
00106 /// <summary>
00107 /// Parses a tree from an NCBI ASN.1 binary format file.  Note that the tree can only contain a single
      file, and this method will always return a collection with a single element.
00108 /// </summary>
00109 /// <param name="inputFile">The path to the input file.</param>
00110 /// <returns>A <see cref="IEnumerable{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00111         public static IEnumerable<TreeNode> ParseTrees(string inputFile)
00112         {
00113             yield return ParseAllTrees(inputFile)[0];
00114         }
00115 
```

```
00116 /// <summary>
00117 /// Parses a tree from an NCBI ASN.1 binary format file.  Note that the tree can only contain a single
      file, and this method will always return a collection with a single element.
00118 /// </summary>
00119 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00120 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00121 /// <returns>A <see cref="IEnumerable{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00122         public static IEnumerable<TreeNode> ParseTrees(Stream inputStream, bool keepOpen = false)
00123         {
00124             yield return ParseAllTrees(inputStream, keepOpen)[0];
00125         }
00126 
00127 /// <summary>
00128 /// Parses a tree from an NCBI ASN.1 binary format file.  Note that the tree can only contain a single
      file, and this method will always return a list with a single element.
00129 /// </summary>
00130 /// <param name="inputFile">The path to the input file.</param>
00131 /// <returns>A <see cref="List{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00132         public static List<TreeNode> ParseAllTrees(string inputFile)
00133         {
00134             using FileStream stream = File.OpenRead(inputFile);
00135             using BinaryReader reader = new BinaryReader(stream);
00136             return new List<TreeNode>() { ParseTree(reader) };
00137         }
00138 
00139 /// <summary>
00140 /// Parses a tree from an NCBI ASN.1 binary format file.  Note that the tree can only contain a single
      file, and this method will always return a list with a single element.
00141 /// </summary>
00142 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00143 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00144 /// <returns>A <see cref="List{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00145         public static List<TreeNode> ParseAllTrees(Stream inputStream, bool keepOpen = false)
00146         {
00147             using BinaryReader reader = new BinaryReader(inputStream, Encoding.UTF8, keepOpen);
00148             return new List<TreeNode>() { ParseTree(reader) };
00149         }
00150 
00151 /// <summary>
00152 /// Parses a tree from a <see cref="BinaryReader"/> reading a stream in NCBI ASN.1 binary format into
      a <see cref="TreeNode"/> object.
00153 /// </summary>
00154 /// <param name="reader">The <see cref="BinaryReader"/> that reads a stream in NCBI ASN.1 binary
      format.</param>
00155 /// <returns>The parsed <see cref="TreeNode"/> object.</returns>
00156         public static TreeNode ParseTree(BinaryReader reader)
00157         {
00158             Contract.Requires(reader != null);
00159 
00160             byte currByte = reader.ReadByte();
00161             AssertByte(currByte, ByteTags.ObjectStart);
00162 
00163             currByte = reader.ReadByte();
00164             AssertByte(currByte, ByteTags.UndefinedLength);
00165 
00166             currByte = reader.ReadByte();
00167 
00168             string treetype = null;
00169 
00170             if (currByte == (byte)ByteTags.TreeType)
00171             {
00172                 currByte = reader.ReadByte();
00173                 AssertByte(currByte, ByteTags.UndefinedLength);
00174 
00175                 currByte = reader.ReadByte();
00176                 AssertByte(currByte, ByteTags.String);
00177 
00178                 int length = ReadLength(reader);
00179 
00180                 treetype = ReadString(reader, length);
00181 
00182                 currByte = reader.ReadByte();
00183                 AssertByte(currByte, ByteTags.EndOfContext);
00184                 currByte = reader.ReadByte();
00185                 AssertByte(currByte, ByteTags.EndOfContext);
00186 
00187                 currByte = reader.ReadByte();
00188             }
00189 
00190             AssertByte(currByte, ByteTags.FDict);
00191 
00192             currByte = reader.ReadByte();
```

```
00193             AssertByte(currByte, ByteTags.UndefinedLength);
00194
00195             currByte = reader.ReadByte();
00196             AssertByte(currByte, ByteTags.ArrayStart);
00197
00198             currByte = reader.ReadByte();
00199             AssertByte(currByte, ByteTags.UndefinedLength);
00200
00201             Dictionary<int, string> features = new Dictionary<int, string>();
00202
00203             bool finishedFeatures = false;
00204
00205             while (!finishedFeatures)
00206             {
00207                 currByte = reader.ReadByte();
00208
00209                 if (currByte == (byte)ByteTags.ObjectStart)
00210                 {
00211                     currByte = reader.ReadByte();
00212                     AssertByte(currByte, ByteTags.UndefinedLength);
00213
00214                     currByte = reader.ReadByte();
00215                     AssertByte(currByte, ByteTags.FeatureId);
00216
00217                     currByte = reader.ReadByte();
00218                     AssertByte(currByte, ByteTags.UndefinedLength);
00219
00220                     currByte = reader.ReadByte();
00221                     AssertByte(currByte, ByteTags.Int);
00222
00223                     int idLength = ReadLength(reader);
00224                     int id = ReadInt(reader, idLength);
00225
00226                     currByte = reader.ReadByte();
00227                     AssertByte(currByte, ByteTags.EndOfContext);
00228                     currByte = reader.ReadByte();
00229                     AssertByte(currByte, ByteTags.EndOfContext);
00230
00231                     currByte = reader.ReadByte();
00232                     AssertByte(currByte, ByteTags.FeatureName);
00233
00234                     currByte = reader.ReadByte();
00235                     AssertByte(currByte, ByteTags.UndefinedLength);
00236
00237                     currByte = reader.ReadByte();
00238                     AssertByte(currByte, ByteTags.String);
00239
00240                     int nameLength = ReadLength(reader);
00241                     string name = ReadString(reader, nameLength);
00242
00243                     currByte = reader.ReadByte();
00244                     AssertByte(currByte, ByteTags.EndOfContext);
00245                     currByte = reader.ReadByte();
00246                     AssertByte(currByte, ByteTags.EndOfContext);
00247
00248                     currByte = reader.ReadByte();
00249                     AssertByte(currByte, ByteTags.EndOfContext);
00250                     currByte = reader.ReadByte();
00251                     AssertByte(currByte, ByteTags.EndOfContext);
00252
00253                     features[id] = name;
00254                 }
00255                 else
00256                 {
00257                     AssertByte(currByte, ByteTags.EndOfContext);
00258                     currByte = reader.ReadByte();
00259                     AssertByte(currByte, ByteTags.EndOfContext);
00260
00261                     finishedFeatures = true;
00262                 }
00263             }
00264
00265             currByte = reader.ReadByte();
00266             AssertByte(currByte, ByteTags.EndOfContext);
00267             currByte = reader.ReadByte();
00268             AssertByte(currByte, ByteTags.EndOfContext);
00269
00270             currByte = reader.ReadByte();
00271             AssertByte(currByte, ByteTags.Nodes);
00272
00273             currByte = reader.ReadByte();
00274             AssertByte(currByte, ByteTags.UndefinedLength);
00275
00276             currByte = reader.ReadByte();
00277             AssertByte(currByte, ByteTags.ArrayStart);
00278
00279             currByte = reader.ReadByte();
```

```
00280             AssertByte(currByte, ByteTags.UndefinedLength);
00281
00282             bool finishedNodes = false;
00283             Dictionary<int, (TreeNode node, int?  parent)> nodes = new Dictionary<int, (TreeNode node,
      int?  parent)>();
00284
00285             while (!finishedNodes)
00286             {
00287                 currByte = reader.ReadByte();
00288
00289                 if (currByte == (byte)ByteTags.ObjectStart)
00290                 {
00291                     currByte = reader.ReadByte();
00292                     AssertByte(currByte, ByteTags.UndefinedLength);
00293
00294                     currByte = reader.ReadByte();
00295                     AssertByte(currByte, ByteTags.NodeId);
00296
00297                     currByte = reader.ReadByte();
00298                     AssertByte(currByte, ByteTags.UndefinedLength);
00299
00300                     currByte = reader.ReadByte();
00301                     AssertByte(currByte, ByteTags.Int);
00302
00303                     int idLength = ReadLength(reader);
00304                     int id = ReadInt(reader, idLength);
00305
00306                     currByte = reader.ReadByte();
00307                     AssertByte(currByte, ByteTags.EndOfContext);
00308                     currByte = reader.ReadByte();
00309                     AssertByte(currByte, ByteTags.EndOfContext);
00310
00311                     currByte = reader.ReadByte();
00312
00313                     int?  parent = null;
00314
00315                     TreeNode node = new TreeNode(null);
00316
00317                     if (currByte == (byte)ByteTags.NodeParent)
00318                     {
00319                         currByte = reader.ReadByte();
00320                         AssertByte(currByte, ByteTags.UndefinedLength);
00321
00322                         currByte = reader.ReadByte();
00323                         AssertByte(currByte, ByteTags.Int);
00324
00325                         int parentLength = ReadLength(reader);
00326                         parent = ReadInt(reader, parentLength);
00327
00328                         currByte = reader.ReadByte();
00329                         AssertByte(currByte, ByteTags.EndOfContext);
00330                         currByte = reader.ReadByte();
00331                         AssertByte(currByte, ByteTags.EndOfContext);
00332
00333                         currByte = reader.ReadByte();
00334                     }
00335
00336                     if (currByte == (byte)ByteTags.NodeFeatures)
00337                     {
00338                         currByte = reader.ReadByte();
00339                         AssertByte(currByte, ByteTags.UndefinedLength);
00340
00341                         currByte = reader.ReadByte();
00342                         AssertByte(currByte, ByteTags.ArrayStart);
00343
00344                         currByte = reader.ReadByte();
00345                         AssertByte(currByte, ByteTags.UndefinedLength);
00346
00347                         bool finishedNodeFeatures = false;
00348
00349                         while (!finishedNodeFeatures)
00350                         {
00351                             currByte = reader.ReadByte();
00352
00353                             if (currByte == (byte)ByteTags.ObjectStart)
00354                             {
00355                                 currByte = reader.ReadByte();
00356                                 AssertByte(currByte, ByteTags.UndefinedLength);
00357
00358                                 currByte = reader.ReadByte();
00359                                 AssertByte(currByte, ByteTags.NodeFeatureId);
00360
00361                                 currByte = reader.ReadByte();
00362                                 AssertByte(currByte, ByteTags.UndefinedLength);
00363
00364                                 currByte = reader.ReadByte();
00365                                 AssertByte(currByte, ByteTags.Int);
```

```
00366
00367                                   int featureIdLength = ReadLength(reader);
00368                                   int featureId = ReadInt(reader, featureIdLength);
00369
00370                                   currByte = reader.ReadByte();
00371                                   AssertByte(currByte, ByteTags.EndOfContext);
00372                                   currByte = reader.ReadByte();
00373                                   AssertByte(currByte, ByteTags.EndOfContext);
00374
00375                                   currByte = reader.ReadByte();
00376                                   AssertByte(currByte, ByteTags.NodeFeatureValue);
00377
00378                                   currByte = reader.ReadByte();
00379                                   AssertByte(currByte, ByteTags.UndefinedLength);
00380
00381                                   currByte = reader.ReadByte();
00382                                   AssertByte(currByte, ByteTags.String);
00383
00384                                   int featureValueLength = ReadLength(reader);
00385                                   string featureValue = ReadString(reader, featureValueLength);
00386
00387                                   object valueObject;
00388
00389                                   if (!features[featureId].Equals("label",
      StringComparison.OrdinalIgnoreCase) && double.TryParse(featureValue,
      System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out double
      doubleValue))
00390                                   {
00391                                       valueObject = doubleValue;
00392                                   }
00393                                   else
00394                                   {
00395                                       valueObject = featureValue;
00396                                   }
00397
00398                                   currByte = reader.ReadByte();
00399                                   AssertByte(currByte, ByteTags.EndOfContext);
00400                                   currByte = reader.ReadByte();
00401                                   AssertByte(currByte, ByteTags.EndOfContext);
00402
00403                                   currByte = reader.ReadByte();
00404                                   AssertByte(currByte, ByteTags.EndOfContext);
00405                                   currByte = reader.ReadByte();
00406                                   AssertByte(currByte, ByteTags.EndOfContext);
00407
00408                                   node.Attributes[features[featureId]] = valueObject;
00409                               }
00410                               else
00411                               {
00412                                   AssertByte(currByte, ByteTags.EndOfContext);
00413                                   currByte = reader.ReadByte();
00414                                   AssertByte(currByte, ByteTags.EndOfContext);
00415
00416                                   finishedNodeFeatures = true;
00417                               }
00418                           }
00419
00420                           currByte = reader.ReadByte();
00421                           AssertByte(currByte, ByteTags.EndOfContext);
00422                           currByte = reader.ReadByte();
00423                           AssertByte(currByte, ByteTags.EndOfContext);
00424
00425                           currByte = reader.ReadByte();
00426                       }
00427
00428                       AssertByte(currByte, ByteTags.EndOfContext);
00429                       currByte = reader.ReadByte();
00430                       AssertByte(currByte, ByteTags.EndOfContext);
00431
00432                       nodes[id] = (node, parent);
00433                   }
00434                   else
00435                   {
00436                       AssertByte(currByte, ByteTags.EndOfContext);
00437                       currByte = reader.ReadByte();
00438                       AssertByte(currByte, ByteTags.EndOfContext);
00439
00440                       finishedNodes = true;
00441                   }
00442               }
00443
00444               currByte = reader.ReadByte();
00445
00446               string label = null;
00447
00448               if (currByte == (byte)ByteTags.Label)
00449               {
```

```
00450                 currByte = reader.ReadByte();
00451                 AssertByte(currByte, ByteTags.UndefinedLength);
00452 
00453                 currByte = reader.ReadByte();
00454                 AssertByte(currByte, ByteTags.String);
00455 
00456                 int length = ReadLength(reader);
00457 
00458                 label = ReadString(reader, length);
00459 
00460                 currByte = reader.ReadByte();
00461                 AssertByte(currByte, ByteTags.EndOfContext);
00462                 currByte = reader.ReadByte();
00463                 AssertByte(currByte, ByteTags.EndOfContext);
00464             }
00465 
00466             TreeNode tree = null;
00467 
00468             foreach (KeyValuePair<int, (TreeNode node, int?  parent)> kvp in nodes)
00469             {
00470                 if (kvp.Value.parent != null)
00471                 {
00472                     int parent = kvp.Value.parent.Value;
00473 
00474                     nodes[parent].node.Children.Add(kvp.Value.node);
00475                     kvp.Value.node.Parent = nodes[parent].node;
00476                 }
00477                 else
00478                 {
00479                     tree = kvp.Value.node;
00480                 }
00481 
00482                 if (kvp.Value.node.Attributes.TryGetValue("dist", out object distValue) && distValue
    is double branchLength)
00483                 {
00484                     kvp.Value.node.Length = branchLength;
00485                 }
00486             }
00487 
00488             foreach (KeyValuePair<int, (TreeNode node, int?  parent)> kvp in nodes)
00489             {
00490                 if (kvp.Value.node.Children.Count == 0)
00491                 {
00492                     if (kvp.Value.node.Attributes.TryGetValue("label", out object labelValue) &&
    labelValue is string nodeLabel)
00493                     {
00494                         kvp.Value.node.Name = nodeLabel;
00495                     }
00496                 }
00497             }
00498 
00499             if (tree != null)
00500             {
00501                 if (!string.IsNullOrEmpty(treetype))
00502                 {
00503                     tree.Attributes["Tree-treetype"] = treetype;
00504                 }
00505 
00506                 if (!string.IsNullOrEmpty(label))
00507                 {
00508                     tree.Attributes["TreeName"] = label;
00509                 }
00510             }
00511 
00512             return tree;
00513         }
00514 
00515 /// <summary>
00516 /// Writes a <see cref="TreeNode"/> to a file in NCBI ASN.1 binary format.
00517 /// </summary>
00518 /// <param name="tree">The tree to write.</param>
00519 /// <param name="outputFile">The path to the output file.</param>
00520 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
    ASN.1 tree format.</param>
00521 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
    format.</param>
00522         public static void WriteTree(TreeNode tree, string outputFile, string treeType = null, string
    label = null)
00523         {
00524             using FileStream stream = File.Create(outputFile);
00525             WriteTree(tree, stream, false, treeType, label);
00526         }
00527 
00528 /// <summary>
00529 /// Writes a <see cref="TreeNode"/> to a file in NCBI ASN.1 binary format.
00530 /// </summary>
00531 /// <param name="tree">The tree to write.</param>
```

```
00532 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00533 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00534 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00535 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00536         public static void WriteTree(TreeNode tree, Stream outputStream, bool keepOpen = false, string
      treeType = null, string label = null)
00537         {
00538             using BinaryWriter sw = new BinaryWriter(outputStream, Encoding.UTF8, keepOpen);
00539             WriteTree(tree, sw, treeType, label);
00540         }
00541 
00542 /// <summary>
00543 /// Writes a collection of <see cref="TreeNode"/>s to a file in NCBI ASN.1 binary format.  Note that
      only one tree can be saved in each file; if the collection contains more than one tree an exception
      will be thrown.
00544 /// </summary>
00545 /// <param name="trees">The collection of trees to write.  If this contains more than one tree, an
      exception will be thrown.</param>
00546 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00547 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00548 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00549 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00550         public static void WriteAllTrees(IEnumerable<TreeNode> trees, Stream outputStream, bool
      keepOpen = false, string treeType = null, string label = null)
00551         {
00552             Contract.Requires(trees != null);
00553 
00554             bool firstTree = true;
00555 
00556             foreach (TreeNode tree in trees)
00557             {
00558                 if (firstTree)
00559                 {
00560                     WriteTree(tree, outputStream, keepOpen, treeType, label);
00561                     firstTree = false;
00562                 }
00563                 else
00564                 {
00565                     throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved
      in an NCBI ASN.1 file!");
00566                 }
00567             }
00568         }
00569 
00570 /// <summary>
00571 /// Writes a collection of <see cref="TreeNode"/>s to a file in NCBI ASN.1 binary format.  Note that
      only one tree can be saved in each file; if the collection contains more than one tree an exception
      will be thrown.
00572 /// </summary>
00573 /// <param name="trees">The collection of trees to write.  If this contains more than one tree, an
      exception will be thrown.</param>
00574 /// <param name="outputFile">The path to the output file.</param>
00575 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00576 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00577         public static void WriteAllTrees(IEnumerable<TreeNode> trees, string outputFile, string
      treeType = null, string label = null)
00578         {
00579             Contract.Requires(trees != null);
00580 
00581             bool firstTree = true;
00582 
00583             foreach (TreeNode tree in trees)
00584             {
00585                 if (firstTree)
00586                 {
00587                     WriteTree(tree, outputFile, treeType, label);
00588                     firstTree = false;
00589                 }
00590                 else
00591                 {
00592                     throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved
      in an NCBI ASN.1 file!");
00593                 }
00594             }
00595         }
00596 
00597 /// <summary>
00598 /// Writes a list of <see cref="TreeNode"/>s to a file in NCBI ASN.1 binary format.  Note that only
      one tree can be saved in each file; if the tree contains more than one tree an exception will be
```

```
    thrown.
00599 /// </summary>
00600 /// <param name="trees">The list of trees to write.  If this contains more than one tree, an exception
    will be thrown.</param>
00601 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00602 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
    after the end of this method.</param>
00603 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
    ASN.1 tree format.</param>
00604 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
    format.</param>
00605         public static void WriteAllTrees(List<TreeNode> trees, Stream outputStream, bool keepOpen =
    false, string treeType = null, string label = null)
00606         {
00607             Contract.Requires(trees != null);
00608 
00609             if (trees.Count > 1)
00610             {
00611                 throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved in an
    NCBI ASN.1 file!");
00612             }
00613 
00614             WriteTree(trees[0], outputStream, keepOpen, treeType, label);
00615         }
00616 
00617 /// <summary>
00618 /// Writes a list of <see cref="TreeNode"/>s to a file in NCBI ASN.1 binary format.  Note that only
    one tree can be saved in each file; if the list contains more than one tree an exception will be
    thrown.
00619 /// </summary>
00620 /// <param name="trees">The list of trees to write.  If this contains more than one tree, an exception
    will be thrown.</param>
00621 /// <param name="outputFile">The path to the output file.</param>
00622 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
    ASN.1 tree format.</param>
00623 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
    format.</param>
00624         public static void WriteAllTrees(List<TreeNode> trees, string outputFile, string treeType =
    null, string label = null)
00625         {
00626             Contract.Requires(trees != null);
00627 
00628             if (trees.Count > 1)
00629             {
00630                 throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved in an
    NCBI ASN.1 file!");
00631             }
00632 
00633             WriteTree(trees[0], outputFile, treeType, label);
00634         }
00635 
00636 /// <summary>
00637 /// Writes a <see cref="TreeNode"/> to a <see cref="BinaryWriter"/> in NCBI ASN.1 binary format.
00638 /// </summary>
00639 /// <param name="tree">The tree to write.</param>
00640 /// <param name="writer">The <see cref="BinaryWriter"/> on which the tree will be written..</param>
00641 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
    ASN.1 tree format.</param>
00642 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
    format.</param>
00643         public static void WriteTree(TreeNode tree, BinaryWriter writer, string treeType = null,
    string label = null)
00644         {
00645             Contract.Requires(writer != null);
00646 
00647             if (tree == null)
00648             {
00649                 throw new ArgumentNullException(nameof(tree));
00650             }
00651 
00652             writer.Write((byte)ByteTags.ObjectStart);
00653             writer.Write((byte)ByteTags.UndefinedLength);
00654 
00655             if (!string.IsNullOrEmpty(treeType))
00656             {
00657                 writer.Write((byte)ByteTags.TreeType);
00658                 writer.Write((byte)ByteTags.UndefinedLength);
00659 
00660                 WriteString(writer, treeType);
00661 
00662                 writer.Write((byte)ByteTags.EndOfContext);
00663                 writer.Write((byte)ByteTags.EndOfContext);
00664             }
00665 
00666             writer.Write((byte)ByteTags.FDict);
00667             writer.Write((byte)ByteTags.UndefinedLength);
00668 
```

```
00669             writer.Write((byte)ByteTags.ArrayStart);
00670             writer.Write((byte)ByteTags.UndefinedLength);
00671 
00672             HashSet<string> attributes = new HashSet<string>(StringComparer.OrdinalIgnoreCase) {
      "label", "dist" };
00673 
00674             foreach (TreeNode node in tree.GetChildrenRecursiveLazy())
00675             {
00676                 foreach (string attribute in node.Attributes.Keys)
00677                 {
00678                     attributes.Add(attribute);
00679                 }
00680             }
00681 
00682             Dictionary<string, int> featureIndex = new Dictionary<string,
      int>(StringComparer.OrdinalIgnoreCase);
00683             int currInd = 0;
00684 
00685             foreach (string attribute in attributes)
00686             {
00687                 featureIndex.Add(attribute, currInd);
00688 
00689                 writer.Write((byte)ByteTags.ObjectStart);
00690                 writer.Write((byte)ByteTags.UndefinedLength);
00691 
00692                 writer.Write((byte)ByteTags.FeatureId);
00693                 writer.Write((byte)ByteTags.UndefinedLength);
00694 
00695                 WriteInt(writer, currInd);
00696 
00697                 writer.Write((byte)ByteTags.EndOfContext);
00698                 writer.Write((byte)ByteTags.EndOfContext);
00699 
00700                 writer.Write((byte)ByteTags.FeatureName);
00701                 writer.Write((byte)ByteTags.UndefinedLength);
00702 
00703                 WriteString(writer, attribute);
00704 
00705                 writer.Write((byte)ByteTags.EndOfContext);
00706                 writer.Write((byte)ByteTags.EndOfContext);
00707 
00708                 writer.Write((byte)ByteTags.EndOfContext);
00709                 writer.Write((byte)ByteTags.EndOfContext);
00710 
00711                 currInd++;
00712             }
00713 
00714             writer.Write((byte)ByteTags.EndOfContext);
00715             writer.Write((byte)ByteTags.EndOfContext);
00716 
00717             writer.Write((byte)ByteTags.EndOfContext);
00718             writer.Write((byte)ByteTags.EndOfContext);
00719 
00720             writer.Write((byte)ByteTags.Nodes);
00721             writer.Write((byte)ByteTags.UndefinedLength);
00722 
00723             writer.Write((byte)ByteTags.ArrayStart);
00724             writer.Write((byte)ByteTags.UndefinedLength);
00725 
00726             Dictionary<TreeNode, int> nodeIndex = new Dictionary<TreeNode, int>();
00727             currInd = 0;
00728 
00729             foreach (TreeNode node in tree.GetChildrenRecursiveLazy())
00730             {
00731                 nodeIndex.Add(node, currInd);
00732 
00733                 writer.Write((byte)ByteTags.ObjectStart);
00734                 writer.Write((byte)ByteTags.UndefinedLength);
00735 
00736                 writer.Write((byte)ByteTags.NodeId);
00737                 writer.Write((byte)ByteTags.UndefinedLength);
00738 
00739                 WriteInt(writer, currInd);
00740 
00741                 writer.Write((byte)ByteTags.EndOfContext);
00742                 writer.Write((byte)ByteTags.EndOfContext);
00743 
00744                 if (node.Parent != null)
00745                 {
00746                     writer.Write((byte)ByteTags.NodeParent);
00747                     writer.Write((byte)ByteTags.UndefinedLength);
00748 
00749                     WriteInt(writer, nodeIndex[node.Parent]);
00750 
00751                     writer.Write((byte)ByteTags.EndOfContext);
00752                     writer.Write((byte)ByteTags.EndOfContext);
00753                 }
```

```
00754
00755                 List<(int, string)> nodeFeatures = new List<(int, string)>();
00756
00757                 bool hasLabel = false;
00758                 bool hasDist = false;
00759
00760                 foreach (KeyValuePair<string, object> attribute in node.Attributes)
00761                 {
00762                     if (attribute.Value != null)
00763                     {
00764                         int attributeIndex = featureIndex[attribute.Key];
00765
00766                         if (attribute.Value is string stringValue &&
      !string.IsNullOrEmpty(stringValue))
00767                         {
00768                             nodeFeatures.Add((attributeIndex, stringValue));
00769
00770                             if (attribute.Key.Equals("label", StringComparison.OrdinalIgnoreCase))
00771                             {
00772                                 hasLabel = true;
00773                             }
00774
00775                             if (attribute.Key.Equals("dist", StringComparison.OrdinalIgnoreCase))
00776                             {
00777                                 hasDist = true;
00778                             }
00779                         }
00780                         else if (attribute.Value is double doubleValue && !double.IsNaN(doubleValue))
00781                         {
00782                             nodeFeatures.Add((attributeIndex,
      doubleValue.ToString(System.Globalization.CultureInfo.InvariantCulture)));
00783
00784                             if (attribute.Key.Equals("label", StringComparison.OrdinalIgnoreCase))
00785                             {
00786                                 hasLabel = true;
00787                             }
00788
00789                             if (attribute.Key.Equals("dist", StringComparison.OrdinalIgnoreCase))
00790                             {
00791                                 hasDist = true;
00792                             }
00793                         }
00794                     }
00795                 }
00796
00797                 if (!hasLabel && node.Name != null)
00798                 {
00799                     nodeFeatures.Add((featureIndex["label"], node.Name));
00800                 }
00801
00802                 if (!hasDist && !double.IsNaN(node.Length))
00803                 {
00804                     nodeFeatures.Add((featureIndex["dist"],
      node.Length.ToString(System.Globalization.CultureInfo.InvariantCulture)));
00805                 }
00806
00807                 if (nodeFeatures.Count > 0)
00808                 {
00809                     writer.Write((byte)ByteTags.NodeFeatures);
00810                     writer.Write((byte)ByteTags.UndefinedLength);
00811
00812                     writer.Write((byte)ByteTags.ArrayStart);
00813                     writer.Write((byte)ByteTags.UndefinedLength);
00814
00815                     for (int i = 0; i < nodeFeatures.Count; i++)
00816                     {
00817                         writer.Write((byte)ByteTags.ObjectStart);
00818                         writer.Write((byte)ByteTags.UndefinedLength);
00819
00820                         writer.Write((byte)ByteTags.NodeFeatureId);
00821                         writer.Write((byte)ByteTags.UndefinedLength);
00822
00823                         WriteInt(writer, nodeFeatures[i].Item1);
00824
00825                         writer.Write((byte)ByteTags.EndOfContext);
00826                         writer.Write((byte)ByteTags.EndOfContext);
00827
00828                         writer.Write((byte)ByteTags.NodeFeatureValue);
00829                         writer.Write((byte)ByteTags.UndefinedLength);
00830
00831                         WriteString(writer, nodeFeatures[i].Item2);
00832
00833                         writer.Write((byte)ByteTags.EndOfContext);
00834                         writer.Write((byte)ByteTags.EndOfContext);
00835
00836                         writer.Write((byte)ByteTags.EndOfContext);
00837                         writer.Write((byte)ByteTags.EndOfContext);
```

```
00838                     }
00839
00840                     writer.Write((byte)ByteTags.EndOfContext);
00841                     writer.Write((byte)ByteTags.EndOfContext);
00842
00843                     writer.Write((byte)ByteTags.EndOfContext);
00844                     writer.Write((byte)ByteTags.EndOfContext);
00845                 }
00846
00847                 writer.Write((byte)ByteTags.EndOfContext);
00848                 writer.Write((byte)ByteTags.EndOfContext);
00849
00850                 currInd++;
00851             }
00852
00853             writer.Write((byte)ByteTags.EndOfContext);
00854             writer.Write((byte)ByteTags.EndOfContext);
00855
00856             writer.Write((byte)ByteTags.EndOfContext);
00857             writer.Write((byte)ByteTags.EndOfContext);
00858
00859             if (!string.IsNullOrEmpty(label))
00860             {
00861                 writer.Write((byte)ByteTags.Label);
00862                 writer.Write((byte)ByteTags.UndefinedLength);
00863
00864                 WriteString(writer, label);
00865
00866                 writer.Write((byte)ByteTags.EndOfContext);
00867                 writer.Write((byte)ByteTags.EndOfContext);
00868             }
00869             else if (tree.Attributes.TryGetValue("TreeName", out object treeNameValue) &&
    treeNameValue != null && treeNameValue is string treeName && !string.IsNullOrEmpty(treeName))
00870             {
00871                 writer.Write((byte)ByteTags.Label);
00872                 writer.Write((byte)ByteTags.UndefinedLength);
00873
00874                 WriteString(writer, treeName);
00875
00876                 writer.Write((byte)ByteTags.EndOfContext);
00877                 writer.Write((byte)ByteTags.EndOfContext);
00878             }
00879
00880             writer.Write((byte)ByteTags.EndOfContext);
00881             writer.Write((byte)ByteTags.EndOfContext);
00882         }
00883
00884 /// <summary>
00885 /// Throws an exception if the byte that has been read does not correspond to the tag that was
    expected.
00886 /// </summary>
00887 /// <param name="observed">The byte that has been read.</param>
00888 /// <param name="expected">The tag that was expected.</param>
00889         private static void AssertByte(byte observed, ByteTags expected)
00890         {
00891             if (observed != (byte)expected)
00892             {
00893                 throw new Exception("Unexpected byte:  0x" + observed.ToString("X2",
    System.Globalization.CultureInfo.InvariantCulture) + "!  Was expecting:  " + expected.ToString() +
    "(0x" + ((byte)expected).ToString("X2", System.Globalization.CultureInfo.InvariantCulture) + ").");
00894             }
00895         }
00896
00897 /// <summary>
00898 /// Reads an UTF8-encoded <see cref="string"/> with the specified length from a <see
    cref="BinaryReader"/>.
00899 /// </summary>
00900 /// <param name="reader">The <see cref="BinaryReader"/> from which the string will be read.</param>
00901 /// <param name="length">The length of the string to read.</param>
00902 /// <returns>The string that has been read.</returns>
00903         private static string ReadString(BinaryReader reader, int length)
00904         {
00905             byte[] buffer = new byte[length];
00906
00907             reader.Read(buffer, 0, length);
00908
00909             // Wishful thinking.
00910             return System.Text.Encoding.UTF8.GetString(buffer);
00911         }
00912
00913 /// <summary>
00914 /// Writes an UTF8-encoded <see cref="string"/> to a <see cref="BinaryWriter"/>.
00915 /// </summary>
00916 /// <param name="writer">The <see cref="BinaryWriter"/> on which the string will be written.</param>
00917 /// <param name="str">The <see cref="string"/> to write.</param>
00918         private static void WriteString(BinaryWriter writer, string str)
00919         {
```

```
00920             writer.Write((byte)ByteTags.String);
00921
00922             byte[] bytes = System.Text.Encoding.UTF8.GetBytes(str);
00923
00924             WriteLength(writer, bytes.Length);
00925
00926             writer.Write(bytes);
00927         }
00928
00929 /// <summary>
00930 /// Writes an <see cref="int"/> to a <see cref="BinaryWriter"/>.
00931 /// </summary>
00932 /// <param name="writer">The <see cref="BinaryWriter"/> on which the <see cref="int"/> will be
      written.</param>
00933 /// <param name="value">The <see cref="int"/> to write.</param>
00934         private static void WriteInt(BinaryWriter writer, int value)
00935         {
00936             writer.Write((byte)ByteTags.Int);
00937
00938             WriteLength(writer, 4);
00939
00940             // Not the optimal way to store this, but who cares.
00941             writer.Write((byte)((value » 24) & 0b11111111));
00942             writer.Write((byte)((value » 16) & 0b11111111));
00943             writer.Write((byte)((value » 8) & 0b11111111));
00944             writer.Write((byte)(value & 0b11111111));
00945         }
00946
00947 /// <summary>
00948 /// Writes a length to a <see cref="BinaryWriter"/>.
00949 /// </summary>
00950 /// <param name="writer">The <see cref="BinaryWriter"/> on which the <paramref name="length"/> will be
      written.</param>
00951 /// <param name="length">The length to write.</param>
00952         private static void WriteLength(BinaryWriter writer, int length)
00953         {
00954             if (length < 128)
00955             {
00956                 writer.Write((byte)length);
00957             }
00958             else
00959             {
00960                 int lengthLength = 1;
00961                 int shiftedLength = length » 8;
00962
00963                 while (shiftedLength != 0)
00964                 {
00965                     shiftedLength »= 8;
00966                     lengthLength++;
00967                 }
00968
00969                 byte lengthByte = 0b10000000;
00970                 lengthByte |= (byte)lengthLength;
00971
00972                 writer.Write(lengthByte);
00973
00974                 for (int i = 0; i < lengthLength; i++)
00975                 {
00976                     byte currByte = (byte)((length » (8 * (lengthLength - 1 - i))) & 0b11111111);
00977                     writer.Write(currByte);
00978                 }
00979             }
00980         }
00981
00982 /// <summary>
00983 /// Reads a length from a <see cref="BinaryReader"/>.
00984 /// </summary>
00985 /// <param name="reader">The <see cref="BinaryReader"/> from which the length will be read.</param>
00986 /// <returns>The length that has been read.</returns>
00987         private static int ReadLength(BinaryReader reader)
00988         {
00989             byte currByte = reader.ReadByte();
00990
00991             if ((currByte & 0b10000000) == 0)
00992             {
00993                 return currByte;
00994             }
00995             else
00996             {
00997                 int additionalBytes = currByte & 0b01111111;
00998
00999                 if (additionalBytes > 4)
01000                 {
01001                     // We could use a long or something even bigger, but most of the things we will
      want to use the length for have int indexers, thus it is better to fail directly here.
01002                     throw new OverflowException("The length specified in the ASN stream exceeds the
      capability of the Int32 type!");
```

```
01003                 }
01004
01005                 int length = 0;
01006
01007                 for (int i = 0; i < additionalBytes; i++)
01008                 {
01009                     byte digit = reader.ReadByte();
01010
01011                     if (additionalBytes == 4 && i == 0 && digit > 127)
01012                     {
01013                         throw new OverflowException("The length specified in the ASN stream exceeds
      the capability of the Int32 type!");
01014                     }
01015
01016                     length |= (digit << ((additionalBytes - 1 - i) * 8));
01017                 }
01018
01019                 return length;
01020             }
01021         }
01022
01023 /// <summary>
01024 /// Reads an <see cref="int"/> from a <see cref="BinaryReader"/>.
01025 /// </summary>
01026 /// <param name="reader">The <see cref="BinaryReader"/> from which the <see cref="int"/> will be
      read.</param>
01027 /// <param name="length">The length (in bytes) of the <see cref="int"/> to read.</param>
01028 /// <returns>The <see cref="int"/> that has been read.</returns>
01029         private static int ReadInt(BinaryReader reader, int length)
01030         {
01031             if (length > 4)
01032             {
01033                 // See comment for the length above.
01034                 throw new OverflowException("The integer specified in the ASN stream exceeds the
      capability of the Int32 type!");
01035             }
01036
01037             byte[] buffer = new byte[length];
01038
01039             reader.Read(buffer, 0, length);
01040
01041             bool needComplement = false;
01042
01043             if (length < 4 && ((buffer[0] & 0b10000000) != 0))
01044             {
01045                 needComplement = true;
01046             }
01047
01048             int value = 0;
01049
01050             for (int i = 0; i < length; i++)
01051             {
01052                 value |= (buffer[i] << ((length - 1 - i) * 8));
01053             }
01054
01055             // The problem here is that we need the 2's complement based on the number of bytes that
      are actually used to store the data.
01056             // Maybe I could get away with just setting the bits from the unused bytes to 1.
01057             if (needComplement)
01058             {
01059                 // Mask to the number of bytes used.
01060                 int maskPattern = 0;
01061
01062                 for (int i = 0; i < length; i++)
01063                 {
01064                     maskPattern |= (0b11111111 << ((length - 1 - i) * 8));
01065                 }
01066
01067                 // Perform the 2's complement and mask the unused bytes back to 0 to get the absolute
      value of the number.
01068                 value = ((~value) + 1) & maskPattern;
01069
01070                 // Negate it.
01071                 value = -value;
01072             }
01073
01074             return value;
01075         }
01076
01077     }
01078 }
```

8.5 NcbiAsnText.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.IO;
00005 using System.Text;
00006
00007 namespace PhyloTree.Formats
00008 {
00009 /// <summary>
00010 /// Contains methods to read and write trees in the NCBI ASN.1 text format.
00011 /// </summary>
00012     public static class NcbiAsnText
00013     {
00014 /// <summary>
00015 /// Parses a tree from an NCBI ASN.1 text format file.  Note that the tree can only contain a single
      file, and this method will always return a collection with a single element.
00016 /// </summary>
00017 /// <param name="inputFile">The path to the input file.</param>
00018 /// <returns>A <see cref="IEnumerable{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00019         public static IEnumerable<TreeNode> ParseTrees(string inputFile)
00020         {
00021             yield return ParseAllTrees(inputFile)[0];
00022         }
00023
00024 /// <summary>
00025 /// Parses a tree from an NCBI ASN.1 text format file.  Note that the tree can only contain a single
      file, and this method will always return a collection with a single element.
00026 /// </summary>
00027 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00028 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00029 /// <returns>A <see cref="IEnumerable{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00030         public static IEnumerable<TreeNode> ParseTrees(Stream inputStream, bool keepOpen = false)
00031         {
00032             yield return ParseAllTrees(inputStream, keepOpen)[0];
00033         }
00034
00035 /// <summary>
00036 /// Parses a tree from an NCBI ASN.1 text format file.  Note that the tree can only contain a single
      file, and this method will always return a list with a single element.
00037 /// </summary>
00038 /// <param name="inputFile">The path to the input file.</param>
00039 /// <returns>A <see cref="List{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00040         public static List<TreeNode> ParseAllTrees(string inputFile)
00041         {
00042             using StreamReader reader = new StreamReader(inputFile);
00043             return new List<TreeNode>() { ParseTree(reader) };
00044         }
00045
00046 /// <summary>
00047 /// Parses a tree from an NCBI ASN.1 text format file.  Note that the tree can only contain a single
      file, and this method will always return a list with a single element.
00048 /// </summary>
00049 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00050 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00051 /// <returns>A <see cref="List{T}"/> containing the tree defined in the file.  This will always
      consist of a single element.</returns>
00052         public static List<TreeNode> ParseAllTrees(Stream inputStream, bool keepOpen = false)
00053         {
00054             using StreamReader reader = new StreamReader(inputStream, Encoding.UTF8, true, 1024,
      keepOpen);
00055             return new List<TreeNode>() { ParseTree(reader) };
00056         }
00057
00058 /// <summary>
00059 /// Parses a tree from an NCBI ASN.1 format string into a <see cref="TreeNode"/> object.
00060 /// </summary>
00061 /// <param name="source">The NCBI ASN.1 format tree string.</param>
00062 /// <returns>The parsed <see cref="TreeNode"/> object.</returns>
00063         public static TreeNode ParseTree(string source)
00064         {
00065             using StringReader reader = new StringReader(source);
00066             return ParseTree(reader);
00067         }
00068
00069 /// <summary>
00070 /// Parses a tree from a <see cref="TextReader"/> that reads an NCBI ASN.1 format string into a <see
      cref="TreeNode"/> object.
00071 /// </summary>
00072 /// <param name="reader">The <see cref="TextReader"/> that reads the NCBI ASN.1 format string.</param>
00073 /// <returns>The parsed <see cref="TreeNode"/> object.</returns>
```

```
00074         public static TreeNode ParseTree(TextReader reader)
00075         {
00076             Contract.Requires(reader != null);
00077 
00078             bool eof = false;
00079 
00080             string currToken = ReadToken(reader, ref eof);
00081             AssertToken(currToken, "BioTreeContainer");
00082 
00083             currToken = ReadToken(reader, ref eof);
00084             AssertToken(currToken, "::=");
00085 
00086             currToken = ReadToken(reader, ref eof);
00087             AssertToken(currToken, "{");
00088 
00089             currToken = ReadToken(reader, ref eof);
00090 
00091 
00092             string treetype = null;
00093 
00094             if (currToken.Equals("treetype", StringComparison.OrdinalIgnoreCase))
00095             {
00096                 currToken = ReadToken(reader, ref eof);
00097                 treetype = currToken[1..^1];
00098 
00099                 currToken = ReadToken(reader, ref eof);
00100                 AssertToken(currToken, ",");
00101 
00102                 currToken = ReadToken(reader, ref eof);
00103             }
00104 
00105             AssertToken(currToken, "fdict");
00106 
00107             currToken = ReadToken(reader, ref eof);
00108             AssertToken(currToken, "{");
00109 
00110             Dictionary<int, string> features = new Dictionary<int, string>();
00111 
00112             bool finishedFeatures = false;
00113 
00114             while (!finishedFeatures)
00115             {
00116                 currToken = ReadToken(reader, ref eof);
00117                 AssertToken(currToken, "{");
00118 
00119                 currToken = ReadToken(reader, ref eof);
00120                 AssertToken(currToken, "id");
00121 
00122                 currToken = ReadToken(reader, ref eof);
00123                 int id = int.Parse(currToken, System.Globalization.CultureInfo.InvariantCulture);
00124 
00125                 currToken = ReadToken(reader, ref eof);
00126                 AssertToken(currToken, ",");
00127 
00128                 currToken = ReadToken(reader, ref eof);
00129                 AssertToken(currToken, "name");
00130 
00131                 string name = ReadToken(reader, ref eof)[1..^1];
00132 
00133                 currToken = ReadToken(reader, ref eof);
00134                 AssertToken(currToken, "}");
00135 
00136                 features[id] = name;
00137 
00138                 currToken = ReadToken(reader, ref eof);
00139 
00140                 if (currToken == "}")
00141                 {
00142                     finishedFeatures = true;
00143                 }
00144                 else
00145                 {
00146                     AssertToken(currToken, ",");
00147                 }
00148             }
00149 
00150             currToken = ReadToken(reader, ref eof);
00151             AssertToken(currToken, ",");
00152 
00153             currToken = ReadToken(reader, ref eof);
00154             AssertToken(currToken, "nodes");
00155 
00156             currToken = ReadToken(reader, ref eof);
00157             AssertToken(currToken, "{");
00158 
00159             bool finishedNodes = false;
00160 
```

```
00161             Dictionary<int, (TreeNode node, int?  parent)> nodes = new Dictionary<int, (TreeNode node,
      int?  parent)>();
00162
00163             while (!finishedNodes)
00164             {
00165                 currToken = ReadToken(reader, ref eof);
00166                 AssertToken(currToken, "{");
00167
00168                 currToken = ReadToken(reader, ref eof);
00169                 AssertToken(currToken, "id");
00170
00171                 currToken = ReadToken(reader, ref eof);
00172                 int id = int.Parse(currToken, System.Globalization.CultureInfo.InvariantCulture);
00173
00174                 currToken = ReadToken(reader, ref eof);
00175                 AssertToken(currToken, ",");
00176
00177                 currToken = ReadToken(reader, ref eof);
00178
00179                 int?  parent = null;
00180                 if (currToken.Equals("parent", StringComparison.OrdinalIgnoreCase))
00181                 {
00182                     currToken = ReadToken(reader, ref eof);
00183                     parent = int.Parse(currToken, System.Globalization.CultureInfo.InvariantCulture);
00184
00185                     currToken = ReadToken(reader, ref eof);
00186                     AssertToken(currToken, ",");
00187
00188                     currToken = ReadToken(reader, ref eof);
00189                 }
00190
00191                 TreeNode node = new TreeNode(null);
00192
00193                 if (currToken.Equals("features", StringComparison.OrdinalIgnoreCase))
00194                 {
00195                     currToken = ReadToken(reader, ref eof);
00196                     AssertToken(currToken, "{");
00197
00198                     bool finishedNodeFeatures = false;
00199
00200                     while (!finishedNodeFeatures)
00201                     {
00202                         currToken = ReadToken(reader, ref eof);
00203                         AssertToken(currToken, "{");
00204
00205                         currToken = ReadToken(reader, ref eof);
00206                         AssertToken(currToken, "featureid");
00207
00208                         currToken = ReadToken(reader, ref eof);
00209                         int featureId = int.Parse(currToken,
      System.Globalization.CultureInfo.InvariantCulture);
00210
00211                         currToken = ReadToken(reader, ref eof);
00212                         AssertToken(currToken, ",");
00213
00214                         currToken = ReadToken(reader, ref eof);
00215                         AssertToken(currToken, "value");
00216
00217                         string value = ReadToken(reader, ref eof)[1..^1];
00218
00219                         object valueObject;
00220
00221                         if (!features[featureId].Equals("label", StringComparison.OrdinalIgnoreCase)
      && double.TryParse(value, System.Globalization.NumberStyles.Any,
      System.Globalization.CultureInfo.InvariantCulture, out double doubleValue))
00222                         {
00223                             valueObject = doubleValue;
00224                         }
00225                         else
00226                         {
00227                             valueObject = value;
00228                         }
00229
00230                         currToken = ReadToken(reader, ref eof);
00231                         AssertToken(currToken, "}");
00232
00233                         node.Attributes[features[featureId]] = valueObject;
00234
00235                         currToken = ReadToken(reader, ref eof);
00236
00237                         if (currToken == "}")
00238                         {
00239                             finishedNodeFeatures = true;
00240                         }
00241                         else
00242                         {
00243                             AssertToken(currToken, ",");
```

```
00244                         }
00245                     }
00246
00247                     currToken = ReadToken(reader, ref eof);
00248                 }
00249
00250                 AssertToken(currToken, "}");
00251
00252                 nodes[id] = (node, parent);
00253
00254                 currToken = ReadToken(reader, ref eof);
00255
00256                 if (currToken == "}")
00257                 {
00258                     finishedNodes = true;
00259                 }
00260                 else
00261                 {
00262                     AssertToken(currToken, ",");
00263                 }
00264             }
00265
00266             currToken = ReadToken(reader, ref eof);
00267
00268             string label = null;
00269
00270             if (currToken.Equals("label", StringComparison.OrdinalIgnoreCase))
00271             {
00272                 label = ReadToken(reader, ref eof)[1..^1];
00273             }
00274
00275             TreeNode tree = null;
00276
00277             foreach (KeyValuePair<int, (TreeNode node, int?  parent)> kvp in nodes)
00278             {
00279                 if (kvp.Value.parent != null)
00280                 {
00281                     int parent = kvp.Value.parent.Value;
00282
00283                     nodes[parent].node.Children.Add(kvp.Value.node);
00284                     kvp.Value.node.Parent = nodes[parent].node;
00285                 }
00286                 else
00287                 {
00288                     tree = kvp.Value.node;
00289                 }
00290
00291                 if (kvp.Value.node.Attributes.TryGetValue("dist", out object distValue) && distValue
    is double branchLength)
00292                 {
00293                     kvp.Value.node.Length = branchLength;
00294                 }
00295             }
00296
00297             foreach (KeyValuePair<int, (TreeNode node, int?  parent)> kvp in nodes)
00298             {
00299                 if (kvp.Value.node.Children.Count == 0)
00300                 {
00301                     if (kvp.Value.node.Attributes.TryGetValue("label", out object labelValue) &&
    labelValue is string nodeLabel)
00302                     {
00303                         kvp.Value.node.Name = nodeLabel;
00304                     }
00305                 }
00306             }
00307
00308             if (tree != null)
00309             {
00310                 if (!string.IsNullOrEmpty(treetype))
00311                 {
00312                     tree.Attributes["Tree-treetype"] = treetype;
00313                 }
00314
00315                 if (!string.IsNullOrEmpty(label))
00316                 {
00317                     tree.Attributes["TreeName"] = label;
00318                 }
00319             }
00320
00321             return tree;
00322         }
00323
00324
00325 /// <summary>
00326 /// Writes a <see cref="TreeNode"/> to a file in NCBI ASN.1 text format.
00327 /// </summary>
00328 /// <param name="tree">The tree to write.</param>
```

```
00329 /// <param name="outputFile">The path to the output file.</param>
00330 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00331 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00332         public static void WriteTree(TreeNode tree, string outputFile, string treeType = null, string
      label = null)
00333         {
00334             File.WriteAllText(outputFile, WriteTree(tree, treeType, label));
00335         }
00336 
00337 /// <summary>
00338 /// Writes a <see cref="TreeNode"/> to a file in NCBI ASN.1 text format.
00339 /// </summary>
00340 /// <param name="tree">The tree to write.</param>
00341 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00342 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00343 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00344 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00345         public static void WriteTree(TreeNode tree, Stream outputStream, bool keepOpen = false, string
      treeType = null, string label = null)
00346         {
00347             using StreamWriter sw = new StreamWriter(outputStream, Encoding.UTF8, 1024, keepOpen);
00348             sw.Write(WriteTree(tree, treeType, label));
00349         }
00350 
00351 /// <summary>
00352 /// Writes a collection of <see cref="TreeNode"/>s to a file in NCBI ASN.1 text format.  Note that
      only one tree can be saved in each file; if the collection contains more than one tree an exception
      will be thrown.
00353 /// </summary>
00354 /// <param name="trees">The collection of trees to write.  If this contains more than one tree, an
      exception will be thrown.</param>
00355 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00356 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00357 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00358 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00359         public static void WriteAllTrees(IEnumerable<TreeNode> trees, Stream outputStream, bool
      keepOpen = false, string treeType = null, string label = null)
00360         {
00361             Contract.Requires(trees != null);
00362 
00363             bool firstTree = true;
00364 
00365             foreach (TreeNode tree in trees)
00366             {
00367                 if (firstTree)
00368                 {
00369                     WriteTree(tree, outputStream, keepOpen, treeType, label);
00370                     firstTree = false;
00371                 }
00372                 else
00373                 {
00374                     throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved
      in an NCBI ASN.1 file!");
00375                 }
00376             }
00377         }
00378 
00379 /// <summary>
00380 /// Writes a collection of <see cref="TreeNode"/>s to a file in NCBI ASN.1 text format.  Note that
      only one tree can be saved in each file; if the collection contains more than one tree an exception
      will be thrown.
00381 /// </summary>
00382 /// <param name="trees">The collection of trees to write.  If this contains more than one tree, an
      exception will be thrown.</param>
00383 /// <param name="outputFile">The path to the output file.</param>
00384 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00385 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00386         public static void WriteAllTrees(IEnumerable<TreeNode> trees, string outputFile, string
      treeType = null, string label = null)
00387         {
00388             Contract.Requires(trees != null);
00389 
00390             bool firstTree = true;
00391 
00392             foreach (TreeNode tree in trees)
00393             {
00394                 if (firstTree)
```

```
00395                 {
00396                     WriteTree(tree, outputFile, treeType, label);
00397                     firstTree = false;
00398                 }
00399                 else
00400                 {
00401                     throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved
      in an NCBI ASN.1 file!");
00402                 }
00403             }
00404         }
00405 
00406 /// <summary>
00407 /// Writes a list of <see cref="TreeNode"/>s to a file in NCBI ASN.1 text format.  Note that only one
      tree can be saved in each file; if the tree contains more than one tree an exception will be thrown.
00408 /// </summary>
00409 /// <param name="trees">The list of trees to write.  If this contains more than one tree, an exception
      will be thrown.</param>
00410 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00411 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00412 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00413 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00414         public static void WriteAllTrees(List<TreeNode> trees, Stream outputStream, bool keepOpen =
      false, string treeType = null, string label = null)
00415         {
00416             Contract.Requires(trees != null);
00417 
00418             if (trees.Count > 1)
00419             {
00420                 throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved in an
      NCBI ASN.1 file!");
00421             }
00422 
00423             WriteTree(trees[0], outputStream, keepOpen, treeType, label);
00424         }
00425 
00426 /// <summary>
00427 /// Writes a list of <see cref="TreeNode"/>s to a file in NCBI ASN.1 text format.  Note that only one
      tree can be saved in each file; if the list contains more than one tree an exception will be thrown.
00428 /// </summary>
00429 /// <param name="trees">The list of trees to write.  If this contains more than one tree, an exception
      will be thrown.</param>
00430 /// <param name="outputFile">The path to the output file.</param>
00431 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00432 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00433         public static void WriteAllTrees(List<TreeNode> trees, string outputFile, string treeType =
      null, string label = null)
00434         {
00435             Contract.Requires(trees != null);
00436 
00437             if (trees.Count > 1)
00438             {
00439                 throw new ArgumentOutOfRangeException(nameof(trees), "Only one tree can be saved in an
      NCBI ASN.1 file!");
00440             }
00441 
00442             WriteTree(trees[0], outputFile, treeType, label);
00443         }
00444 
00445 /// <summary>
00446 /// Writes a <see cref="TreeNode"/> to a <see cref="string"/> in NCBI ASN.1 text format.
00447 /// </summary>
00448 /// <param name="tree">The tree to write.</param>
00449 /// <param name="treeType">An optional value for the <c>treetype</c> property defined in the NCBI
      ASN.1 tree format.</param>
00450 /// <param name="label">An optional value for the <c>label</c> property defined in the NCBI ASN.1 tree
      format.</param>
00451 /// <returns>A <see cref="string"/> containing the NCBI ASN.1 representation of the <see
      cref="TreeNode"/>.</returns>
00452         public static string WriteTree(TreeNode tree, string treeType = null, string label = null)
00453         {
00454             if (tree == null)
00455             {
00456                 throw new ArgumentNullException(nameof(tree));
00457             }
00458 
00459             StringBuilder builder = new StringBuilder();
00460 
00461             builder.Append("BioTreeContainer ::= {\n");
00462 
00463             if (!string.IsNullOrEmpty(treeType))
00464             {
```

```
00465                 builder.Append("  treetype \"" + treeType.Replace("\"", "\"\"",
      StringComparison.OrdinalIgnoreCase) + "\",\n");
00466             }
00467
00468             builder.Append("  fdict {\n");
00469
00470             HashSet<string> attributes = new HashSet<string>(StringComparer.OrdinalIgnoreCase) {
      "label", "dist" };
00471
00472             foreach (TreeNode node in tree.GetChildrenRecursiveLazy())
00473             {
00474                 foreach (string attribute in node.Attributes.Keys)
00475                 {
00476                     attributes.Add(attribute);
00477                 }
00478             }
00479
00480             Dictionary<string, int> featureIndex = new Dictionary<string,
      int>(StringComparer.OrdinalIgnoreCase);
00481             int currInd = 0;
00482
00483             foreach (string attribute in attributes)
00484             {
00485                 featureIndex.Add(attribute, currInd);
00486
00487                 if (currInd > 0)
00488                 {
00489                     builder.Append(",\n");
00490                 }
00491
00492                 builder.Append("    {\n");
00493                 builder.Append("      id " +
      currInd.ToString(System.Globalization.CultureInfo.InvariantCulture) + ",\n");
00494                 builder.Append("      name \"" + attribute.Replace("\"", "\"\"",
      StringComparison.OrdinalIgnoreCase) + "\"\n");
00495
00496                 builder.Append("    }");
00497
00498                 currInd++;
00499             }
00500
00501             builder.Append("\n  },\n");
00502
00503             builder.Append("  nodes {\n");
00504
00505             Dictionary<TreeNode, int> nodeIndex = new Dictionary<TreeNode, int>();
00506             currInd = 0;
00507
00508             foreach (TreeNode node in tree.GetChildrenRecursiveLazy())
00509             {
00510                 nodeIndex.Add(node, currInd);
00511
00512                 if (currInd > 0)
00513                 {
00514                     builder.Append(",\n");
00515                 }
00516
00517                 builder.Append("    {\n");
00518                 builder.Append("      id " +
      currInd.ToString(System.Globalization.CultureInfo.InvariantCulture));
00519
00520                 if (node.Parent != null)
00521                 {
00522                     builder.Append(",\n      parent " +
      nodeIndex[node.Parent].ToString(System.Globalization.CultureInfo.InvariantCulture));
00523                 }
00524
00525                 List<(int, string)> nodeFeatures = new List<(int, string)>();
00526
00527                 bool hasLabel = false;
00528                 bool hasDist = false;
00529
00530                 foreach (KeyValuePair<string, object> attribute in node.Attributes)
00531                 {
00532                     if (attribute.Value != null)
00533                     {
00534                         int attributeIndex = featureIndex[attribute.Key];
00535
00536                         if (attribute.Value is string stringValue &&
      !string.IsNullOrEmpty(stringValue))
00537                         {
00538                             nodeFeatures.Add((attributeIndex, stringValue));
00539
00540                             if (attribute.Key.Equals("label", StringComparison.OrdinalIgnoreCase))
00541                             {
00542                                 hasLabel = true;
00543                             }
```

```
00544
00545                           if (attribute.Key.Equals("dist", StringComparison.OrdinalIgnoreCase))
00546                           {
00547                               hasDist = true;
00548                           }
00549                       }
00550                       else if (attribute.Value is double doubleValue && !double.IsNaN(doubleValue))
00551                       {
00552                           nodeFeatures.Add((attributeIndex,
      doubleValue.ToString(System.Globalization.CultureInfo.InvariantCulture)));
00553
00554                           if (attribute.Key.Equals("label", StringComparison.OrdinalIgnoreCase))
00555                           {
00556                               hasLabel = true;
00557                           }
00558
00559                           if (attribute.Key.Equals("dist", StringComparison.OrdinalIgnoreCase))
00560                           {
00561                               hasDist = true;
00562                           }
00563                       }
00564                   }
00565               }
00566
00567               if (!hasLabel && node.Name != null)
00568               {
00569                   nodeFeatures.Add((featureIndex["label"], node.Name));
00570               }
00571
00572               if (!hasDist && !double.IsNaN(node.Length))
00573               {
00574                   nodeFeatures.Add((featureIndex["dist"],
      node.Length.ToString(System.Globalization.CultureInfo.InvariantCulture)));
00575               }
00576
00577               if (nodeFeatures.Count > 0)
00578               {
00579                   builder.Append(",\n      features {\n");
00580
00581                   for (int i = 0; i < nodeFeatures.Count; i++)
00582                   {
00583                       builder.Append("        {\n");
00584                       builder.Append("          featureid " +
      nodeFeatures[i].Item1.ToString(System.Globalization.CultureInfo.InvariantCulture) + ",\n");
00585                       builder.Append("          value \"" + nodeFeatures[i].Item2.Replace("\"",
      "\"\"", StringComparison.OrdinalIgnoreCase) + "\"\n");
00586                       builder.Append("        }");
00587
00588                       if (i < nodeFeatures.Count - 1)
00589                       {
00590                           builder.Append(",\n");
00591                       }
00592                       else
00593                       {
00594                           builder.Append("\n");
00595                       }
00596                   }
00597
00598                   builder.Append("      }");
00599               }
00600
00601               builder.Append("\n    }");
00602
00603               currInd++;
00604           }
00605
00606           builder.Append("\n  }");
00607
00608           if (!string.IsNullOrEmpty(label))
00609           {
00610               builder.Append(",\n  label \"" + label.Replace("\"", "\"\"",
      StringComparison.OrdinalIgnoreCase) + "\"");
00611           }
00612           else if (tree.Attributes.TryGetValue("TreeName", out object treeNameValue) &&
      treeNameValue != null && treeNameValue is string treeName && !string.IsNullOrEmpty(treeName))
00613           {
00614               builder.Append(",\n  label \"" + treeName.Replace("\"", "\"\"",
      StringComparison.OrdinalIgnoreCase) + "\"");
00615           }
00616
00617           builder.Append("\n}\n");
00618
00619           return builder.ToString();
00620       }
00621
00622 /// <summary>
00623 /// Throws an exception if the token that has been read is different than what was expected.
```

```
00624 /// </summary>
00625 /// <param name="token">The token that has been read.</param>
00626 /// <param name="expected">The token that was expected.</param>
00627         private static void AssertToken(string token, string expected)
00628         {
00629             if (!token.Equals(expected, StringComparison.OrdinalIgnoreCase))
00630             {
00631                 throw new Exception("Unexpected token:  \"" + token + "\"!  Was expecting:  \"" +
      expected + "\".");
00632             }
00633         }
00634 
00635 /// <summary>
00636 /// Reads a token from the <see cref="TextReader"/>.  A token is usually a word, a curly bracket or a
      comma.
00637 /// </summary>
00638 /// <param name="reader">The <see cref="TextReader"/> from which the token will be read.</param>
00639 /// <param name="eof">This parameter will be set to <see langword="true" /> if the reader reaches the
      end of the file.  If this is already <see langword="true" /> when the method starts, an exception is
      thrown.</param>
00640 /// <returns>The token that has been read.</returns>
00641         private static string ReadToken(TextReader reader, ref bool eof)
00642         {
00643             if (eof)
00644             {
00645                 throw new IndexOutOfRangeException("Trying to read beyond the end of the string!");
00646             }
00647 
00648             StringBuilder tokenBuilder = new StringBuilder();
00649 
00650             int charInt = reader.Peek();
00651 
00652             while (charInt >= 0 && char.IsWhiteSpace((char)charInt))
00653             {
00654                 reader.Read();
00655                 charInt = reader.Peek();
00656             }
00657 
00658             if (charInt >= 0)
00659             {
00660                 bool firstChar = true;
00661                 bool quotesOpen = false;
00662 
00663                 while (!IsBreakCharacter(charInt, firstChar, quotesOpen))
00664                 {
00665                     charInt = reader.Read();
00666 
00667                     if ((!quotesOpen || (char)charInt != '\n') && (char)charInt != '\r')
00668                     {
00669                         tokenBuilder.Append((char)charInt);
00670                     }
00671 
00672                     if ((char)charInt == '\"')
00673                     {
00674                         quotesOpen = !quotesOpen;
00675                     }
00676                     charInt = reader.Peek();
00677                     firstChar = false;
00678                 }
00679 
00680                 if (charInt < 0)
00681                 {
00682                     eof = true;
00683                 }
00684                 else
00685                 {
00686                     eof = false;
00687                 }
00688 
00689                 return tokenBuilder.ToString();
00690             }
00691             else
00692             {
00693                 eof = true;
00694                 return tokenBuilder.ToString();
00695             }
00696         }
00697 
00698 /// <summary>
00699 /// Determines whether a character breaks the current token.
00700 /// </summary>
00701 /// <param name="charInt">The character that was read.</param>
00702 /// <param name="firstChar">A <see langword="bool" /> specifying whether this character is the first
      character in the token.</param>
00703 /// <param name="quotesOpen">A <see langword="bool" /> specifying whether the character being read is
      currently within a double-quoted string.</param>
00704 /// <returns><see langword="true"/> if the character breaks the current token; otherwise, <see
```

```
    langword="false"/>.  </returns>
00705         private static bool IsBreakCharacter(int charInt, bool firstChar, bool quotesOpen)
00706         {
00707             if (charInt < 0)
00708             {
00709                 return true;
00710             }
00711             else if (firstChar)
00712             {
00713                 return char.IsWhiteSpace((char)charInt);
00714             }
00715             else if (quotesOpen)
00716             {
00717                 return false;
00718             }
00719             else
00720             {
00721                 char c = (char)charInt;
00722 
00723                 return char.IsWhiteSpace(c) || c == ',' || c == '{' || c == '}';
00724             }
00725         }
00726     }
00727 }
```

8.6 NEXUS.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.IO;
00005 using System.Linq;
00006 using System.Text;
00007 using PhyloTree.Extensions;
00008 
00009 namespace PhyloTree.Formats
00010 {
00011 /// <summary>
00012 /// Contains methods to read and write trees in NEXUS format.
00013 /// </summary>
00014     public static class NEXUS
00015     {
00016 /// <summary>
00017 /// Possible states while reading a NEXUS file
00018 /// </summary>
00019         private enum NEXUSStatus
00020         {
00021 /// <summary>
00022 /// At the root of the NEXUS structure
00023 /// </summary>
00024             Root,
00025 
00026 /// <summary>
00027 /// Inside a comment at the root of the NEXUS structure
00028 /// </summary>
00029             InCommentInRoot,
00030 
00031 /// <summary>
00032 /// Inside a block that is not a "Trees" block
00033 /// </summary>
00034             InOtherBlock,
00035 
00036 /// <summary>
00037 /// Inside a comment inside a block that is not a "Trees" block.
00038 /// </summary>
00039             InCommentInOtherBlock,
00040 
00041 /// <summary>
00042 /// Inside a "Trees" block
00043 /// </summary>
00044             InTreeBlock,
00045 
00046 /// <summary>
00047 /// Inside a "Translate" statement inside a "Trees" block.
00048 /// </summary>
00049             InTranslateStatement,
00050 
00051 /// <summary>
00052 /// Inside a "Tree" statement inside a "Trees" block.
00053 /// </summary>
00054             InTreeStatement,
00055 
00056 /// <summary>
```

```
00057 /// Inside a comment inside a "Trees" block.
00058 /// </summary>
00059             InCommentInTreeBlock,
00060 
00061 /// <summary>
00062 /// Inside a comment inside a "Translate" statement inside a "Trees" block
00063 /// </summary>
00064             InCommentInTranslateStatement,
00065 
00066 /// <summary>
00067 /// Inside a comment before the equal sign inside a "Tree" statement inside a "Trees" block
00068 /// </summary>
00069             InCommentInTreeStatementName
00070         }
00071 
00072 /// <summary>
00073 /// Parses a NEXUS file and completely loads it into memory.  Can be used to parse a string or a file.
00074 /// </summary>
00075 /// <param name="sourceString">The NEXUS file content.  If this parameter is specified, <paramref
      name="sourceStream"/> is ignored.</param>
00076 /// <param name="sourceStream">The stream to parse.</param>
00077 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00078 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00079 /// <returns>A <see cref="List{T}"/> containing the trees defined in the "Trees" blocks of the NEXUS
      file.</returns>
00080         public static List<TreeNode> ParseAllTrees(string sourceString = null, Stream sourceStream =
      null, bool keepOpen = false, Action<double> progressAction = null)
00081         {
00082             return ParseTrees(sourceString, sourceStream, keepOpen, progressAction).ToList();
00083         }
00084 
00085 
00086 /// <summary>
00087 /// Lazily parses a NEXUS file.  Each tree in the NEXUS file is not read and parsed until it is
      requested.  Can be used to parse a <see cref="string"/> or a <see cref="Stream"/>.
00088 /// </summary>
00089 /// <param name="inputFile">The path to the input file.</param>
00090 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00091 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the "Trees" blocks of
      the NEXUS file.</returns>
00092         [System.Diagnostics.CodeAnalysis.SuppressMessage("Reliability", "CA2000")]
00093         public static IEnumerable<TreeNode> ParseTrees(string inputFile, Action<double> progressAction
      = null)
00094         {
00095             FileStream inputStream = File.OpenRead(inputFile);
00096             return ParseTrees(sourceStream:  inputStream, keepOpen:  false, progressAction:
      progressAction);
00097         }
00098 
00099 /// <summary>
00100 /// Lazily parses a NEXUS file.  Each tree in the NEXUS file is not read and parsed until it is
      requested.  Can be used to parse a <see cref="string"/> or a <see cref="Stream"/>.
00101 /// </summary>
00102 /// <param name="sourceString">The NEXUS file content.  If this parameter is specified, <paramref
      name="sourceStream"/> is ignored.</param>
00103 /// <param name="sourceStream">The stream to parse.</param>
00104 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00105 /// <param name="progressAction">An <see cref="Action" /> that might be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00106 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the "Trees" blocks of
      the NEXUS file.</returns>
00107         public static IEnumerable<TreeNode> ParseTrees(string sourceString = null, Stream sourceStream
      = null, bool keepOpen = false, Action<double> progressAction = null)
00108         {
00109             bool isUsingSourceString = !string.IsNullOrEmpty(sourceString);
00110 
00111             using TextReader reader = isUsingSourceString ?  (TextReader)(new
      StringReader(sourceString)) :  (TextReader)(new StreamReader(sourceStream, Encoding.UTF8, true, 1024,
      keepOpen));
00112 
00113             double totalLength = isUsingSourceString ?  sourceString.Length :
      ((StreamReader)reader).BaseStream.Length;
00114 
00115             Func<long> currentPos;
00116 
00117             if (isUsingSourceString)
00118             {
00119                 System.Reflection.FieldInfo fi = typeof(StringReader).GetField("_pos",
      System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
00120                 currentPos = () =>
```

```
00121                 {
00122                     return (int)fi.GetValue(reader);
00123                 };
00124             }
00125             else
00126             {
00127                 currentPos = () =>
00128                 {
00129                     return ((StreamReader)reader).BaseStream.Position;
00130                 };
00131             }
00132 
00133             NEXUSStatus status = NEXUSStatus.Root;
00134 
00135             string word = reader.NextWord(out bool eof);
00136 
00137             Dictionary<string, string> translateDictionary = new Dictionary<string, string>();
00138 
00139             string treeName;
00140 
00141             while (!eof)
00142             {
00143                 switch (status)
00144                 {
00145                     case NEXUSStatus.Root:
00146                         if (word.Equals("begin", StringComparison.OrdinalIgnoreCase))
00147                         {
00148                             word = reader.NextWord(out _);
00149 
00150                             if (word.Equals("trees", StringComparison.OrdinalIgnoreCase))
00151                             {
00152                                 status = NEXUSStatus.InTreeBlock;
00153                             }
00154                             else
00155                             {
00156                                 status = NEXUSStatus.InOtherBlock;
00157                             }
00158                         }
00159                         else if (word.Equals("[", StringComparison.OrdinalIgnoreCase))
00160                         {
00161                             status = NEXUSStatus.InCommentInRoot;
00162                         }
00163                         break;
00164                     case NEXUSStatus.InCommentInRoot:
00165                         if (word.Equals("]", StringComparison.OrdinalIgnoreCase))
00166                         {
00167                             status = NEXUSStatus.Root;
00168                         }
00169                         break;
00170                     case NEXUSStatus.InOtherBlock:
00171                         if (word.Equals("end", StringComparison.OrdinalIgnoreCase))
00172                         {
00173                             status = NEXUSStatus.Root;
00174                         }
00175                         else if (word.Equals("[", StringComparison.OrdinalIgnoreCase))
00176                         {
00177                             status = NEXUSStatus.InCommentInOtherBlock;
00178                         }
00179                         break;
00180                     case NEXUSStatus.InCommentInOtherBlock:
00181                         if (word.Equals("]", StringComparison.OrdinalIgnoreCase))
00182                         {
00183                             status = NEXUSStatus.InOtherBlock;
00184                         }
00185                         break;
00186                     case NEXUSStatus.InTreeBlock:
00187                         if (word.Equals("translate", StringComparison.OrdinalIgnoreCase))
00188                         {
00189                             status = NEXUSStatus.InTranslateStatement;
00190                         }
00191                         else if (word.Equals("tree", StringComparison.OrdinalIgnoreCase) ||
    word.Equals("utree", StringComparison.OrdinalIgnoreCase))
00192                         {
00193                             status = NEXUSStatus.InTreeStatement;
00194                         }
00195                         else if (word.Equals("end", StringComparison.OrdinalIgnoreCase))
00196                         {
00197                             status = NEXUSStatus.Root;
00198                         }
00199                         else if (word.Equals("[", StringComparison.OrdinalIgnoreCase))
00200                         {
00201                             status = NEXUSStatus.InCommentInTreeBlock;
00202                         }
00203                         break;
00204                     case NEXUSStatus.InCommentInTreeBlock:
00205                         if (word.Equals("]", StringComparison.OrdinalIgnoreCase))
00206                         {
```

```
00207                           status = NEXUSStatus.InTreeBlock;
00208                       }
00209                       break;
00210                   case NEXUSStatus.InTranslateStatement:
00211                       if (word.Equals("[", StringComparison.OrdinalIgnoreCase))
00212                       {
00213                           status = NEXUSStatus.InCommentInTranslateStatement;
00214                       }
00215                       else if (word.Equals(";", StringComparison.OrdinalIgnoreCase))
00216                       {
00217                           status = NEXUSStatus.InTreeBlock;
00218                       }
00219                       else if (word.Equals(",", StringComparison.OrdinalIgnoreCase))
00220                       { }
00221                       else
00222                       {
00223                           string name = word;
00224 
00225                           char initialChar = name[0];
00226 
00227                           while ((initialChar == '\'' || initialChar == '"') &&
      !name.EndsWith(initialChar))
00228                           {
00229                               word = reader.NextWord(out _, out string headingTrivia);
00230                               name += headingTrivia + word;
00231                           }
00232 
00233                           word = reader.NextWord(out _);
00234 
00235                           initialChar = word[0];
00236 
00237                           while ((initialChar == '\'' || initialChar == '"') &&
      !word.EndsWith(initialChar))
00238                           {
00239                               string word2 = reader.NextWord(out _, out string headingTrivia);
00240                               word += headingTrivia + word2;
00241                           }
00242 
00243                           if ((name.StartsWith("'", StringComparison.OrdinalIgnoreCase) &&
      name.EndsWith("'", StringComparison.OrdinalIgnoreCase)) || (name.StartsWith("\"",
      StringComparison.OrdinalIgnoreCase) && name.EndsWith("\"", StringComparison.OrdinalIgnoreCase)))
00244                           {
00245                               name = name[1..^1];
00246                           }
00247 
00248                           if ((word.StartsWith("'", StringComparison.OrdinalIgnoreCase) &&
      word.EndsWith("'", StringComparison.OrdinalIgnoreCase)) || (word.StartsWith("\"",
      StringComparison.OrdinalIgnoreCase) && word.EndsWith("\"", StringComparison.OrdinalIgnoreCase)))
00249                           {
00250                               word = word[1..^1];
00251                           }
00252 
00253                           translateDictionary.Add(name, word);
00254                       }
00255                       break;
00256                   case NEXUSStatus.InCommentInTranslateStatement:
00257                       if (word.Equals("]", StringComparison.OrdinalIgnoreCase))
00258                       {
00259                           status = NEXUSStatus.InTranslateStatement;
00260                       }
00261                       break;
00262                   case NEXUSStatus.InCommentInTreeStatementName:
00263                       if (word.Equals("]", StringComparison.OrdinalIgnoreCase))
00264                       {
00265                           status = NEXUSStatus.InTreeStatement;
00266                       }
00267                       break;
00268                   case NEXUSStatus.InTreeStatement:
00269                       if (word.Equals("[", StringComparison.OrdinalIgnoreCase))
00270                       {
00271                           status = NEXUSStatus.InCommentInTreeStatementName;
00272                       }
00273                       else
00274                       {
00275                           treeName = word;
00276                           bool escaping = false;
00277                           bool openQuotes = false;
00278                           bool openApostrophe = false;
00279                           bool openComment = false;
00280 
00281                           char c = reader.NextToken(ref escaping, out bool escaped, ref openQuotes,
      ref openApostrophe, out eof);
00282 
00283                           while (!eof && c != '=')
00284                           {
00285                               if (c == '[')
00286                               {
```

```
00287                                   openComment = true;
00288                               }
00289
00290                               if (c == ']')
00291                               {
00292                                   openComment = false;
00293                               }
00294
00295                               c = reader.NextToken(ref escaping, out escaped, ref openQuotes, ref
     openApostrophe, out eof);
00296                           }
00297
00298                           StringBuilder preComments = new StringBuilder();
00299                           StringBuilder tree = new StringBuilder();
00300
00301                           c = reader.NextToken(ref escaping, out escaped, ref openQuotes, ref
     openApostrophe, out eof);
00302
00303                           while (!(c == '(' && !openComment) && !eof)
00304                           {
00305                               preComments.Append(c);
00306
00307                               if (c == '[')
00308                               {
00309                                   openComment = true;
00310                               }
00311
00312                               if (c == ']')
00313                               {
00314                                   openComment = false;
00315                               }
00316
00317                               c = reader.NextToken(ref escaping, out escaped, ref openQuotes, ref
     openApostrophe, out eof);
00318                           }
00319
00320
00321
00322                           while (!(c == ';' && !openComment && !escaped && !openQuotes &&
     !openApostrophe) && !eof)
00323                           {
00324                               tree.Append(c);
00325
00326                               if (c == '[')
00327                               {
00328                                   openComment = true;
00329                               }
00330
00331                               if (c == ']')
00332                               {
00333                                   openComment = false;
00334                               }
00335
00336
00337                               c = reader.NextToken(ref escaping, out escaped, ref openQuotes, ref
     openApostrophe, out eof);
00338                           }
00339
00340
00341                           TreeNode parsedTree = NWKA.ParseTree(tree.ToString());
00342
00343                           if (!parsedTree.Attributes.ContainsKey("TreeName"))
00344                           {
00345                               parsedTree.Attributes.Add("TreeName", treeName);
00346                           }
00347
00348                           List<TreeNode> nodes = parsedTree.GetChildrenRecursive();
00349
00350                           foreach (TreeNode node in nodes)
00351                           {
00352                               if (!string.IsNullOrEmpty(node.Name) &&
     translateDictionary.TryGetValue(node.Name, out string newName))
00353                               {
00354                                   node.Name = newName;
00355                               }
00356                           }
00357
00358                           bool tempEof = false;
00359
00360                           string tempGuid = Guid.NewGuid().ToString();
00361
00362                           parsedTree.Name = tempGuid;
00363
00364                           string preCommentsString = preComments.ToString();
00365
00366                           if (preCommentsString != "[&R]" && preCommentsString != "[&U]")
00367                           {
```

```
00368                                 using StringReader sr = new StringReader(preCommentsString);
00369                                 NWKA.ParseAttributes(sr, ref tempEof, parsedTree,
      parsedTree.Children.Count);
00370                             }
00371 
00372                             if (parsedTree.Name == tempGuid)
00373                             {
00374                                 parsedTree.Name = null;
00375                             }
00376 
00377                             yield return parsedTree;
00378 
00379                             double progress = Math.Max(0, Math.Min(1, currentPos() / totalLength));
00380 
00381                             progressAction?.Invoke(progress);
00382 
00383                             status = NEXUSStatus.InTreeBlock;
00384                         }
00385                         break;
00386                 }
00387 
00388                 word = reader.NextWord(out eof);
00389             }
00390         }
00391 
00392 /// <summary>
00393 /// Lazily parses trees from a file in NEXUS format.  Each tree in the file is not read and parsed
      until it is requested.
00394 /// </summary>
00395 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00396 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00397 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00398 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the file.</returns>
00399         public static IEnumerable<TreeNode> ParseTrees(Stream inputStream, bool keepOpen = false,
      Action<double> progressAction = null)
00400         {
00401             return ParseTrees(null, inputStream, keepOpen, progressAction);
00402         }
00403 
00404 /// <summary>
00405 /// Parses trees from a file in NEXUS format and completely loads them in memory.
00406 /// </summary>
00407 /// <param name="inputFile">The path to the input file.</param>
00408 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00409 /// <returns>A <see cref="List{T}"/> containing the trees defined in the file.</returns>
00410         public static List<TreeNode> ParseAllTrees(string inputFile, Action<double> progressAction =
      null)
00411         {
00412             using FileStream inputStream = File.OpenRead(inputFile);
00413             return ParseAllTrees(inputStream, false, progressAction);
00414         }
00415 
00416 /// <summary>
00417 /// Parses trees from a file in NEXUS format and completely loads them in memory.
00418 /// </summary>
00419 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00420 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00421 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00422 /// <returns>A <see cref="List{T}"/> containing the trees defined in the file.</returns>
00423         public static List<TreeNode> ParseAllTrees(Stream inputStream, bool keepOpen = false,
      Action<double> progressAction = null)
00424         {
00425             return ParseTrees(inputStream, keepOpen, progressAction).ToList();
00426         }
00427 
00428 /// <summary>
00429 /// Writes a single tree in NEXUS format.
00430 /// </summary>
00431 /// <param name="tree">The tree to be written.</param>
00432 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00433 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00434 /// <param name="translate">If this is <c>true</c>, a <c>Taxa</c> block and a <c>Translate</c>
      statement in the <c>Trees</c> block are added to the NEXUS file.</param>
00435 /// <param name="translateQuotes">If this is <c>true</c>, entries in the <c>Taxa</c> block and a
      <c>Translate</c> statement in the <c>Trees</c> block are placed between single quotes.  Otherwise,
      they are not.  This has no effect if <paramref name="translate"/> is <c>false</c>.</param>
00436 /// <param name="additionalNexusBlocks">A <see cref="TextReader"/> that can read additional NEXUS
      blocks that will be placed at the end of the file.</param>
```

```
00437         public static void WriteTree(TreeNode tree, Stream outputStream, bool keepOpen = false, bool
      translate = true, bool translateQuotes = true, TextReader additionalNexusBlocks = null)
00438         {
00439             WriteAllTrees(new List<TreeNode>() { tree }, outputStream, keepOpen, null, translate,
      translateQuotes, additionalNexusBlocks);
00440         }
00441 
00442 /// <summary>
00443 /// Writes a single tree in NEXUS format.
00444 /// </summary>
00445 /// <param name="tree">The tree to be written.</param>
00446 /// <param name="outputFile">The file on which the tree should be written.</param>
00447 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00448 /// <param name="translate">If this is <c>true</c>, a <c>Taxa</c> block and a <c>Translate</c>
      statement in the <c>Trees</c> block are added to the NEXUS file.</param>
00449 /// <param name="translateQuotes">If this is <c>true</c>, entries in the <c>Taxa</c> block and a
      <c>Translate</c> statement in the <c>Trees</c> block are placed between single quotes.  Otherwise,
      they are not.  This has no effect if <paramref name="translate"/> is <c>false</c>.</param>
00450 /// <param name="additionalNexusBlocks">A <see cref="TextReader"/> that can read additional NEXUS
      blocks that will be placed at the end of the file.</param>
00451         public static void WriteTree(TreeNode tree, string outputFile, bool append = false, bool
      translate = true, bool translateQuotes = true, TextReader additionalNexusBlocks = null)
00452         {
00453             using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
      File.Create(outputFile);
00454             WriteAllTrees(new List<TreeNode>() { tree }, outputStream, false, null, translate,
      translateQuotes, additionalNexusBlocks);
00455         }
00456 
00457 /// <summary>
00458 /// Writes trees in NEXUS format.
00459 /// </summary>
00460 /// <param name="trees">A collection of trees to be written.  If <paramref name="translate"/> is
      <c>true</c>, each tree will be accessed twice.  Otherwise, each tree will be accessed once.</param>
00461 /// <param name="outputFile">The file on which the trees should be written.</param>
00462 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00463 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
      written, with a value between 0 and 1 depending on how many trees have been written so far.</param>
00464 /// <param name="translate">If this is <c>true</c>, a <c>Taxa</c> block and a <c>Translate</c>
      statement in the <c>Trees</c> block are added to the NEXUS file.</param>
00465 /// <param name="translateQuotes">If this is <c>true</c>, entries in the <c>Taxa</c> block and a
      <c>Translate</c> statement in the <c>Trees</c> block are placed between single quotes.  Otherwise,
      they are not.  This has no effect if <paramref name="translate"/> is <c>false</c>.</param>
00466 /// <param name="additionalNexusBlocks">A <see cref="TextReader"/> that can read additional NEXUS
      blocks that will be placed at the end of the file.</param>
00467         public static void WriteAllTrees(IList<TreeNode> trees, string outputFile, bool append =
      false, Action<double> progressAction = null, bool translate = true, bool translateQuotes = true,
      TextReader additionalNexusBlocks = null)
00468         {
00469             using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
      File.Create(outputFile);
00470             WriteAllTrees(trees, outputStream, false, progressAction, translate, translateQuotes,
      additionalNexusBlocks);
00471         }
00472 
00473 /// <summary>
00474 /// Writes trees in NEXUS format.
00475 /// </summary>
00476 /// <param name="trees">A collection of trees to be written.  If <paramref name="translate"/> is
      <c>true</c>, each tree will be accessed twice.  Otherwise, each tree will be accessed once.</param>
00477 /// <param name="outputStream">The <see cref="Stream"/> on which the trees should be written.</param>
00478 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00479 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
      written, with a value between 0 and 1 depending on how many trees have been written so far.</param>
00480 /// <param name="translate">If this is <c>true</c>, a <c>Taxa</c> block and a <c>Translate</c>
      statement in the <c>Trees</c> block are added to the NEXUS file.</param>
00481 /// <param name="translateQuotes">If this is <c>true</c>, entries in the <c>Taxa</c> block and a
      <c>Translate</c> statement in the <c>Trees</c> block are placed between single quotes.  Otherwise,
      they are not.  This has no effect if <paramref name="translate"/> is <c>false</c>.</param>
00482 /// <param name="additionalNexusBlocks">A <see cref="TextReader"/> that can read additional NEXUS
      blocks that will be placed at the end of the file.</param>
00483         public static void WriteAllTrees(IList<TreeNode> trees, Stream outputStream, bool keepOpen =
      false, Action<double> progressAction = null, bool translate = true, bool translateQuotes = true,
      TextReader additionalNexusBlocks = null)
00484         {
00485             Contract.Requires(trees != null);
00486 
00487             using StreamWriter sw = new StreamWriter(outputStream, Encoding.UTF8, 8192, keepOpen);
00488 
00489             sw.WriteLine("#NEXUS");
00490             sw.WriteLine();
00491 
00492             Dictionary<string, int> translationLabels = new Dictionary<string, int>();
00493 
00494             if (translate)
00495             {
```

```
00496                 int index = 0;
00497
00498                 for (int i = 0; i < trees.Count; i++)
00499                 {
00500                     foreach (string label in trees[i].GetLeafNames())
00501                     {
00502                         if (!translationLabels.ContainsKey(label))
00503                         {
00504                             translationLabels[label] = index;
00505                             index++;
00506                         }
00507                     }
00508                 }
00509
00510                 sw.WriteLine("Begin Taxa;");
00511                 sw.WriteLine("\tDimensions ntax=" +
      index.ToString(System.Globalization.CultureInfo.InvariantCulture) + ";");
00512                 sw.WriteLine("\tTaxLabels");
00513
00514                 if (!translateQuotes)
00515                 {
00516                     foreach (KeyValuePair<string, int> kvp in translationLabels)
00517                     {
00518                         sw.WriteLine("\t\t" + kvp.Key);
00519                     }
00520                 }
00521                 else
00522                 {
00523                     foreach (KeyValuePair<string, int> kvp in translationLabels)
00524                     {
00525                         sw.WriteLine("\t\t'" + kvp.Key + "'");
00526                     }
00527                 }
00528
00529                 sw.WriteLine("\t\t;");
00530                 sw.WriteLine("End;");
00531                 sw.WriteLine();
00532                 sw.WriteLine("Begin Trees;");
00533                 sw.WriteLine("\tTranslate\n");
00534
00535                 int count = 0;
00536
00537                 if (!translateQuotes)
00538                 {
00539                     foreach (KeyValuePair<string, int> kvp in translationLabels)
00540                     {
00541                         count++;
00542                         sw.WriteLine("\t\t" + (kvp.Value +
      1).ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + kvp.Key + (count < index ?  ","
      :  ""));
00543                     }
00544                 }
00545                 else
00546                 {
00547                     foreach (KeyValuePair<string, int> kvp in translationLabels)
00548                     {
00549                         count++;
00550                         sw.WriteLine("\t\t" + (kvp.Value +
      1).ToString(System.Globalization.CultureInfo.InvariantCulture) + " '" + kvp.Key + "'" + (count < index
      ?  "," :  ""));
00551                     }
00552                 }
00553
00554                 sw.WriteLine("\t\t;");
00555             }
00556             else
00557             {
00558                 sw.WriteLine("Begin Trees;");
00559             }
00560
00561             for (int i = 0; i < trees.Count; i++)
00562             {
00563                 TreeNode tree = trees[i].Clone();
00564
00565                 foreach (TreeNode leaf in tree.GetLeaves())
00566                 {
00567                     if (translationLabels.TryGetValue(leaf.Name, out int translation))
00568                     {
00569                         leaf.Name = (translation +
      1).ToString(System.Globalization.CultureInfo.InvariantCulture);
00570                     }
00571                 }
00572
00573                 string treeName = "";
00574
00575                 if (tree.Attributes.TryGetValue("TreeName", out object value))
00576                 {
```

```
00577                   treeName = value.ToString();
00578               }
00579               else
00580               {
00581                   treeName = "tree" + (i +
    1).ToString(System.Globalization.CultureInfo.InvariantCulture);
00582               }
00583
00584               sw.WriteLine("\tTree " + treeName + " = " + NWKA.WriteTree(tree, true, true) + ";");
00585               progressAction?.Invoke((double)(i + 1) / trees.Count);
00586           }
00587
00588           sw.WriteLine("End;");
00589
00590           if (additionalNexusBlocks != null)
00591           {
00592               sw.WriteLine();
00593
00594               char[] buffer = new char[1024];
00595
00596               int bytesRead;
00597
00598               while ((bytesRead = additionalNexusBlocks.Read(buffer, 0, 1024)) > 0)
00599               {
00600                   sw.Write(buffer, 0, bytesRead);
00601               }
00602           }
00603       }
00604
00605 /// <summary>
00606 /// Writes trees in NEXUS format.
00607 /// </summary>
00608 /// <param name="trees">An <see cref="IEnumerable{T}"/> containing the trees to be written.  It will
    only be enumerated once.</param>
00609 /// <param name="outputFile">The file on which the trees should be written.</param>
00610 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00611 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
    written, with the number of trees written so far.</param>
00612 /// <param name="additionalNexusBlocks">A <see cref="TextReader"/> that can read additional NEXUS
    blocks that will be placed at the end of the file.</param>
00613       public static void WriteAllTrees(IEnumerable<TreeNode> trees, string outputFile, bool append =
    false, Action<int> progressAction = null, TextReader additionalNexusBlocks = null)
00614       {
00615           using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
    File.Create(outputFile);
00616           WriteAllTrees(trees, outputStream, false, progressAction, additionalNexusBlocks);
00617       }
00618
00619 /// <summary>
00620 /// Writes trees in NEXUS format.
00621 /// </summary>
00622 /// <param name="trees">An <see cref="IEnumerable{T}"/> containing the trees to be written.  It will
    only be enumerated once.</param>
00623 /// <param name="outputStream">The <see cref="Stream"/> on which the trees should be written.</param>
00624 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
    after the end of this method.</param>
00625 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
    written, with the number of trees written so far.</param>
00626 /// <param name="additionalNexusBlocks">A <see cref="TextReader"/> that can read additional NEXUS
    blocks that will be placed at the end of the file.</param>
00627       public static void WriteAllTrees(IEnumerable<TreeNode> trees, Stream outputStream, bool
    keepOpen = false, Action<int> progressAction = null, TextReader additionalNexusBlocks = null)
00628       {
00629           Contract.Requires(trees != null);
00630
00631           using StreamWriter sw = new StreamWriter(outputStream, Encoding.UTF8, 8192, keepOpen);
00632
00633           sw.WriteLine("#NEXUS");
00634           sw.WriteLine();
00635           sw.WriteLine("Begin Trees;");
00636
00637           int treeIndex = 0;
00638           foreach (TreeNode tree in trees)
00639           {
00640               string treeName = "";
00641
00642               if (tree.Attributes.TryGetValue("TreeName", out object value))
00643               {
00644                   treeName = value.ToString();
00645               }
00646               else
00647               {
00648                   treeName = "tree" + (treeIndex +
    1).ToString(System.Globalization.CultureInfo.InvariantCulture);
00649               }
00650
00651               sw.WriteLine("\tTree " + treeName + " = " + NWKA.WriteTree(tree, true, true) + ";");
```

```
00652
00653                 treeIndex++;
00654                 progressAction?.Invoke(treeIndex);
00655             }
00656
00657             sw.WriteLine("End;");
00658
00659             if (additionalNexusBlocks != null)
00660             {
00661                 sw.WriteLine();
00662
00663                 char[] buffer = new char[1024];
00664
00665                 int bytesRead;
00666
00667                 while ((bytesRead = additionalNexusBlocks.Read(buffer, 0, 1024)) > 0)
00668                 {
00669                     sw.Write(buffer, 0, bytesRead);
00670                 }
00671             }
00672         }
00673     }
00674 }
```

8.7 NWKA.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.Globalization;
00005 using System.IO;
00006 using System.Linq;
00007 using System.Text;
00008 using PhyloTree.Extensions;
00009
00010 namespace PhyloTree.Formats
00011 {
00012 /// <summary>
00013 /// Contains methods to read and write trees in Newick and Newick-with-Attributes (NWKA) format.
00014 /// </summary>
00015     public static class NWKA
00016     {
00017 /// <summary>
00018 /// Parse a Newick-with-Attributes string into a TreeNode object.
00019 /// </summary>
00020 /// <param name="source">The Newick-with-Attributes string.  This string must specify only a single
      tree.</param>
00021 /// <param name="parent">The parent node of this node.  If parsing a whole tree, this parameter should
      be left equal to <c>null</c>.</param>
00022 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00023 /// <returns>The parsed <see cref="TreeNode"/> object.</returns>
00024         public static TreeNode ParseTree(string source, bool debug = false, TreeNode parent = null)
00025         {
00026             Contract.Requires(source != null);
00027
00028             source = source.Trim();
00029             if (source.EndsWith(";", StringComparison.OrdinalIgnoreCase))
00030             {
00031                 source = source[0..^1];
00032             }
00033
00034             if (debug)
00035             {
00036                 Console.WriteLine("Parsing:  " + source);
00037             }
00038
00039             if (source.StartsWith("(", StringComparison.OrdinalIgnoreCase))
00040             {
00041                 using StringReader sr = new StringReader(source);
00042
00043                 StringBuilder childrenBuilder = new StringBuilder();
00044
00045                 sr.Read();
00046
00047                 bool closed = false;
00048                 int openCount = 0;
00049                 int openSquareCount = 0;
00050                 int openCurlyCount = 0;
00051
00052                 bool escaping = false;
00053                 bool openQuotes = false;
00054                 bool openApostrophe = false;
```

```
00055                 bool eof = false;
00056
00057                 List<int> commas = new List<int>();
00058                 int position = 0;
00059
00060                 while (!closed && !eof)
00061                 {
00062                     char c = sr.NextToken(ref escaping, out bool escaped, ref openQuotes, ref
      openApostrophe, out eof);
00063
00064                     if (!escaped)
00065                     {
00066                         if (!openQuotes && !openApostrophe)
00067                         {
00068                             switch (c)
00069                             {
00070                                 case '(':
00071                                     openCount++;
00072                                     break;
00073                                 case ')':
00074                                     if (openCount > 0)
00075                                     {
00076                                         openCount--;
00077                                     }
00078                                     else
00079                                     {
00080                                         closed = true;
00081                                     }
00082                                     break;
00083                                 case '[':
00084                                     openSquareCount++;
00085                                     break;
00086                                 case ']':
00087                                     openSquareCount--;
00088                                     break;
00089                                 case '{':
00090                                     openCurlyCount++;
00091                                     break;
00092                                 case '}':
00093                                     openCurlyCount--;
00094                                     break;
00095                                 case ',':
00096                                     if (openCount == 0 && openSquareCount == 0 && openCurlyCount == 0)
00097                                     {
00098                                         commas.Add(position);
00099                                     }
00100                                     break;
00101                             }
00102                         }
00103                     }
00104
00105                     if (!closed && !eof)
00106                     {
00107                         childrenBuilder.Append(c);
00108                         position++;
00109                     }
00110                 }
00111
00112                 List<string> children = new List<string>();
00113
00114                 if (commas.Count > 0)
00115                 {
00116                     for (int i = 0; i < commas.Count; i++)
00117                     {
00118                         children.Add(childrenBuilder.ToString(i > 0 ?  commas[i - 1] + 1 :  0,
      commas[i] - (i > 0 ?  commas[i - 1] + 1 :  0)));
00119                     }
00120                     children.Add(childrenBuilder.ToString(commas.Last() + 1, childrenBuilder.Length -
      commas.Last() - 1));
00121                 }
00122                 else
00123                 {
00124                     children.Add(childrenBuilder.ToString());
00125                 }
00126
00127                 if (debug)
00128                 {
00129                     Console.WriteLine();
00130                     Console.WriteLine("Children:  ");
00131                     for (int i = 0; i < children.Count; i++)
00132                     {
00133                         Console.WriteLine(" - " + children[i]);
00134                     }
00135
00136                     Console.WriteLine();
00137                 }
00138
```

```
00139                 TreeNode tbr = new TreeNode(parent);
00140
00141                 ParseAttributes(sr, ref eof, tbr, children.Count);
00142
00143                 if (debug)
00144                 {
00145                     Console.WriteLine("Attributes:");
00146
00147                     foreach (KeyValuePair<string, object> kvp in tbr.Attributes)
00148                     {
00149                         Console.WriteLine(" - " + kvp.Key + " = " + kvp.Value.ToString());
00150                     }
00151
00152                     Console.WriteLine();
00153                     Console.WriteLine();
00154                 }
00155
00156                 for (int i = 0; i < children.Count; i++)
00157                 {
00158                     tbr.Children.Add(ParseTree(children[i], debug, tbr));
00159                 }
00160
00161                 return tbr;
00162             }
00163             else
00164             {
00165                 using StringReader sr = new StringReader(source);
00166
00167                 bool eof = false;
00168
00169                 TreeNode tbr = new TreeNode(parent);
00170
00171                 ParseAttributes(sr, ref eof, tbr, 0);
00172
00173                 if (debug)
00174                 {
00175                     Console.WriteLine();
00176                     Console.WriteLine("Attributes:");
00177
00178                     foreach (KeyValuePair<string, object> kvp in tbr.Attributes)
00179                     {
00180                         Console.WriteLine(" - " + kvp.Key + " = " + kvp.Value.ToString());
00181                     }
00182
00183                     Console.WriteLine();
00184                 }
00185
00186                 return tbr;
00187             }
00188         }
00189
00190 /// <summary>
00191 /// Lazily parses trees from a string in Newick-with-Attributes (NWKA) format.  Each tree in the
      string is not read and parsed until it is requested.
00192 /// </summary>
00193 /// <param name="source">The <see cref="string"/> from which the trees should be read.</param>
00194 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00195 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the string.</returns>
00196         [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1031")]
00197         public static IEnumerable<TreeNode> ParseTreesFromSource(string source, bool debug = false)
00198         {
00199             bool escaping = false;
00200             bool openQuotes = false;
00201             bool openApostrophe = false;
00202             bool eof = false;
00203
00204             while (!eof)
00205             {
00206                 using StringReader sr = new StringReader(source);
00207
00208                 StringBuilder sb = new StringBuilder();
00209
00210                 char c = sr.NextToken(ref escaping, out bool escaped, ref openQuotes, ref
      openApostrophe, out eof);
00211
00212                 while (!eof && !(c == ';' && !escaped && !openQuotes && !openApostrophe))
00213                 {
00214                     sb.Append(c);
00215                     c = sr.NextToken(ref escaping, out escaped, ref openQuotes, ref openApostrophe,
      out eof);
00216                 }
00217
00218                 string treeString = sb.ToString().Trim();
00219
00220                 int index = treeString.IndexOf("(", StringComparison.OrdinalIgnoreCase);
00221
```

```
00222                 string treeName = "";
00223 
00224                 if (index > 0)
00225                 {
00226                     treeName = treeString.Substring(0, index);
00227                     treeString = treeString.Substring(index);
00228                 }
00229 
00230                 if (treeString.Length > 0)
00231                 {
00232                     TreeNode tbr = null;
00233 
00234                     try
00235                     {
00236                         tbr = ParseTree(treeString, debug, null);
00237                         if (!tbr.Attributes.ContainsKey("TreeName") &&
      !string.IsNullOrWhiteSpace(treeName))
00238                         {
00239                             tbr.Attributes["TreeName"] = treeName;
00240                         }
00241                     }
00242                     catch
00243                     {
00244                         yield break;
00245                     }
00246 
00247                     yield return tbr;
00248                 }
00249             }
00250         }
00251 
00252 /// <summary>
00253 /// Parses trees from a string in Newick-with-Attributes (NWKA) format and completely loads them in
      memory.
00254 /// </summary>
00255 /// <param name="source">The <see cref="string"/> from which the trees should be read.</param>
00256 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00257 /// <returns>A <see cref="List{T}"/> containing the trees defined in the string.</returns>
00258         public static List<TreeNode> ParseAllTreesFromSource(string source, bool debug = false)
00259         {
00260             return ParseTreesFromSource(source, debug).ToList();
00261         }
00262 
00263 /// <summary>
00264 /// Lazily parses trees from a file in Newick-with-Attributes (NWKA) format.  Each tree in the file is
      not read and parsed until it is requested.
00265 /// </summary>
00266 /// <param name="inputFile">The path to the input file.</param>
00267 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00268 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00269 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the file.</returns>
00270         public static IEnumerable<TreeNode> ParseTrees(string inputFile, Action<double> progressAction
      = null, bool debug = false)
00271         {
00272             FileStream inputStream = File.OpenRead(inputFile);
00273             return ParseTrees(inputStream, false, progressAction, debug);
00274         }
00275 
00276 /// <summary>
00277 /// Lazily parses trees from a file in Newick-with-Attributes (NWKA) format.  Each tree in the file is
      not read and parsed until it is requested.
00278 /// </summary>
00279 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00280 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00281 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00282 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00283 /// <returns>A lazy <see cref="IEnumerable{T}"/> containing the trees defined in the file.</returns>
00284         [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1031")]
00285         public static IEnumerable<TreeNode> ParseTrees(Stream inputStream, bool keepOpen = false,
      Action<double> progressAction = null, bool debug = false)
00286         {
00287             bool escaping = false;
00288             bool openQuotes = false;
00289             bool openApostrophe = false;
00290             bool eof = false;
00291 
00292             using StreamReader sr = new StreamReader(inputStream, Encoding.UTF8, true, 1024,
      keepOpen);
00293 
```

```
00294             while (!eof)
00295             {
00296                 StringBuilder sb = new StringBuilder();
00297 
00298                 char c = sr.NextToken(ref escaping, out bool escaped, ref openQuotes, ref
      openApostrophe, out eof);
00299 
00300                 while (!eof && !(c == ';' && !escaped && !openQuotes && !openApostrophe))
00301                 {
00302                     sb.Append(c);
00303                     c = sr.NextToken(ref escaping, out escaped, ref openQuotes, ref openApostrophe,
      out eof);
00304                 }
00305 
00306                 string treeString = sb.ToString().Trim();
00307 
00308                 int index = treeString.IndexOf("(", StringComparison.OrdinalIgnoreCase);
00309 
00310                 string treeName = "";
00311 
00312                 if (index > 0)
00313                 {
00314                     treeName = treeString.Substring(0, index);
00315                     treeString = treeString.Substring(index);
00316                 }
00317 
00318                 if (treeString.Length > 0)
00319                 {
00320                     TreeNode tbr = null;
00321 
00322                     try
00323                     {
00324                         tbr = ParseTree(treeString, debug, null);
00325                         if (!tbr.Attributes.ContainsKey("TreeName") &&
      !string.IsNullOrWhiteSpace(treeName))
00326                         {
00327                             tbr.Attributes["TreeName"] = treeName;
00328                         }
00329                         progressAction?.Invoke((double)inputStream.Position / inputStream.Length);
00330                     }
00331                     catch
00332                     {
00333                         yield break;
00334                     }
00335 
00336                     yield return tbr;
00337                 }
00338             }
00339         }
00340 
00341 /// <summary>
00342 /// Parses trees from a file in Newick-with-Attributes (NWKA) format and completely loads them in
      memory.
00343 /// </summary>
00344 /// <param name="inputFile">The path to the input file.</param>
00345 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00346 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00347 /// <returns>A <see cref="List{T}"/> containing the trees defined in the file.</returns>
00348         public static List<TreeNode> ParseAllTrees(string inputFile, Action<double> progressAction =
      null, bool debug = false)
00349         {
00350             using FileStream inputStream = File.OpenRead(inputFile);
00351             return ParseAllTrees(inputStream, false, progressAction, debug);
00352         }
00353 
00354 /// <summary>
00355 /// Parses trees from a file in Newick-with-Attributes (NWKA) format and completely loads them in
      memory.
00356 /// </summary>
00357 /// <param name="inputStream">The <see cref="Stream"/> from which the file should be read.</param>
00358 /// <param name="keepOpen">Determines whether the stream should be disposed at the end of this method
      or not.</param>
00359 /// <param name="progressAction">An <see cref="Action" /> that will be called after each tree is
      parsed, with the approximate progress (as determined by the position in the stream), ranging from 0 to
      1.</param>
00360 /// <param name="debug">When this is <c>true</c>, debug information is printed to the standard output
      during the parsing.</param>
00361 /// <returns>A <see cref="List{T}"/> containing the trees defined in the file.</returns>
00362         public static List<TreeNode> ParseAllTrees(Stream inputStream, bool keepOpen = false,
      Action<double> progressAction = null, bool debug = false)
00363         {
00364             return ParseTrees(inputStream, keepOpen, progressAction, debug).ToList();
00365         }
00366 
```

```
00367 /// <summary>
00368 /// Parse the attributes of a node in the tree.
00369 /// </summary>
00370 /// <param name="sr">The <see cref="TextReader"/> from which the attributes should be read.</param>
00371 /// <param name="eof">A <see cref="bool"/> indicating whether we have reach the end of the
      stream.</param>
00372 /// <param name="node">The <see cref="TreeNode"/> whose attributes we are parsing.</param>
00373 /// <param name="childCount">The number of children of <paramref name="node"/>.</param>
00374         internal static void ParseAttributes(TextReader sr, ref bool eof, TreeNode node, int
      childCount)
00375         {
00376             StringBuilder attributeValue = new StringBuilder();
00377             StringBuilder attributeName = new StringBuilder();
00378 
00379             int openSquareCount = 0;
00380             int openCurlyCount = 0;
00381 
00382 
00383             bool escaping = false;
00384             bool escaped = false;
00385             bool openQuotes = false;
00386             bool openApostrophe = false;
00387 
00388             bool nameFinished = false;
00389             char lastSeparator = ',';
00390 
00391             bool start = true;
00392             bool closedOuterBrackets = false;
00393 
00394             bool withinBrackets = false;
00395 
00396             char expectedClosingBrackets = '\0';
00397 
00398             int supportCount = 0;
00399             int lengthCount = 0;
00400 
00401             while (!eof)
00402             {
00403                 char c2;
00404 
00405                 if (!closedOuterBrackets)
00406                 {
00407                     c2 = sr.NextToken(ref escaping, out escaped, ref openQuotes, ref openApostrophe,
      out eof);
00408                 }
00409                 else
00410                 {
00411                     c2 = ',';
00412                 }
00413 
00414                 if (start)
00415                 {
00416                     if (c2 == '[' && !openApostrophe && !openQuotes && !escaped)
00417                     {
00418                         expectedClosingBrackets = ']';
00419                         c2 = ',';
00420                         start = false;
00421                     }
00422                 }
00423 
00424 
00425                 if (c2 == '=' && !escaped && !openQuotes && !openApostrophe)
00426                 {
00427                     nameFinished = true;
00428 
00429                     if (closedOuterBrackets)
00430                     {
00431                         closedOuterBrackets = false;
00432                         expectedClosingBrackets = '\0';
00433                         start = true;
00434                         withinBrackets = false;
00435                     }
00436 
00437                     if (expectedClosingBrackets != '\0')
00438                     {
00439                         withinBrackets = true;
00440                     }
00441                 }
00442                 else if ((eof || ((c2 == ':'  || c2 == '/' || c2 == ',') && openSquareCount == 0 &&
      openCurlyCount == 0)) && !escaped && !openQuotes && !openApostrophe)
00443                 {
00444                     if (attributeValue.Length > 0)
00445                     {
00446                         string name = attributeName.ToString();
00447 
00448                         if (name.StartsWith("&", StringComparison.OrdinalIgnoreCase))
00449                         {
```

```
00450                           name = name.Substring(1);
00451                       }
00452
00453                       if (name.StartsWith("!", StringComparison.OrdinalIgnoreCase))
00454                       {
00455                           name = name.Substring(1);
00456                       }
00457
00458                       if (name.Equals("Name", StringComparison.OrdinalIgnoreCase))
00459                       {
00460                           string value = attributeValue.ToString();
00461
00462                           if ((value.StartsWith("\"", StringComparison.OrdinalIgnoreCase) &&
      value.EndsWith("\"", StringComparison.OrdinalIgnoreCase)) || (value.StartsWith("'",
      StringComparison.OrdinalIgnoreCase) && value.EndsWith("'", StringComparison.OrdinalIgnoreCase)))
00463                           {
00464                               value = value[1..^1];
00465                           }
00466
00467                           node.Name = value;
00468                       }
00469                       else if (name.Equals("Support", StringComparison.OrdinalIgnoreCase))
00470                       {
00471                           supportCount = Math.Max(supportCount, 1);
00472                           node.Support = double.Parse(attributeValue.ToString(),
      CultureInfo.InvariantCulture);
00473                       }
00474                       else if (name.Equals("Length", StringComparison.OrdinalIgnoreCase))
00475                       {
00476                           lengthCount = Math.Max(lengthCount, 1);
00477                           node.Length = double.Parse(attributeValue.ToString(),
      CultureInfo.InvariantCulture);
00478                       }
00479                       else
00480                       {
00481                           string value = attributeValue.ToString();
00482                           if (double.TryParse(value, NumberStyles.Any, CultureInfo.InvariantCulture,
      out double result))
00483                           {
00484                               node.Attributes.Add(name, result);
00485                           }
00486                           else
00487                           {
00488                               if ((value.StartsWith("\"", StringComparison.OrdinalIgnoreCase) &&
      value.EndsWith("\"", StringComparison.OrdinalIgnoreCase)) || (value.StartsWith("'",
      StringComparison.OrdinalIgnoreCase) && value.EndsWith("'", StringComparison.OrdinalIgnoreCase)))
00489                               {
00490                                   value = value[1..^1];
00491                               }
00492                               node.Attributes.Add(name, value);
00493                           }
00494                       }
00495                   }
00496                   else if (attributeName.Length > 0)
00497                   {
00498                       switch (lastSeparator)
00499                       {
00500                           case ':':
00501                               if (double.TryParse(attributeName.ToString(), NumberStyles.Any,
      CultureInfo.InvariantCulture, out double result))
00502                               {
00503                                   if (lengthCount == 0)
00504                                   {
00505                                       node.Length = result;
00506                                       lengthCount++;
00507                                   }
00508                                   else
00509                                   {
00510                                       lengthCount++;
00511                                       node.Attributes["Length" +
      lengthCount.ToString(System.Globalization.CultureInfo.InvariantCulture)] = result;
00512                                   }
00513                               }
00514                               else
00515                               {
00516                                   string name = "Unknown";
00517
00518                                   if (node.Attributes.ContainsKey(name))
00519                                   {
00520                                       int ind = 2;
00521                                       string newName = name +
      ind.ToString(System.Globalization.CultureInfo.InvariantCulture);
00522
00523                                       while (node.Attributes.ContainsKey(newName))
00524                                       {
00525                                           ind++;
00526                                           newName = name +
```

```
ind.ToString(System.Globalization.CultureInfo.InvariantCulture);
00527                                       }
00528
00529                                       name = newName;
00530                                   }
00531
00532                                   node.Attributes.Add(name, attributeName.ToString());
00533                               }
00534                               break;
00535                           case '/':
00536                               if (double.TryParse(attributeName.ToString(), NumberStyles.Any,
CultureInfo.InvariantCulture, out double result2))
00537                               {
00538                                   if (supportCount == 0)
00539                                   {
00540                                       node.Support = result2;
00541                                       supportCount++;
00542                                   }
00543                                   else
00544                                   {
00545                                       supportCount++;
00546                                       node.Attributes["Support" +
supportCount.ToString(System.Globalization.CultureInfo.InvariantCulture)] = result2;
00547                                   }
00548                               }
00549                               else
00550                               {
00551                                   string name = "Unknown";
00552
00553                                   if (node.Attributes.ContainsKey(name))
00554                                   {
00555                                       int ind = 2;
00556                                       string newName = name +
ind.ToString(System.Globalization.CultureInfo.InvariantCulture);
00557
00558                                       while (node.Attributes.ContainsKey(newName))
00559                                       {
00560                                           ind++;
00561                                           newName = name +
ind.ToString(System.Globalization.CultureInfo.InvariantCulture);
00562                                       }
00563
00564                                       name = newName;
00565                                   }
00566
00567                                   node.Attributes.Add(name, attributeName.ToString());
00568                               }
00569                               break;
00570                           case ',':
00571                               bool isName = false;
00572
00573                               string value = attributeName.ToString();
00574
00575                               if ((value.StartsWith("\"", StringComparison.OrdinalIgnoreCase) &&
value.EndsWith("\"", StringComparison.OrdinalIgnoreCase)) || (value.StartsWith("'",
StringComparison.OrdinalIgnoreCase) && value.EndsWith("'", StringComparison.OrdinalIgnoreCase)))
00576                               {
00577                                   value = value[1..^1];
00578                                   isName = true;
00579                               }
00580
00581                               if (childCount == 0 && node.Attributes.Count == 3 &&
string.IsNullOrEmpty(node.Name) && double.IsNaN(node.Length) && double.IsNaN(node.Support))
00582                               {
00583                                   isName = true;
00584                               }
00585
00586                               if (string.IsNullOrEmpty(node.Name) && !withinBrackets &&
!closedOuterBrackets && (isName || !int.TryParse(value.Substring(0, 1), out _)))
00587                               {
00588                                   node.Name = value;
00589                               }
00590                               else
00591                               {
00592                                   if (double.IsNaN(node.Support) && double.TryParse(value,
NumberStyles.Any, CultureInfo.InvariantCulture, out double result3))
00593                                   {
00594                                       if (supportCount == 0)
00595                                       {
00596                                           node.Support = result3;
00597                                           supportCount++;
00598                                       }
00599                                       else
00600                                       {
00601                                           supportCount++;
00602                                           node.Attributes["Support" +
supportCount.ToString(System.Globalization.CultureInfo.InvariantCulture)] = result3;
```

```
00603                                       }
00604                                   }
00605                                   else
00606                                   {
00607
00608                                       string name = "Unknown";
00609
00610                                       if (node.Attributes.ContainsKey(name))
00611                                       {
00612                                           int ind = 2;
00613                                           string newName = name +
      ind.ToString(System.Globalization.CultureInfo.InvariantCulture);
00614
00615                                           while (node.Attributes.ContainsKey(newName))
00616                                           {
00617                                               ind++;
00618                                               newName = name +
      ind.ToString(System.Globalization.CultureInfo.InvariantCulture);
00619                                           }
00620
00621                                           name = newName;
00622                                       }
00623
00624                                       node.Attributes.Add(name, value);
00625                                   }
00626                               }
00627                               break;
00628                           }
00629                       }
00630
00631                       lastSeparator = c2;
00632                       nameFinished = false;
00633                       attributeName.Clear();
00634                       attributeValue.Clear();
00635
00636                       if (closedOuterBrackets)
00637                       {
00638                           closedOuterBrackets = false;
00639                           expectedClosingBrackets = '\0';
00640                           start = true;
00641                           withinBrackets = false;
00642                       }
00643
00644                       if (expectedClosingBrackets != '\0')
00645                       {
00646                           withinBrackets = true;
00647                       }
00648
00649                   }
00650                   else
00651                   {
00652                       if (closedOuterBrackets)
00653                       {
00654                           closedOuterBrackets = false;
00655                           expectedClosingBrackets = '\0';
00656                           start = true;
00657                           withinBrackets = false;
00658                       }
00659
00660                       if (expectedClosingBrackets != '\0')
00661                       {
00662                           withinBrackets = true;
00663                       }
00664
00665                       if (c2 == '[' && !escaped && !openQuotes && !openApostrophe)
00666                       {
00667                           openSquareCount++;
00668                       }
00669                       else if (c2 == ']' && !escaped && !openQuotes && !openApostrophe)
00670                       {
00671                           if (openSquareCount > 0)
00672                           {
00673                               openSquareCount--;
00674                           }
00675                           else if (expectedClosingBrackets == c2)
00676                           {
00677                               closedOuterBrackets = true;
00678                           }
00679                       }
00680                       else if (c2 == '{' && !escaped && !openQuotes && !openApostrophe)
00681                       {
00682                           openCurlyCount++;
00683                       }
00684                       else if (c2 == '}' && !escaped && !openQuotes && !openApostrophe)
00685                       {
00686                           if (openCurlyCount > 0)
00687                           {
```

```
00688                               openCurlyCount--;
00689                           }
00690                       }
00691
00692
00693                       if (!closedOuterBrackets && !escaping)
00694                       {
00695                           if (!nameFinished)
00696                           {
00697                               attributeName.Append(c2);
00698                           }
00699                           else
00700                           {
00701                               attributeValue.Append(c2);
00702                           }
00703
00704                       }
00705                   }
00706               }
00707
00708               if (double.IsNaN(node.Support) && node.Attributes.ContainsKey("prob"))
00709               {
00710                   node.Support = Convert.ToDouble(node.Attributes["prob"],
      System.Globalization.CultureInfo.InvariantCulture);
00711               }
00712           }
00713
00714 /// <summary>
00715 /// Writes a <see cref="TreeNode"/> to a <see cref="string"/>.
00716 /// </summary>
00717 /// <param name="tree">The tree to write.</param>
00718 /// <param name="nwka">If this is false, a Newick-compliant string is produced, only including the
      <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see cref="TreeNode.Support"/>
      attributes of each branch.
00719 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
00720 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
      of the nodes are placed between single quotes.</param>
00721 /// <returns>A <see cref="string"/> containing the Newick or NWKA representation of the <see
      cref="TreeNode"/>.</returns>
00722           public static string WriteTree(TreeNode tree, bool nwka, bool singleQuoted = false)
00723           {
00724               Contract.Requires(tree != null);
00725
00726               if (!nwka)
00727               {
00728                   if (tree.Children.Count == 0)
00729                   {
00730                       StringBuilder tbr = new StringBuilder();
00731                       if (singleQuoted)
00732                       {
00733                           tbr.Append("'");
00734                           tbr.Append(tree.Name.Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00735                           tbr.Append("'");
00736                       }
00737                       else
00738                       {
00739                           tbr.Append(tree.Name);
00740                       }
00741                       if (!double.IsNaN(tree.Length))
00742                       {
00743                           tbr.Append(":");
00744                           tbr.Append(tree.Length.ToString(CultureInfo.InvariantCulture));
00745                       }
00746                       if (tree.Parent == null)
00747                       {
00748                           tbr.Append(";");
00749                       }
00750                       return tbr.ToString();
00751                   }
00752                   else
00753                   {
00754                       StringBuilder tbr = new StringBuilder("(");
00755
00756                       for (int i = 0; i < tree.Children.Count; i++)
00757                       {
00758                           tbr.Append(WriteTree(tree.Children[i], false, singleQuoted));
00759                           if (i < tree.Children.Count - 1)
00760                           {
00761                               tbr.Append(",");
00762                           }
00763                       }
00764                       tbr.Append(")");
00765                       if (!string.IsNullOrEmpty(tree.Name) && (singleQuoted ||
      double.IsNaN(tree.Support)))
00766                       {
```

```
00767                         if (singleQuoted)
00768                         {
00769                             tbr.Append("'");
00770                             tbr.Append(tree.Name.Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00771                             tbr.Append("'");
00772                         }
00773                         else
00774                         {
00775                             tbr.Append(tree.Name);
00776                         }
00777                     }
00778                     if (!double.IsNaN(tree.Support))
00779                     {
00780                         tbr.Append(tree.Support.ToString(CultureInfo.InvariantCulture));
00781                     }
00782                     if (!double.IsNaN(tree.Length))
00783                     {
00784                         tbr.Append(":");
00785                         tbr.Append(tree.Length.ToString(CultureInfo.InvariantCulture));
00786                     }
00787                     if (tree.Parent == null)
00788                     {
00789                         tbr.Append(";");
00790                     }
00791                     return tbr.ToString();
00792                 }
00793             }
00794             else
00795             {
00796                 if (tree.Children.Count == 0)
00797                 {
00798                     StringBuilder tbr = new StringBuilder();
00799
00800                     if (!string.IsNullOrEmpty(tree.Name))
00801                     {
00802                         tbr.Append("'");
00803                         tbr.Append(tree.Name.Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00804                         tbr.Append("'");
00805                     }
00806
00807                     if (!double.IsNaN(tree.Length))
00808                     {
00809                         tbr.Append(":");
00810                         tbr.Append(tree.Length.ToString(CultureInfo.InvariantCulture));
00811                     }
00812
00813                     if (tree.Attributes.Count > 3)
00814                     {
00815                         tbr.Append("[");
00816                         bool first = true;
00817                         foreach (KeyValuePair<string, object> attribute in tree.Attributes)
00818                         {
00819                             if (!attribute.Key.Equals("Name", StringComparison.OrdinalIgnoreCase) &&
      !attribute.Key.Equals("Length", StringComparison.OrdinalIgnoreCase))
00820                             {
00821                                 if (attribute.Value is double)
00822                                 {
00823                                     tbr.Append((!first ?  "," :  ""));
00824                                     tbr.Append(attribute.Key);
00825                                     tbr.Append("=");
00826
      tbr.Append(((double)attribute.Value).ToString(CultureInfo.InvariantCulture));
00827                                 }
00828                                 else
00829                                 {
00830                                     if (!attribute.Value.ToString().Contains('\'',
      StringComparison.OrdinalIgnoreCase))
00831                                     {
00832                                         tbr.Append(!first ?  "," :  "");
00833                                         tbr.Append(attribute.Key);
00834                                         tbr.Append("='");
00835                                         tbr.Append(attribute.Value.ToString().Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00836                                         tbr.Append("'");
00837                                     }
00838                                     else
00839                                     {
00840                                         tbr.Append(!first ?  "," :  "");
00841                                         tbr.Append(attribute.Key);
00842                                         tbr.Append("=\"");
00843                                         tbr.Append(attribute.Value.ToString().Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
```

```
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00844                                     tbr.Append("\"");
00845                                 }
00846
00847                             }
00848                             first = false;
00849                         }
00850                     }
00851                     tbr.Append("]");
00852                 }
00853
00854                 if (tree.Parent == null)
00855                 {
00856                     tbr.Append(";");
00857                 }
00858                 return tbr.ToString();
00859             }
00860             else
00861             {
00862                 StringBuilder tbr = new StringBuilder("(");
00863
00864                 for (int i = 0; i < tree.Children.Count; i++)
00865                 {
00866                     tbr.Append(WriteTree(tree.Children[i], true, true));
00867                     if (i < tree.Children.Count - 1)
00868                     {
00869                         tbr.Append(",");
00870                     }
00871                 }
00872                 tbr.Append(")");
00873
00874                 if (!string.IsNullOrEmpty(tree.Name))
00875                 {
00876                     tbr.Append("'");
00877                     tbr.Append(tree.Name.Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00878                     tbr.Append("'");
00879                 }
00880                 if (tree.Support >= 0)
00881                 {
00882                     tbr.Append(tree.Support.ToString(CultureInfo.InvariantCulture));
00883                 }
00884                 if (!double.IsNaN(tree.Length))
00885                 {
00886                     tbr.Append(":");
00887                     tbr.Append(tree.Length.ToString(CultureInfo.InvariantCulture));
00888                 }
00889
00890                 if (tree.Attributes.Count > 3)
00891                 {
00892                     tbr.Append("[");
00893                     bool first = true;
00894                     foreach (KeyValuePair<string, object> attribute in tree.Attributes)
00895                     {
00896                         if (!attribute.Key.Equals("Name", StringComparison.OrdinalIgnoreCase) &&
      !attribute.Key.Equals("Support", StringComparison.OrdinalIgnoreCase) &&
      !attribute.Key.Equals("Length", StringComparison.OrdinalIgnoreCase))
00897                             {
00898                                 if (attribute.Value is double)
00899                                 {
00900                                     tbr.Append(!first ?  "," :  "");
00901                                     tbr.Append(attribute.Key);
00902                                     tbr.Append("=");
00903
      tbr.Append(((double)attribute.Value).ToString(CultureInfo.InvariantCulture));
00904                                 }
00905                                 else
00906                                 {
00907                                     if (!attribute.Value.ToString().Contains('\",
      StringComparison.OrdinalIgnoreCase))
00908                                     {
00909                                         tbr.Append(!first ?  "," :  "");
00910                                         tbr.Append(attribute.Key);
00911                                         tbr.Append("='");
00912                                         tbr.Append(attribute.Value.ToString().Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
      StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00913                                         tbr.Append("'");
00914                                     }
00915                                     else
00916                                     {
00917                                         tbr.Append(!first ?  "," :  "");
00918                                         tbr.Append(attribute.Key);
00919                                         tbr.Append("=\"");
00920                                         tbr.Append(attribute.Value.ToString().Replace("\\", "\\\\",
      StringComparison.OrdinalIgnoreCase).Replace("'", "\\'",
```

```
    StringComparison.OrdinalIgnoreCase).Replace("\"", "\\\"", StringComparison.OrdinalIgnoreCase));
00921                                       tbr.Append("\"");
00922                                   }
00923
00924                                   }
00925                                   first = false;
00926                               }
00927                           }
00928                           tbr.Append("]");
00929                       }
00930
00931                       if (tree.Parent == null)
00932                       {
00933                           tbr.Append(";");
00934                       }
00935                       return tbr.ToString();
00936                   }
00937               }
00938           }
00939
00940 /// <summary>
00941 /// Writes a single tree in Newick o Newick-with-Attributes format.
00942 /// </summary>
00943 /// <param name="tree">The tree to be written.</param>
00944 /// <param name="outputStream">The <see cref="Stream"/> on which the tree should be written.</param>
00945 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
    after the end of this method.</param>
00946 /// <param name="nwka">If this is false, a Newick-compliant string is produced for each tree, only
    including the <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see
    cref="TreeNode.Support"/> attributes of each branch.
00947 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
00948 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
    of the nodes are placed between single quotes.</param>
00949           public static void WriteTree(TreeNode tree, Stream outputStream, bool keepOpen = false, bool
    nwka = true, bool singleQuoted = false)
00950           {
00951               WriteAllTrees(new List<TreeNode> { tree }, outputStream, keepOpen, null, nwka,
    singleQuoted);
00952           }
00953
00954 /// <summary>
00955 /// Writes a single tree in Newick o Newick-with-Attributes format.
00956 /// </summary>
00957 /// <param name="tree">The tree to be written.</param>
00958 /// <param name="outputFile">The file on which the tree should be written.</param>
00959 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00960 /// <param name="nwka">If this is false, a Newick-compliant string is produced for each tree, only
    including the <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see
    cref="TreeNode.Support"/> attributes of each branch.
00961 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
00962 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
    of the nodes are placed between single quotes.</param>
00963           public static void WriteTree(TreeNode tree, string outputFile, bool append = false, bool nwka
    = true, bool singleQuoted = false)
00964           {
00965               using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
    File.Create(outputFile);
00966               WriteAllTrees(new List<TreeNode>() { tree }, outputStream, false, null, nwka,
    singleQuoted);
00967           }
00968
00969 /// <summary>
00970 /// Writes trees in Newick o Newick-with-Attributes format.
00971 /// </summary>
00972 /// <param name="trees">An <see cref="IEnumerable{T}"/> containing the trees to be written.  It will
    ony be enumerated once.</param>
00973 /// <param name="outputFile">The file on which the trees should be written.</param>
00974 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
00975 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
    written, with the number of trees written so far.</param>
00976 /// <param name="nwka">If this is false, a Newick-compliant string is produced for each tree, only
    including the <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see
    cref="TreeNode.Support"/> attributes of each branch.
00977 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
00978 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
    of the nodes are placed between single quotes.</param>
00979           public static void WriteAllTrees(IEnumerable<TreeNode> trees, string outputFile, bool append =
    false, Action<int> progressAction = null, bool nwka = true, bool singleQuoted = false)
00980           {
00981               using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
    File.Create(outputFile);
00982               WriteAllTrees(trees, outputStream, false, progressAction, nwka, singleQuoted);
00983           }
00984
00985 /// <summary>
00986 /// Writes trees in Newick o Newick-with-Attributes format.
00987 /// </summary>
```

```
00988 /// <param name="trees">An <see cref="IEnumerable{T}"/> containing the trees to be written.  It will
      ony be enumerated once.</param>
00989 /// <param name="outputStream">The <see cref="Stream"/> on which the trees should be written.</param>
00990 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
00991 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
      written, with the number of trees written so far.</param>
00992 /// <param name="nwka">If this is false, a Newick-compliant string is produced for each tree, only
      including the <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see
      cref="TreeNode.Support"/> attributes of each branch.
00993 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
00994 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
      of the nodes are placed between single quotes.</param>
00995         public static void WriteAllTrees(IEnumerable<TreeNode> trees, Stream outputStream, bool
      keepOpen = false, Action<int> progressAction = null, bool nwka = true, bool singleQuoted = false)
00996         {
00997             Contract.Requires(trees != null);
00998             using StreamWriter sw = new StreamWriter(outputStream, Encoding.UTF8, 1024, keepOpen);
00999             int count = 0;
01000             foreach (TreeNode tree in trees)
01001             {
01002                 sw.WriteLine(WriteTree(tree, nwka, singleQuoted));
01003                 count++;
01004                 progressAction?.Invoke(count);
01005             }
01006         }
01007
01008 /// <summary>
01009 /// Writes trees in Newick o Newick-with-Attributes format.
01010 /// </summary>
01011 /// <param name="trees">A collection of trees to be written.  Each tree will only be accessed
      once.</param>
01012 /// <param name="outputFile">The file on which the trees should be written.</param>
01013 /// <param name="append">Specifies whether the file should be overwritten or appended to.</param>
01014 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
      written, with a value between 0 and 1 depending on how many trees have been written so far.</param>
01015 /// <param name="nwka">If this is false, a Newick-compliant string is produced for each tree, only
      including the <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see
      cref="TreeNode.Support"/> attributes of each branch.
01016 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
01017 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
      of the nodes are placed between single quotes.</param>
01018         public static void WriteAllTrees(IList<TreeNode> trees, string outputFile, bool append =
      false, Action<double> progressAction = null, bool nwka = true, bool singleQuoted = false)
01019         {
01020             using FileStream outputStream = append ?  new FileStream(outputFile, FileMode.Append) :
      File.Create(outputFile);
01021             WriteAllTrees(trees, outputStream, false, progressAction, nwka, singleQuoted);
01022         }
01023
01024 /// <summary>
01025 /// Writes trees in Newick o Newick-with-Attributes format.
01026 /// </summary>
01027 /// <param name="trees">A collection of trees to be written.  Each tree will only be accessed
      once.</param>
01028 /// <param name="outputStream">The <see cref="Stream"/> on which the trees should be written.</param>
01029 /// <param name="keepOpen">Determines whether the <paramref name="outputStream"/> should be kept open
      after the end of this method.</param>
01030 /// <param name="progressAction">An <see cref="Action"/> that will be invoked after each tree is
      written, with a value between 0 and 1 depending on how many trees have been written so far.</param>
01031 /// <param name="nwka">If this is false, a Newick-compliant string is produced for each tree, only
      including the <see cref="TreeNode.Name"/>, <see cref="TreeNode.Length"/> and <see
      cref="TreeNode.Support"/> attributes of each branch.
01032 /// Otherwise, a Newick-with-Attributes string is produced, including all attributes.</param>
01033 /// <param name="singleQuoted">If <paramref name="nwka"/> is false, this determines whether the names
      of the nodes are placed between single quotes.</param>
01034         public static void WriteAllTrees(IList<TreeNode> trees, Stream outputStream, bool keepOpen =
      false, Action<double> progressAction = null, bool nwka = true, bool singleQuoted = false)
01035         {
01036             using StreamWriter sw = new StreamWriter(outputStream, Encoding.UTF8, 1024, keepOpen);
01037             for (int i = 0; i < trees.Count; i++)
01038             {
01039                 sw.WriteLine(WriteTree(trees[i], nwka, singleQuoted));
01040                 progressAction?.Invoke((double)i / trees.Count);
01041             }
01042         }
01043
01044     }
01045 }
```

8.8 LikelihoodScores.cs

```
00001 using MathNet.Numerics.LinearAlgebra;
```

```
00002 using MathNet.Numerics.LinearAlgebra.Complex;
00003 using PhyloTree.SequenceSimulation;
00004 using PhyloTree.TreeBuilding;
00005 using System;
00006 using System.Collections.Generic;
00007 using System.Linq;
00008 using System.Reflection;
00009 using System.Text;
00010 using System.Threading.Tasks;
00011
00012 namespace PhyloTree.SequenceScores
00013 {
00014 /// <summary>
00015 /// Contains methods to compute likelihood scores on a tree.
00016 /// </summary>
00017     public static class LikelihoodScores
00018     {
00019         private static int MaxInd(this double[] arr, double[] multipArr)
00020         {
00021             int tbr = 0;
00022
00023             for (int i = 0; i < arr.Length; i++)
00024             {
00025                 if (arr[i] > arr[tbr] && multipArr[i] > 0)
00026                 {
00027                     tbr = i;
00028                 }
00029             }
00030
00031             return tbr;
00032         }
00033
00034         private static int MaxInd(this double[] arr)
00035         {
00036             int tbr = 0;
00037
00038             for (int i = 0; i < arr.Length; i++)
00039             {
00040                 if (arr[i] > arr[tbr])
00041                 {
00042                     tbr = i;
00043                 }
00044             }
00045
00046             return tbr;
00047         }
00048
00049         private static double Log1p(double x)
00050         {
00051             if (x <= -1)
00052             {
00053                 return double.NegativeInfinity;
00054             }
00055             else if (Math.Abs(x) > 0.0001)
00056             {
00057                 return Math.Log(1 + x);
00058             }
00059             else
00060             {
00061                 return (1 - 0.5 * x) * x;
00062             }
00063         }
00064
00065         const double Tolerance = 1e-7;
00066
00067         private static void TimesLogVectorAndAdd(this Matrix<double> mat, double[] logVector, double[]
      addToVector)
00068         {
00069             int maxInd = logVector.MaxInd();
00070
00071             for (int i = 0; i < mat.RowCount; i++)
00072             {
00073                 double toBeAdded = logVector[maxInd] + Math.Log(mat[i, maxInd]);
00074
00075                 double log1pArg = 0;
00076
00077                 for (var j = 0; j < mat.ColumnCount; j++)
00078                 {
00079                     if (j != maxInd)
00080                     {
00081                         log1pArg += mat[i, j] / mat[i, maxInd] * Math.Exp(logVector[j] -
      logVector[maxInd]);
00082                     }
00083                 }
00084
00085                 if (!double.IsNaN(log1pArg))
00086                 {
```

```
00087                     toBeAdded += Log1p(log1pArg);
00088                     addToVector[i] += toBeAdded;
00089                     if (addToVector[i] > Tolerance)
00090                     {
00091                         addToVector[i] = double.NaN;
00092                     }
00093                     else if (addToVector[i] > 0)
00094                     {
00095                         addToVector[i] = 0;
00096                     }
00097                 }
00098                 else
00099                 {
00100                     double logArg = 0;
00101                     for (var j = 0; j < mat.ColumnCount; j++)
00102                     {
00103                         logArg += mat[i, j] * Math.Exp(logVector[j]);
00104                     }
00105 
00106                     addToVector[i] += Math.Log(logArg);
00107 
00108                     if (addToVector[i] > Tolerance)
00109                     {
00110                         addToVector[i] = double.NaN;
00111                     }
00112                     else if (addToVector[i] > 0)
00113                     {
00114                         addToVector[i] = 0;
00115                     }
00116                 }
00117             }
00118         }
00119 
00120         private static double LogSumExpTimes(double[] logs, double[] multipliers)
00121         {
00122             int maxInd = logs.MaxInd(multipliers);
00123 
00124             double log1pArg = 0;
00125 
00126             for (int i = 0; i < logs.Length; i++)
00127             {
00128                 if (i != maxInd)
00129                 {
00130                     log1pArg += Math.Exp(logs[i] - logs[maxInd]) * (multipliers[i] /
      multipliers[maxInd]);
00131                 }
00132             }
00133 
00134             if (!double.IsNaN(log1pArg))
00135             {
00136                 return logs[maxInd] + Math.Log(multipliers[maxInd]) + Log1p(log1pArg);
00137             }
00138             else
00139             {
00140                 double logArg = 0;
00141 
00142                 for (int i = 0; i < logs.Length; i++)
00143                 {
00144                     logArg += Math.Exp(logs[i]) * multipliers[i];
00145                 }
00146 
00147                 return Math.Log(logArg);
00148             }
00149         }
00150 
00151 /// <summary>
00152 /// Computes the likelihood for the specified character using the specified rate matrix and
      evolutionary rate.
00153 /// </summary>
00154 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00155 /// <param name="tipStates">The character states at the tips of the tree.
00156 /// This <see cref="Dictionary{String, Char}"/> should contain an entry for each terminal node of the
      tree,
00157 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00158 /// is the character state.</param>
00159 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00160 /// <param name="rate">The overall evolutionary rate of the character.</param>
00161 /// <returns>The log-likelihood for the specified character.</returns>
00162 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00163 /// contain an entry for one of the tips in the tree.</exception>
00164         public static double GetLogLikelihood(this TreeNode tree, Dictionary<string, char> tipStates,
      RateMatrix rateMatrix, double rate)
00165         {
00166             char[] states = rateMatrix.GetStates();
00167 
00168             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
```

```
00169
00170             for (int i = 0; i < states.Length; i++)
00171             {
00172                 stateIndices[states[i]] = i;
00173             }
00174
00175             Matrix<double> actualMatrix = rateMatrix.GetMatrix();
00176
00177             List<TreeNode> nodes = tree.GetChildrenRecursive();
00178
00179             Dictionary<string, int> indices = new Dictionary<string, int>();
00180             int[][] children = new int[nodes.Count][];
00181
00182             for (int i = nodes.Count - 1; i >= 0; i--)
00183             {
00184                 indices[nodes[i].Id] = i;
00185
00186                 children[i] = new int[nodes[i].Children.Count];
00187
00188                 for (int j = 0; j < nodes[i].Children.Count; j++)
00189                 {
00190                     children[i][j] = indices[nodes[i].Children[j].Id];
00191                 }
00192             }
00193
00194             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00195             double[][] logLikelihoods = new double[nodes.Count][];
00196             MatrixExponential cachedExp = actualMatrix.FastExponential(1);
00197
00198             for (int i = nodes.Count - 1; i >= 0; i--)
00199             {
00200                 if (nodes[i].Children.Count == 0)
00201                 {
00202                     char state;
00203
00204                     if (!tipStates.TryGetValue(nodes[i].Name, out state) &&
    !tipStates.TryGetValue(nodes[i].Id, out state))
00205                     {
00206                         throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
    nodes[i].Id + ") has no associated state!");
00207                     }
00208
00209                     logLikelihoods[i] = new double[states.Length];
00210
00211                     if (state != '-')
00212                     {
00213                         int index = stateIndices[state];
00214                         for (int j = 0; j < states.Length; j++)
00215                         {
00216                             if (j != index)
00217                             {
00218                                 logLikelihoods[i][j] = double.NegativeInfinity;
00219                             }
00220                             else
00221                             {
00222                                 logLikelihoods[i][j] = 0;
00223                             }
00224                         }
00225                     }
00226                 }
00227                 else
00228                 {
00229                     logLikelihoods[i] = new double[states.Length];
00230
00231                     for (int j = 0; j < children[i].Length; j++)
00232                     {
00233                         Matrix<double> matrix = actualMatrix.FastExponential(rate *
    nodes[children[i][j]].Length, cachedExp).Result;
00234                         matrix.TimesLogVectorAndAdd(logLikelihoods[children[i][j]],
    logLikelihoods[i]);
00235                     }
00236                 }
00237             }
00238
00239             return LogSumExpTimes(logLikelihoods[0], equilibriumFrequencies);
00240         }
00241
00242 /// <summary>
00243 /// Estimates the maximum-likelihood evolutionary rate for the specified character under the specified
    rate matrix and computes the
00244 /// likelihood.
00245 /// </summary>
00246 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00247 /// <param name="tipStates">The character states at the tips of the tree.
00248 /// This <see cref="Dictionary{String, Char}"/> should contain an entry for each terminal node of the
    tree,
00249 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
```

```
00250 /// is the character state.</param>
00251 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00252 /// <returns>The maximum-likelihood rate estimate and the log-likelihood for the specified
      character.</returns>
00253 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00254 /// contain an entry for one of the tips in the tree.</exception>
00255         public static (double rateMLE, double logLikelihood) GetLogLikelihood(this TreeNode tree,
      Dictionary<string, char> tipStates, RateMatrix rateMatrix)
00256         {
00257             char[] states = rateMatrix.GetStates();
00258 
00259             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00260 
00261             for (int i = 0; i < states.Length; i++)
00262             {
00263                 stateIndices[states[i]] = i;
00264             }
00265 
00266             Matrix<double> actualMatrix = rateMatrix.GetMatrix();
00267 
00268             List<TreeNode> nodes = tree.GetChildrenRecursive();
00269 
00270             Dictionary<string, int> indices = new Dictionary<string, int>();
00271             int[][] children = new int[nodes.Count][];
00272 
00273             for (int i = nodes.Count - 1; i >= 0; i--)
00274             {
00275                 indices[nodes[i].Id] = i;
00276 
00277                 children[i] = new int[nodes[i].Children.Count];
00278 
00279                 for (int j = 0; j < nodes[i].Children.Count; j++)
00280                 {
00281                     children[i][j] = indices[nodes[i].Children[j].Id];
00282                 }
00283             }
00284 
00285             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00286 
00287             double likelihoodFunction(double rate)
00288             {
00289                 double[][] logLikelihoods = new double[nodes.Count][];
00290                 MatrixExponential cachedExp = actualMatrix.FastExponential(1);
00291 
00292                 for (int i = nodes.Count - 1; i >= 0; i--)
00293                 {
00294                     if (nodes[i].Children.Count == 0)
00295                     {
00296                         char state;
00297 
00298                         if (!tipStates.TryGetValue(nodes[i].Name, out state) &&
      !tipStates.TryGetValue(nodes[i].Id, out state))
00299                         {
00300                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00301                         }
00302 
00303                         logLikelihoods[i] = new double[states.Length];
00304 
00305                         if (state != '-')
00306                         {
00307                             int index = stateIndices[state];
00308                             for (int j = 0; j < states.Length; j++)
00309                             {
00310                                 if (j != index)
00311                                 {
00312                                     logLikelihoods[i][j] = double.NegativeInfinity;
00313                                 }
00314                                 else
00315                                 {
00316                                     logLikelihoods[i][j] = 0;
00317                                 }
00318                             }
00319                         }
00320                     }
00321                     else
00322                     {
00323                         logLikelihoods[i] = new double[states.Length];
00324 
00325                         for (int j = 0; j < children[i].Length; j++)
00326                         {
00327                             Matrix<double> matrix = actualMatrix.FastExponential(rate *
      nodes[children[i][j]].Length, cachedExp).Result;
00328                             matrix.TimesLogVectorAndAdd(logLikelihoods[children[i][j]],
      logLikelihoods[i]);
00329                         }
```

```
00330                     }
00331                 }
00332 
00333                 return LogSumExpTimes(logLikelihoods[0], equilibriumFrequencies);
00334             }
00335 
00336             double mleRate = MathNet.Numerics.FindMinimum.OfScalarFunction(x =>
      -likelihoodFunction(x), 1);
00337 
00338             return (mleRate, likelihoodFunction(mleRate));
00339         }
00340 
00341 /// <summary>
00342 /// Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and
      evolutionary rate,
00343 /// and returns the likelihood for each site.
00344 /// </summary>
00345 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00346 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00347 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00348 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00349 /// is the character state sequence.</param>
00350 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00351 /// <param name="rate">The overall evolutionary rate of the character.</param>
00352 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00353 /// <returns>A <see cref="T:double[]"/> array containing the log-likelihood for each site in the
      alignment.</returns>
00354 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00355 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00356 /// contain an entry for one of the tips in the tree.</exception>
00357         public static double[] GetLogLikelihoods(this TreeNode tree, Dictionary<string,
      IReadOnlyList<char» tipStates, RateMatrix rateMatrix, double rate, int maxParallelism = -1)
00358         {
00359             char[] states = rateMatrix.GetStates();
00360 
00361             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00362 
00363             for (int i = 0; i < states.Length; i++)
00364             {
00365                 stateIndices[states[i]] = i;
00366             }
00367 
00368             Matrix<double> actualMatrix = rateMatrix.GetMatrix();
00369 
00370             List<TreeNode> nodes = tree.GetChildrenRecursive();
00371 
00372             Dictionary<string, int> indices = new Dictionary<string, int>();
00373             int[][] children = new int[nodes.Count][];
00374 
00375             for (int i = nodes.Count - 1; i >= 0; i--)
00376             {
00377                 indices[nodes[i].Id] = i;
00378 
00379                 children[i] = new int[nodes[i].Children.Count];
00380 
00381                 for (int j = 0; j < nodes[i].Children.Count; j++)
00382                 {
00383                     children[i][j] = indices[nodes[i].Children[j].Id];
00384                 }
00385             }
00386 
00387             int sequenceLength = 0;
00388 
00389             foreach (KeyValuePair<string, IReadOnlyList<char» kvp in tipStates)
00390             {
00391                 if (sequenceLength == 0)
00392                 {
00393                     sequenceLength = kvp.Value.Count;
00394                 }
00395                 else
00396                 {
00397                     if (sequenceLength != kvp.Value.Count)
00398                     {
00399                         throw new ArgumentException("Not all the sequences have the same length!");
00400                     }
00401                 }
00402             }
00403 
00404             double[] siteLikelihoods = new double[sequenceLength];
00405 
00406 
00407             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00408             MatrixExponential cachedExp = actualMatrix.FastExponential(1);
00409 
```

```
00410             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
      maxParallelism }, siteIndex =>
00411             {
00412                 double[][] logLikelihoods = new double[nodes.Count][];
00413 
00414                 for (int i = nodes.Count - 1; i >= 0; i--)
00415                 {
00416                     if (nodes[i].Children.Count == 0)
00417                     {
00418                         IReadOnlyList<char> seq;
00419 
00420                         if (!tipStates.TryGetValue(nodes[i].Name, out seq) &&
      !tipStates.TryGetValue(nodes[i].Id, out seq))
00421                         {
00422                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00423                         }
00424 
00425                         logLikelihoods[i] = new double[states.Length];
00426 
00427                         if (seq[siteIndex] != '-')
00428                         {
00429                             int index = stateIndices[seq[siteIndex]];
00430                             for (int j = 0; j < states.Length; j++)
00431                             {
00432                                 if (j != index)
00433                                 {
00434                                     logLikelihoods[i][j] = double.NegativeInfinity;
00435                                 }
00436                                 else
00437                                 {
00438                                     logLikelihoods[i][j] = 0;
00439                                 }
00440                             }
00441                         }
00442                     }
00443                     else
00444                     {
00445                         logLikelihoods[i] = new double[states.Length];
00446 
00447                         for (int j = 0; j < children[i].Length; j++)
00448                         {
00449                             Matrix<double> matrix = actualMatrix.FastExponential(rate *
      nodes[children[i][j]].Length, cachedExp).Result;
00450                             matrix.TimesLogVectorAndAdd(logLikelihoods[children[i][j]],
      logLikelihoods[i]);
00451                         }
00452                     }
00453                 }
00454 
00455                 siteLikelihoods[siteIndex] = LogSumExpTimes(logLikelihoods[0],
      equilibriumFrequencies);
00456             }); ;
00457 
00458             return siteLikelihoods;
00459         }
00460 
00461 /// <summary>
00462 /// Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and
      evolutionary rate,
00463 /// and returns the total likelihood for the tree.
00464 /// </summary>
00465 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00466 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00467 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00468 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00469 /// is the character state sequence.</param>
00470 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00471 /// <param name="rate">The overall evolutionary rate of the character.</param>
00472 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00473 /// <returns>The log-likelihood for the alignment.</returns>
00474 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00475 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00476 /// contain an entry for one of the tips in the tree.</exception>
00477         public static double GetLogLikelihood(this TreeNode tree, Dictionary<string,
      IReadOnlyList<char» tipStates, RateMatrix rateMatrix, double rate, int maxParallelism = -1)
00478         {
00479             return GetLogLikelihoods(tree, tipStates, rateMatrix, rate, maxParallelism).Sum();
00480         }
00481 
00482 
00483 /// <summary>
00484 /// Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and
      evolutionary rate,
```

```
00485 /// and returns the likelihood for each site.
00486 /// </summary>
00487 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00488 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00489 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00490 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00491 /// is the character state sequence.</param>
00492 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00493 /// <param name="rate">The overall evolutionary rate of the character.</param>
00494 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00495 /// <returns>A <see cref="T:double[]"/> array containing the log-likelihood for each site in the
      alignment.</returns>
00496 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00497 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00498 /// contain an entry for one of the tips in the tree.</exception>
00499         public static double[] GetLogLikelihoods(this TreeNode tree, Dictionary<string, Sequence>
      tipStates, RateMatrix rateMatrix, double rate, int maxParallelism)
00500         {
00501             return GetLogLikelihoods(tree, new Dictionary<string, IReadOnlyList<char»(from el in
      tipStates select new KeyValuePair<string, IReadOnlyList<char»(el.Key, el.Value)), rateMatrix, rate,
      maxParallelism);
00502         }
00503
00504 /// <summary>
00505 /// Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and
      evolutionary rate,
00506 /// and returns the total likelihood for the tree.
00507 /// </summary>
00508 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00509 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00510 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00511 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00512 /// is the character state sequence.</param>
00513 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00514 /// <param name="rate">The overall evolutionary rate of the character.</param>
00515 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00516 /// <returns>The log-likelihood for the alignment.</returns>
00517 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00518 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00519 /// contain an entry for one of the tips in the tree.</exception>
00520         public static double GetLogLikelihood(this TreeNode tree, Dictionary<string, Sequence>
      tipStates, RateMatrix rateMatrix, double rate, int maxParallelism = -1)
00521         {
00522             return GetLogLikelihoods(tree, tipStates, rateMatrix, rate, maxParallelism).Sum();
00523         }
00524
00525 /// <summary>
00526 /// Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and
      evolutionary rate,
00527 /// and returns the likelihood for each site.
00528 /// </summary>
00529 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00530 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00531 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00532 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00533 /// is the character state sequence.</param>
00534 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00535 /// <param name="rate">The overall evolutionary rate of the character.</param>
00536 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00537 /// <returns>A <see cref="T:double[]"/> array containing the log-likelihood for each site in the
      alignment.</returns>
00538 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00539 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00540 /// contain an entry for one of the tips in the tree.</exception>
00541         public static double[] GetLogLikelihoods(this TreeNode tree, Dictionary<string, string>
      tipStates, RateMatrix rateMatrix, double rate, int maxParallelism = -1)
00542         {
00543             return GetLogLikelihoods(tree, new Dictionary<string, IReadOnlyList<char»(from el in
      tipStates select new KeyValuePair<string, IReadOnlyList<char»(el.Key, el.Value.ToCharArray())),
      rateMatrix, rate, maxParallelism);
00544         }
00545
00546 /// <summary>
00547 /// Computes the likelihood for a sequence alignment on a tree, using the specified rate matrix and
      evolutionary rate,
00548 /// and returns the total likelihood for the tree.
00549 /// </summary>
00550 /// <param name="tree">The tree on which the likelihood should be computed.</param>
```

```
00551 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00552 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00553 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00554 /// is the character state sequence.</param>
00555 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00556 /// <param name="rate">The overall evolutionary rate of the character.</param>
00557 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00558 /// <returns>The log-likelihood for the alignment.</returns>
00559 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00560 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00561 /// contain an entry for one of the tips in the tree.</exception>
00562         public static double GetLogLikelihood(this TreeNode tree, Dictionary<string, string>
      tipStates, RateMatrix rateMatrix, double rate, int maxParallelism = -1)
00563         {
00564             return GetLogLikelihoods(tree, tipStates, rateMatrix, rate, maxParallelism).Sum();
00565         }
00566 
00567 
00568 /// <summary>
00569 /// Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the
      specified rate matrix and computes the
00570 /// likelihood.
00571 /// </summary>
00572 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00573 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00574 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00575 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00576 /// is the character state sequence.</param>
00577 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00578 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00579 /// <returns>A <see cref="T:double[]"/> array containing the log-likelihood for each site in the
      alignment.</returns>
00580 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00581 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00582 /// contain an entry for one of the tips in the tree.</exception>
00583         public static (double rateMLE, double[] logLikelihoods) GetLogLikelihoods(this TreeNode tree,
      Dictionary<string, IReadOnlyList<char» tipStates, RateMatrix rateMatrix, int maxParallelism = -1)
00584         {
00585             char[] states = rateMatrix.GetStates();
00586 
00587             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00588 
00589             for (int i = 0; i < states.Length; i++)
00590             {
00591                 stateIndices[states[i]] = i;
00592             }
00593 
00594             Matrix<double> actualMatrix = rateMatrix.GetMatrix();
00595 
00596             List<TreeNode> nodes = tree.GetChildrenRecursive();
00597 
00598             Dictionary<string, int> indices = new Dictionary<string, int>();
00599             int[][] children = new int[nodes.Count][];
00600 
00601             for (int i = nodes.Count - 1; i >= 0; i--)
00602             {
00603                 indices[nodes[i].Id] = i;
00604 
00605                 children[i] = new int[nodes[i].Children.Count];
00606 
00607                 for (int j = 0; j < nodes[i].Children.Count; j++)
00608                 {
00609                     children[i][j] = indices[nodes[i].Children[j].Id];
00610                 }
00611             }
00612 
00613             int sequenceLength = 0;
00614 
00615             foreach (KeyValuePair<string, IReadOnlyList<char» kvp in tipStates)
00616             {
00617                 if (sequenceLength == 0)
00618                 {
00619                     sequenceLength = kvp.Value.Count;
00620                 }
00621                 else
00622                 {
00623                     if (sequenceLength != kvp.Value.Count)
00624                     {
00625                         throw new ArgumentException("Not all the sequences have the same length!");
00626                     }
00627                 }
```

```
00628             }
00629
00630             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00631             MatrixExponential cachedExp = actualMatrix.FastExponential(1);
00632
00633             double[] likelihoodFunction(double rate)
00634             {
00635                 double[] siteLikelihoods = new double[sequenceLength];
00636
00637                 Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
      maxParallelism }, siteIndex =>
00638                 {
00639                     double[][] logLikelihoods = new double[nodes.Count][];
00640
00641                     for (int i = nodes.Count - 1; i >= 0; i--)
00642                     {
00643                         if (nodes[i].Children.Count == 0)
00644                         {
00645                             IReadOnlyList<char> seq;
00646
00647                             if (!tipStates.TryGetValue(nodes[i].Name, out seq) &&
      !tipStates.TryGetValue(nodes[i].Id, out seq))
00648                             {
00649                                 throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00650                             }
00651
00652                             logLikelihoods[i] = new double[states.Length];
00653
00654                             if (seq[siteIndex] != '-')
00655                             {
00656                                 int index = stateIndices[seq[siteIndex]];
00657                                 for (int j = 0; j < states.Length; j++)
00658                                 {
00659                                     if (j != index)
00660                                     {
00661                                         logLikelihoods[i][j] = double.NegativeInfinity;
00662                                     }
00663                                     else
00664                                     {
00665                                         logLikelihoods[i][j] = 0;
00666                                     }
00667                                 }
00668                             }
00669                         }
00670                         else
00671                         {
00672                             logLikelihoods[i] = new double[states.Length];
00673
00674                             for (int j = 0; j < children[i].Length; j++)
00675                             {
00676                                 Matrix<double> matrix = actualMatrix.FastExponential(rate *
      nodes[children[i][j]].Length, cachedExp).Result;
00677                                 matrix.TimesLogVectorAndAdd(logLikelihoods[children[i][j]],
      logLikelihoods[i]);
00678                             }
00679                         }
00680                     }
00681
00682                     siteLikelihoods[siteIndex] = LogSumExpTimes(logLikelihoods[0],
      equilibriumFrequencies);
00683                 });
00684
00685                 return siteLikelihoods;
00686             }
00687
00688             double mleRate = MathNet.Numerics.FindMinimum.OfScalarFunction(x =>
      -likelihoodFunction(x).Sum(), 1);
00689
00690             return (mleRate, likelihoodFunction(mleRate));
00691         }
00692
00693
00694 /// <summary>
00695 /// Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the
      specified rate matrix and returns
00696 /// the total likelihood for the alignment.
00697 /// </summary>
00698 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00699 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00700 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00701 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00702 /// is the character state sequence.</param>
00703 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00704 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00705 /// <returns>The log-likelihood for the alignment.</returns>
```

```
00706 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00707 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00708 /// contain an entry for one of the tips in the tree.</exception>
00709         public static (double rateMLE, double logLikelihood) GetLogLikelihood(this TreeNode tree,
      Dictionary<string, IReadOnlyList<char>> tipStates, RateMatrix rateMatrix, int maxParallelism = -1)
00710         {
00711             (double rateMLE, double[] logLikelihoods) = GetLogLikelihoods(tree, tipStates, rateMatrix,
      maxParallelism);
00712 
00713             return (rateMLE, logLikelihoods.Sum());
00714         }
00715 
00716 /// <summary>
00717 /// Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the
      specified rate matrix and computes the
00718 /// likelihood.
00719 /// </summary>
00720 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00721 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00722 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00723 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00724 /// is the character state sequence.</param>
00725 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00726 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00727 /// <returns>A <see cref="T:double[]"/> array containing the log-likelihood for each site in the
      alignment.</returns>
00728 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00729 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00730 /// contain an entry for one of the tips in the tree.</exception>
00731         public static (double rateMLE, double[] logLikelihoods) GetLogLikelihoods(this TreeNode tree,
      Dictionary<string, Sequence> tipStates, RateMatrix rateMatrix, int maxParallelism = -1)
00732         {
00733             return GetLogLikelihoods(tree, new Dictionary<string, IReadOnlyList<char>>(from el in
      tipStates select new KeyValuePair<string, IReadOnlyList<char>>(el.Key, el.Value)), rateMatrix,
      maxParallelism);
00734         }
00735 
00736 /// <summary>
00737 /// Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the
      specified rate matrix and returns
00738 /// the total likelihood for the alignment.
00739 /// </summary>
00740 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00741 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00742 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00743 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00744 /// is the character state sequence.</param>
00745 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00746 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00747 /// <returns>The log-likelihood for the alignment.</returns>
00748 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00749 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00750 /// contain an entry for one of the tips in the tree.</exception>
00751         public static (double rateMLE, double logLikelihood) GetLogLikelihood(this TreeNode tree,
      Dictionary<string, Sequence> tipStates, RateMatrix rateMatrix, int maxParallelism = -1)
00752         {
00753             (double rateMLE, double[] logLikelihoods) = GetLogLikelihoods(tree, tipStates, rateMatrix,
      maxParallelism);
00754 
00755             return (rateMLE, logLikelihoods.Sum());
00756         }
00757 
00758 
00759 /// <summary>
00760 /// Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the
      specified rate matrix and computes the
00761 /// likelihood.
00762 /// </summary>
00763 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00764 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00765 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00766 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00767 /// is the character state sequence.</param>
00768 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00769 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00770 /// <returns>A <see cref="T:double[]"/> array containing the log-likelihood for each site in the
      alignment.</returns>
00771 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
```

```
      length.</exception>
00772 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00773 /// contain an entry for one of the tips in the tree.</exception>
00774         public static (double rateMLE, double[] logLikelihoods) GetLogLikelihoods(this TreeNode tree,
      Dictionary<string, string> tipStates, RateMatrix rateMatrix, int maxParallelism = -1)
00775         {
00776             return GetLogLikelihoods(tree, new Dictionary<string, IReadOnlyList<char»(from el in
      tipStates select new KeyValuePair<string, IReadOnlyList<char»(el.Key, el.Value.ToCharArray())),
      rateMatrix, maxParallelism);
00777         }
00778
00779 /// <summary>
00780 /// Estimates the maximum-likelihood evolutionary rate for the specified sequence alignment under the
      specified rate matrix and returns
00781 /// the total likelihood for the alignment.
00782 /// </summary>
00783 /// <param name="tree">The tree on which the likelihood should be computed.</param>
00784 /// <param name="tipStates">The aligned sequences at the tips of the tree.
00785 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00786 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00787 /// is the character state sequence.</param>
00788 /// <param name="rateMatrix">The rate matrix describing the evolution of the character.</param>
00789 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00790 /// <returns>The log-likelihood for the alignment.</returns>
00791 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00792 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00793 /// contain an entry for one of the tips in the tree.</exception>
00794         public static (double rateMLE, double logLikelihood) GetLogLikelihood(this TreeNode tree,
      Dictionary<string, string> tipStates, RateMatrix rateMatrix, int maxParallelism = -1)
00795         {
00796             (double rateMLE, double[] logLikelihoods) = GetLogLikelihoods(tree, tipStates, rateMatrix,
      maxParallelism);
00797
00798             return (rateMLE, logLikelihoods.Sum());
00799         }
00800
00801
00802     }
00803 }
```

8.9 ParsimonyScore.cs

```
00001 using PhyloTree.SequenceSimulation;
00002 using System;
00003 using System.Collections.Generic;
00004 using System.Data.SqlTypes;
00005 using System.Linq;
00006 using System.Reflection;
00007 using System.Text;
00008 using System.Threading.Tasks;
00009
00010 namespace PhyloTree.SequenceScores
00011 {
00012 /// <summary>
00013 /// Exception that is thrown when not enough data has been supplied.
00014 /// </summary>
00015     public class MissingDataException :  Exception
00016     {
00017 /// <inheritdoc/>
00018         public MissingDataException(string message) :  base(message) { }
00019     }
00020
00021 /// <summary>
00022 /// Contains methods to compute parsimony scores for a tree.
00023 /// </summary>
00024     public static class ParsimonyScore
00025     {
00026 /// <summary>
00027 /// Computes the parsimony score (minimum number of state changes) for a character on a tree.
00028 /// </summary>
00029 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00030 /// <param name="tipStates">The character states at the tips of the tree.
00031 /// This <see cref="Dictionary{String, Char}"/> should contain an entry for each terminal node of the
      tree,
00032 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00033 /// is the character state.</param>
00034 /// <returns>The parsimony score (minimum number of state changes) for the specified
      character.</returns>
00035 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
```

```
00036 /// contain an entry for one of the tips in the tree.</exception>
00037 /// <remarks>The character states can be anything; the only thing that is taken into account when
      computing
00038 /// parsimony score is whether two states are the same or not.  Note that this is case
      sensitive.</remarks>
00039         public static int GetParsimonyScore(this TreeNode tree, Dictionary<string, char> tipStates)
00040         {
00041             List<TreeNode> nodes = tree.GetChildrenRecursive();
00042 
00043             Dictionary<string, int> indices = new Dictionary<string, int>();
00044             int[][] children = new int[nodes.Count][];
00045 
00046             for (int i = nodes.Count - 1; i >= 0; i--)
00047             {
00048                 indices[nodes[i].Id] = i;
00049 
00050                 children[i] = new int[nodes[i].Children.Count];
00051 
00052                 for (int j = 0; j < nodes[i].Children.Count; j++)
00053                 {
00054                     children[i][j] = indices[nodes[i].Children[j].Id];
00055                 }
00056             }
00057 
00058             HashSet<char>[] states = new HashSet<char>[nodes.Count];
00059 
00060             int count = 0;
00061 
00062             for (int i = nodes.Count - 1; i >= 0; i--)
00063             {
00064                 if (nodes[i].Children.Count == 0)
00065                 {
00066                     char state;
00067 
00068                     if (tipStates.TryGetValue(nodes[i].Name, out state))
00069                     {
00070                         states[i] = new HashSet<char>() { state };
00071                     }
00072                     else if (tipStates.TryGetValue(nodes[i].Id, out state))
00073                     {
00074                         states[i] = new HashSet<char>() { state };
00075                     }
00076                     else
00077                     {
00078                         throw new Exception("Tip " + nodes[i].Name + " (id:  " + nodes[i].Id + ") has
      no associated state!");
00079                     }
00080                 }
00081                 else
00082                 {
00083                     HashSet<char> intersection = new HashSet<char>(states[children[i][0]]);
00084 
00085                     for (int j = 1; j < nodes[i].Children.Count; j++)
00086                     {
00087                         intersection.IntersectWith(states[children[i][j]]);
00088                     }
00089 
00090                     if (intersection.Count > 0)
00091                     {
00092                         states[i] = intersection;
00093                     }
00094                     else
00095                     {
00096                         HashSet<char> union = new HashSet<char>(states[children[i][0]]);
00097 
00098                         for (int j = 1; j < nodes[i].Children.Count; j++)
00099                         {
00100                             union.UnionWith(states[children[i][j]]);
00101                         }
00102 
00103                         states[i] = union;
00104 
00105                         count++;
00106                     }
00107                 }
00108             }
00109 
00110             return count;
00111         }
00112 
00113 /// <summary>
00114 /// Computes the parsimony score (minimum number of state changes) for a character sequence on a tree,
00115 /// returning the score for each character in the sequence.
00116 /// </summary>
00117 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00118 /// <param name="tipStates">The character state sequences at the tips of the tree.
00119 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
```

```
the tree,
00120 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00121 /// is the character state sequence.</param>
00122 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00123 /// <returns>An <see cref="T:int[]"/> array containing the parsimony score (minimum number of state
changes)
00124 /// for each character in the sequence.</returns>
00125 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
length.</exception>
00126 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
does not
00127 /// contain an entry for one of the tips in the tree.</exception>
00128 /// <remarks>The character states can be anything; the only thing that is taken into account when
computing
00129 /// parsimony score is whether two states are the same or not.  Note that this is case
sensitive.</remarks>
00130         public static int[] GetParsimonyScores(this TreeNode tree, Dictionary<string, string>
tipStates, int maxParallelism = -1)
00131         {
00132             List<TreeNode> nodes = tree.GetChildrenRecursive();
00133 
00134             Dictionary<string, int> indices = new Dictionary<string, int>();
00135             int[][] children = new int[nodes.Count][];
00136 
00137             for (int i = nodes.Count - 1; i >= 0; i--)
00138             {
00139                 indices[nodes[i].Id] = i;
00140 
00141                 children[i] = new int[nodes[i].Children.Count];
00142 
00143                 for (int j = 0; j < nodes[i].Children.Count; j++)
00144                 {
00145                     children[i][j] = indices[nodes[i].Children[j].Id];
00146                 }
00147             }
00148 
00149             int sequenceLength = 0;
00150 
00151             foreach (KeyValuePair<string, string> kvp in tipStates)
00152             {
00153                 if (sequenceLength == 0)
00154                 {
00155                     sequenceLength = kvp.Value.Length;
00156                 }
00157                 else
00158                 {
00159                     if (sequenceLength != kvp.Value.Length)
00160                     {
00161                         throw new ArgumentException("Not all the sequences have the same length!");
00162                     }
00163                 }
00164             }
00165 
00166             int[] siteScores = new int[sequenceLength];
00167 
00168 
00169             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
maxParallelism }, k =>
00170             {
00171                 HashSet<char>[] states = new HashSet<char>[nodes.Count];
00172 
00173                 for (int i = nodes.Count - 1; i >= 0; i--)
00174                 {
00175                     if (nodes[i].Children.Count == 0)
00176                     {
00177                         string seq;
00178 
00179                         if (tipStates.TryGetValue(nodes[i].Name, out seq))
00180                         {
00181                             states[i] = new HashSet<char>() { seq[k] };
00182                         }
00183                         else if (tipStates.TryGetValue(nodes[i].Id, out seq))
00184                         {
00185                             states[i] = new HashSet<char>() { seq[k] };
00186                         }
00187                         else
00188                         {
00189                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
nodes[i].Id + ") has no associated state!");
00190                         }
00191                     }
00192                     else
00193                     {
00194                         HashSet<char> intersection = new HashSet<char>(states[children[i][0]]);
00195 
00196                         for (int j = 1; j < nodes[i].Children.Count; j++)
00197                         {
```

```
00198                             intersection.IntersectWith(states[children[i][j]]);
00199                         }
00200 
00201                         if (intersection.Count > 0)
00202                         {
00203                             states[i] = intersection;
00204                         }
00205                         else
00206                         {
00207                             HashSet<char> union = new HashSet<char>(states[children[i][0]]);
00208 
00209                             for (int j = 1; j < nodes[i].Children.Count; j++)
00210                             {
00211                                 union.UnionWith(states[children[i][j]]);
00212                             }
00213 
00214                             states[i] = union;
00215 
00216                             siteScores[k]++;
00217                         }
00218                     }
00219                 }
00220             });
00221 
00222             return siteScores;
00223         }
00224 
00225 /// <summary>
00226 /// Computes the parsimony score (minimum number of state changes) for a character sequence on a tree,
00227 /// returning the total score for the sequence.
00228 /// </summary>
00229 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00230 /// <param name="tipStates">The character state sequences at the tips of the tree.
00231 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00232 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00233 /// is the character state sequence.</param>
00234 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00235 /// <returns>The total parsimony score (minimum number of state changes) for the sequence.</returns>
00236 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00237 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00238 /// contain an entry for one of the tips in the tree.</exception>
00239 /// <remarks>The character states can be anything; the only thing that is taken into account when
      computing
00240 /// parsimony score is whether two states are the same or not.  Note that this is case
      sensitive.</remarks>
00241         public static int GetParsimonyScore(this TreeNode tree, Dictionary<string, string> tipStates,
      int maxParallelism = -1)
00242         {
00243             return GetParsimonyScores(tree, tipStates, maxParallelism).Sum();
00244         }
00245 
00246 /// <summary>
00247 /// Computes the parsimony score (minimum number of state changes) for a character sequence on a tree,
00248 /// returning the score for each character in the sequence.
00249 /// </summary>
00250 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00251 /// <param name="tipStates">The character state sequences at the tips of the tree.
00252 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00253 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00254 /// is the character state sequence.</param>
00255 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00256 /// <returns>An <see cref="T:int[]"/> array containing the parsimony score (minimum number of state
      changes)
00257 /// for each character in the sequence.</returns>
00258 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00259 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00260 /// contain an entry for one of the tips in the tree.</exception>
00261 /// <remarks>The character states can be anything; the only thing that is taken into account when
      computing
00262 /// parsimony score is whether two states are the same or not.  Note that this is case
      sensitive.</remarks>
00263         public static int[] GetParsimonyScores(this TreeNode tree, Dictionary<string,
      IReadOnlyList<char» tipStates, int maxParallelism = -1)
00264         {
00265             List<TreeNode> nodes = tree.GetChildrenRecursive();
00266 
00267             Dictionary<string, int> indices = new Dictionary<string, int>();
00268             int[][] children = new int[nodes.Count][];
00269 
00270             for (int i = nodes.Count - 1; i >= 0; i--)
00271             {
```

```
00272                 indices[nodes[i].Id] = i;
00273
00274                 children[i] = new int[nodes[i].Children.Count];
00275
00276                 for (int j = 0; j < nodes[i].Children.Count; j++)
00277                 {
00278                     children[i][j] = indices[nodes[i].Children[j].Id];
00279                 }
00280             }
00281
00282             int sequenceLength = 0;
00283
00284             foreach (KeyValuePair<string, IReadOnlyList<char» kvp in tipStates)
00285             {
00286                 if (sequenceLength == 0)
00287                 {
00288                     sequenceLength = kvp.Value.Count;
00289                 }
00290                 else
00291                 {
00292                     if (sequenceLength != kvp.Value.Count)
00293                     {
00294                         throw new ArgumentException("Not all the sequences have the same length!");
00295                     }
00296                 }
00297             }
00298
00299             int[] siteScores = new int[sequenceLength];
00300
00301
00302             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
    maxParallelism }, k =>
00303             {
00304                 HashSet<char>[] states = new HashSet<char>[nodes.Count];
00305
00306                 for (int i = nodes.Count - 1; i >= 0; i--)
00307                 {
00308                     if (nodes[i].Children.Count == 0)
00309                     {
00310                         IReadOnlyList<char> seq;
00311
00312                         if (tipStates.TryGetValue(nodes[i].Name, out seq))
00313                         {
00314                             states[i] = new HashSet<char>() { seq[k] };
00315                         }
00316                         else if (tipStates.TryGetValue(nodes[i].Id, out seq))
00317                         {
00318                             states[i] = new HashSet<char>() { seq[k] };
00319                         }
00320                         else
00321                         {
00322                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
    nodes[i].Id + ") has no associated state!");
00323                         }
00324                     }
00325                     else
00326                     {
00327                         HashSet<char> intersection = new HashSet<char>(states[children[i][0]]);
00328
00329                         for (int j = 1; j < nodes[i].Children.Count; j++)
00330                         {
00331                             intersection.IntersectWith(states[children[i][j]]);
00332                         }
00333
00334                         if (intersection.Count > 0)
00335                         {
00336                             states[i] = intersection;
00337                         }
00338                         else
00339                         {
00340                             HashSet<char> union = new HashSet<char>(states[children[i][0]]);
00341
00342                             for (int j = 1; j < nodes[i].Children.Count; j++)
00343                             {
00344                                 union.UnionWith(states[children[i][j]]);
00345                             }
00346
00347                             states[i] = union;
00348
00349                             siteScores[k]++;
00350                         }
00351                     }
00352                 }
00353             });
00354
00355             return siteScores;
00356         }
```

```
00357
00358 /// <summary>
00359 /// Computes the parsimony score (minimum number of state changes) for a character sequence on a tree,
00360 /// returning the total score for the sequence.
00361 /// </summary>
00362 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00363 /// <param name="tipStates">The character state sequences at the tips of the tree.
00364 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00365 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00366 /// is the character state sequence.</param>
00367 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00368 /// <returns>The total parsimony score (minimum number of state changes) for the sequence.</returns>
00369 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00370 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00371 /// contain an entry for one of the tips in the tree.</exception>
00372 /// <remarks>The character states can be anything; the only thing that is taken into account when
      computing
00373 /// parsimony score is whether two states are the same or not.  Note that this is case
      sensitive.</remarks>
00374         public static int GetParsimonyScore(this TreeNode tree, Dictionary<string, IReadOnlyList<char»
      tipStates, int maxParallelism = -1)
00375         {
00376             return GetParsimonyScores(tree, tipStates, maxParallelism).Sum();
00377         }
00378
00379 /// <summary>
00380 /// Computes the parsimony score (minimum number of state changes) for a character sequence on a tree,
00381 /// returning the score for each character in the sequence.
00382 /// </summary>
00383 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00384 /// <param name="tipStates">The character state sequences at the tips of the tree.
00385 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00386 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00387 /// is the character state sequence.</param>
00388 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00389 /// <returns>An <see cref="T:int[]"/> array containing the parsimony score (minimum number of state
      changes)
00390 /// for each character in the sequence.</returns>
00391 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00392 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00393 /// contain an entry for one of the tips in the tree.</exception>
00394 /// <remarks>The character states can be anything; the only thing that is taken into account when
      computing
00395 /// parsimony score is whether two states are the same or not.  Note that this is case
      sensitive.</remarks>
00396         public static int[] GetParsimonyScores(this TreeNode tree, Dictionary<string, Sequence>
      tipStates, int maxParallelism = -1)
00397         {
00398             List<TreeNode> nodes = tree.GetChildrenRecursive();
00399
00400             Dictionary<string, int> indices = new Dictionary<string, int>();
00401             int[][] children = new int[nodes.Count][];
00402
00403             for (int i = nodes.Count - 1; i >= 0; i--)
00404             {
00405                 indices[nodes[i].Id] = i;
00406
00407                 children[i] = new int[nodes[i].Children.Count];
00408
00409                 for (int j = 0; j < nodes[i].Children.Count; j++)
00410                 {
00411                     children[i][j] = indices[nodes[i].Children[j].Id];
00412                 }
00413             }
00414
00415             int sequenceLength = 0;
00416
00417             foreach (KeyValuePair<string, Sequence> kvp in tipStates)
00418             {
00419                 if (sequenceLength == 0)
00420                 {
00421                     sequenceLength = kvp.Value.Length;
00422                 }
00423                 else
00424                 {
00425                     if (sequenceLength != kvp.Value.Length)
00426                     {
00427                         throw new ArgumentException("Not all the sequences have the same length!");
00428                     }
00429                 }
00430             }
```

```
00431
00432             int[] siteScores = new int[sequenceLength];
00433
00434
00435             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
      maxParallelism }, k =>
00436             {
00437                 HashSet<char>[] states = new HashSet<char>[nodes.Count];
00438
00439                 for (int i = nodes.Count - 1; i >= 0; i--)
00440                 {
00441                     if (nodes[i].Children.Count == 0)
00442                     {
00443                         Sequence seq;
00444
00445                         if (tipStates.TryGetValue(nodes[i].Name, out seq))
00446                         {
00447                             states[i] = new HashSet<char>() { seq[k] };
00448                         }
00449                         else if (tipStates.TryGetValue(nodes[i].Id, out seq))
00450                         {
00451                             states[i] = new HashSet<char>() { seq[k] };
00452                         }
00453                         else
00454                         {
00455                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00456                         }
00457                     }
00458                     else
00459                     {
00460                         HashSet<char> intersection = new HashSet<char>(states[children[i][0]]);
00461
00462                         for (int j = 1; j < nodes[i].Children.Count; j++)
00463                         {
00464                             intersection.IntersectWith(states[children[i][j]]);
00465                         }
00466
00467                         if (intersection.Count > 0)
00468                         {
00469                             states[i] = intersection;
00470                         }
00471                         else
00472                         {
00473                             HashSet<char> union = new HashSet<char>(states[children[i][0]]);
00474
00475                             for (int j = 1; j < nodes[i].Children.Count; j++)
00476                             {
00477                                 union.UnionWith(states[children[i][j]]);
00478                             }
00479
00480                             states[i] = union;
00481
00482                             siteScores[k]++;
00483                         }
00484                     }
00485                 }
00486             });
00487
00488             return siteScores;
00489         }
00490
00491 /// <summary>
00492 /// Computes the parsimony score (minimum number of state changes) for a character sequence on a tree,
00493 /// returning the total score for the sequence.
00494 /// </summary>
00495 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00496 /// <param name="tipStates">The character state sequences at the tips of the tree.
00497 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00498 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00499 /// is the character state sequence.</param>
00500 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00501 /// <returns>The total parsimony score (minimum number of state changes) for the sequence.</returns>
00502 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00503 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00504 /// contain an entry for one of the tips in the tree.</exception>
00505 /// <remarks>The character states can be anything; the only thing that is taken into account when
      computing
00506 /// parsimony score is whether two states are the same or not.  Note that this is case
      sensitive.</remarks>
00507         public static int GetParsimonyScore(this TreeNode tree, Dictionary<string, Sequence>
      tipStates, int maxParallelism = -1)
00508         {
00509             return GetParsimonyScores(tree, tipStates, maxParallelism).Sum();
```

```
00510         }
00511
00512 /// <summary>
00513 /// Computes the Sankoff parsimony score for a character on a tree.
00514 /// </summary>
00515 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00516 /// <param name="tipStates">The character states at the tips of the tree.
00517 /// This <see cref="Dictionary{String, Char}"/> should contain an entry for each terminal node of the
      tree,
00518 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00519 /// is the character state.</param>
00520 /// <param name="states">The character states.</param>
00521 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
      correspond to the
00522 /// states in the <paramref name="states"/> array.</param>
00523 /// <returns>The Sankoff parsimony score for the specified character.</returns>
00524 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00525 /// contain an entry for one of the tips in the tree.</exception>
00526 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
      the same
00527 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
00528         public static double GetSankoffParsimonyScore(this TreeNode tree, Dictionary<string, char>
      tipStates, char[] states, double[,] transitionCosts)
00529         {
00530             List<TreeNode> nodes = tree.GetChildrenRecursive();
00531
00532             Dictionary<string, int> indices = new Dictionary<string, int>();
00533             int[][] children = new int[nodes.Count][];
00534
00535             for (int i = nodes.Count - 1; i >= 0; i--)
00536             {
00537                 indices[nodes[i].Id] = i;
00538
00539                 children[i] = new int[nodes[i].Children.Count];
00540
00541                 for (int j = 0; j < nodes[i].Children.Count; j++)
00542                 {
00543                     children[i][j] = indices[nodes[i].Children[j].Id];
00544                 }
00545             }
00546
00547             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00548
00549             for (int i = 0; i < states.Length; i++)
00550             {
00551                 stateIndices[states[i]] = i;
00552             }
00553
00554             double[][] stateScores = new double[nodes.Count][];
00555
00556             for (int i = nodes.Count - 1; i >= 0; i--)
00557             {
00558                 if (nodes[i].Children.Count == 0)
00559                 {
00560                     char state;
00561
00562                     if (!tipStates.TryGetValue(nodes[i].Name, out state) &&
      !tipStates.TryGetValue(nodes[i].Id, out state))
00563                     {
00564                         throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00565                     }
00566
00567                     stateScores[i] = new double[states.Length];
00568
00569                     int index = stateIndices[state];
00570                     for (int j = 0; j < states.Length; j++)
00571                     {
00572                         if (j != index)
00573                         {
00574                             stateScores[i][j] = double.PositiveInfinity;
00575                         }
00576                         else
00577                         {
00578                             stateScores[i][j] = 0;
00579                         }
00580                     }
00581                 }
00582                 else
00583                 {
00584                     stateScores[i] = new double[states.Length];
00585
00586                     for (int j = 0; j < states.Length; j++)
00587                     {
00588                         stateScores[i][j] = 0;
00589
```

```
00590                         for (int l = 0; l < children[i].Length; l++)
00591                         {
00592                             double scoreL = double.PositiveInfinity;
00593 
00594                             for (int k = 0; k < states.Length; k++)
00595                             {
00596                                 scoreL = Math.Min(scoreL, transitionCosts[j, k] +
      stateScores[children[i][l]][k]);
00597                             }
00598 
00599                             stateScores[i][j] += scoreL;
00600                         }
00601                     }
00602                 }
00603             }
00604 
00605             return stateScores[0].Min();
00606         }
00607 
00608 /// <summary>
00609 /// Computes the Sankoff parsimony score for a character sequence on a tree,
00610 /// returning the score for each character in the sequence.
00611 /// </summary>
00612 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00613 /// <param name="tipStates">The character state sequences at the tips of the tree.
00614 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00615 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00616 /// is the character state sequence.</param>
00617 /// <param name="states">The character states.</param>
00618 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
      correspond to the
00619 /// states in the <paramref name="states"/> array.</param>
00620 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00621 /// <returns>An <see cref="T:double[]"/> array containing the Sankoff parsimony score
00622 /// for each character in the sequence.</returns>
00623 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00624 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00625 /// contain an entry for one of the tips in the tree.</exception>
00626 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
      the same
00627 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
00628         public static double[] GetSankoffParsimonyScores(this TreeNode tree, Dictionary<string,
      string> tipStates, char[] states, double[,] transitionCosts, int maxParallelism = -1)
00629         {
00630             List<TreeNode> nodes = tree.GetChildrenRecursive();
00631 
00632             Dictionary<string, int> indices = new Dictionary<string, int>();
00633             int[][] children = new int[nodes.Count][];
00634 
00635             for (int i = nodes.Count - 1; i >= 0; i--)
00636             {
00637                 indices[nodes[i].Id] = i;
00638 
00639                 children[i] = new int[nodes[i].Children.Count];
00640 
00641                 for (int j = 0; j < nodes[i].Children.Count; j++)
00642                 {
00643                     children[i][j] = indices[nodes[i].Children[j].Id];
00644                 }
00645             }
00646 
00647             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00648 
00649             for (int i = 0; i < states.Length; i++)
00650             {
00651                 stateIndices[states[i]] = i;
00652             }
00653 
00654             int sequenceLength = 0;
00655 
00656             foreach (KeyValuePair<string, string> kvp in tipStates)
00657             {
00658                 if (sequenceLength == 0)
00659                 {
00660                     sequenceLength = kvp.Value.Length;
00661                 }
00662                 else
00663                 {
00664                     if (sequenceLength != kvp.Value.Length)
00665                     {
00666                         throw new ArgumentException("Not all the sequences have the same length!");
00667                     }
00668                 }
00669             }
```

```
00670
00671             double[] siteScores = new double[sequenceLength];
00672
00673             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
      maxParallelism }, siteIndex =>
00674             {
00675                 double[][] stateScores = new double[nodes.Count][];
00676
00677                 for (int i = nodes.Count - 1; i >= 0; i--)
00678                 {
00679                     if (nodes[i].Children.Count == 0)
00680                     {
00681                         string seq;
00682
00683                         if (!tipStates.TryGetValue(nodes[i].Name, out seq) &&
      !tipStates.TryGetValue(nodes[i].Id, out seq))
00684                         {
00685                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00686                         }
00687
00688                         char state = seq[siteIndex];
00689
00690                         stateScores[i] = new double[states.Length];
00691
00692                         int index = stateIndices[state];
00693                         for (int j = 0; j < states.Length; j++)
00694                         {
00695                             if (j != index)
00696                             {
00697                                 stateScores[i][j] = double.PositiveInfinity;
00698                             }
00699                             else
00700                             {
00701                                 stateScores[i][j] = 0;
00702                             }
00703                         }
00704                     }
00705                     else
00706                     {
00707                         stateScores[i] = new double[states.Length];
00708
00709                         for (int j = 0; j < states.Length; j++)
00710                         {
00711                             stateScores[i][j] = 0;
00712
00713                             for (int l = 0; l < children[i].Length; l++)
00714                             {
00715                                 double scoreL = double.PositiveInfinity;
00716
00717                                 for (int k = 0; k < states.Length; k++)
00718                                 {
00719                                     scoreL = Math.Min(scoreL, transitionCosts[j, k] +
      stateScores[children[i][l]][k]);
00720                                 }
00721
00722                                 stateScores[i][j] += scoreL;
00723                             }
00724                         }
00725                     }
00726                 }
00727
00728                 siteScores[siteIndex] = stateScores[0].Min();
00729             });
00730
00731             return siteScores;
00732         }
00733
00734 /// <summary>
00735 /// Computes the Sankoff parsimony score for a character sequence on a tree,
00736 /// returning the total score for the sequence.
00737 /// </summary>
00738 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00739 /// <param name="tipStates">The character state sequences at the tips of the tree.
00740 /// This <see cref="Dictionary{String, String}"/> should contain an entry for each terminal node of
      the tree,
00741 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00742 /// is the character state sequence.</param>
00743 /// <param name="states">The character states.</param>
00744 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
      correspond to the
00745 /// states in the <paramref name="states"/> array.</param>
00746 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00747 /// <returns>The total Sankoff parsimony score for the sequence.</returns>
00748 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00749 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
```

```
does not
00750 /// contain an entry for one of the tips in the tree.</exception>
00751 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
the same
00752 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
00753         public static double GetSankoffParsimonyScore(this TreeNode tree, Dictionary<string, string>
tipStates, char[] states, double[,] transitionCosts, int maxParallelism = -1)
00754         {
00755             return tree.GetSankoffParsimonyScores(tipStates, states, transitionCosts,
maxParallelism).Sum();
00756         }
00757
00758 /// <summary>
00759 /// Computes the Sankoff parsimony score for a character sequence on a tree,
00760 /// returning the score for each character in the sequence.
00761 /// </summary>
00762 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00763 /// <param name="tipStates">The character state sequences at the tips of the tree.
00764 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
tree,
00765 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00766 /// is the character state sequence.</param>
00767 /// <param name="states">The character states.</param>
00768 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
correspond to the
00769 /// states in the <paramref name="states"/> array.</param>
00770 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00771 /// <returns>An <see cref="T:double[]"/> array containing the Sankoff parsimony score
00772 /// for each character in the sequence.</returns>
00773 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
length.</exception>
00774 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
does not
00775 /// contain an entry for one of the tips in the tree.</exception>
00776 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
the same
00777 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
00778         public static double[] GetSankoffParsimonyScores(this TreeNode tree, Dictionary<string,
IReadOnlyList<char» tipStates, char[] states, double[,] transitionCosts, int maxParallelism = -1)
00779         {
00780             List<TreeNode> nodes = tree.GetChildrenRecursive();
00781
00782             Dictionary<string, int> indices = new Dictionary<string, int>();
00783             int[][] children = new int[nodes.Count][];
00784
00785             for (int i = nodes.Count - 1; i >= 0; i--)
00786             {
00787                 indices[nodes[i].Id] = i;
00788
00789                 children[i] = new int[nodes[i].Children.Count];
00790
00791                 for (int j = 0; j < nodes[i].Children.Count; j++)
00792                 {
00793                     children[i][j] = indices[nodes[i].Children[j].Id];
00794                 }
00795             }
00796
00797             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00798
00799             for (int i = 0; i < states.Length; i++)
00800             {
00801                 stateIndices[states[i]] = i;
00802             }
00803
00804             int sequenceLength = 0;
00805
00806             foreach (KeyValuePair<string, IReadOnlyList<char» kvp in tipStates)
00807             {
00808                 if (sequenceLength == 0)
00809                 {
00810                     sequenceLength = kvp.Value.Count;
00811                 }
00812                 else
00813                 {
00814                     if (sequenceLength != kvp.Value.Count)
00815                     {
00816                         throw new ArgumentException("Not all the sequences have the same length!");
00817                     }
00818                 }
00819             }
00820
00821             double[] siteScores = new double[sequenceLength];
00822
00823             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
maxParallelism }, siteIndex =>
00824             {
00825                 double[][] stateScores = new double[nodes.Count][];
```

```
00826
00827                 for (int i = nodes.Count - 1; i >= 0; i--)
00828                 {
00829                     if (nodes[i].Children.Count == 0)
00830                     {
00831                         IReadOnlyList<char> seq;
00832
00833                         if (!tipStates.TryGetValue(nodes[i].Name, out seq) &&
      !tipStates.TryGetValue(nodes[i].Id, out seq))
00834                         {
00835                             throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00836                         }
00837
00838                         char state = seq[siteIndex];
00839
00840                         stateScores[i] = new double[states.Length];
00841
00842                         int index = stateIndices[state];
00843                         for (int j = 0; j < states.Length; j++)
00844                         {
00845                             if (j != index)
00846                             {
00847                                 stateScores[i][j] = double.PositiveInfinity;
00848                             }
00849                             else
00850                             {
00851                                 stateScores[i][j] = 0;
00852                             }
00853                         }
00854                     }
00855                     else
00856                     {
00857                         stateScores[i] = new double[states.Length];
00858
00859                         for (int j = 0; j < states.Length; j++)
00860                         {
00861                             stateScores[i][j] = 0;
00862
00863                             for (int l = 0; l < children[i].Length; l++)
00864                             {
00865                                 double scoreL = double.PositiveInfinity;
00866
00867                                 for (int k = 0; k < states.Length; k++)
00868                                 {
00869                                     scoreL = Math.Min(scoreL, transitionCosts[j, k] +
      stateScores[children[i][l]][k]);
00870                                 }
00871
00872                                 stateScores[i][j] += scoreL;
00873                             }
00874                         }
00875                     }
00876                 }
00877
00878                 siteScores[siteIndex] = stateScores[0].Min();
00879             });
00880
00881             return siteScores;
00882         }
00883
00884 /// <summary>
00885 /// Computes the Sankoff parsimony score for a character sequence on a tree,
00886 /// returning the total score for the sequence.
00887 /// </summary>
00888 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00889 /// <param name="tipStates">The character state sequences at the tips of the tree.
00890 /// This <see cref="Dictionary{TKey, TValue}"/> should contain an entry for each terminal node of the
      tree,
00891 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00892 /// is the character state sequence.</param>
00893 /// <param name="states">The character states.</param>
00894 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
      correspond to the
00895 /// states in the <paramref name="states"/> array.</param>
00896 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00897 /// <returns>The total Sankoff parsimony score for the sequence.</returns>
00898 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00899 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00900 /// contain an entry for one of the tips in the tree.</exception>
00901 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
      the same
00902 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
00903         public static double GetSankoffParsimonyScore(this TreeNode tree, Dictionary<string,
      IReadOnlyList<char» tipStates, char[] states, double[,] transitionCosts, int maxParallelism = -1)
```

```
00904         {
00905             return tree.GetSankoffParsimonyScores(tipStates, states, transitionCosts,
      maxParallelism).Sum();
00906         }
00907 
00908 
00909 /// <summary>
00910 /// Computes the Sankoff parsimony score for a character sequence on a tree,
00911 /// returning the score for each character in the sequence.
00912 /// </summary>
00913 /// <param name="tree">The tree for which the parsimony score is computed.</param>
00914 /// <param name="tipStates">The character state sequences at the tips of the tree.
00915 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
00916 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
00917 /// is the character state sequence.</param>
00918 /// <param name="states">The character states.</param>
00919 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
      correspond to the
00920 /// states in the <paramref name="states"/> array.</param>
00921 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
00922 /// <returns>An <see cref="T:double[]"/> array containing the Sankoff parsimony score
00923 /// for each character in the sequence.</returns>
00924 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
00925 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
00926 /// contain an entry for one of the tips in the tree.</exception>
00927 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
      the same
00928 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
00929         public static double[] GetSankoffParsimonyScores(this TreeNode tree, Dictionary<string,
      Sequence> tipStates, char[] states, double[,] transitionCosts, int maxParallelism = -1)
00930         {
00931             List<TreeNode> nodes = tree.GetChildrenRecursive();
00932 
00933             Dictionary<string, int> indices = new Dictionary<string, int>();
00934             int[][] children = new int[nodes.Count][];
00935 
00936             for (int i = nodes.Count - 1; i >= 0; i--)
00937             {
00938                 indices[nodes[i].Id] = i;
00939 
00940                 children[i] = new int[nodes[i].Children.Count];
00941 
00942                 for (int j = 0; j < nodes[i].Children.Count; j++)
00943                 {
00944                     children[i][j] = indices[nodes[i].Children[j].Id];
00945                 }
00946             }
00947 
00948             Dictionary<char, int> stateIndices = new Dictionary<char, int>();
00949 
00950             for (int i = 0; i < states.Length; i++)
00951             {
00952                 stateIndices[states[i]] = i;
00953             }
00954 
00955             int sequenceLength = 0;
00956 
00957             foreach (KeyValuePair<string, Sequence> kvp in tipStates)
00958             {
00959                 if (sequenceLength == 0)
00960                 {
00961                     sequenceLength = kvp.Value.Length;
00962                 }
00963                 else
00964                 {
00965                     if (sequenceLength != kvp.Value.Length)
00966                     {
00967                         throw new ArgumentException("Not all the sequences have the same length!");
00968                     }
00969                 }
00970             }
00971 
00972             double[] siteScores = new double[sequenceLength];
00973 
00974             Parallel.For(0, sequenceLength, new ParallelOptions() { MaxDegreeOfParallelism =
      maxParallelism }, siteIndex =>
00975             {
00976                 double[][] stateScores = new double[nodes.Count][];
00977 
00978                 for (int i = nodes.Count - 1; i >= 0; i--)
00979                 {
00980                     if (nodes[i].Children.Count == 0)
00981                     {
00982                         Sequence seq;
```

```
00983
00984                           if (!tipStates.TryGetValue(nodes[i].Name, out seq) &&
      !tipStates.TryGetValue(nodes[i].Id, out seq))
00985                           {
00986                               throw new MissingDataException("Tip " + nodes[i].Name + " (id:  " +
      nodes[i].Id + ") has no associated state!");
00987                           }
00988
00989                           char state = seq[siteIndex];
00990
00991                           stateScores[i] = new double[states.Length];
00992
00993                           int index = stateIndices[state];
00994                           for (int j = 0; j < states.Length; j++)
00995                           {
00996                               if (j != index)
00997                               {
00998                                   stateScores[i][j] = double.PositiveInfinity;
00999                               }
01000                               else
01001                               {
01002                                   stateScores[i][j] = 0;
01003                               }
01004                           }
01005                       }
01006                       else
01007                       {
01008                           stateScores[i] = new double[states.Length];
01009
01010                           for (int j = 0; j < states.Length; j++)
01011                           {
01012                               stateScores[i][j] = 0;
01013
01014                               for (int l = 0; l < children[i].Length; l++)
01015                               {
01016                                   double scoreL = double.PositiveInfinity;
01017
01018                                   for (int k = 0; k < states.Length; k++)
01019                                   {
01020                                       scoreL = Math.Min(scoreL, transitionCosts[j, k] +
      stateScores[children[i][l]][k]);
01021                                   }
01022
01023                                   stateScores[i][j] += scoreL;
01024                               }
01025                           }
01026                       }
01027                   }
01028
01029                   siteScores[siteIndex] = stateScores[0].Min();
01030               });
01031
01032               return siteScores;
01033           }
01034
01035 /// <summary>
01036 /// Computes the Sankoff parsimony score for a character sequence on a tree,
01037 /// returning the total score for the sequence.
01038 /// </summary>
01039 /// <param name="tree">The tree for which the parsimony score is computed.</param>
01040 /// <param name="tipStates">The character state sequences at the tips of the tree.
01041 /// This <see cref="Dictionary{String, Sequence}"/> should contain an entry for each terminal node of
      the tree,
01042 /// where the key is either the <see cref="TreeNode.Name"/> or <see cref="TreeNode.Id"/> and the value
01043 /// is the character state sequence.</param>
01044 /// <param name="states">The character states.</param>
01045 /// <param name="transitionCosts">The transition cost matrix.  Indices in this matrix should
      correspond to the
01046 /// states in the <paramref name="states"/> array.</param>
01047 /// <param name="maxParallelism">The maximum number of concurrent computations to run.</param>
01048 /// <returns>The total Sankoff parsimony score for the sequence.</returns>
01049 /// <exception cref="ArgumentException">Thrown when the sequences do not all have the same
      length.</exception>
01050 /// <exception cref="MissingDataException">Thrown when the <paramref name="tipStates"/> dictionary
      does not
01051 /// contain an entry for one of the tips in the tree.</exception>
01052 /// <remarks>Note that diagonal entries in the cost matrix may not be 0, in which case even retaining
      the same
01053 /// state across a branch will incur a cost.  This may or may not be useful.</remarks>
01054           public static double GetSankoffParsimonyScore(this TreeNode tree, Dictionary<string, Sequence>
      tipStates, char[] states, double[,] transitionCosts, int maxParallelism = -1)
01055           {
01056               return tree.GetSankoffParsimonyScores(tipStates, states, transitionCosts,
      maxParallelism).Sum();
01057           }
01058       }
01059 }
```

8.10 IndelModel.cs

```
00001 using MathNet.Numerics.Distributions;
00002
00003 namespace PhyloTree.SequenceSimulation
00004 {
00005 /// <summary>
00006 /// Represents a model for sequence insertion/deletion.
00007 /// </summary>
00008     public class IndelModel
00009     {
00010 /// <summary>
00011 /// The rate of insertions, expressed as a multiple of the rate of sequence mutation.
00012 /// </summary>
00013         public double InsertionRate { get; }
00014
00015 /// <summary>
00016 /// The rate of deletions, expressed as a multiple of the rate of sequence mutation.
00017 /// </summary>
00018         public double DeletionRate { get; }
00019
00020 /// <summary>
00021 /// The size distribution for insertions.
00022 /// </summary>
00023         public IDiscreteDistribution InsertionSizeDistribution { get; }
00024
00025 /// <summary>
00026 /// The size distribution for deletions.
00027 /// </summary>
00028         public IDiscreteDistribution DeletionSizeDistribution { get; }
00029
00030 /// <summary>
00031 /// Creates a new <see cref="IndelModel"/> with the specified insertion rate, deletion rate, insertion
      size distribution and deletion size distribution.
00032 /// </summary>
00033 /// <param name="insertionRate">The insertion rate, expressed as a multiple of the rate of sequence
      mutation.</param>
00034 /// <param name="deletionRate">The deletion rate, expressed as a multiple of the rate of sequence
      mutation.</param>
00035 /// <param name="insertionSizeDistribution">The size distribution for insertions.</param>
00036 /// <param name="deletionSizeDistribution">The size distribution for deletions.</param>
00037         public IndelModel(double insertionRate, double deletionRate, IDiscreteDistribution
      insertionSizeDistribution, IDiscreteDistribution deletionSizeDistribution)
00038         {
00039             InsertionRate = insertionRate;
00040             DeletionRate = deletionRate;
00041             InsertionSizeDistribution = insertionSizeDistribution;
00042             DeletionSizeDistribution = deletionSizeDistribution;
00043         }
00044
00045 /// <summary>
00046 /// Creates a new <see cref="IndelModel"/> with the specified rate and size distribution for
      insertions and deletions.
00047 /// </summary>
00048 /// <param name="indelRate">The insertion/deletion rate, expressed as a multiple of the rate of
      sequence mutation.</param>
00049 /// <param name="indelSizeDistribution">The size distribution for insertions and deletions.</param>
00050         public IndelModel(double indelRate, IDiscreteDistribution indelSizeDistribution) :
      this(indelRate, indelRate, indelSizeDistribution, indelSizeDistribution) { }
00051
00052 /// <summary>
00053 /// Creates a new <see cref="IndelModel"/> with the specified insertion rate, deletion rate, and size
      distribution for insertions and deletions.
00054 /// </summary>
00055 /// <param name="insertionRate">The insertion rate, expressed as a multiple of the rate of sequence
      mutation.</param>
00056 /// <param name="deletionRate">The deletion rate, expressed as a multiple of the rate of sequence
      mutation.</param>
00057 /// <param name="indelSizeDistribution">The size distribution for insertions and deletions.</param>
00058         public IndelModel(double insertionRate, double deletionRate, IDiscreteDistribution
      indelSizeDistribution) :  this(insertionRate, deletionRate, indelSizeDistribution,
      indelSizeDistribution) { }
00059
00060 /// <summary>
00061 /// Creates a new <see cref="IndelModel"/> with the specified rate for insertions and deletions,
      insertion size distribution and deletion size distribution.
00062 /// </summary>
00063 /// <param name="indelRate">The insertion/deletion rate, expressed as a multiple of the rate of
      sequence mutation.</param>
00064 /// <param name="insertionSizeDistribution">The size distribution for insertions.</param>
00065 /// <param name="deletionSizeDistribution">The size distribution for deletions.</param>
00066         public IndelModel(double indelRate, IDiscreteDistribution insertionSizeDistribution,
      IDiscreteDistribution deletionSizeDistribution) :  this(indelRate, indelRate,
      insertionSizeDistribution, deletionSizeDistribution) { }
00067     }
00068
00069 /// <summary>
```

```
00070 /// Represents an insertion event.
00071 /// </summary>
00072     public struct Insertion
00073     {
00074 /// <summary>
00075 /// The position in the ancestral sequence at which the insertion occurred.
00076 /// </summary>
00077         public int Start { get; }
00078 
00079 /// <summary>
00080 /// The length of the insertion.
00081 /// </summary>
00082         public int Length { get; }
00083 
00084 /// <summary>
00085 /// The end of the insertion in the new sequence.
00086 /// </summary>
00087         public int End => Start + Length;
00088 
00089 /// <summary>
00090 /// Creates a new <see cref="Insertion"/>
00091 /// </summary>
00092 /// <param name="start">The position in the ancestral sequence at which the insertion
      occurred.</param>
00093 /// <param name="length">The length of the insertion.</param>
00094         public Insertion(int start, int length)
00095         {
00096             this.Start = start;
00097             this.Length = length;
00098         }
00099     }
00100 }
```

8.11 RateMatix.cs

```
00001 using MathNet.Numerics.LinearAlgebra;
00002 using PhyloTree.TreeBuilding;
00003 using System;
00004 using System.Collections.Immutable;
00005 using System.Linq;
00006 
00007 namespace PhyloTree.SequenceSimulation
00008 {
00009 /// <summary>
00010 /// Represents a rate matrix for a continuous-type Markov chain model.  This type cannot be
      instantiated directly,
00011 /// please use <see cref="MutableRateMatrix"/> or <see cref="ImmutableRateMatrix"/> instead, or access
      the
00012 /// static members for some pre-baked common rate matrices for DNA and protein evolution.
00013 /// </summary>
00014     public abstract partial class RateMatrix
00015     {
00016         internal abstract Matrix<double> GetMatrix();
00017         internal abstract double[] GetEquilibriumFrequencies();
00018         internal abstract char[] GetStates();
00019         internal abstract MatrixExponential GetExponential();
00020         internal RateMatrix() { }
00021 
00022 /// <summary>
00023 /// Gets the states for the character to which the rate matrix applies.
00024 /// </summary>
00025         public abstract ImmutableArray<char> States { get; }
00026 
00027 /// <summary>
00028 /// Gets the equilibrium frequences of the rate matrix.
00029 /// </summary>
00030         public abstract ImmutableArray<double> EquilibriumFrequencies { get; }
00031 
00032 /// <summary>
00033 /// Gets the rate of going from state number <paramref name="from"/> to state number
00034 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00035 /// sum of the elements on the row is returned.
00036 /// </summary>
00037 /// <param name="from">The row number.</param>
00038 /// <param name="to">The column number.</param>
00039 /// <returns>The rate of going from state number <paramref name="from"/> to state number <paramref
      name="to"/>.
00040 /// </returns>
00041         public abstract double this[int from, int to] { get; }
00042 
00043 /// <summary>
00044 /// Gets the rate of going from state <paramref name="from"/> to state
00045 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
```

```
00046 /// sum of the elements on the row is returned.
00047 /// </summary>
00048 /// <param name="from">The row state.</param>
00049 /// <param name="to">The column state.</param>
00050 /// <returns>The rate of going from state <paramref name="from"/> to state <paramref name="to"/>.
00051 /// </returns>
00052 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state is not part of the rate
      matrix.</exception>
00053         public abstract double this[char from, char to] { get; }
00054     }
00055
00056 /// <summary>
00057 /// Represents a rate matrix whose values can be changed after initialisation.
00058 /// </summary>
00059     [System.ComponentModel.EditorBrowsable(System.ComponentModel.EditorBrowsableState.Never)]
00060     [Obsolete("Please do not use this class.  Use PhyloTree.SequenceSimulation.RateMatrix or
      PhyloTree.SequenceSimulation.MutableRateMatrix instead.")]
00061     public abstract class IMutableRateMatrix :  RateMatrix
00062     {
00063 /// <summary>
00064 /// Gets the rate of going from state number <paramref name="from"/> to state number
00065 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00066 /// sum of the elements on the row is returned.
00067 /// </summary>
00068 /// <param name="from">The row number.</param>
00069 /// <param name="to">The column number.</param>
00070 /// <returns>The rate of going from state number <paramref name="from"/> to state number <paramref
      name="to"/>.
00071 /// </returns>
00072         public sealed override double this[int from, int to] => GetThis(from, to);
00073
00074 /// <summary>
00075 /// Gets the rate of going from state number <paramref name="from"/> to state number
00076 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00077 /// sum of the elements on the row is returned.
00078 /// </summary>
00079 /// <param name="from">The row number.</param>
00080 /// <param name="to">The column number.</param>
00081 /// <returns>The rate of going from state number <paramref name="from"/> to state number <paramref
      name="to"/>.
00082 /// </returns>
00083         protected internal abstract double GetThis(int from, int to);
00084
00085 /// <summary>
00086 /// Gets the rate of going from state <paramref name="from"/> to state
00087 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00088 /// sum of the elements on the row is returned.
00089 /// </summary>
00090 /// <param name="from">The row state.</param>
00091 /// <param name="to">The column state.</param>
00092 /// <returns>The rate of going from state <paramref name="from"/> to state <paramref name="to"/>.
00093 /// </returns>
00094 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state is not part of the rate
      matrix.</exception>
00095         public sealed override double this[char from, char to] => GetThis(from, to);
00096
00097 /// <summary>
00098 /// Gets the rate of going from state <paramref name="from"/> to state
00099 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00100 /// sum of the elements on the row is returned.
00101 /// </summary>
00102 /// <param name="from">The row state.</param>
00103 /// <param name="to">The column state.</param>
00104 /// <returns>The rate of going from state <paramref name="from"/> to state <paramref name="to"/>.
00105 /// </returns>
00106 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state is not part of the rate
      matrix.</exception>
00107         protected internal abstract double GetThis(char from, char to);
00108     }
00109
00110 /// <summary>
00111 /// Represents a rate matrix whose values can be changed after initialisation.
00112 /// </summary>
00113 #pragma warning disable 0618
00114     public class MutableRateMatrix :  IMutableRateMatrix
00115     {
00116 #pragma warning restore 0618
00117 /// <summary>
00118 /// Gets the states for the character to which the rate matrix applies.
00119 /// </summary>
00120         public override ImmutableArray<char> States { get; }
00121
00122         private readonly double[,] rates;
00123         private double[] equilibriumFrequencies;
00124         private MatrixExponential exponential;
00125
00126 /// <summary>
```

```
00127 /// Gets or sets the rate of going from state number <paramref name="from"/> to state number
00128 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00129 /// sum of the elements on the row is returned, but this value cannot be set.
00130 /// </summary>
00131 /// <param name="from">The row number.</param>
00132 /// <param name="to">The column number.</param>
00133 /// <returns>The rate of going from state number <paramref name="from"/> to state number <paramref
      name="to"/>.
00134 /// </returns>
00135 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state index is &lt; 0 or greater than
      the
00136 /// number of states in the rate matrix.</exception>
00137 /// <exception cref="ArgumentException">Thrown when attempting to set the value of a diagonal
      entry.</exception>
00138         public new double this[int from, int to]
00139         {
00140             get
00141             {
00142                 if (from >= 0 && from < States.Length)
00143                 {
00144                     if (to >= 0 && to < States.Length)
00145                     {
00146                         if (from != to)
00147                         {
00148                             return rates[from, to];
00149                         }
00150                         else
00151                         {
00152                             double tbr = 0;
00153 
00154                             for (int i = 0; i < States.Length; i++)
00155                             {
00156                                 if (i != from)
00157                                 {
00158                                     tbr += rates[from, i];
00159                                 }
00160                             }
00161 
00162                             return -tbr;
00163                         }
00164                     }
00165                     else
00166                     {
00167                         throw new ArgumentOutOfRangeException(nameof(to));
00168                     }
00169                 }
00170                 else
00171                 {
00172                     throw new ArgumentOutOfRangeException(nameof(from));
00173                 }
00174             }
00175 
00176             set
00177             {
00178                 if (from >= 0 && from < States.Length)
00179                 {
00180                     if (to >= 0 && to < States.Length)
00181                     {
00182                         if (from != to)
00183                         {
00184                             rates[from, to] = value;
00185                             this.equilibriumFrequencies = null;
00186                             this.exponential = null;
00187                         }
00188                         else
00189                         {
00190                             throw new ArgumentException("The value of the diagonal entries of a rate
      matrix cannot be set!");
00191                         }
00192                     }
00193                     else
00194                     {
00195                         throw new ArgumentOutOfRangeException(nameof(to));
00196                     }
00197                 }
00198                 else
00199                 {
00200                     throw new ArgumentOutOfRangeException(nameof(from));
00201                 }
00202             }
00203         }
00204 
00205 /// <summary>
00206 /// Gets or sets the rate of going from state <paramref name="from"/> to state
00207 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00208 /// sum of the elements on the row is returned, but this value cannot be set.
00209 /// </summary>
```

```
00210 /// <param name="from">The row state.</param>
00211 /// <param name="to">The column state.</param>
00212 /// <returns>The rate of going from state <paramref name="from"/> to state <paramref name="to"/>.
00213 /// </returns>
00214 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state is not part of the rate
      matrix.</exception>
00215 /// <exception cref="ArgumentException">Thrown when attempting to set the value of a diagonal
      entry.</exception>
00216         public new double this[char from, char to]
00217         {
00218             get
00219             {
00220                 int fromInt = States.IndexOf(from);
00221                 int toInt = States.IndexOf(to);
00222 
00223                 if (fromInt < 0)
00224                 {
00225                     throw new ArgumentException("Character " + from + " is not found in the rate
      matrix!", nameof(from));
00226                 }
00227 
00228                 if (toInt < 0)
00229                 {
00230                     throw new ArgumentException("Character " + to + " is not found in the rate
      matrix!", nameof(to));
00231                 }
00232 
00233                 return this[fromInt, toInt];
00234             }
00235 
00236             set
00237             {
00238                 int fromInt = States.IndexOf(from);
00239                 int toInt = States.IndexOf(to);
00240 
00241                 if (fromInt < 0)
00242                 {
00243                     throw new ArgumentException("Character " + from + " is not found in the rate
      matrix!", nameof(from));
00244                 }
00245 
00246                 if (toInt < 0)
00247                 {
00248                     throw new ArgumentException("Character " + to + " is not found in the rate
      matrix!", nameof(to));
00249                 }
00250 
00251                 this[fromInt, toInt] = value;
00252             }
00253         }
00254 
00255 /// <inheritdoc/>
00256         protected internal sealed override double GetThis(int from, int to) => this[from, to];
00257 
00258 /// <inheritdoc/>
00259         protected internal sealed override double GetThis(char from, char to) => this[from, to];
00260 
00261 /// <summary>
00262 /// Gets the equilibrium frequences of the rate matrix.
00263 /// </summary>
00264         public override ImmutableArray<double> EquilibriumFrequencies
00265         {
00266             get
00267             {
00268                 if (this.equilibriumFrequencies == null)
00269                 {
00270                     Matrix<double> rateMatrix = Matrix<double>.Build.DenseOfArray(rates);
00271 
00272                     for (int i = 0; i < States.Length; i++)
00273                     {
00274                         rateMatrix[i, i] = this[i, i];
00275                     }
00276 
00277                     Vector<double>[] kernel = rateMatrix.Transpose().Kernel();
00278 
00279                     if (kernel == null || kernel.Length == 0)
00280                     {
00281                         throw new ArgumentException("The kernel of the transpose of the rate matrix is
      empty!");
00282                     }
00283 
00284                     Vector<double> equilibriumFrequencies = kernel[0] / kernel[0].Sum();
00285 
00286                     this.equilibriumFrequencies = equilibriumFrequencies.ToArray();
00287                 }
00288 
00289                 return ImmutableArray.Create<double>(equilibriumFrequencies);
```

```
00290             }
00291         }
00292 
00293 /// <summary>
00294 /// Creates a new <see cref="MutableRateMatrix"/> with the specified <paramref name="states"/>.
00295 /// </summary>
00296 /// <param name="states">The possible states of the character described by the <see
      cref="MutableRateMatrix"/>.</param>
00297         public MutableRateMatrix(ReadOnlySpan<char> states)
00298         {
00299             this.States = ImmutableArray.Create<char>(states);
00300             this.equilibriumFrequencies = null;
00301             this.rates = new double[states.Length, states.Length];
00302             this.exponential = null;
00303         }
00304 
00305 /// <summary>
00306 /// Creates a new <see cref="MutableRateMatrix"/> with the specified <paramref name="states"/> and
      <paramref name="rates"/>.
00307 /// </summary>
00308 /// <param name="states">The possible states of the character described by the <see
      cref="MutableRateMatrix"/>.</param>
00309 /// <param name="rates">A 2D <see langword="double"/> array containing the rates used to initialise
      the matrix.
00310 /// The number of rows and columns in the array must be equal to the number of states.  Diagonal
      entries are ignored.</param>
00311 /// <exception cref="ArgumentException">Thrown if the number of rows or columns of the <paramref
      name="rates"/> matrix does
00312 /// not correspond to the number of <paramref name="states"/>.</exception>
00313         public MutableRateMatrix(ReadOnlySpan<char> states, double[,] rates)
00314         {
00315             if (states.Length != rates.GetLength(0) || states.Length != rates.GetLength(1))
00316             {
00317                 throw new ArgumentException("The size of the rate matrix does not correspond to the
      number of states!");
00318             }
00319 
00320             this.States = ImmutableArray.Create<char>(states);
00321             this.equilibriumFrequencies = null;
00322             this.rates = new double[states.Length, states.Length];
00323             this.exponential = null;
00324 
00325             for (int i = 0; i < states.Length; i++)
00326             {
00327                 for (int j = 0; j < states.Length; j++)
00328                 {
00329                     if (i != j)
00330                     {
00331                         this.rates[i, j] = rates[i, j];
00332                     }
00333                 }
00334             }
00335         }
00336 
00337         internal override Matrix<double> GetMatrix()
00338         {
00339             Matrix<double> tbr = Matrix<double>.Build.DenseOfArray(this.rates);
00340 
00341             for (int i= 0; i < this.States.Length; i++)
00342             {
00343                 tbr[i, i] = this[i, i];
00344             }
00345 
00346             return tbr;
00347         }
00348         internal override double[] GetEquilibriumFrequencies()
00349         {
00350             return this.EquilibriumFrequencies.ToArray();
00351         }
00352         internal override char[] GetStates()
00353         {
00354             return this.States.ToArray();
00355         }
00356 
00357         internal override MatrixExponential GetExponential()
00358         {
00359             if (this.exponential == null)
00360             {
00361                 this.exponential = this.GetMatrix().FastExponential(1);
00362             }
00363 
00364             return this.exponential;
00365         }
00366     }
00367 
00368 /// <summary>
00369 /// Represents a rate matrix whose values cannot be changed after initialisation.
```

```
00370 /// </summary>
00371     public class ImmutableRateMatrix :  RateMatrix
00372     {
00373 /// <summary>
00374 /// Gets the states for the character to which the rate matrix applies.
00375 /// </summary>
00376         public override ImmutableArray<char> States { get; }
00377
00378 /// <summary>
00379 /// Gets the equilibrium frequences of the rate matrix.
00380 /// </summary>
00381         public override ImmutableArray<double> EquilibriumFrequencies { get; }
00382
00383         private readonly Matrix<double> rates;
00384         private readonly MatrixExponential exponential;
00385
00386 /// <summary>
00387 /// Gets the rate of going from state number <paramref name="from"/> to state number
00388 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00389 /// sum of the elements on the row is returned.
00390 /// </summary>
00391 /// <param name="from">The row number.</param>
00392 /// <param name="to">The column number.</param>
00393 /// <returns>The rate of going from state number <paramref name="from"/> to state number <paramref
      name="to"/>.
00394 /// </returns>
00395 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state index is &lt; 0 or greater than
      the
00396 /// number of states in the rate matrix.</exception>
00397         public override double this[int from, int to]
00398         {
00399             get
00400             {
00401                 if (from >= 0 && from < States.Length)
00402                 {
00403                     if (to >= 0 && to < States.Length)
00404                     {
00405                         return rates[from, to];
00406                     }
00407                     else
00408                     {
00409                         throw new ArgumentOutOfRangeException(nameof(to));
00410                     }
00411                 }
00412                 else
00413                 {
00414                     throw new ArgumentOutOfRangeException(nameof(from));
00415                 }
00416             }
00417         }
00418
00419 /// <summary>
00420 /// Gets the rate of going from state <paramref name="from"/> to state
00421 /// <paramref name="to"/>.  If <c><paramref name="from"/> == <paramref name="to"/></c>, the negative
00422 /// sum of the elements on the row is returned.
00423 /// </summary>
00424 /// <param name="from">The row state.</param>
00425 /// <param name="to">The column state.</param>
00426 /// <returns>The rate of going from state <paramref name="from"/> to state <paramref name="to"/>.
00427 /// </returns>
00428 /// <exception cref="ArgumentOutOfRangeException">Thrown if the state is not part of the rate
      matrix.</exception>
00429         public override double this[char from, char to]
00430         {
00431             get
00432             {
00433                 int fromInt = States.IndexOf(from);
00434                 int toInt = States.IndexOf(to);
00435
00436                 if (fromInt < 0)
00437                 {
00438                     throw new ArgumentException("Character " + from + " is not found in the rate
      matrix!", nameof(from));
00439                 }
00440
00441                 if (toInt < 0)
00442                 {
00443                     throw new ArgumentException("Character " + to + " is not found in the rate
      matrix!", nameof(to));
00444                 }
00445
00446                 return this[fromInt, toInt];
00447             }
00448         }
00449
00450 /// <summary>
00451 /// Creates a new <see cref="ImmutableRateMatrix"/> with the specified <paramref name="states"/> and
```

```
    <paramref name="rates"/>.
00452 /// </summary>
00453 /// <param name="states">The possible states of the character described by the <see
    cref="ImmutableRateMatrix"/>.</param>
00454 /// <param name="rates">A 2D <see langword="double"/> array containing the rates used to initialise
    the matrix.
00455 /// The number of rows and columns in the array must be equal to the number of states.  Diagonal
    entries are ignored.</param>
00456 /// <exception cref="ArgumentException">Thrown if the number of rows or columns of the <paramref
    name="rates"/> matrix does
00457 /// not correspond to the number of <paramref name="states"/>.</exception>
00458         public ImmutableRateMatrix(ReadOnlySpan<char> states, double[,] rates)
00459         {
00460             if (states.Length != rates.GetLength(0) || states.Length != rates.GetLength(1))
00461             {
00462                 throw new ArgumentException("The size of the rate matrix does not correspond to the
    number of states!");
00463             }
00464
00465             this.States = ImmutableArray.Create<char>(states);
00466
00467
00468             this.rates = Matrix<double>.Build.Dense(states.Length, states.Length);
00469
00470             for (int i = 0; i < states.Length; i++)
00471             {
00472                 double diag = 0;
00473
00474                 for (int j = 0; j < states.Length; j++)
00475                 {
00476                     if (i != j)
00477                     {
00478                         this.rates[i, j] = rates[i, j];
00479                         diag += rates[i, j];
00480                     }
00481                 }
00482
00483                 this.rates[i, i] = -diag;
00484             }
00485
00486             Vector<double>[] kernel = this.rates.Transpose().Kernel();
00487
00488             if (kernel == null || kernel.Length == 0)
00489             {
00490                 throw new ArgumentException("The kernel of the transpose of the rate matrix is
    empty!");
00491             }
00492
00493             Vector<double> equilibriumFrequencies = kernel[0] / kernel[0].Sum();
00494
00495             this.EquilibriumFrequencies =
    ImmutableArray.Create<double>(equilibriumFrequencies.ToArray());
00496             this.exponential = this.rates.FastExponential(1);
00497         }
00498
00499 /// <summary>
00500 /// Creates a new <see cref="ImmutableRateMatrix"/> with the specified <paramref name="states"/> and
    <paramref name="rates"/>.
00501 /// </summary>
00502 /// <param name="states">The possible states of the character described by the <see
    cref="ImmutableRateMatrix"/>.</param>
00503 /// <param name="rates">A 2D <see langword="double"/> array containing the rates used to initialise
    the matrix.
00504 /// The number of rows and columns in the array must be equal to the number of states.  Diagonal
    entries are ignored.</param>
00505 /// <param name="equilibriumFrequencies">Equilibrium frequencies for the rate matrix.  These are not
    checked, so they better
00506 /// be correct!</param>
00507 /// <exception cref="ArgumentException">Thrown if the number of rows or columns of the <paramref
    name="rates"/> matrix, or the
00508 /// number of equilibrium frequencies, does not correspond to the number of <paramref
    name="states"/>.</exception>
00509 /// <remarks>Using this constructor is faster than the <see
    cref="ImmutableRateMatrix(ReadOnlySpan{char}, double[,])"/> constructor,
00510 /// as equilbrium frequencies are not computed.  This is especially useful for pre-baked rate
    matrices.</remarks>
00511         public ImmutableRateMatrix(ReadOnlySpan<char> states, double[,] rates, double[]
    equilibriumFrequencies)
00512         {
00513             if (states.Length != rates.GetLength(0) || states.Length != rates.GetLength(1))
00514             {
00515                 throw new ArgumentException("The size of the rate matrix does not correspond to the
    number of states!");
00516             }
00517
00518             if (states.Length != equilibriumFrequencies.Length)
00519             {
```

```
00520                 throw new ArgumentException("The number of equilibrium frequencies does not correspond
      to the number of states!");
00521             }
00522
00523             this.States = ImmutableArray.Create<char>(states);
00524
00525
00526             this.rates = Matrix<double>.Build.Dense(states.Length, states.Length);
00527
00528             for (int i = 0; i < states.Length; i++)
00529             {
00530                 double diag = 0;
00531
00532                 for (int j = 0; j < states.Length; j++)
00533                 {
00534                     if (i != j)
00535                     {
00536                         this.rates[i, j] = rates[i, j];
00537                         diag += rates[i, j];
00538                     }
00539                 }
00540
00541                 this.rates[i, i] = -diag;
00542             }
00543
00544             this.EquilibriumFrequencies = ImmutableArray.Create<double>(equilibriumFrequencies);
00545             this.exponential = this.rates.FastExponential(1);
00546         }
00547
00548         internal override Matrix<double> GetMatrix()
00549         {
00550             return this.rates.Clone();
00551         }
00552         internal override double[] GetEquilibriumFrequencies()
00553         {
00554             return this.EquilibriumFrequencies.ToArray();
00555         }
00556         internal override char[] GetStates()
00557         {
00558             return this.States.ToArray();
00559         }
00560         internal override MatrixExponential GetExponential()
00561         {
00562             return this.exponential;
00563         }
00564     }
00565 }
```

8.12 RateMatrices.cs

```
00001 using System;
00002 using System.Buffers.Text;
00003 using System.IO;
00004 using System.Text;
00005
00006 namespace PhyloTree.SequenceSimulation
00007 {
00008     partial class RateMatrix
00009     {
00010 /// <summary>
00011 /// Contains rate matrices for DNA sequence evolution.
00012 /// </summary>
00013         public static class DNA
00014         {
00015 /// <summary>
00016 /// DNA sequence evolution matrix from Jukes &amp;amp; Cantor, 1969.
00017 /// </summary>
00018             public static readonly ImmutableRateMatrix JC69Matrix = new ImmutableRateMatrix(new
      char[4] { 'A', 'C', 'G', 'T' },
00019                 new double[4, 4] { { -0.999999999, 0.333333333, 0.333333333, 0.333333333 }, {
      0.333333333, -0.999999999, 0.333333333, 0.333333333 }, { 0.333333333, 0.333333333, -0.999999999,
      0.333333333 }, { 0.333333333, 0.333333333, 0.333333333, -0.999999999 } },
00020                 new double[4] { 0.25, 0.25, 0.25, 0.25 });
00021
00022 /// <summary>
00023 /// Kimura 2-parameter DNA sequence evolution matrix, from Kimura 1980.
00024 /// </summary>
00025 /// <param name="kappa">The transition/transversion rate ratio.</param>
00026 /// <returns>The Kimura 2-parameter DNA sequence evolution matrix with the specified
      transition/transversion rate ratio.</returns>
00027 /// <remarks>See also <see href="https://doi.org/10.1007/BF01731581"/>.</remarks>
00028             public static ImmutableRateMatrix K80Matrix(double kappa)
00029             {
```

```
00030                 double v = 1.0 / (2 + kappa);
00031                 double t = kappa * v;
00032 
00033                 return new ImmutableRateMatrix(new char[4] { 'A', 'C', 'G', 'T' },
00034                 new double[4, 4] { { -1, v, t, v }, { v, -1, v, t }, { t, v, -1, v }, { v, t, v, -1 } },
00035                 new double[4] { 0.25, 0.25, 0.25, 0.25 });
00036             }
00037 
00038 /// <summary>
00039 /// Hasegawa-Kishino-Yano DNA sequence evolution matrix, from Hasegawa, Kishino &amp;amp; Yano, 1985.
00040 /// </summary>
00041 /// <param name="kappa">The transition/transversion rate ratio.</param>
00042 /// <param name="piA">The equilibrium frequency for A.</param>
00043 /// <param name="piC">The equilibrium frequency for C.</param>
00044 /// <param name="piG">The equilibrium frequency for G.</param>
00045 /// <param name="piT">The equilibrium frequency for C.</param>
00046 /// <returns>The HKY85 DNA sequence evolution matrix with the specified parameters.</returns>
00047 /// <remarks>See also <see href="https://doi.org/10.1007/BF02101694"/>.</remarks>
00048             public static ImmutableRateMatrix HKY85Matrix(double kappa, double piA, double piC, double piG, double piT)
00049             {
00050                 double total = piA + piC + piG + piT;
00051 
00052                 piA /= total;
00053                 piC /= total;
00054                 piG /= total;
00055                 piT /= total;
00056 
00057                 double v = 4.0 / (2 + kappa);
00058                 double t = kappa * v;
00059 
00060                 return new ImmutableRateMatrix(new char[4] { 'A', 'C', 'G', 'T' },
00061                 new double[4, 4] { { -1, v * piC, t * piG, v * piT }, { v * piA, -1, v * piG, t * piT }, { t * piA, v * piC, -1, v * piT }, { v * piA, t * piC, v * piG, -1 } },
00062                 new double[4] { piA, piC, piG, piT });
00063             }
00064 
00065 /// <summary>
00066 /// General Time Reversible DNA Evolution matrix, from Tavaré, 1986.
00067 /// </summary>
00068 /// <param name="AC">The weight for A-C and C-A transversions.</param>
00069 /// <param name="AG">The weight for A-G and G-A transitions.</param>
00070 /// <param name="AT">The weight for A-T and T-A transversions.</param>
00071 /// <param name="CG">The weight for C-G and G-C transversions.</param>
00072 /// <param name="CT">The weight for C-T and T-C transitions.</param>
00073 /// <param name="GT">The weight for G-T and T-G transversions.</param>
00074 /// <param name="piA">The equilibrium frequency for A.</param>
00075 /// <param name="piC">The equilibrium frequency for C.</param>
00076 /// <param name="piG">The equilibrium frequency for G.</param>
00077 /// <param name="piT">The equilibrium frequency for T.</param>
00078 /// <returns>The GTR DNA sequence evolution matrix with the specified parameters.</returns>
00079             public static ImmutableRateMatrix GTRMatrix(double AC, double AG, double AT, double CG, double CT, double GT, double piA, double piC, double piG, double piT)
00080             {
00081                 double total = piA + piC + piG + piT;
00082 
00083                 piA /= total;
00084                 piC /= total;
00085                 piG /= total;
00086                 piT /= total;
00087 
00088 
00089                 double totalRate = AC * (piA + piC) + AG * (piA + piG) + AT * (piA + piT) + CG * (piG + piC) + GT * (piG + piT) + CT * (piC + piT);
00090 
00091                 AC *= 4.0 / totalRate;
00092                 AG *= 4.0 / totalRate;
00093                 AT *= 4.0 / totalRate;
00094                 CG *= 4.0 / totalRate;
00095                 GT *= 4.0 / totalRate;
00096                 CT *= 4.0 / totalRate;
00097 
00098                 return new ImmutableRateMatrix(new char[4] { 'A', 'C', 'G', 'T' },
00099                 new double[4, 4] { { -1, AC * piC, AG * piG, AT * piT }, { AC * piA, -1, CG * piG, CT * piT }, { AG * piA, CG * piC, -1, GT * piT }, { AT * piA, CT * piC, GT * piG, -1 } },
00100                 new double[4] { piA, piC, piG, piT });
00101             }
00102 
00103         }
00104 
00105 /// <summary>
00106 /// Contains rate matrices for protein sequence evolution.
00107 /// </summary>
00108         public static class Protein
00109         {
00110 /// <summary>
```

```
00111 /// cpREV10 protein sequence evolution matrix.
00112 /// </summary>
00113 /// <remarks>Citation:  Adachi, J., P. J. Waddell, W. Martin, and M. Hasegawa.  2000.  Plastid genome
      phylogeny and a model of amino acid substitution for proteins encoded by chloroplast DNA.Journal of
      Molecular Evolution 50:348-358.</remarks>
00114             public static readonly ImmutableRateMatrix cpREV10Matrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00115     new double[20, 20] {
00116         { -0.9437513720455278, 0.012847710436231344, 0.018338109198681868, 0.012792540450461161,
      0.011995334156082019, 0.011817016880646248, 0.04866879405415725, 0.109802064179106 51,
      0.0031990710320167486, 0.023027557989147417, 0.039243001985321 49, 0.02343620995517 37,
      0.008019351503023013, 0.006779603108501619, 0.041611961767160456, 0.2990370857274685,
      0.14336708945035886, 0.0004992883712201553, 0.0033874371262892305, 0.1258821446744802,  },
00117         { 0.015620002221183033, -0.9290073903841098, 0.028840110061363115, 0.0031433099392561706,
      0.014756591943879672, 0.13134200290909362, 0.014824963319102005, 0.040123160294019376,
      0.03465660284684811, 0.02159826128637275, 0.04043822031989981, 0.44508937719952757,
      0.005418480745285821, 0.005284102422802733, 0.007388246272944815, 0.04718413032994893,
      0.03359497469209902, 0.008202594670045409, 0.019538253424846 81, 0.011964005485591093,  },
00118         { 0.03376895718293856, 0.04368221548318657, -1.5278289181105693, 0.324199525130258 57,
      0.009646472011916481, 0.057805534804689924, 0.10289694935297775, 0.1078206735473031,
      0.06810143636338685, 0.026680205118460455, 0.022509945301225017, 0.24131351775878002,
      0.0026442186036994804, 0.009670904434186132, 0.014691570174936242, 0.25552964087777535,
      0.14903757880921634, 0.001426538203486158, 0.045609421307537 14, 0.010793613644609356,  },
00119         { 0.026033337035305058, 0.005261443321504264, 0.35827979866147175, -1.0583849477284581,
      0.00017930245375309442, 0.030107049377442668, 0.35999302375529935, 0.07116494685893972,
      0.016043825933329612, 0.001588107447527408, 0.001992030557630532 6, 0.040914061447167645,
      0.0020373487602274687, 0.0021934010056917004, 0.014436803062076077, 0.0723081477783633,
      0.02845943715954885, 0.0006419421915687711, 0.016997675580129885, 0.009753265341514479,  },
00120         { 0.09952172843782332, 0.10070157799065138, 0.04346212664709623, 0.000731002311454923 5,
      -1.0776282387650977, 0.0007526762344360668, 0.000975326534151447 8, 0.050030113453036505,
      0.021375610986657363, 0.044467008530767425, 0.07888441008216908, 0.004766686770543803,
      0.006892307508003564, 0.0723822331878261, 0.024202875721715776, 0.2856784618158726,
      0.06162645039060202, 0.015513602962911969, 0.08867826477035735, 0.07698577442902095,  },
00121         { 0.02335562236881654, 0.21351671153546378, 0.062042589711839 97, 0.029240092458196938,
      0.00017930245375309442, -1.2609486437675692, 0.304496943962082 05, 0.021960412835821307,
      0.06150941120650385, 0.014610588517252153, 0.056972073948233 23, 0.32900069314190 87,
      0.008756264884381887, 0.000997000457132591, 0.027429925817944546, 0.04853224833937604,
      0.025784678028012303, 0.0018901631196191592, 0.023651569935340875, 0.007022351045890424,  },
00122         { 0.07423220103209842, 0.018598590345782515, 0.085227776231759 34, 0.269812953158012 3,
      0.00017930245375309442, 0.23498552039094, -1.2353834634686571, 0.06257892078779154,
      0.007852265260404747, 0.023503990223405637, 0.016334650572570 37, 0.2610754066616595,
      0.004898306593738381, 0.01445650662842257, 0.015710638626376904, 0.06961191175950907,
      0.03947944478147942, 0.0022467976704906 99, 0.008589572713090549, 0.026008707577371 94,  },
00123         { 0.09892668073415921, 0.029733272723849683, 0.0527523581794681, 0.031506199623707 2,
      0.005432864348718762, 0.010010593917999687, 0.036964875644339 87, -0.4183750849112877,
      0.0009209446910351246, 0.006352429790109632, 0.003984061115261065, 0.026117471263604586,
      0.0009103047652080178, 0.0024925011428314774, 0.002377826386694 8827, 0.084686322228557 68,
      0.009843113604054491, 0.002924403317146624, 0.000604899486837 3625, 0.011833961947704233,  },
00124         { 0.009818287110457907, 0.08748679011338487, 0.113502393938978 08, 0.024196176509157 97,
      0.007907238210511466, 0.09551461414993687, 0.015800289853253453, 0.003137201833688758,
      -0.5969035825782917, 0.0046055115978294835, 0.013147401680361517, 0.030288322187830414,
      0.0004334784596228656, 0.012661905805583906, 0.012908200384915079, 0.037134523350583176,
      0.003423691688366779, 0.0024607784010136228, 0.119225688855644 16, 0.003251088447171 4926,  },
00125         { 0.02157047925782419, 0.016640843993594886, 0.013571816499464994, 0.000731002311454923 5,
      0.005020468705086645, 0.006924621356811813, 0.014434832705441427, 0.006604635439344753,
      0.0014056524231588746, -1.3692288044820726, 0.347609332306527 96, 0.034260561163283 58,
      0.07681238304517178, 0.045263820753819635, 0.009935917401546476, 0.026472135457841477,
      0.11126997987192032, 0.0014976865113660466, 0.005383605432852527, 0.623818851243266 1,  },
00126         { 0.029306099405457694, 0.02483890684338 06, 0.009128662288330 62, 0.000731002311454923 5,
      0.007100377168622 54, 0.021526540304871 51, 0.007997677580041872, 0.003302317719672376 3,
      0.0031990710320167486, 0.2771247495935327, -0.8190049409352597, 0.021648702416219 77,
      0.05856293989504915, 0.126419657964412 54, 0.018597999238792122, 0.063238990260399 09,
      0.016690496980788047, 0.005670489358857478, 0.011432600301226153, 0.112487660272133 65,  },
00127         { 0.03510781451618281, 0.5484136969065608, 0.196306631510118 68, 0.030117295231942846,
      0.0008606517780148533, 0.24936163646866888, 0.256413345828415 63, 0.043425478013691 75,
      0.014783585829774 37, 0.054789706939695 57, 0.043426266156345615, -1.7664716174510826,
      0.008366134270721307, 0.007178403291354655, 0.025646556027923 38, 0.106378766562066 67,
      0.09821715531002197, 0.0003566345508715395, 0.014941017324882856, 0.032380840933828064,  },
00128         { 0.027520956294465344, 0.015294893376465885, 0.004927861943258123, 0.003435710863838 14,
      0.0028509090146742015, 0.015204059935608546, 0.011021189835911 36, 0.003467433605655 9956,
      0.0004847077321237498, 0.2814126397018567, 0.269123328335884 9, 0.019166053056561 54,
      -0.8532534829230717, 0.032601914948235 72, 0.008492237095338867, 0.011397724988792858,
      0.06900878559364289, 0.003067057137495 24, 0.013005338967003295, 0.061770680496258 36,  },
00129         { 0.010115810962289964, 0.006485034791621 5354, 0.007836108336000623, 0.001608205085200 8318,
      0.013017358142474657, 0.000752676234436 0668, 0.014142234745195994, 0.004127897149590471,
      0.006155788197971622, 0.072100078117744 33, 0.252589474707551 5, 0.007150030155815704,
      0.014174745629667708, -0.690702016099449, 0.003651661950995 7132, 0.059684860962818 51,
      0.015834574058696354, 0.016690496980788047, 0.143361178380454 92, 0.041223801510134 53,  },
00130         { 0.07289334369885415, 0.010645245790020257, 0.013975739609568119, 0.012427039294733 7,
      0.005110119931963191, 0.024311442372284955, 0.018043540881801785, 0.004623244807541327 5,
      0.007367557528280 9965, 0.018580857136070675, 0.043625469212108664, 0.02999040426467143,
      0.004334784596228656, 0.004287101965670142, -0.46882626357714885, 0.147312531575580 8,
      0.027817494967980 08, 0.001747509299270 5434, 0.005867525022322417, 0.015865311622196884,  },
00131         { 0.3629790992351105, 0.047108271599514924, 0.168435936913003 04, 0.043219363758404 8,
      0.04179540196984631, 0.02980597888366824, 0.055398547139802 23, 0.114095077214680 61,
      0.014686644283349619, 0.03430312086659201, 0.102788776773735 49, 0.086197585767333 77,
```

```
      0.00403134967449265, 0.048553922262357184, 0.1020766898859732, -1.541418270784083,
      0.23013627567740444, 0.0026034322213622386, 0.031575753212910326, 0.021717270827105572,  },
00132         { 0.19934098072747872, 0.038420772161682305, 0.11253297847473058, 0.019444661484700965,
      0.010327821336178239, 0.018139497249909205, 0.035989549110188424, 0.015190661510492932,
      0.0015510647427959994, 0.16516317454285043, 0.031075676699036308, 0.09116288448665023,
      0.027959360645674836, 0.014755606765562345, 0.022079816447881057, 0.26361834893433805,
      -1.1709149706682367, 0.0010342401975274645, 0.004294786356545274, 0.09883308879401338,  },
00133         { 0.0020826669628244044, 0.02814260381269723, 0.0032313848808249988, 0.0013158041606188625,
      0.007799656738259608, 0.003989184042511154, 0.0061445571651541206, 0.013539502650656744,
      0.0033444833516538736, 0.006670051279615114, 0.031673285866632547, 0.0009930597438632923,
      0.0037279147527566445, 0.04665962139380526, 0.004161196176716045, 0.008946601335289017,
      0.0031027205925823935, -0.19775425252959558, 0.020929522244572746, 0.0013004435378868597,  },
00134         { 0.008330667851297617, 0.03952200448478785, 0.06091160500355123, 0.02054116495188335,
      0.026285739720203646, 0.02942964076645021, 0.013849636784950558, 0.0016511588598361882,
      0.09553589400159108, 0.014134156282993932, 0.03764937753921707, 0.02452857567342332,
      0.00931978688189161, 0.23628910834042408, 0.008237469982478702, 0.06397432735645024,
      0.0075963159335637916, 0.012339555460155268, -0.725601366883686, 0.015475181008536305,  },
00135         { 0.144001544286716, 0.011257041525078892, 0.006705123627711872, 0.005482517335911926,
      0.01061470526218319, 0.004064451665956476, 0.019506530683028957, 0.015025545624509314,
      0.0012117693303093744, 0.7618151425788976, 0.172310643235041 06, 0.024727187622195976,
      0.02059022683208612, 0.031604914491103135, 0.010360529256313419, 0.020466882506757066,
      0.081312677598711, 0.0003566345508715395, 0.007198303893364615, -1.3486123719067455,  },
00136     }, new double[20] {
00137 0.0755, 0.0621, 0.041, 0.0371, 0.0091, 0.0382, 0.0495, 0.0838, 0.0246, 0.0806, 0.1011, 0.0504, 0.022,
      0.0506, 0.0431, 0.0622, 0.0543, 0.0181, 0.0307, 0.066,  });
00138
00139
00140 /// <summary>
00141 /// cpREV64 protein sequence evolution matrix.
00142 /// </summary>
00143 /// <remarks>Citation:  Zhong B, Yonezawa T, Zhong Y and Hasegawa M. 2010.  The position of Gnetales
      among seed plants:  Overcoming pitfalls of chloroplast phygenomics.  MBE Advance Access published July
      2,2010.</remarks>
00144             public static readonly ImmutableRateMatrix cpREV64Matrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00145                 new double[20, 20] {
00146         { -0.9240449511486136, 0.004224894241913516, 0.0020699970330603783, 0.034710712310833336,
      0.0023550698326207296, 0.002308940541097105, 0.040431342068098494, 0.0931133780673936,
      0.00073664466322927 68, 0.0013731506301627398, 0.004458309590169038, 0.0020970530508021548,
      0.003349596574850681, 0.002580571926424513, 0.042206412685540914, 0.1986320236568338,
      0.26028069740380083, 0.00065510671186349 18, 0.000847714597924398, 0.227613335562008 02,  },
00147         { 0.004239348065205315, -1.131555515858714, 0.00487599301120889, 0.003911646108575524,
      0.042717343579 13849, 0.18407176805500367, 0.011668105429408559, 0.14499195788817684,
      0.09118608581259549, 0.04632094792415643, 0.042182467660830135, 0.355550351780051,
      0.013373389310038168, 0.0030006650307261777, 0.03193248328182372, 0.0655778365637257,
      0.04593188777714132, 0.033102156793572916, 0.0016568967141249592, 0.005264185073570671,  },
00148         { 0.0029349332759113713, 0.006889827532966657, -1.1128393338439788, 0.2648390291616608,
      0.0013164236500215142, 0.006813267170450474, 0.007055133515238663, 0.013718651806711675,
      0.09084407221895334, 0.023435104088110763, 0.002743575132411716, 0.3191515023982708,
      0.002474701947091175, 0.00522115715346355, 0.0016197636447301885, 0.21469028761634684,
      0.11997965837544187, 0.0011753385124609707, 0.026934204724961552, 0.0010027019187753658,  },
00149         { 0.05498108336873969, 0.006174845430488986, 0.29587157592543006, -0.917698663275372,
      0.0012922690876357981, 0.004390772176512528, 0.23737810765887624, 0.11850076819245775,
      0.05324888565628773, 0.005126429019274229, 0.006744622200512135, 0.005042913288833753,
      0.0009498855958531784, 0.004500997546089267, 0.006201380811252722, 0.027528452502022367,
      0.004454001238995522, 0.0006743745563300651, 0.078182405417663 79, 0.006454893602116418,  },
00150         { 0.012718044195615943, 0.2298992451330478, 0.005013992813412915, 0.004405748774921906,
      -1.2489795800538401, 3.7851484280280415E-05, 5.427025780952818E-05, 0.06843557338060767,
      0.029413169053226128, 0.00311247476170221, 0.02320607299498243, 4.992983454290845E-05,
      0.002624683883278519, 0.14283165546256607, 0.0030081324830703504, 0.3732953084972516,
      0.004119951146070859, 0.09992304140364908, 0.22314160166273583, 0.023688832831068016,  },
00151         { 0.003978465107346525, 0.3160870876680836, 0.008279988132241513, 0.004776325774681693,
      1.2077281192857928E-05, -1.0188696394432908, 0.18592990325544353, 0.005597840679501665,
      0.11841562961410627, 0.0032955615123905 76, 0.029836379564977415, 0.249898821887256 76,
      0.0023747139896329456, 0.0014403192147485655, 0.067382167620775 84, 0.008464208096886187,
      0.0030064508363219777, 0.0032755335593174593, 0.0036220532820406093, 0.0031961123660964783,  },
00152         { 0.04858945090119937, 0.013974650184790863, 0.0059799914288410925, 0.18010042188832562,
      1.2077281192857928E-05, 0.1296791851442407, -0.7793777082361288, 0.12764653606359888,
      0.0016311417542933 99, 0.003936365139799855, 0.005830097156374896, 0.213000674160047 45,
      0.002399710978997503, 0.004621024147318314, 0.003748595863518436, 0.012973178765320887,
      0.006458301796543507, 0.0017533738464581693, 0.0035064558368691, 0.013536475903467439,  },
00153         { 0.07702569330780733, 0.11953200785967627, 0.008003988527833461, 0.06188635895988435,
      0.010483080075400681, 0.0026874553838999093, 0.08786354739362612, -0.5624017163101381,
      0.0005787922353944319, 0.0022885843836045667, 0.003886731437583265, 0.014629441521072175,
      0.0009498855958531784, 0.0018604123190502303, 0.00120325299322814, 0.108136191293863 73,
      0.0035075259757089737, 0.009441243788620912, 0.0008091821162005616, 0.047628341141829 88,  },
00154         { 0.0018261807050115198, 0.2252843606534192, 0.15883777233683302, 0.083338649723756 43,
      0.013502400373615163, 0.17036953074554215, 0.0033647559841907472, 0.0017345421824577982,
      -1.121509083433005, 0.007689643528911344, 0.01977660407946779, 0.004593544777947577,
      0.0008998916171240637, 0.006601463067597591, 0.061967529151249215, 0.015188111725253 72,
      0.002115650588522873, 0.0010982671345946774, 0.3426308274883521, 0.000689357569158064,  },
00155         { 0.000978311091970457, 0.032889176713972915, 0.011775983121410151, 0.002305812442949782 3,
      0.0004106275605571696, 0.001362653434090095, 0.0023336210858097115, 0.001971070661883861 6,
      0.0022099339896878307, -1.0353164407747015, 0.23434702956016742, 0.018923407291762298,
      0.13370889611101713, 0.03660811337485937, 0.0009718581868381133, 0.028635918981988 79,
      0.148206891227576, 0.0006358388673969185, 0.004161508026174317, 0.3728797760445892,  },
```

```
00156         { 0.002543608839123188, 0.023984399619478267, 0.0011039984176322015, 0.0024293381095363783,
      0.0024516880821501595, 0.009879237397153188, 0.002767783148285937, 0.0026806561001620514,
      0.004551411669238032, 0.18766391945557445, -0.9318349012177747, 0.005392422130634112,
      0.03569570081258786, 0.3254521292325612, 0.07210262167113239, 0.12680491195615476,
      0.00595722665715651, 0.01903663033297441, 0.003699118245488281, 0.09763809934075125,  },
00157         { 0.00273927105751728, 0.4628534137948639, 0.29403157856270973, 0.004158697441748715,
      1.2077281192857928E-05, 0.18944667882280347, 0.23151691981544725, 0.023100948157278856,
      0.002420403893467624, 0.03469493925544523, 0.012346088095852723, -1.4045870212393752,
      0.02089748310876992, 0.005641250257765215, 0.009811139790937142, 0.01890603490799812,
      0.07805637171339652, 0.0004431604227311857, 0.007745028826491 09, 0.005765536032958353,  },
00158         { 0.008739579088269419, 0.03477412952959587, 0.004553993472732832, 0.0015646584434302095,
      0.0012681145252500825, 0.0035958910066266393, 0.005209944749714705, 0.002996027406063469,
      0.0009471145670090703, 0.48966551471603303, 0.16324272037849713, 0.041741341677871 46,
      -1.0584063625463715, 0.002280505423351895, 0.004859290934190566, 0.0049404494412798475,
      0.16435264571893476, 0.002138730735789635, 0.0013101043786104 33, 0.12026156138312045,  },
00159         { 0.0028044917969819775, 0.003249918647625782, 0.0040019942639167305, 0.0030881416646648878,
      0.028743929239001872, 0.0009084356227267297, 0.00417880985133367, 0.002444127620735988,
      0.0028939611769721587, 0.05584145895995142, 0.6199336642945307, 0.004693404447033395,
      0.0009498855958531784, -1.1080774565472404, 0.005831149121028678, 0.1972872429311603,
      0.002004300557547985, 0.006570334963101492, 0.09860462073129701, 0.06404758506177649,  },
00160         { 0.05948131439180379, 0.044848877337235 79, 0.0016099976923802939, 0.005517479774201266,
      0.0007850232775357653, 0.05511176111208828, 0.0043958908825717825, 0.002049913488359216,
      0.035227400145142924, 0.0019224210882227836, 0.17810375234572726, 0.010585124923096559,
      0.002624683883278519, 0.007561675877429968, -0.7936628558377392, 0.2757591535116379,
      0.10088312806324856, 0.0002119462891323 0622, 0.002157818976534831, 0.004825502984106448,  },
00161         { 0.1637692767958545, 0.05388365117763547, 0.12484382106057478, 0.014328977324045077,
      0.056992689949096564, 0.004050108817990004, 0.00890032228076262, 0.10777814379180953,
      0.005051277690715041, 0.033138701874594126, 0.18324795571899924, 0.011933230455755117,
      0.001549813340602554, 0.14967317173262176, 0.1613284590151268, -1.2801057249450514,
      0.08061742242581896, 0.010847796434680762, 0.10184134919609925, 0.006329555862269497,  },
00162         { 0.3049069569974591, 0.0536236576858254, 0.0991298579165581, 0.0032940177756425465,
      0.0008937188082714867, 0.0020439801511351425, 0.006295349905905268, 0.004967098067947331,
      0.0009997320429540185, 0.24368846516621423, 0.012231772465335567, 0.07000162802915763,
      0.07379111260417322, 0.002160478822122848, 0.08385747783574576, 0.11454367592795515,
      -1.1010517138981215, 0.0002890176669985 9934, 0.0012715718968865967, 0.023062144131833412,  },
00163         { 0.002217505141799703, 0.11166720473242188, 0.002805995978148512, 0.001441132776843614,
      0.06263278026616122, 0.0064347523276476705, 0.004938593460667064, 0.03863298497292368,
      0.0014995980644310278, 0.0030209313863580277, 0.11294384295094899, 0.0011483861944868942,
      0.0027746658194658627, 0.020464535509552535, 0.0005090685740580593, 0.044535973444363774,
      0.0008351252323116603, -0.4299980688628356, 0.0083615485340724 68, 0.003133443496173018,  },
00164         { 0.0014348562682233374, 0.0027949300369581724, 0.032153953913537876, 0.0835445258347341,
      0.06993953538784027, 0.0035580395223463587, 0.004938593460667064, 0.0016556993559824439,
      0.23393729805124036, 0.009886684537171727, 0.010974300529646864, 0.010035896743124597,
      0.0008498976383949 49, 0.15357403627256577, 0.0025916218315683013, 0.20907385046794577,
      0.001837275511085653, 0.004181122249246404, -0.8414742762467692, 0.004512158634489147,  },
00165         { 0.23688172573578004, 0.005459863328011314, 0.0007359989450881345, 0.004241047886139779,
      0.004565212290900297, 0.001930425698294301, 0.011722375686858088, 0.05992054812126939,
      0.0002893961176972159 4, 0.5446830832978868, 0.17810375234572726, 0.004593544777947577,
      0.04796922259058551, 0.06133359322804308, 0.003563480018406415, 0.00798957960547201,
      0.020488405699379403, 0.0009633922233286645, 0.0027743386841162115, -1.1982089862809313,  },
00166                 }, new double[20] {
00167 0.061007, 0.060799, 0.043028, 0.038515, 0.011297, 0.035406, 0.050764, 0.073749, 0.024609, 0.085629,
      0.10693, 0.046704, 0.023382, 0.056136, 0.043289, 0.073994, 0.052078, 0.018023, 0.036043, 0.05862,  });
00168
00169 /// <summary>
00170 /// Dayhoff rate matrix prepared using the DCMut method.
00171 /// </summary>
00172 /// <remarks>Citation:  Kosiol, C., and Goldman, N. 2005.  Different versions of the Dayhoff rate
      matrix.  Molecular Biology and Evolution 22:193-199.</remarks>
00173           public static readonly ImmutableRateMatrix DayhoffDCMutMatrix = new
      ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M',
      'F', 'P', 'S', 'T', 'W', 'Y', 'V' },
00174                 new double[20, 20] {
00175         { -1.3623986751406103, 0.011216122929753615, 0.040752145486210 89, 0.057576485934644386,
      0.012338160536299818, 0.03476973314298245, 0.09944979848871842, 0.21647322349976,
      0.007851661018147275, 0.024675861797894636, 0.03551787472812525, 0.021310893678932345,
      0.010842489238126737, 0.007477700770853093, 0.12898664811681773, 0.28863048867549157,
      0.22058676413223588, 0, 0.007477590682196854, 0.13646503228341925,  },
00176         { 0.023890747665280 73, -0.8855562289329317, 0.013538555564265432, 0, 0.007963723046926064,
      0.09556264852403254, 0, 0.007964592076507517, 0.08202699614264515, 0.023890853259394145,
      0.013538758668458058, 0.3798629822363342, 0.013538324412692237, 0.00557469248008023,
      0.05335595959843808, 0.1091011237158389, 0.015927721743446917, 0.021502578062410793,
      0.0023893839341503856, 0.01592658780203039,  },
00177         { 0.08781688216702353, 0.013696603601125673, -1.8256124276959287, 0.4286073551723533, 0,
      0.04028258250341676, 0.07573002406793428, 0.125682172003666, 0.18208049950842342,
      0.029003963908851682, 0.02980975565873418, 0.25941809748423034, 0, 0.005639720922154997,
      0.021753265713734603, 0.348041125597730 15, 0.13615668291566402, 0.0024170342797049187,
      0.02900327158135115, 0.01047339060982967,  },
00178         { 0.1070248013745469, 0, 0.369718650459306, -1.4441348614583254, 0, 0.052817347167176155,
      0.5775132064769238, 0.11258451822733964, 0.029884506190084027, 0.009035056661334731, 0,
      0.05907188400659008, 0, 0, 0.006949729536254814, 0.06810651485145659, 0.039613573658568824, 0, 0,
      0.011815072848743806,  },
00179         { 0.03211408594868239, 0.009731377412662478, 0, 0, -0.27248134844081473, 0, 0,
      0.009732495458345085, 0.009731418523907841, 0.016543600832013443, 0, 0, 0, 0, 0.00973138550488719,
      0.11385712605720422, 0.009731586553261595, 0, 0.029193902838159744, 0.03211436931169077,  },
00180         { 0.07918919198924669, 0.10217996536993929, 0.04257496734487378, 0.06471453918232599, 0,
      -1.2670296761329414, 0.35932863442359847, 0.02554564592606376, 0.20691730281211046,
```

```
      0.006812433024761571, 0.0638619306864426, 0.12516832504496758, 0.017030112244370135, 0,
      0.07918914306015874, 0.040021197667144 81, 0.03150547653639623, 0, 0, 0.022990810820541465,  },
00181         { 0.17493968489655903, 0, 0.06181942929769269, 0.5465212803146855, 0, 0.277531130827271 54,
      -1.3896432373759031, 0.07365798383730666, 0.015126346314963283, 0.023018382375429323,
      0.009864546993981213, 0.06839640421142074, 0.004603843819303388, 0, 0.026306806958860677,
      0.05655964267803548, 0.020387627678277787, 0, 0.00657644144732821, 0.02433368572478725,  },
00182         { 0.21284546724894585, 0.003676518691570707, 0.05734642687731034, 0.059552448182547096,
      0.0036765398927080238, 0.01102840117480216, 0.04117139822441429, -0.6675757464664498,
      0.0036760819059608496, 0, 0.006249585522037318, 0.022056426827991756, 0.002573354196587162,
      0.006249489813870492, 0.0180126882014295, 0.16542241777580216, 0.018380127585801872, 0, 0,
      0.0356583743446702,  },
00183         { 0.020348364601211148, 0.09980166721850016, 0.21897970659819077, 0.04166532538569316,
      0.009689446553118505, 0.23545082896806221, 0.02228525336803984, 0.009689311694309847,
      -0.899177679102265, 0.0029071411123445506, 0.038757672307039734, 0.022286480823590103, 0,
      0.019379330839898495, 0.048447252617343 18, 0.02519136890045013, 0.013565519747726473,
      0.0029069132625710397, 0.038757196716837125, 0.029068898387338428,  },
00184         { 0.058285902804998256, 0.02649328909944852, 0.0317922319786014, 0.011481081598169537,
      0.015013297572271806, 0.007065272064258903, 0.03090875885308828, 0, 0.002649655073900977,
      -1.307905911011769, 0.22342788209874964, 0.037973493642359794, 0.05033866995926752,
      0.07948173609034728, 0.006182426263280824, 0.017662800835054313, 0.1139229031549702, 0,
      0.011480571419439645, 0.5837459385035623,  },
00185         { 0.03625438887774136, 0.006487919966430503, 0.0141203186709226, 0, 0, 0.028621415448174856,
      0.005724088388906469, 0.006487906935327751, 0.01526522939290707, 0.0965516695654074,
      -0.544962428880948, 0.01488338938787044, 0.07899736097412406, 0.06373194514573775,
      0.016409642852597245, 0.012211769364421086, 0.01984424689848479, 0.004961276366172161,
      0.0087772462284949, 0.11563261441722752,  },
00186         { 0.023070715244148787, 0.19306315062430904, 0.1303263194727004, 0.034403366597791904, 0,
      0.059496207484937245, 0.042092840553567545, 0.02428478888286683, 0.00930964076997398,
      0.017403987108660222, 0.01578511037363424, -0.7629424484914241, 0.03642769162023646, 0,
      0.017403847478879002, 0.06799681465067023, 0.08094965120313709, 0, 0.004047300054177565,
      0.0068810163717335756,  },
00187         { 0.06403264148649551, 0.037536204282299424, 0, 0, 0, 0.04415962474807696,
      0.01545640780655438, 0.01545652152565455, 0, 0.12585861723836114, 0.45705807230179,
      0.19872141593494547, -1.280653993800121, 0.037537332254881164, 0.008831407463926015,
      0.0441615254400708, 0.061826136036067555, 0, 0, 0.17001808728099788,  },
00188         { 0.016381113221917865, 0.005733360686040473, 0.005733309773825074, 0, 0, 0, 0,
      0.013923860791177009, 0.01638121601897183, 0.07371425418456577, 0.1367788303782746, 0,
      0.013924073789506735, -0.5585836389587654, 0.005733812245284265, 0.032760540286916225,
      0.008190569210524143, 0.008190657121324649, 0.21294800841803904, 0.008190032832397692,  },
00189         { 0.22174861267707138, 0.04306377607368807, 0.017354539055598216, 0.006427539913641191,
      0.006427553243697588, 0.05977467773808943, 0.02570986875833404, 0.03149448158849784,
      0.03213788843217168, 0.004499703534873253, 0.027637685180922314, 0.027637708148138533,
      0.0025708317741772, 0.00449970758917612, -0.7629424613771453, 0.172899056330566, 0.046921403830399 91,
      0, 0, 0.03213742750810255,  },
00190         { 0.36143421801499853, 0.06414005151806881, 0.20225072639187414, 0.04588137695096761,
      0.05477749022865104, 0.02200455490688913, 0.04026329249382839, 0.21067897845479658,
      0.012172249896722092, 0.009363871273579105, 0.014981387493552335, 0.07865317044282724,
      0.009363941889092151, 0.018726766148170117, 0.1259399539335281, -1.6348763369454706,
      0.32585309481582925, 0.007959287178873075, 0.010299999151816374, 0.020131925761406284,  },
00191         { 0.3282952922440182, 0.011128890885073158, 0.09403653793252927, 0.031716843027645755,
      0.005564468728158905, 0.02058764655973212, 0.017249140769107628, 0.02782104925750872,
      0.007790290874907184, 0.0717802638409045, 0.028933848903590 01, 0.111286066046251 87,
      0.015580625618190439, 0.00556447197979171, 0.04062001206184736, 0.38727547364287096,
      -1.321525317126502, 0, 0.012797746419287583, 0.10349664833508647,  },
00192         { 0, 0.08381374624212419, 0.009312514770061871, 0, 0, 0, 0, 0, 0.009312704114196284, 0,
      0.040354456526334 77, 0, 0, 0.03104238755758757, 0, 0.05277142405607509, 0, -0.245231453756032 97,
      0.018624220489653193, 0,  },
00193         { 0.02177764551971404, 0.003266992928282102, 0.039198431494089775, 0, 0.032666021647431445, 0,
      0.01088819176648503, 0, 0.04355455931352278, 0.014155380310785223, 0.025043435162643377,
      0.010888178755858558, 0, 0.2831049669341572, 0, 0.023955175858601 68, 0.025043644567386468,
      0.006533044852868718, -0.5585870939834543, 0.018511424871627873,  },
00194         { 0.18371687734104064, 0.010066150799688667, 0.00654315845880023, 0.0085570644112352,
      0.016610470013590294, 0.013589935843811825, 0.018623063969045898, 0.04882350918492406,
      0.015100393938068 71, 0.3327057802719861, 0.152508623084942 23, 0.008556986906478721,
      0.038757020329067055, 0.005033128122162629, 0.025166488861068 59, 0.021643422211177053,
      0.09362002513725134, 0, 0.008556936037263504, -1.0081790349221962,  },
00195             }, new double[20] {
00196 0.087127, 0.040904, 0.040432, 0.046872, 0.033474, 0.038255, 0.04953, 0.088612, 0.033619, 0.036886,
      0.085357, 0.080481, 0.014753, 0.039772, 0.05068, 0.069577, 0.058542, 0.010494, 0.029916, 0.064718,
      });
00197
00198 /// <summary>
00199 /// Dayhoff protein sequence evolution matrix.
00200 /// </summary>
00201 /// <remarks>Citation:  Dayhoff, M. O., Schwartz, R. M., and Orcutt, B. C. (1978) A model of
      evolutionary change in proteins.  Matrices for detecting distant relationships In M. O. Dayhoff,
      (ed.), Atlas of protein sequence and structure, volume 5, pp.  345-358 National biomedical research
      foundation Washington DC.</remarks>
00202         public static readonly ImmutableRateMatrix DayhoffMatrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00203             new double[20, 20] {
00204         { -1.3634324059950833, 0.011242118443529615, 0.0403338717440125, 0.05725499007813634,
      0.012266727712977061, 0.034657467579094915, 0.09982794497014537, 0.21648229991482412,
      0.007870789934693798, 0.02440582715719387, 0.03562391223475259, 0.02130054526124033,
      0.010812632726865662, 0.007287331878480668, 0.12897193852228542, 0.28967273494278667,
      0.22108540714005337, 0, 0.007308586291080078, 0.137027279462931,  },
```

```
00205         { 0.02394611905020058, -0.8852989321897626, 0.013170243834779593, 0, 0.007837076038846456,
      0.09579479802761066, 0.0005041815402532595, 0.008118086246805904, 0.08212998192723962,
      0.02403035289323704, 0.013033138622470464, 0.3801328077390582, 0.013515790908582079,
      0.005667924794373852, 0.0531364386711816, 0.10906992953835976, 0.015493856025987567,
      0.021471170962385596, 0.0024361954303600258, 0.0158108399380305,  },
00206         { 0.08691554321924656, 0.013323992229368432, -1.8300026751803307, 0.4317980501726116, 0,
      0.04010920405221097, 0.07461886795748239, 0.12537933203400228, 0.1830814180461383,
      0.028911518324675818, 0.029541780877599715, 0.2605220535797856, 0.00015017545453980088,
      0.005667924794373852, 0.021667285671743954, 0.3505819163732993, 0.1364651165365828,
      0.002456900159874969, 0.028929820735525 31, 0.009881774961269062,  },
00207         { 0.10642719577866926, 0, 0.37247095845236033, -1.4469383668345417, 0, 0.05218090624268223,
      0.5813213159120081, 0.11275119787230423, 0.0294299101905942, 0.009011382334963 89, 0,
      0.05816687359800243, 0, 0, 0.006706540803158843, 0.06728339809184532, 0.03933055760442998, 0, 0,
      0.011858129953522875,  },
00208         { 0.031928158733600775, 0.00957661941485856, 0, 0, -0.2724085940053223, 0, 0,
      0.009922105412762 77, 0.009581831224844623, 0.016520867614100467, 0, 0, 0, 0, 0.009801867327693692,
      0.11402765360828522, 0.009534680631376963, 0, 0.029234345164320313, 0.03228046487347894,  },
00209         { 0.07893350353584636, 0.10242819026326982, 0.04239172234319682, 0.06393473892058558, 0,
      -1.2672023302191789, 0.3609939828213337, 0.025256268323396145, 0.20737820436628004,
      0.006758536751222917, 0.063427941296022 92, 0.12534551634499116, 0.017120001817537298, 0,
      0.07893082637563 87, 0.03966179255940355, 0.031583629591436194, 0, 0, 0.023057474909627815,  },
00210         { 0.17560487303480427, 0.0004163747571677635, 0.060912377735855616, 0.5501250296674267, 0,
      0.2788173796250782, -1.3922160008508242, 0.07306277622125314, 0.0147149550952971,
      0.022903930101366557, 0.009557634989811672, 0.06799789448780566, 0.004505263636194026, 0,
      0.02631027545854623, 0.05595145736058715, 0.020261196341676047, 0, 0.006699537433490071,
      0.024375044904463686,  },
00211         { 0.2128543915573385, 0.0037473728145098716, 0.05720824665732386, 0.05964061466472536,
      0.0037481668011874352, 0.010903472946232107, 0.04083870476051401, -0.6659124733120019,
      0.0034220825803016513, 0, 0.006082131357152882, 0.022119797002057264, 0.0025529827271766146,
      0.006072776565400556, 0.01754018363903082, 0.16572963319465053, 0.01787752618383181, 0, 0,
      0.03557438986056862,  },
00212         { 0.020398545857578276, 0.09992994172026323, 0.22019001411272132, 0.04103274288933104,
      0.009540788221204 38, 0.2359823073363092, 0.021679806230890154, 0.009020095829784339,
      -0.8963041168652541, 0.0026283198476978018, 0.03823053995924669, 0.02130054526124033, 0,
      0.019432885009281 78, 0.04849344888437933, 0.02478862034962722, 0.013110185868143324,
      0.0028841871442010503, 0.03867460245696541, 0.02898653988638925,  },
00213         { 0.05764806438011251, 0.026647984458736864, 0.03169089922743839, 0.011450998015627269,
      0.014992667204749741, 0.007009375465434926, 0.030755073955448825, 0, 0.00239545780621115 57,
      -1.307919107128715, 0.22330110839832726, 0.037685580077579044, 0.05045895272537309,
      0.07935094712123393, 0.006190653049069 7005, 0.016997911096887235, 0.11441616757652358, 0,
      0.011267403865415119, 0.5856598627045464,  },
00214         { 0.03636262522437866, 0.0062456213575164 52, 0.013993384074453318, 0, 0, 0.028426911609819423,
      0.005545996942785853, 0.006314067080849036, 0.015057163353327263, 0.09649688583690501,
      -0.5428351122425857, 0.014746531334704842, 0.07914246454247505, 0.06356172805119248,
      0.016508408130852536, 0.012040187026961791, 0.01966527880221499, 0.004913800319749938,
      0.008526684006260091, 0.11528737454813906,  },
00215         { 0.023059225752045002, 0.19319788732584228, 0.13087929810812218, 0.03387586912956401, 0,
      0.059579691456196 87, 0.041847067841020 53, 0.024354258740417 71, 0.008897414708784293,
      0.017271816142014125, 0.015639766346964557, -0.7617023241918524, 0.036492635453171614, 0,
      0.017024295884941677, 0.067991644387548 94, 0.08104478536670419, 0, 0.003958817574335042,
      0.006587849974179375,  },
00216         { 0.06385631746720155, 0.037473728145098716, 0.00041157011983686227, 0, 0,
      0.044392711281087 87, 0.015125446207597783, 0.015334162910633373, 0, 0.126159352689494 46,
      0.4578976036027956, 0.19907817301851538, -1.2815987439676373, 0.03724636293445674,
      0.008770091819515 41, 0.043911270333625366, 0.061975424103950266, 0, 0, 0.16996652933382786,  },
00217         { 0.015964079366800388, 0.005829246600348689, 0.0057619816777160 71, 0, 0, 0, 0,
      0.013530143744676508, 0.016425996385447925, 0.07359295573553844, 0.13641351758185752, 0,
      0.013816141817660 18, -0.5559170097905118, 0.005674765294980558, 0.0325793296023672,
      0.007746928012993783, 0.008118452702195 55, 0.212558051298912 28, 0.007905419969015 25,  },
00218         { 0.2217233245388943, 0.042886599988279 64, 0.017285945033148214, 0.006202623925131438,
      0.006474106292960116, 0.05957969145619687, 0.025713258552916233, 0.030668325821266745,
      0.032167576254835 52, 0.004505691167481945, 0.027804029061270 32, 0.027053307446958878,
      0.002552982727176 6116, 0.004453369481293 7405, -0.7606764221492248, 0.173520342447390 54,
      0.046481568077962 69, 0, 0, 0.031621679876061,  },
00219         { 0.362739358945631 03, 0.06412171260383558, 0.20372720931924682, 0.045326867145191276,
      0.054859532271925186, 0.021806945892464215, 0.03983034168000749, 0.21107024241695352,
      0.011977289031055778, 0.009011382334963 89, 0.014770890438799858, 0.078648167118425 83,
      0.009310878181467653, 0.018623181467228 37, 0.126392499751839 72, -1.6381001567603015,
      0.3277546467035831, 0.008011630956114029, 0.010353830573903011, 0.019763549922538124,  },
00220         { 0.32903741361571914, 0.010825743686361 85, 0.09424955744264146, 0.031490244542974 99,
      0.005451878983545 36, 0.020638716648225 06, 0.017142172368610 82, 0.027060287489353015,
      0.007528581676663632, 0.072091058679711 12, 0.028672904969435 02, 0.111418236751103 25,
      0.015618247272139 29, 0.005263073023347149, 0.040239244818953 05, 0.389535462636999 2,
      -1.3224820952097895, 0, 0.012790026009390136, 0.10342924459461 62,  },
00221         { 0, 0.08369132619072046, 0.009466112756247832, 0, 0, 0, 0, 0, 0.009239622966814458, 0,
      0.039968291775576086, 0, 0, 0.030768734598029483, 0, 0.053118472177772616, 0, -0.24482855062165615,
      0.018575990156495 2, 0,  },
00222         { 0.021285439155733 85, 0.003330998057342108, 0.039099161384501 92, 0, 0.032711273901272 16, 0,
      0.011091993885571706, 0, 0.043460448769830 97, 0.013892547766402665, 0.024328525428611 53,
      0.010650272630620164, 0, 0.2825865361766392, 0, 0.024080374053923582, 0.025028536657364 53,
      0.006516126510972743, -0.5565082143064893, 0.018445979927702 25,  },
00223         { 0.18447380601636001, 0.009992994172026325, 0.006173551797552934, 0.008588248511720452,
      0.016696379387107663, 0.013629341182790135, 0.018654716989370598, 0.048708517480835 42,
      0.015057163353327263, 0.3337966206576208, 0.152053283928822 06, 0.008192517408169358,
      0.038745267271268626, 0.004858221252320445, 0.024762612196278802, 0.021247388871109045,
      0.093559053695386 46, 0, 0.008526684006260091, -1.0077163681783265,  },
00224                 }, new double[20] {
```

```
00225 0.087127, 0.040904, 0.040432, 0.046872, 0.033474, 0.038255, 0.04953, 0.088612, 0.033618, 0.036886,
      0.085357, 0.080482, 0.014753, 0.039772, 0.05068, 0.069577, 0.058542, 0.010494, 0.029916, 0.064718,
      });
00226
00227 /// <summary>
00228 /// JTT rate matrix prepared using the DCMut method.
00229 /// </summary>
00230 /// <remarks>Citation:  Kosiol, C., and Goldman, N. 2005.  Different versions of the Dayhoff rate
      matrix.  Molecular Biology and Evolution 22:193-199.</remarks>
00231             public static readonly ImmutableRateMatrix JTTDCMutMatrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00232                 new double[20, 20] {
00233         { -1.2928238294558667, 0.0281468321036338, 0.024614600367361905, 0.04398651167844037,
      0.012079403154917593, 0.023702588958263795, 0.06837370272266732, 0.13480824638984204,
      0.005241984242941388, 0.019714444850004956, 0.029286525644394715, 0.02279125790163828,
      0.011395663392627488, 0.005811688665702252, 0.1026737054748352, 0.2749756507287224,
      0.27805047906557645, 0.001253517732522138, 0.004672159901677867, 0.2012448664800967,  },
00234         { 0.042372677774830114, -1.0730468079009121, 0.019899963891619254, 0.008234343880836473,
      0.02144398001615488, 0.1286615569965933, 0.020414596270321 92, 0.105330778291203, 0.07651173701536058,
      0.013037891186496875, 0.03516769402919296, 0.402801924578646, 0.01046462729060837,
      0.0027447855893767354, 0.037226258193186555, 0.07085037488484398, 0.0394563353990006,
      0.018699100194728757, 0.007891244981756711, 0.01183693743615493,  },
00235         { 0.04446780927551758, 0.023880798580698637, -1.344537769632745, 0.295009899334524,
      0.0065879109067194675, 0.032733138046859 38, 0.037056873750929115, 0.05990772650112313,
      0.09593610420320996, 0.02676328387400876, 0.012969764615616119, 0.1560506238237598,
      0.008029008285944623, 0.0030879993553142035, 0.006381952645238555, 0.357803692027710 06,
      0.14266751262276905, 0.0004117497087699 7505, 0.02346908595465238, 0.0113228415458786,  },
00236         { 0.06594416237157509, 0.008200294437649804, 0.24481638372284728, -1.1118339420349186,
      0.0022209500768317863, 0.022209099141806955, 0.4978323036565298, 0.09857409362409543,
      0.024601175148095643, 0.006321111081400822, 0.005808615017632677, 0.017425854623235242,
      0.004612719437281214, 0.001366719492569896, 0.006662791256273321, 0.041685204954757415,
      0.025796832915417768, 0.000854209702677 8119, 0.015204716626662972, 0.021696704747577775,  },
00237         { 0.04578367203970986, 0.053990102454993825, 0.013821650842609915, 0.0056149657028990016,
      -0.5688423247061868, 0.003887271421698895, 0.003455410936044447, 0.0423281682516939,
      0.017277029104714372, 0.008206575635560036, 0.015549187971200196, 0.0030234566610853 55,
      0.0099343372886161, 0.028506662166914717, 0.006478831486992899, 0.15246952911127407,
      0.02850685374370605, 0.016413220403586278, 0.07083507808607835, 0.042760321396808555,  },
00238         { 0.04436882668493392, 0.15998327160992334, 0.033916955050819, 0.027730408511758134,
      0.0019198260432194024, -1.085975363440827, 0.21907319436408487, 0.017918097449998605,
      0.1354541882784847, 0.004266292118008781, 0.06697998376803889, 0.18302323301954904,
      0.011092342269847127, 0.0019198034124306795, 0.08425818511359968, 0.038822967267195155,
      0.03178346454366181, 0.002559775102831853, 0.008532456156288018, 0.012372092676153421,  },
00239         { 0.08501034517420 99, 0.016860369488415176, 0.02550342527672323, 0.41286581003181216,
      0.0011334888122297854, 0.14550896851801504, -0.9954711393982095, 0.08642681130654857,
      0.005809101308966388, 0.006092452649881123, 0.009209459632988349, 0.10683081729263876,
      0.004250574312540166, 0.0018418900633370018, 0.01005957617294942, 0.022102847266284607,
      0.020119101431290 14, 0.0017002290050114987, 0.0021252680446281077, 0.032020603609740 05,  },
00240         { 0.13868393385464625, 0.0719794623124709, 0.0341145452220037, 0.067641877104876 57,
      0.011488782878390938, 0.009847351224595027, 0.07151143661122189, -0.6464123500849578,
      0.004806506586645025, 0.002930806961712203, 0.006569460821216215, 0.016646933123044185,
      0.0031652556960925855, 0.002110132192390 3704, 0.010902458772896487, 0.13258892881656903,
      0.01922583332646164, 0.008089023701026895, 0.0017584405904144183, 0.032355695027378,  },
00241         { 0.0175307572066728, 0.16997170764448788, 0.17759637512203677, 0.054878721170766025,
      0.015244349006417905, 0.24199993146686077, 0.01562540325111178, 0.01562517221923145,
      -1.1764684141862969, 0.009908786400262 94, 0.05106802896101087, 0.032394152072257 66,
      0.008003268875922363, 0.019055066910921306, 0.05983334721937921, 0.052592706723000 95,
      0.028963863346040532, 0.0019055539004 2003399, 0.19588693236151777, 0.008384305224189346,  },
00242         { 0.0288248142452982, 0.012662892775380375, 0.021660497169502496, 0.006164793776414593,
      0.0031657658946056 03, 0.003332348354687336, 0.007164591733067988, 0.004165426607646437,
      0.004332089973886571, -1.3316082253048758, 0.22060181933545445, 0.012496427761896219,
      0.11730001248756994, 0.032656787727727 09, 0.0051651250514565756, 0.028658383869591374,
      0.15495404467597154, 0.001999438748254489, 0.010163619142062917, 0.6561393459744018,  },
00243         { 0.024706357454966 65, 0.01970735645584512, 0.006056476225000312, 0.003268561242210 1587,
      0.0034608552520328916, 0.03018587342362003, 0.006248738291878475, 0.005383470509374663,
      0.01288205057140096, 0.12728229347329637, -0.6981256368991662, 0.009036686224416 82,
      0.0936354296765098, 0.10507350553334836, 0.055565386260599 55, 0.041914616902577 89, 0.0165349860290201,
      0.007883059658979 36, 0.008075228108673786, 0.1212247056054147,  },
00244         { 0.029442698323233072, 0.34565628866872716, 0.11158912637745276, 0.015015733985657 46,
      0.0010304998089036592, 0.12630873257951028, 0.11100005252329369, 0.020904214618224 53,
      0.012513274346645 23, 0.011041122575802921, 0.013838137728878968, -0.9308324462978204,
      0.015163167135207383, 0.0010304817870151412, 0.011335453228417254, 0.033564885037917 19,
      0.058590973102479134, 0.0013249421856138253, 0.0029442659363769336, 0.008583896348463604,  },
00245         { 0.03740896385428094, 0.022819359168727748, 0.014589645325787961, 0.010100346494830895,
      0.008604186635168952, 0.019452578198607365, 0.011222794225729608, 0.010100320922433648,
      0.007855946381452279, 0.2633614229289768, 0.36436395461076637, 0.038531567361859 1,
      -1.2023274819127545, 0.018330271046941446, 0.008604088155051141, 0.020200984726305084,
      0.128312666267339, 0.0029927514752885976, 0.006359525998133059, 0.209116108135074 53,  },
00246         { 0.011021416585818072, 0.003457698441569405, 0.003241599323246931, 0.001728851262387 5156,
      0.014263177944309487, 0.0019449555537079 1053, 0.002809416326560411, 0.003889869642789394,
      0.010805393605075363, 0.042357134815171116, 0.23620409974460654, 0.001517246246317759,
      0.010589315723984385, -0.6593402855810772, 0.007779805873558802, 0.066777127905040 34,
      0.008428101673216818, 0.007996000968987817, 0.18368958456784779, 0.040843989393496 06,  },
00247         { 0.15617245211364647, 0.03761301877166006, 0.005373358609283612, 0.006759966851062 2355,
      0.0026000202589394634, 0.06846602823853234, 0.012306019651649911, 0.016119811302900897,
      0.027213445324606 04, 0.005373336872279362, 0.10018637512050751, 0.013346726751056158,
      0.00398670387204875, 0.006239917914496521, -0.7498377485952056, 0.197253669988995 35,
```

```
      0.0714129684775884, 0.0010400024683787474, 0.0038133040232129814, 0.014559925119511456,  },
00248         { 0.30978641944025004, 0.053021730897698495, 0.22313104992320928, 0.03132515606926285,
      0.04531959810696265, 0.02336547979418542, 0.020027819978039055, 0.14519969553098042,
      0.017716939224843017, 0.022081972614738717, 0.05597482829770282, 0.02927143319876874,
      0.0069327351613295 32, 0.039669871659820964, 0.14609926642556123, -1.5126034362160459,
      0.28988722192837724, 0.0046218087255856 33, 0.021054663288747173, 0.02811574594998253,  },
00249         { 0.3652126853607153, 0.03442568297731935, 0.10372760504542759, 0.022601213758852892,
      0.009878849021986652, 0.022301870153239985, 0.021254363622857177, 0.024546958391490736,
      0.011375584799242108, 0.13920125729811592, 0.02574454205697478, 0.05957219518184667,
      0.05133997689571543, 0.005837365610845853, 0.061667181433225 63, 0.3379738835241043,
      -1.3833244226535613, 0.0011974335573889572, 0.006735463849310419, 0.07873031011490168,  },
00250         { 0.006720694751472976, 0.06659597925797058, 0.0012219798485858927, 0.003054860299008701,
      0.023217333744721406, 0.007331677280788136, 0.007331763189858459, 0.04215703939721844,
      0.0030548969680201177, 0.007331786799455233, 0.05009998943842553, 0.005498842784573897,
      0.004887854564900059, 0.022605881645722 39, 0.0036658345934789944, 0.021995179987659024,
      0.004887794148387765, -0.3232055466560041, 0.025049873804275216, 0.01649628415148131,  },
00251         { 0.011116972242911317, 0.012472627775548787, 0.030910928738093684, 0.024131833474673684,
      0.044468456443908705, 0.010845778479811236, 0.00406724052004178, 0.004067118542309471,
      0.13937000793934815, 0.016539989929081064, 0.022776277999237756, 0.005422961789365835,
      0.004609539105355152, 0.23047205716295296, 0.005965200721326142, 0.04446814236680109,
      0.012201525354733218, 0.011117078626074653, -0.6464118214487702, 0.011388084237174712,  },
00252         { 0.2330443084248831, 0.0091053502075777, 0.007257986808252491, 0.01675909777478478,
      0.013064401084850704, 0.007653757455864278, 0.029823631469462892, 0.036421240218692856,
      0.0029031923187926557, 0.51967019131781, 0.16640407618065717, 0.0076538630478935644,
      0.07376750769690897, 0.024940592553083965, 0.011084794289016073, 0.028899821728953477,
      0.06941182220905501, 0.0035630025250193756, 0.005542370282240858, -1.2669710075937994,  },
00253             }, new double[20] {
00254 0.076862, 0.051057, 0.042546, 0.051269, 0.020279, 0.041061, 0.06182, 0.074714, 0.022983, 0.052569,
      0.091111, 0.059498, 0.023414, 0.04053, 0.050532, 0.068225, 0.058518, 0.014336, 0.032303, 0.066374,
      });
00255
00256 /// <summary>
00257 /// JTT protein sequence evolution matrix.
00258 /// </summary>
00259 /// <remarks>Citation:  David T. Jones, William R. Taylor, Janet M. Thornton, The rapid generation of
      mutation data matrices from protein sequences, Bioinformatics, Volume 8, Issue 3, June 1992, Pages
      275-282</remarks>
00260         public static readonly ImmutableRateMatrix JTTMatrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00261             new double[20, 20] {
00262         { -1.2950937799320588, 0.030917490151747835, 0.023747789032223136, 0.0430550839914495,
      0.01143616250764773, 0.023954388392780166, 0.0669498871239073, 0.1350331935609873,
      0.006388430901105962, 0.019958649097766023, 0.028432625984777658, 0.021178239133551047,
      0.01326802254617771, 0.0062069526368011465, 0.10183318987882857, 0.26805292506989725,
      0.286875236434852, 0.001323590961771736, 0.0036415483377568894, 0.20284037418803064,  },
00263         { 0.045904616551553326, -1.064824819239263, 0.01978982419351928, 0.008504707948928296,
      0.023076542202932027, 0.13027825266248863, 0.018490921205650587, 0.10334942747405174,
      0.07760760502084281, 0.012196952226412 57, 0.036014659580717, 0.3908897851506851, 0.010810981333922578,
      0.002068984212267049, 0.038843587891924304, 0.0716226069631207, 0.038652663435432694,
      0.018530273464804306, 0.0066209969777397985, 0.011571430742270204,  },
00264         { 0.04273878092730827, 0.023987707876356082, -1.3290708876081345, 0.2806553623146338,
      0.006943384379643266, 0.036141708803142006, 0.036981842411301175, 0.061104406024804 32,
      0.09251394379009006, 0.026057125210972306, 0.011373050393910632, 0.1591393397749693,
      0.007371123636765393, 0.004137968424534098, 0.007873700248363036, 0.3566947653707892,
      0.14011590495344 35, 0.0011765252993526543, 0.023173489422089296, 0.010890758345666074,  },
00265         { 0.0641081713909624, 0.008528962800482163, 0.23220060387062622, -1.0844802001931104,
      0.0020242171876365 6664, 0.020592368969232073, 0.48905298499082756, 0.09806879979289582,
      0.026500157811995106, 0.006098476113206285, 0.006634279396447868, 0.015732406213495066,
      0.0036855618183826966, 0.001655187369813639, 0.007873700248363036, 0.041838948622021,
      0.022950018914788 16, 0.0005882626496763272, 0.015228293048801536, 0.021100844294728017,  },
00266         { 0.0443216987394307 94, 0.060235799778405 27, 0.014952311612881235, 0.005315442468080185,
      -0.5871369440659445, 0.0037822718514916058, 0.0031880898630432045, 0.044508147598314 26,
      0.016325990080604126, 0.009424917629500621, 0.021798346588328712, 0.004235647826710209,
      0.007616827757990907, 0.032276153711365 96, 0.007348786898472166, 0.158137043096791 25,
      0.025365810379502703, 0.016912551178194405, 0.0691894184173809, 0.042201688589456034,  },
00267         { 0.04511315764549206, 0.16524865425934188, 0.03782055290317018, 0.026045668093592 91,
      0.0018379546887291, -1.0836982081041098, 0.205950605152591, 0.019613759958579162, 0.1412553054800096,
      0.004989662274441506, 0.06823830236346379, 0.17668702362984304, 0.010565277212697064,
      0.001655187369813639, 0.086085789382102 51, 0.037584140287578185, 0.030801341175110426,
      0.002647181923543472, 0.007945196373287758, 0.013613447932082592,  },
00268         { 0.08310318513643274, 0.015458745075873918, 0.025506884516091517, 0.40769443730175015,
      0.0010210859381828332, 0.13574153422575427, -0.9801115223946416, 0.089770670759650 78,
      0.006151822349213149, 0.006652883032588674, 0.008529787795432973, 0.109521750947792 57,
      0.004422674182059236, 0.002068984212267049, 0.009448440298035642, 0.021274041672214 07,
      0.019326331717716347, 0.0014706566241908178, 0.0023173489422089297, 0.030630257847185833,  },
00269         { 0.1416711441849663, 0.07302924397912851, 0.035621683548334705, 0.069107520850424,
      0.012048814070557431, 0.010926563126531305, 0.07587653874042827, -0.672368935861015,
      0.005441996693534709, 0.003326441516294337, 0.005686525196955316, 0.016337498760167952,
      0.0034398576971571837, 0.002068984212267049, 0.012597920397380856, 0.14253607920383424,
      0.01993027958389498, 0.008088611433049498, 0.0026483987910959194, 0.031991602640394 09,  },
00270         { 0.021369390463654134, 0.17484373740988432, 0.1719515835481342, 0.059532955642498074,
      0.0140909859469231, 0.2508906994822765, 0.016578067287824662, 0.017350633809512336,
      -1.178711030662898, 0.008870510710118231, 0.05307423517158295, 0.02722916460027992,
      0.008108236000441934, 0.016551873698136 39, 0.060365033523744993, 0.051766834737520 89,
      0.027781601844217244, 0.0011765252993526543, 0.189691563412245 22, 0.007487396362645425,  },
00271         { 0.028492520618205515, 0.011727323850662974, 0.020669371935453 47, 0.005846986714888204,
```

```
      0.0034716921898216333, 0.0037822718514916058, 0.0076514156713036905, 0.004526252298133653,
      0.0037857368302850153, -1.321841529041229, 0.21703571168379457, 0.012706943480130 63,
      0.11769227406702079, 0.03682791897835347, 0.005249133498908689, 0.028363588896285423,
      0.14796722721376576, 0.001323590961771736, 0.010593595164383678, 0.6541261731365685,  },
00272         { 0.023743767181837926, 0.020256286651145136, 0.005277286451605142, 0.0037208097276561296,
      0.004696995315641032, 0.030258174811932846, 0.005738561753477768, 0.004526252298133653,
      0.013250078905997553, 0.1269591845385672, -0.6934459584265384, 0.008471295653420419,
      0.0953331990354991, 0.10262161692844561, 0.05354116168886863, 0.041838948622021, 0.015098696654465895,
      0.007647414445792253, 0.007945196373287758, 0.12252103138874333,  },
00273         { 0.027701061712144245, 0.34435687306946733, 0.11566052806434601, 0.013820150417008481,
      0.0014295203134559664, 0.12271370895950542, 0.1154088530421 64, 0.020368135341601437,
      0.010647384835176606, 0.011642545300703018, 0.013268558792895737, -0.934850886589255,
      0.015970767879658353, 0.0016551873698136 39, 0.011023180347708248, 0.033329331953135 37,
      0.06220663021639949, 0.0014706566241908178, 0.0026483987910959194, 0.009529413552457816,  },
00274         { 0.04273878092730827, 0.023454647701325947, 0.013193216129012853, 0.007973163702120277,
      0.006330732816733566, 0.018070854401571003, 0.011477123506955535, 0.010561255362311856,
      0.007808082212462843, 0.265560914384 16457, 0.3677286294031104, 0.03933101553373766,
      -1.2268230758919016, 0.017793264225496618, 0.008398613598253904, 0.020564906949806932,
      0.1364922177563717, 0.003529575898057963, 0.005958897279965819, 0.21985718410313387,  },
00275         { 0.011871883590918963, 0.0026653008751506757, 0.0043977387096709505, 0.002126176987232074,
      0.015928940635652197, 0.0016810097117740468, 0.0031880898630432045, 0.0037718769151113773,
      0.009464342075712538, 0.049342215825032 67, 0.235043041474153 06, 0.0024203701866915483,
      0.010565277212697064, -0.6613164475676792, 0.008923526948144772, 0.06524039446145646,
      0.007247374394143629, 0.007794480108211334, 0.1774427190034266, 0.042201688589456034,  },
00276         { 0.15354302777588524, 0.03944645295223, 0.006596608064506427, 0.007973163702120277,
      0.0028590406269119327, 0.06892139818273592, 0.011477123506955535, 0.018105009192534 61,
      0.027209983467673546, 0.005544069193823895, 0.09667092834824036, 0.012706943480130 63,
      0.0039312659396082104, 0.007034546321707966, -0.7552371624354566, 0.202103395886033 65,
      0.07126584820907902, 0.0008823939745144908, 0.0033104984888698992, 0.015655465121894 98,  },
00277         { 0.2991714664911579, 0.05383907767804365, 0.22120625709644887, 0.031361110561673095,
      0.04554043284295436, 0.022273378681006123, 0.019128539185259227, 0.151629459187477 38,
      0.017272424288175 38, 0.022176276775295 58, 0.0559174977700606, 0.028439349693625694,
      0.007125419515539881, 0.038069309505713696, 0.14960030471889765, -1.5027024186870435,
      0.288083132167209 27, 0.005147298184667862, 0.020856140479880367, 0.025865551070956 93,  },
00278         { 0.3759429803791005, 0.034115851201928 65, 0.102027538064366 07, 0.020198681378704703,
      0.008577121880735799, 0.021432873825119096, 0.020403775123476 51, 0.024894387639735 09,
      0.010883993387069418, 0.135826969524868543, 0.023693854987313818, 0.062324532307307376,
      0.055529131396965 96, 0.004965562109440917, 0.061939775287122543, 0.338257262588203 66,
      -1.3859691600005937, 0.0017647879490289816, 0.0069520468266267895, 0.076235308419662 53,  },
00279         { 0.007123130154551379, 0.067165582053797 02, 0.003518190967736761, 0.002126176987232074,
      0.023484976578205162, 0.007564543702983 2115, 0.006376179726086409, 0.041490646066225 15,
      0.0018928684151425076, 0.004989662274441506, 0.049283218373612735, 0.006050925466728 87,
      0.005896898909412315, 0.021931232650030715, 0.003149480099345214, 0.024819715284249744,
      0.007247374394143629, -0.32463215129000 39, 0.023504539270976288, 0.017016809915103 24,  },
00280         { 0.008706047966673906, 0.010661203500602703, 0.030784170967696655, 0.024451035353168848,
      0.042681392216042426, 0.010086058270644281, 0.004463325808260486, 0.006035003064178204,
      0.13557670023458 21, 0.017741021420236463, 0.022746100787821265, 0.004840740373383097,
      0.004422674182059236, 0.221795107555027 63, 0.005249133498908689, 0.044675487511649 54,
      0.012682905189751351, 0.010441662031754806, -0.6289305282781077, 0.010890758345666074,  },
00281         { 0.23585475400625674, 0.009062022975512297, 0.007036381935473522, 0.016477871651048574,
      0.012661465633467131, 0.008405048558870235, 0.028692808767388 84, 0.035455643002046 95,
      0.002602694070820948, 0.532785049526476 3, 0.170595755908659 47, 0.008471295653420419,
      0.079362431155840 74, 0.025655404232111403, 0.012073007047489986, 0.026947119451471154,
      0.06764216101200721, 0.0036766415604770446, 0.005296797582191839, -1.2887543537310306,  },
00282             }, new double[20] {
00283 0.076748, 0.051691, 0.042645, 0.051544, 0.019803, 0.040752, 0.06183, 0.073152, 0.022944, 0.053761,
      0.091904, 0.058676, 0.023826, 0.040126, 0.050901, 0.068765, 0.058565, 0.014261, 0.032102, 0.066005,
      });
00284
00285 /// <summary>
00286 /// LG protein sequence evolution matrix.
00287 /// </summary>
00288 /// <remarks>Citation:  Le, S. Q., and O. Gascuel.  2008.  An improved general amino acid replacement
      matrix.  Mol.  Biol.  Evol.  25:1307-1320.</remarks>
00289         public static readonly ImmutableRateMatrix LGMatrix = new ImmutableRateMatrix(new char[20]
      { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W', 'Y', 'V'
      },
00290             new double[20, 20] {
00291         { -1.1067338146342782, 0.024154024927070 75, 0.011802691482520505, 0.021292785735778922,
      0.032707583772433 98, 0.040161355014497 44, 0.07551422096426 56, 0.120323102779573 8,
      0.008148405583470372, 0.009459412313586 69, 0.039786265332239 38, 0.035204010843190754,
      0.026203347629781198, 0.010900800991046372, 0.052679209167095 39, 0.293837883449863 34,
      0.11580014295907595, 0.002214816042388920 6, 0.007596130578979939, 0.178947625067418 82,  },
00292         { 0.034138862996438674, -0.8972349426496831, 0.0320578288496216, 0.006679402858433231,
      0.007024219195952 55, 0.11626980890287 75, 0.026464824349800683, 0.022724202880760998,
      0.055099591307020 56, 0.008017488013846944, 0.030377638870143175, 0.415083933576266 2,
      0.011286041180255 83, 0.002265312434125001, 0.014875014297072504, 0.053341985462032 91,
      0.031337577019803 45, 0.007275078196707338, 0.010908559589943561, 0.012002112668580422,  },
00293         { 0.022231021863329113, 0.042722133636912 64, -1.2057263781461713, 0.27353408635810855,
      0.006948228206112117, 0.070217671300723 65, 0.039388721400607 82, 0.083726310766268,
      0.102388967492425 33, 0.012090408037701342, 0.006886415331449 23, 0.140750823545330 35,
      0.008648793662154066, 0.003846631305053691, 0.007237128158951048, 0.249156353791609 33,
      0.108286424832342 24, 0.000556115322517 7468, 0.021232385138771175, 0.005877760186603 77,  },
00294         { 0.03173368387591601, 0.007043136456752117, 0.21643180922593536, -0.8768633936933968,
      0.000822011475092 1946, 0.021672358995537 41, 0.381289937256337 9, 0.049207236035669275,
      0.021051504757072577, 0.000674905677315 9029, 0.001517231466189 2029, 0.018566556395458143,
      0.000595571423706 2459, 0.000748315339947 6692, 0.017644981519325996, 0.077094510195668 22,
```

```
      0.023049603099298425, 0.0003663233204790077, 0.004687135098964841, 0.0026665820787303476,  },
00295         { 0.1998962524968126 4, 0.030373490456889665, 0.022545085831952 41, 0.0033709015054951773,
      -0.881465950895653, 0.003511728287905172, 0.0002544177278345814, 0.033153148585610186,
      0.014544483215235169, 0.02024256151550664, 0.05978018781750131, 0.0008704580421267664,
      0.02083334389924056, 0.04748945095271597, 0.003372021206141705, 0.1730809437911865,
      0.06189066865163613, 0.008212897761992521, 0.04043466249836566, 0.13760924665150423,  },
00296         { 0.07789137526863038, 0.1595469222615319, 0.072301792827298 47, 0.028203252371556665,
      0.0011144118738349452, -1.2691818974788556, 0.30019626789891457, 0.015605534402873038,
      0.10929780308105866, 0.004599586362504471, 0.05861780897467263, 0.212220505269286,
      0.038990695967470325, 0.0015405860423647036, 0.027926197326510697, 0.07607217772166998,
      0.058462184974554555, 0.002894787620402179, 0.008927506330739462, 0.014772500902981838,  },
00297         { 0.08340467961278217, 0.020680981462188136, 0.023096979272948 82, 0.2825719239980337,
      4.597829387025368E-05, 0.17095662913747173, -0.8557924238442528, 0.020316331453091894,
      0.009624849681843527, 0.002794639832215944, 0.007011811352069536, 0.11857920648247577,
      0.004050086163206694, 0.0008082544705877128, 0.018761190231708975, 0.03803975625403533,
      0.03272089960471838, 0.0009541319219113988, 0.004164326318210327, 0.017209768300882663,  },
00298         { 0.16592194297521287, 0.02217093034781469, 0.06129688241511 82, 0.04552980250386882,
      0.0074803753815518605, 0.011095641924096572, 0.025365207516979196, -0.5120723831682655,
      0.007072708326863787, 0.0005495840899003681, 0.004454376620124722, 0.019463982495425563,
      0.0032528904540912065, 0.0038492522992967324, 0.008810547196716409, 0.1081563982143966,
      0.007027352341146176, 0.003290549168241193, 0.0018969251043713395, 0.005387033793049132,  },
00299         { 0.028819585589920302, 0.13788084264397393, 0.19226041997000842, 0.04995859675116145,
      0.008416997510869934, 0.1993175369360554, 0.03082104626814 81, 0.018140365794560007,
      -1.1104356813772154, 0.00687418895767167, 0.03686572559034427, 0.04575147416594887,
      0.01031484574096407, 0.02930954740908149, 0.02276214962148 01, 0.06153836062795258,
      0.03162308553515779, 0.007317323646211892, 0.18410480512319857, 0.008358783494506673,  },
00300         { 0.012032721881462191, 0.007215700516154446, 0.00816511508275157, 0.0005760428590981431,
      0.00421317017111684, 0.0030167372498708 07, 0.0032185769427258487, 0.0005069662783373941,
      0.002472328042678221, -1.4372501178721084, 0.4171548210309419, 0.010437425772882753,
      0.09962573216498272, 0.04781067313240412, 0.003501700565625465, 0.00398471595097322,
      0.05595095491068815, 0.013684731336462 36, 0.008066693173681614, 0.7479315690120867,  },
00301         { 0.03174918354436106, 0.0171511742517201, 0.00291752259634 28337, 0.0008123874783689061,
      0.007805495400682415, 0.02411837000505121, 0.005066032109579534, 0.0025776949391719014,
      0.008317773292277491, 0.26169590749811017, -0.8062356289837133, 0.009022160469804795,
      0.14715281200107683, 0.11140050501605434, 0.01114106287444 8184, 0.011330815327276427,
      0.016396361631731245, 0.007594033178823971, 0.01039539519147503, 0.1195909421773568,  },
00302         { 0.0430873114756613, 0.35945070845278493, 0.0914597094117511, 0.015247568883774697,
      0.00017432067633117612, 0.13392559347233715, 0.13140264822375403, 0.017275640314864015,
      0.01583241803374283, 0.010042710120202372, 0.01383784336700818 5, -0.9917970816959694,
      0.015306661151214025, 0.0010127687545984632, 0.01746005758965857 3, 0.04653754141365703,
      0.06153252460498217, 0.0006116337113357429, 0.00457698792717349 5, 0.013007515319751833,  },
00303         { 0.09027030995147402, 0.027508711152659637, 0.015818500786730045, 0.0013766831584882471,
      0.011743321424969507, 0.06925771001289108, 0.012632541853484138, 0.008126485990424273,
      0.010046986037177108, 0.2698111905441519, 0.6352685184034985, 0.043083539295386955,
      -1.5662058090195037, 0.07729153044389553, 0.004466489950015163, 0.021566758386236153,
      0.10935198049903058, 0.008532122369838514, 0.016697478635025367, 0.13335495012412696,  },
00304         { 0.020374514943928716, 0.0029956938886432478, 0.00381707820652070 35, 0.0009384810508936634,
      0.014523451065555923, 0.0014846832582166 77, 0.0013677770443830553, 0.005217355658947017 5,
      0.015488982372701448, 0.07025124130988708, 0.2609255694174195, 0.0015693966117584803,
      0.04193461101644949, -0.8233635354517814, 0.004225605731036188 5, 0.022490382039859 29,
      0.008930652235446718, 0.030113774624914682, 0.2707331446415206, 0.04598114033369558,  },
00305         { 0.09457616603100734, 0.018894713324081 13, 0.006898113731341692, 0.021255712070010962,
      0.000990550371113879, 0.02585075582220394, 0.030495880198163457, 0.011470716271982939,
      0.01155422013597156, 0.0049422162138415 53, 0.025065114683542243, 0.025611256137419 25,
      0.002327666004604859, 0.004058845904502562, -0.42319902976049834, 0.08317722369405971,
      0.030930612370144 82, 0.0011658984208928898, 0.0031088562222796 47, 0.020824512153333863,  },
00306         { 0.37963602942704533, 0.04876062566353879, 0.17090439503750812, 0.06683363489877918,
      0.03658918198321126, 0.0506762499661637, 0.04449750790400465, 0.10133443476671826,
      0.022479697564225038, 0.004047224363361642, 0.018345156027940516, 0.04912536848738081,
      0.008088283277325782, 0.01554631993480281, 0.05985791675223278, -1.450887708044806,
      0.35031104611356817, 0.003049981739078181, 0.013895785580947473, 0.0069088685569738345,  },
00307         { 0.17182153439304704, 0.03289836913440098, 0.08530296798820032, 0.02294795247666373,
      0.015025795791585504, 0.04472625396171046, 0.04395740648006773, 0.007561455912029168,
      0.013266539252321435, 0.06526438914526325, 0.030487133950749024, 0.07459607576860863,
      0.04709849127241636, 0.007089617558951848, 0.025563161160905623, 0.40231172873331256,
      -1.253857044124528, 0.0017259110607713705, 0.008528721530820873, 0.1536835385527021,  },
00308         { 0.01451323099681107, 0.03372908581153698, 0.00193646969081159837, 0.001610656787506408,
      0.008805756534634281, 0.009780524359434413, 0.005660739910653852, 0.015636517293174643,
      0.013557000672225, 0.007049576045752452, 0.06235905862680738, 0.003274617748407 84,
      0.016229134801107555, 0.10557540976157308, 0.00425544227217991 65, 0.015469064519009399,
      0.007622130175312781, -0.4497155647482195, 0.10934283724709949, 0.013310084276876978,  },
00309         { 0.017584414005493425, 0.017866658820700713, 0.02609491526775575, 0.0072803950013258035,
      0.015315568108369393, 0.010655764912465397, 0.008728076820828707, 0.003184423794739847 3,
      0.12049957307946432, 0.014680177063285846, 0.030156233376271047, 0.008656812182562 08,
      0.011220138549332957, 0.3353111838567005, 0.004008608637950393, 0.02489768380024133,
      0.013306104061275123, 0.03862774628088135, -0.7255847780856934, 0.017510300466049 44,  },
00310         { 0.20461730694868233, 0.009709896087943908, 0.00356820598656581 52, 0.0020458951572852 39,
      0.025745886646282707, 0.008709424043152421, 0.017816802950048248, 0.004466952385382708,
      0.0027023674927284867, 0.672323926346541, 0.17136231711968256, 0.012152161187845727,
      0.04426265001082964, 0.028129842197000456, 0.013263214821074284, 0.006114539012265575,
      0.11843369515463918, 0.0023225805441276933, 0.008649172233327819, -1.3563968363254058,  },
00311             }, new double[20] {
00312 0.079066, 0.055941, 0.041977, 0.053052, 0.012937, 0.040767, 0.071586, 0.057337, 0.022355, 0.062157,
      0.099081, 0.0646, 0.022951, 0.042302, 0.04404, 0.061197, 0.053287, 0.012066, 0.034155, 0.069147,  });
00313
00314 /// <summary>
00315 /// mtArt protein sequence evolution matrix.
```

```
00316 /// </summary>
00317 /// <remarks>Citation:  Abascal, F., D. Posada, and R. Zardoya.  2007.  MtArt:  A new Model of amino
      acid replacement for Arthropoda.Mol.Biol.Evol.  24:1-5.</remarks>
00318             public static readonly ImmutableRateMatrix mtArtMatrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00319                 new double[20, 20] {
00320         { -1.134670547403153, 3.370446316512181E-05, 7.378664583737619E-05, 0.00018639435381819712,
      0.022800818771334448, 3.4728892450987694E-05, 4.4914012498910615E-05, 0.12608061732525924,
      4.5337467925739656E-05, 0.02226920654138213, 0.005497819811489196, 4.0159846986345429E-05,
      0.06874968051825148, 0.01065743002661581, 0.018948927673472008, 0.5664242417108847,
      0.11097986395747535, 5.507879189314304E-05, 0.0003646801834152355, 0.18138315640899758,  },
00321         { 0.00010006861955580907, -0.09804006835306175, 7.378664583737619E-05, 0.0007455774152727885,
      0.003231612109322992, 0.026741247187260526, 4.4914012498910615E-05, 0.00012608061732525926,
      0.00929418092477663, 0.0017130158877986252, 0.002748909905744598, 0.041967040100730824,
      0.002840895889183945, 0.004099011548698388, 7.734256193253882E-05, 0.0025249223256057266,
      9.096710160448799E-05, 5.507879189314304E-05, 0.001458720733660942, 0.00010669597435823387,  },
00322         { 0.00010006861955580907, 3.370446316512181E-05, -1.0197695620084046, 0.09319717690909855,
      0.00879716629760148, 0.04549484911079388, 0.041096321436503214, 0.07627877348178184,
      0.04080372113316568, 0.017986666821885565, 0.017867914387339884, 0.09377324271311625,
      0.044886155049106334, 0.01639604619479355, 0.006574117764265799, 0.3349730285303597,
      0.07550269433172503, 0.002203151675725722, 0.09153472603722412, 0.012270037051196893,  },
00323         { 0.00050034309777904554, 0.0006740892633024363, 0.18446661459344044, -0.43333720116102487,
      0.0009874370334042478, 3.4728892450987694E-05, 0.19357939387030473, 0.0075648370395155545,
      4.5337467925739656E-05, 0.005995555607295189, 0.001374454952872299, 0.00040159846986345286,
      0.00011363583556735781, 0.00016396046194793552, 7.734256193253882E-05, 0.037032194108888399,
      9.096710160448799E-05, 5.507879189314304E-05, 7.293603668304711E-05, 0.00010669597435823387,  },
00324         { 0.12708714683587752, 0.006066803369721926, 0.036155456460314433, 0.0020503378920001683,
      -1.5771152263270305, 3.4728892450987694E-05, 4.4914012498910615E-05, 0.05106265001673,
      0.002702480750544379, 0.053960000465656695, 0.10858194127691162, 4.015984698634529E-05,
      0.17727190348507815, 0.1508436249921007, 7.734256193253882E-05, 0.5588494747340674,
      0.0832348979681065, 0.006058667108245735, 0.02625697320589696, 0.18671795512690925,  },
00325         { 0.00010006861955580907, 0.025952436637143795, 0.0966605060469628, 3.727887076363942E-05,
      1.7953400607349960E-05, -0.5070219107370064, 0.0588373563735729, 0.0018912092598788886,
      0.07117982464341126, 0.009421587382892439, 0.021991279245956785, 0.07007893299117253,
      0.038068004915064487, 0.00016396046194793552, 0.01508179957684507, 0.043765320310499255,
      0.020012762352987355, 0.0019277577162600066, 0.031727175957125486, 0.00010669597435823387,  },
00326         { 0.00010006861955580907, 3.370446316512181E-05, 0.0675147809411992, 0.1606719329912859,
      1.7953400607349960E-05, 0.045494849110793388, -0.39743610272595964, 0.027737735811557033,
      0.0034003100944304474, 0.005995555607295189, 0.002748909905744598, 0.021284718902763,
      0.00011363583556735781, 0.00016396046194793552, 0.0030937024773015527, 0.026090864031259174,
      0.019557926844964917, 0.0030293335541228673, 0.0029174414673218884, 0.007468718205076371,  },
00327         { 0.10006861955580908, 3.370446316512181E-05, 0.044640920731612584, 0.0022367322458183656,
      0.0072711272245976733, 0.0005209333867648154, 0.009881082749760335, -0.39713290679119967,
      4.5337467925739656E-05, 0.0025695238316979377, 0.001374454952872299, 4.015984698634529E-05,
      0.03181803395886019, 0.0008198023097396776, 7.734256193253882E-05, 0.19021081519563l4,
      9.096710160448799E-05, 0.0005507879189314305, 0.00328212165073712, 0.001600439615373508,  },
00328         { 0.00010006861955580907, 0.006909414948849972, 0.06640798125363856, 3.727887076363942E-05,
      0.0010772040364409974, 0.05452436114805068, 0.003368550937418296, 0.00012608061732525926,
      -0.24070232554483748, 0.0001713015887798625l, 0.0082467291723379l, 4.0159846986345l9E-05,
      0.00011363583556735781, 0.011477232336355487, 0.0003867128096626941, 0.009258048527220997,
      0.008641874652426359, 5.507879189314304E-05, 0.06965391503230998, 0.0001066959743582338l,  },
00329         { 0.01300892054225518, 0.00033704463165121814, 0.007747597812924499, 0.0013047604767273797,
      0.005655321191315237, 0.0019100890848043233, 0.0015719904374618713, 0.0018912092598788886,
      4.5337467925739656E-05, -2.2240595932646205, 0.707844300729234, 0.0006023977047951793,
      0.292612276585946l5, 0.09673667254928195, 7.734256193253882E-05, 0.005891485426413362,
      0.092786443636577l5, 5.507879189314304E-05, 0.004376162200982826, 0.9896051621726191,  },
00330         { 0.0020013723911161815, 0.00033704463165121814, 0.004796131979429452, 0.00018639435381819712,
      0.007091593239903233, 0.002778311396079015l, 0.00044914012498910614, 0.0006304030866262962,
      0.0013601240377721896, 0.441101591108146, -1.27161033335833, 0.0008031969397269057,
      0.5028385723855583, 0.2156080074615352, 0.004640553715952329, 0.00673312620161527,
      0.021832104385077114, 0.005783273148780l2, 0.007293603668304711, 0.045345789102249394,  },
00331         { 0.00010006861955580907, 0.03522116400755229, 0.17229181803027338, 0.00037278870763639423,
      1.7953400607349960E-05, 0.060601917326973l3, 0.023804426624422625, 0.00012608061732525926,
      4.5337467925739656E-05, 0.0025695238316979377, 0.005497819811489196, -0.5904469531358115,
      0.060226992850699636, 0.009017825407136453, 0.006574117764265799, 0.121196271629074l7,
      0.031838485561570797, 0.004406303351451444, 0.042667581459582556, 0.013870476666570401,  },
00332         { 0.060541514831264485, 0.0008426115791280453, 0.029145725105763593, 3.727887076363942E-05,
      0.028007304947465936, 0.011634178971080877, 4.4914012498910615E-05, 0.035302572851072l9,
      4.5337467925739656E-05, 0.441101591108146, 1.2163926332919845, 0.021284718902763, -2.531354626483312,
      0.2639763437361762, 0.0019335640483134704, 0.0942637668226138, 0.13144746181848513,
      0.019552971122065783, 0.025892293022481725, 0.14990784397331855,  },
00333         { 0.00650446027112759, 0.0008426115791280453, 0.007378664583737618, 3.727887076363942E-05,
      0.016517128558761l6, 3.4728892450987694E-05, 4.4914012498910615E-05, 0.0006304030866262962,
      0.0031736227548017754, 0.10106793738011889, 0.36148165260541l6, 0.0022087915842489906,
      0.18295369526344607, -1.0567822790168042, 0.005800692144940411, 0.030299067907268717,
      0.006367697112314158, 0.014871273811148621, 0.28882670526486653, 0.027740953331408202,  },
00334         { 0.024516811791173l2, 3.370446316512181E-05, 0.006271864896176976, 3.727887076363942E-05,
      1.7953400607349960E-05, 0.0067721340279426005, 0.0017965604999564246, 0.00012608061732525926,
      0.0002266873396286982l, 0.00017130158877986254, 0.016493459434467l9, 0.0034135869938393495,
      0.002840895889183945, 0.012297034646095164, -0.19330604876941077, 0.07322274744256607,
      0.021377268877054673, 5.507879189314304E-05, 0.006564243301470l4, 0.017071355897317418,  },
00335         { 0.33673090480529755, 0.0005055669474768272, 0.1468354252163786, 0.008201351568000673,
      0.059605290016401864, 0.0090295120372568, 0.006961671937331145, 0.14247109757754295,
      0.0024935607359156805, 0.005995555607295189, 0.010995639622978393, 0.028915089831016861,
      0.06363606791772038, 0.029512883150628392, 0.033644014440654l8, -1.2297585323065077,
      0.30019143529481035, 0.0005507879189314305, 0.010940405502457066, 0.032542272179261l3,  },
```

```
00336         { 0.12208371585808706, 3.370446316512181E-05, 0.06124291604502224, 3.727887076363942E-05,
      0.01642736155572521, 0.007640356339217293, 0.009656512687265782, 0.0001260806173252592 6,
      0.004307059452945267, 0.1747276205554598, 0.06597383773787036, 0.014055946445220851,
      0.16420378239483205, 0.011477232336355487, 0.01817550205414662, 0.5554829116332598,
      -1.5326952365300517, 5.507879189314304E-05, 0.016775288437100834, 0.2902130502543961,  },
00337         { 0.00010006861955580907, 3.3704463165121 81E-05, 0.002951465833495047, 3.727887076363942E-05,
      0.0019748740668084955, 0.0012155112357845695, 0.0024702706874400837, 0.0012608061732525923,
      4.5337467925739656E-05, 0.00017130158877986254, 0.028863554010318277, 0.003212787758907623,
      0.04034072162641202, 0.04426932472594259, 7.734256193253882E-05, 0.0016832815504038176,
      9.096710160448799E-05, -0.1427631412866295, 0.01385784696977895, 0.00010669597435823387,  },
00338         { 0.0005003430977790454, 0.0006740892633024363, 0.09260224052590711, 3.727887076363942E-05,
      0.006463224218645984, 0.015107068216179647, 0.0017965604999564246, 0.005673627779636666,
      0.04329728186908136, 0.010278095326791751, 0.027489099057445978, 0.023493510487011995,
      0.04034072162641202, 0.6492834293138247, 0.006960830573928494, 0.025249223256057267,
      0.020922433369032236, 0.01046497045969718, -0.9817009875550362, 0.0010669597435823386,  },
00339         { 0.1701166532448754, 3.370446316512181E-05, 0.008485464271298261, 3.727887076363942E-05,
      0.031418451062862424, 3.4728892450987694E-05, 0.0031439808749237426, 0.0018912092598788886,
      4.5337467925739656E-05, 1.588822235933225, 0.11682867099414541, 0.005220780108224888,
      0.1596583489721377, 0.04262972010646324, 0.012374809909206 21, 0.05134008728731644, 0.2474305163642073,
      5.507879189314304E-05, 0.000729360366830471, -2.4402964172417945,  },
00340                 }, new double[20] {
00341 0.054116, 0.018227, 0.039903, 0.02016, 0.009709, 0.018781, 0.024289, 0.068183, 0.024518, 0.092638,
      0.148658, 0.021718, 0.061453, 0.088668, 0.041826, 0.09103, 0.049194, 0.029786, 0.039443, 0.0577,  });
00342
00343 /// <summary>
00344 /// mtmam protein sequence evolution matrix.
00345 /// </summary>
00346 /// <remarks>Citation:  Yang, Z., R. Nielsen, and M. Hasegawa.    1998.    Models of amino acid
      substitution and applications to Mitochondrial protein evolution, Molecular Biology and Evolution
      15:1600-1611.</remarks>
00347             public static readonly ImmutableRateMatrix mtmamMatrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00348                 new double[20, 20] {
00349         { -1.2307900321918033, 0.005840880738946804, 0.0007935979264873374, 0.0020296266969913653, 0,
      0, 0, 0.043098319392711075, 0.002198266256369925, 0.06733182407541004, 0.034893508830240116, 0,
      0.042294801492142646, 0, 0.028180662369565356, 0.24596575734066914, 0.587728704382441,
      0.0014532762028799367, 0, 0.16898080648694874,  },
00350         { 0.0219667906051695, -0.19984680584026565, 0.0015871958529746748, 0, 0.011993248664039885,
      0.058079464249975794, 0, 0.009945766013702556, 0.06374972143472782, 0, 0.009969573951497179,
      0.010961571359606348, 0, 0, 0.004785395496718645, 0.0021575943626374484, 0, 0.004650483849215797, 0,
      0,  },
00351         { 0.0013729244128230938, 0.0007301100923683505, -0.9246596320493131, 0.15941795147277635, 0,
      0.0018887630650398634, 0, 0.02596950014689001, 0.1258507431771782, 0.01705739543243721, 0,
      0.0894464222943878, 0.011686721464934153, 0.003636662498128224, 0.017546450154635032,
      0.3207623619121007, 0.09493415195604774, 0.001743931443455924, 0.052615542526110476, 0,  },
00352         { 0.007551084270527016, 0, 0.3428343042425298, -0.5616838124325348, 0, 0.011568673773369164,
      0.13320937995053203, 0.04365086194902788, 0.0030226161025086 46, 0, 0, 0, 0, 0.00303055208177352,
      0.0010634212214930322, 0.011507169934066393, 0, 0, 0, 0.004245748906707255,  },
00353         { 0, 0.0339501192951283, 0, 0, -0.7492715143033871, 0, 0, 0, 0.08380890102410338,
      0.03680806382789082, 0.0448630827817373, 0, 0, 0.004242772914482928, 0, 0.2495617479450649,
      0.09838630293626766, 0.01889259063743918, 0.17875793294127276, 0,  },
00354         { 0, 0.04490177068065355, 0.0031743917059493497, 0.00904106437750699, 0, -0.4518409314300142,
      0.06414652039797149, 0, 0.1511308051254323, 0, 0.03323191317165726, 0.05305400538049473,
      0.012243232010883398, 0, 0.027117241148072322, 0.021575943626374484, 0, 0, 0.018213072412884397,
      0.014010971392133942,  },
00355         { 0, 0, 0, 0.10498705368982608, 0, 0.06469013497761532, -0.26205992328983235,
      0.012155936238969793, 0.006045232205017292, 0, 0, 0.0471347568463073, 0, 0, 0, 0.01510316053846214,
      0.0034521509802199176, 0, 0, 0.00849149781341451,  },
00356         { 0.05354405210010066, 0.003285495415657577, 0.018649551272452432, 0.014576409914756169, 0, 0,
      0.00515045054290282, -0.17821630235644514, 0, 0, 0, 0, 0, 0, 0.08055018953846474, 0, 0,
      0.00033727911875711846, 0.0021228744533536275,  },
00357         { 0.005491697651292375, 0.042346385357364326, 0.18173392516560027, 0.0020296266969913653,
      0.019666348615764332, 0.12985246072149062, 0.00515045054290282, 0, -0.98725070051135695, 0,
      0.043201487123154435, 0, 0, 0, 0.028180662369565356, 0.014383962417582991, 0.0008630377450549794, 0,
      0.5143506561046056, 0,  },
00358         { 0.05148466548086601, 0, 0.007539180301629706, 0, 0.0026436730926109432, 0, 0, 0, 0,
      -1.954687244793166, 0.38549019279122415, 0.0013153885631527619, 0.21036098636881476,
      0.034548293732218 13, 0.0026585530537325806, 0, 0.3106935882197926, 0, 0.005396465900113895,
      0.9425562572890106,  },
00359         { 0.014415706334642486, 0.0010951651385525257, 0, 0, 0.0017409554512315963,
      0.004721907662599658, 0, 0, 0.007144365333202255, 0.2082797758066017, -0.8860977168038853,
      0.0008769257087685079, 0.3389149224830904, 0.1491031624232572, 0.022863556262100192,
      0.05322066094505706, 0.0293432833318693, 0.003487862886911848, 0.00843197796892796,
      0.04245748906707255,  },
00360         { 0, 0.00912637615460438, 0.16189397700341684, 0, 0, 0.05713508271745587,
      0.050333948487459375, 0, 0, 0.005386545926032802, 0.006646382634331451, -0.4454246921943641,
      0.032834122211100548, 0, 0.00957079099343729, 0.046747877857144716, 0.04315188725274897, 0,
      0.022597700956726936, 0,  },
00361         { 0.052171127687277566, 0, 0.008332778228117043, 0, 0, 0.005194098428859624, 0, 0, 0,
      0.33935239334006656, 1.0119117560769635, 0.012934654204335492, -2.423750243225364,
      0.006667214579901744, 0, 0.03380231168132003, 0.5963590818329908, 0.0037785181274878356, 0,
      0.35324630903804366,  },
00362         { 0, 0, 0.0023807937794620123, 0.0009225575895415298, 0.0004513588206896732, 0, 0, 0, 0,
      0.05117218629731163, 0.4087525320113843, 0, 0.006121616005441699, -0.7830440660624641,
      0.009039080382690775, 0.06472783087912345, 0.006904301960439835, 0, 0.2300243589923548,
      0.002547449344024353,  },
```

```
00363         { 0.036382496939811985, 0.0016427477078287886, 0.013094365787041068, 0.0003690230358166119, 0,
      0.012040864539629128, 0, 0, 0.014563513948455075, 0.004488788271694003, 0.07144861331906031,
      0.003946165689458286, 0, 0.010303877078029967, -0.3856082404827892, 0.14527802041758822,
      0.0673169441142884, 0.00203458668403191115, 0.0026982329500569477, 0,  },
00364         { 0.23477007459274904, 0.0005475825692762629, 0.17697233760667624, 0.002952184286532895,
      0.02237450153990237, 0.0070828614938994875, 0.004916339154589056, 0.061884766307482 58,
      0.005495665640924811, 0, 0.12295807873513186, 0.014250042767488254, 0.026155995659614 53,
      0.05454993747192336, 0.10740554337079625, -1.413251091457548, 0.5299051754637574,
      0.004941139089791785, 0.036088865707011676, 0,  },
00365         { 0.4674807625662634, 0, 0.043647885956803564, 0, 0.007350700794088963, 0,
      0.0009364455532550581, 0, 0.0002747832820462406, 0.32319275556196814, 0.05649425239181734,
      0.010961571359606348, 0.38454878725092856, 0.004848883330837632, 0.04147342763822826,
      0.4415876462197978, -1.8834221509946032, 0, 0, 0.10062424908896195,  },
00366         { 0.0034323110320577345, 0.002920440369473402, 0.0023807937794620123, 0, 0.004191189049261251,
      0, 0, 0, 0, 0, 0.019939147902994357, 0, 0.0072346370973401 89, 0, 0.003721974275225613,
      0.012226368054945543, 0, -0.06076876922335976, 0.004721907662599658, 0,  },
00367         { 0, 0, 0.061900638266012324, 0, 0.034174310709360965, 0.012749150689019077, 0,
      0.0005525425563168088, 0.4190445051205169, 0.014364122469420808, 0.04153989146457157,
      0.014688505621872508, 0, 0.4133673039539081 4, 0.004253684885972129, 0.076954198934069, 0,
      0.004069173368063823, -1.097658028039104, 0,  },
00368         { 0.27321195815179566, 0, 0, 0.0018451151790830596, 0, 0.0077911476432894365,
      0.004682227766275 29, 0.002762712781584043, 0, 1.993021992632137, 0.16615956585828628, 0,
      0.4630167742297721, 0.0036366624981282 24, 0, 0, 0.20453994557803012, 0, 0, -3.120668102318381,  },
00369                 }, new double[20] {
00370 0.0692, 0.0184, 0.04, 0.0186, 0.0065, 0.0238, 0.0236, 0.0557, 0.0277, 0.0905, 0.1675, 0.0221, 0.0561,
      0.0611, 0.0536, 0.0725, 0.087, 0.0293, 0.034, 0.0428,  });
00371
00372 /// <summary>
00373 /// mtREV24 protein sequence evolution matrix.
00374 /// </summary>
00375 /// <remarks>Citation:  Adachi, J. and Hasegawa, M. (1996) MOLPHY version 2.3:  programs for molecular
      phylogenetics based on maximum likelihood.Computer Science Monographs of Institute of Statistical
      Mathematics 28:1-150.</remarks>
00376             public static readonly ImmutableRateMatrix mtREV24Matrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00377                 new double[20, 20] {
00378         { -1.1143824850663313, 0.0044013246586137745, 0.010503638078279842, 0.003355108141402304,
      0.0035934524334597455, 0.00047468988984186527, 0.0023432691656867484, 0.067553467994893 62,
      0.0038894590552937675, 0.08485576457755911, 0.04299930898143552, 0.00192154467407987 03,
      0.07656518073307889, 0.003883163168333759, 0.029308253211259617, 0.27907688186384716,
      0.41314929369792014, 0.0005506402722165637, 0.002137003916711252, 0.083821040552417 51,  },
00379         { 0.016678703969483774, -0.23671605576792737, 0.005160228874078854, 0.000360764316279817 57,
      0.0061957523769296755, 0.05521143092429148, 0.0004557022942481906 4, 0.012888380150131723,
      0.046234195662315763, 0.0016709084122433656, 0.026313009973714274, 0.032500767573551 88,
      0.0010253301620584288, 0.0028590322228391417, 0.012757265805821715, 0.004345960827251165,
      0.0017876321577876388, 0.006361344197449249, 0.0006265906545912622, 0.003283055212859990 5,  },
00380         { 0.019391331836824323, 0.0025139576566025183, -1.3045719361998, 0.15083366187703234,
      0.0035340912135510996, 0.043361672253133754, 0.015122120869657061, 0.029828513330526 3,
      0.13882570655416523, 0.023832430511471162, 0.025603673376220052, 0.139909598458423 13,
      0.0352983399474957, 0.00926594664971321, 0.03956155483184391, 0.3557284061895204, 0.204941713627904 03,
      0.003095177951196263, 0.06310757245399154, 0.0008164666105280082,  },
00381         { 0.012714094009524519, 0.00036076431627981757, 0.30960593753706644, -0.6463999134005746,
      0.00011392557356204766, 0.013810977426557007, 0.13996056516238506, 0.031770444686190 96,
      0.03189636242539112, 0.0038167065837558982, 0.003208903655331009, 0.000530953133627 3326,
      0.0010253301620584288, 0.0030358167312876 17, 0.007247465303391948, 0.049661956340542 29,
      0.024072873432515 27, 0.005755639898011029, 0.006994730412568774, 0.0008164666105280082,  },
00382         { 0.04312142920151694, 0.019619882526943973, 0.022971592888082145, 0.000360764316279817 57,
      -0.7729845169449252, 0.018797719637737863, 0.00045570229424819064, 0.017186372314830 45,
      0.03959133537651188, 0.055166360368434905, 0.043320199346968624, 0.000436714698654 51596,
      0.0033350212639584684, 0.04315980413155889, 0.01686937940312973, 0.199345769402307 17,
      0.15467315357550065, 0.009737638498145547, 0.08401921108958729, 0.0008164666105280082,  },
00383         { 0.0013671068827445718, 0.04196068750246152, 0.06764420871488866, 0.010496342844183323,
      0.0045114527130570865, -0.8026638277479705, 0.07520526914971719, 0.003777532176004738 2,
      0.1629655362448266, 0.0073344085042682 46, 0.0670491974298111, 0.107013489157668 19,
      0.025563099882477774, 0.01164948950500128, 0.07408819892052 72, 0.038933766960688 83,
      0.08158650035518297, 0.0005506402722165637, 0.012802236426964 63, 0.008164666105280008,  },
00384         { 0.007029807497060245, 0.00036076431627981757, 0.024573446413192725, 0.110802114086888 17,
      0.00011392557356204766, 0.07833882203095541, -0.4046911193864726, 0.015826460736146874,
      0.013744620776692794, 0.0029108983392239682, 0.003208903655331009, 0.072140671220898 1,
      0.0010253301620584288, 0.0016276366812325177, 0.006923676831162 97, 0.039365482923660 8,
      0.012736879124236928, 0.0005506402722165637, 0.004326773362230189, 0.009084265340295838,  },
00385         { 0.0868544588505775, 0.0043728432652232 63, 0.020773428926616534, 0.010779258018529076,
      0.0018413970337318336, 0.001686398292859258, 0.006782768884915174, -0.256905666097395 24,
      0.0005316526766228891, 0.005258964371165961, 0.0040702409522882805, 0.005224486894956394,
      0.0010253301620584288, 0.0011582433312141511, 0.0010253301620584288, 0.090610405128433 64,
      0.009599928462734 58, 0.003164732511897303, 0.0010586084217041 85, 0.001087189749808347 7,  },
00386         { 0.010001466142183972, 0.03137320419942855, 0.19336437698615874, 0.021643960217229686,
      0.008483857580681117, 0.14550494307573805, 0.011781103522879538, 0.001063305353245778 2,
      -0.8695592256534729, 0.010781756386370347, 0.019405422631449103, 0.029344929251169504,
      0.006459580020968103, 0.029358420437512 38, 0.032902305255001 27, 0.055734789019681326,
      0.038485657704678 11, 0.0020518595406806687, 0.221001821719888 64, 0.0008164666105280082,  },
00387         { 0.06942744374527565, 0.00036076431627981757, 0.010562099885765628, 0.000824061648765477 9,
      0.003761342752393289, 0.002083638779621661, 0.0007938813652429005, 0.003346613690741975 4,
      0.003430558850208747, -1.8824460168416577, 0.5557990020699378, 0.004498161396141515,
      0.2795265880752763, 0.0516149804494222, 0.011132926970139678, 0.034321578056271625,
      0.31664294033350954, 0.0005506402722165637, 0.0082479117217513, 0.5255208824626959,  },
```

```
00388         { 0.0183192322287772 62, 0.002958267393494504, 0.005908540009896934, 0.00036076431627981757,
      0.0015379952430876432, 0.009918520329853712, 0.00045570229424819064, 0.001348718895432803,
      0.003215099607577366, 0.2894101312553522, -1.0052316921878106, 0.0034201656399890515,
      0.2900766957954038, 0.13171055481164712, 0.021639862893969998, 0.05296459875727785,
      0.10863303112709499, 0.009401458121423855, 0.014559988105370642, 0.0393923653616329,  },
00389         { 0.006015270284076116, 0.026848460169455895, 0.23723801477732614, 0.00043861345821388347,
      0.00011392557356204766, 0.11631900995398717, 0.07527722214354586, 0.012720489831198178,
      0.03572426169707592, 0.017210356646106664, 0.02513078231122391, -0.8512394937511274,
      0.049307587845936134, 0.0039258352910627025, 0.027036337431119627, 0.07611907217134119,
      0.11716725580345616, 0.006955456070103963, 0.016875075681807836, 0.0008164666105280082,  },
00390         { 0.10208690764410518, 0.00036076431627981757, 0.025493245517635783, 0.00036076431627981757,
      0.00037055791821760767, 0.011834768464110083, 0.00045570229424819064, 0.0010633053532457782,
      0.003349411862724201, 0.45552481019674657, 0.9078326220263564, 0.02100138000845428,
      -2.3151812152638804, 0.055364031232036 42, 0.010166958027989895, 0.07998294793994032,
      0.4539296522974506, 0.0062917896367482105, 0.013178190819719387, 0.166533405391591 73,  },
00391         { 0.004583405706885749, 0.0008905182333433393, 0.0059241298252264785, 0.0009455822605649956,
      0.004245226635891039, 0.004774380944672655, 0.0006403816450750889, 0.0010633053532457782,
      0.013475996266399124, 0.07446095540244513, 0.3649030125109568, 0.0014802329785974123,
      0.0490107817463929, -0.7696317073019369, 0.009341297423806, 0.046258579732446 59, 0.029091994490919027,
      0.0022721156495672944, 0.15354109313926306, 0.002728717356238343,  },
00392         { 0.03907767094834616, 0.004488667598344678, 0.0285722340452206, 0.0025500340882305002,
      0.0018743754892366368, 0.03430009209283667, 0.003077189702739098, 0.0010633053532457782,
      0.01706045457563029, 0.018142547655042435, 0.0677247560940913, 0.011515477053995396,
      0.010166958027989895, 0.010552206349114187, -0.49271381499180406, 0.12224813651489619,
      0.11019720926515916, 0.00122010291896407, 0.00534580763732861, 0.0035365895813923723,  },
00393         { 0.27907688186384716, 0.0011468507738579465, 0.19268622001932356, 0.013105238478754215,
      0.016612147450192266, 0.013518668389128 07, 0.013121827641220269, 0.07047475954433728,
      0.021674640174320515, 0.04194859540210976, 0.1243196831941661, 0.024315814721400 66,
      0.05998721095495524, 0.039191296717767256, 0.09168610238617214, -1.5495382021878925,
      0.5132652889194018, 0.011180895632692119, 0.021409613313718 28, 0.0008164666105280082,  },
00394         { 0.34589243193314245, 0.00039494198834843185, 0.09293868408707276, 0.005318425525788258,
      0.010791150249453535, 0.023717005917204353, 0.0035544778951358868, 0.006251116208292285,
      0.012530214136406828, 0.3240067296435911, 0.21347653791254714, 0.03133542887766851,
      0.2850255956286318, 0.020635019348210007, 0.0691935965153325, 0.4297104744441504, -1.978334373936851,
      0.0028952085891807743, 0.012772555817010306, 0.08789477921968357,  },
00395         { 0.0013671068827445718, 0.0041677772328115766, 0.004162480692988078, 0.0037709364849037768,
      0.002014683827202527, 0.00047468988984186527, 0.0004557022942481 9064, 0.0061112076091809995,
      0.0019811057634158183, 0.0016709084122433656, 0.054787807673125 22, 0.0055163961935307294,
      0.011715746220151837, 0.004779277745641 55, 0.0022719157801399923, 0.027759465019097672,
      0.008585790988605053, -0.15255743575279707, 0.008656844570010858, 0.002307592472913 37,  },
00396         { 0.0046625540000972765, 0.00036076431627981757, 0.07458167653653545, 0.004027269025418384,
      0.015276220198106781, 0.009698663959821689, 0.0031467442634401373, 0.00179642641258892,
      0.1875166972168752, 0.02199443125800346, 0.07456478756992845, 0.011761416384290308,
      0.021564312250449906, 0.2838183842877 2865, 0.008747685224719543, 0.046711883593567 16,
      0.03328605455342079, 0.00760753007667621, -0.8119399677384762, 0.0008164666105280082,  },
00397         { 0.14035150976218747, 0.00145065230335674, 0.0007405162281533098, 0.00036076431627981757,
      0.00011392557356204766, 0.004746898898418653, 0.0050702876317930265, 0.0014158750230062204,
      0.0005316526766228891, 1.0754845966678428, 0.1548211568864177, 0.000436711469865415596,
      0.2091349742126966, 0.0038709711332683473, 0.004441298544074142, 0.0013671068827445718,
      0.17578955843936714, 0.0015562832956857616, 0.0006265906545912622, -1.7823113338287229,  },
00398                 }, new double[20] {
00399 0.072, 0.019, 0.039, 0.019, 0.006, 0.025, 0.024, 0.056, 0.028, 0.088, 0.169, 0.023, 0.054, 0.061,
      0.054, 0.072, 0.086, 0.029, 0.033, 0.043,  });
00400
00401 /// <summary>
00402 /// MTZoa protein sequence evolution matrix.
00403 /// </summary>
00404 /// <remarks>Citation:  Rota-Stabelli, O., Z. Yang, and M. Telford.  2009.  MtZoa:  a general
      mitochondrial amino acid substitutions model for animal evolutionary studies.  Mol.  Phyl.
      Evol.</remarks>
00405             public static readonly ImmutableRateMatrix MtZoaMatrix = new ImmutableRateMatrix(new
      char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
      'Y', 'V' },
00406                 new double[20, 20] {
00407         { -1.3402360111467704, 0.0006819733423697274, 0.0005075154567039491, 0.003273270856518464,
      0.026678231787002286, 0.0013354946587779033, 0.004197731337389084, 0.20453023598839784,
      0.0006058416032545011, 0.027745616471483454, 0.024016535573254 66, 3.7871853912569246E-05,
      0.06792019148365738, 0.021271659203919963, 0.027199769417465006, 0.5102073279990578,
      0.20943420686797376, 0.0008526255743195561, 0.0007716262528185448, 0.2089682854184940 8,  },
00408         { 0.002232938338281448, -0.17538894562565346, 0.010030893732501583, 0.0006505879963266512,
      0.0059817980263700546, 0.04226017344899 94, 0.0015710807344614117, 0.005090278946133648,
      0.016252359530783793, 0.0001671422679005 0273, 0.006312038195534879, 0.05534971449321995,
      0.0018908185174937584, 0.000160540824180 52806, 0.003300942890469 0542, 0.009335472895625497,
      0.0017725190742594974, 0.004648184582580804, 0.005070686804236152, 0.0033107743262948738,  },
00409         { 0.0011503015682055944, 0.006943728576855407, -1.0116367447720385, 0.12544149804173244,
      0.009261507255230885, 0.03481433336512808, 0.04271375746816963, 0.07020343046542657,
      0.060294409993458836, 0.020307785549911 08, 0.012162219937737938, 0.07828112203728063,
      0.03667192756297106, 0.010354883159644057, 0.009946841243280085, 0.33243777209676556,
      0.10225219409634474, 0.0017602592502081156, 0.052029655904336156, 0.004609117199351687,  },
00410         { 0.010894032498888278, 0.0006613074835100387, 0.1841982569331392, -0.503506159255173,
      0.0009300667962441165, 0.00353082834444021, 0.21690733390157868, 0.018381562861038175,
      0.014645562235195766, 0.0012535670092537704, 0.0007697607555530341, 3.7871853912569246E-05,
      9.951676407861888E-05, 0.0016054082418052802, 8.802514374584145E-05, 0.04066468701992801,
      0.0012739980846240137, 5.5008101569003614E-05, 0.005107430911513226, 0.002401934315155105,  },
00411         { 0.18438657490354382, 0.012626839763269802, 0.028241742 47305505, 0.0019314331140947456,
      -1.5337206152318523, 0.008982573663834939, 0.0008346366401826249, 0.05825541460575176,
      0.009877852226975563, 0.04078271336772266, 0.09190943421303227, 3.7871853912569246E-05,
```

```
      0.1317601956400914, 0.13469375148746301, 0.003565018321706579, 0.5192263441863569,
      0.11022852993051248, 0.009928962333205152, 0.028182730281515425, 0.15826799622562557,  },
00412         { 0.004939530263471082, 0.04773813396588092, 0.05681187730044796, 0.0039238588528451145,
      0.004806976810061697, -0.6173871314482275, 0.08577118884700269, 0.00629215036397076,
      0.11110608184902114, 0.00016714226790050273, 0.035408994755439564, 0.06324599603399064,
      0.04174728253098062, 0.007625689148575082, 0.02297456251766462, 0.07626606670663541,
      0.02182414110182006, 0.00167774709785461, 0.019217168105909472, 0.00584254292875566,  },
00413         { 0.011570680480185686, 0.0013226149670200775, 0.05194569968616892, 0.17964361048569658,
      0.000332866011287368, 0.06392079914753415, -0.4557107027529902, 0.03047097535810559,
      0.00392479995151829, 0.006100692778368349, 0.0015395215111060681, 0.0309034327926565,
      9.951676407861888E-05, 0.00016054082418052806, 0.009066589805821671, 0.030379843999323648,
      0.019109971269360204, 0.0009626417774575633, 0.003674410727707356, 0.01058149441541303,  },
00414         { 0.1957542609893403, 0.0014879418378975873, 0.0296448734415895, 0.005286027470154041,
      0.00806710568531739, 0.0016282058168666211, 0.010580246821138568, -0.5223731374718185,
      0.0019492295061231777, 0.002841418554308546, 0.005388325288871239, 0.001912528622584747,
      0.025973875424519528, 0.004655683901235313, 0.0005721634343479695, 0.2034025492767216,
      0.002880343495671683, 0.003382998246493722, 0.001579996612914163, 0.015385363045723237,  },
00415         { 0.0015562903569840394, 0.012750834916427934, 0.06833546355266704, 0.011303966436175565,
      0.0036713163009636177, 0.07716597905103008, 0.003657672334917974, 0.00523167558352625,
      -0.3489878469157433, 0.0013371381432040219, 0.010160841973300049, 0.004525686542552024,
      0.00353284512479097, 0.010515423983824585, 0.006865961212175634, 0.018275374905843134,
      0.010745896887698203, 0.0012376822853025812, 0.09792304589340105, 0.000194751430958522,  },
00416         { 0.022464712979073962, 4.133171771937742E-05, 0.007254485645827038, 0.0003049631232781177 5,
      0.004777606279653988, 3.6588894761038444E-05, 0.0017920139627450476, 0.0024037428356742225,
      0.0004214550283509574, -2.2398689533654137, 0.6546045465223002, 0.0015906178643279084,
      0.22376344403077453, 0.0724841821175084, 0.001144326868695939, 0.0056962207498731845,
      0.12313468443996445, 0.002667892926096675, 0.004813478053296637, 1.1104726593254923,  },
00417         { 0.010555708508239572, 0.000847300213247237, 0.002358454181153646, 0.00010165437442603925,
      0.0058447355511340796, 0.004207722897519421, 0.00024548136475959555, 0.0024744411543705233,
      0.001738501991947699, 0.35534446155646876, -1.025767491650488, 0.0012687071060710699,
      0.3166125849161259, 0.1879933051153983, 0.005017433193512963, 0.012025354916398943,
      0.027695610535304647, 0.007481101813384491, 0.002094414114793193, 0.08186051814623208,  },
00418         { 0.00013532959625945 17, 0.060406305446870104, 0.1234158175302427, 4.066174977041 57E-05,
      1.95803536051393E-05, 0.061103454250934205, 0.040062558728766, 0.0071405301883263675,
      0.006295484485992425, 0.007019975251821115, 0.010314794124410658, -0.5630458140068918,
      0.04129945709262683, 0.013084077170713035, 0.010695054965119737, 0.11463644259119783,
      0.04182037190831001, 0.001815267351777119, 0.0165348482746831, 0.007205802945465314,  },
00419         { 0.09236244944709626, 0.000785302636668171, 0.022002287740635916, 4.066174977041 57E-05,
      0.025924388173204426, 0.015349041352255626, 4.909627295191911 6E-05, 0.036904522359468 95,
      0.0018702066883073732, 0.3758193893742804, 0.9795975375167911, 0.01571681937371624,
      -2.1963706830761636, 0.1730630084666092, 0.00237667888113771 93, 0.07539581075873811,
      0.16899861548642894, 0.013394472732052382, 0.015212060412708454, 0.1815083336533425,  },
00420         { 0.017931171504381324, 4.133171771937742E-05, 0.003851146700871144, 0.000406617497704157,
      0.01642791667471187, 0.001737972501149326, 4.909627295191911 6E-05, 0.004100502484385439,
      0.003450663044623463, 0.07546473395707698, 0.3605559379010412, 0.003086556093874394,
      0.10727907167675113, -0.9068412374055845, 0.004621320046656676, 0.025474765020266188,
      0.010690505666627593, 0.01600735755658005, 0.2169739534711194, 0.03869061761709304,  },
00421         { 0.041816845244179 85, 0.001549939414476 6535, 0.006746970189123089, 4.066174977041 57E-05,
      0.0007930043210081414, 0.009549701532631034, 0.005056916114047669, 0.0009190781430519088,
      0.004109186526421834, 0.002172849482706 5355, 0.017550545226609177, 0.004601430250377 1635,
      0.002686952630122 7095, 0.008428393269477 72, -0.21411505189688756, 0.0630539991340129,
      0.031517604789176 68, 0.000357552660198 5235, 0.001543225056370897, 0.011620168713858478,  },
00422         { 0.43637028313869874, 0.0024385713454432682, 0.1254458793570585, 0.010450069690996835,
      0.0642529303552646, 0.01763584727482053, 0.009426484406768 47, 0.18176537736818904,
      0.0060847569718169466, 0.006017121644418098, 0.023400726968812235, 0.02743815815965642,
      0.04741973808346189, 0.025847072693065014, 0.035078019782717 82, -1.4340226464349088,
      0.36907170599346967, 0.0028329172308036864, 0.010912999861290847, 0.032133986108156126,  },
00423         { 0.25584060172855017, 0.0006613074835100387, 0.055110207827970006, 0.0004676101223597805,
      0.0194824518371136, 0.007208012267924573, 0.008469107084206047, 0.003676312572207635,
      0.005110142218755357, 0.18577863077140877, 0.0769760755553034, 0.014296624851994889,
      0.1518128236019331, 0.015492189533420957, 0.025043153395691894, 0.5271377618945142,
      -1.6215405071253182, 0.000990145828242065, 0.010655791110351334, 0.25733155743986036,  },
00424         { 0.0020976087420219664, 0.003492530147287392, 0.0019106464252383968, 4.066174977041 57E-05,
      0.0035342538257276427, 0.0011159612902116726, 0.0008591847766585845, 0.008695893199644983,
      0.0011853422672370676, 0.008106399993174382, 0.041874985102085045, 0.0012497711791147849,
      0.024232332053143697, 0.046717379836533 66, 0.0005721634343479695, 0.008148760239401916,
      0.0019940839585419344, -0.19404340450768542, 0.0293217976071047, 0.00889364868043917,  },
00425         { 0.001420960760724558, 0.0028518885226370425, 0.04227305215839953, 0.002825991609043891,
      0.0075090656075709195, 0.009567995980011552, 0.0024548136475959557, 0.003040027703940929,
      0.07019860315970633, 0.010947818547482929, 0.008775272613304588, 0.00852116713032808,
      0.020599970164274107, 0.473996783393009, 0.0018485280186626707, 0.023496910593226887,
      0.016063454110476695, 0.02194823252603244, -0.7384676106562712, 0.010127074409843144,  },
00426         { 0.2178129851796358, 0.0010539588018441 24, 0.002119623377998846 4, 0.000752242370752690 5,
      0.023868451044664802, 0.00164650026424673, 0.004001346245581408, 0.01675550153102326,
      7.902281781580 45E-05, 1.4295678173529995, 0.19413366255047518, 0.002101887892147593,
      0.1391244361819092, 0.04784116560579735, 0.00787825036525281, 0.03916151765537814, 0.2195708003238952,
      0.0037680549574767475, 0.005732080735223 4755, -2.3569693052541187,  },
00427                 }, new double[20] {
00428 0.06888, 0.021037, 0.03039, 0.020696, 0.009966, 0.018623, 0.024989, 0.071968, 0.026814, 0.085072,
      0.156717, 0.019276, 0.050652, 0.081712, 0.044803, 0.080535, 0.056386, 0.027998, 0.037404, 0.066083,
      });
00429
00430 /// <summary>
00431 /// WAG protein sequence evolution matrix.
00432 /// </summary>
00433 /// <remarks>Citation:  Whelan, S. and N. Goldman.  2001.  A general empirical model of protein
      evolution derived from multiple protein families using a maximum likelihood approach.Molecular Biology
```

```
    and Evolution 18, 691-699.</remarks>
00434           public static readonly ImmutableRateMatrix WAGMatrix = new ImmutableRateMatrix(new
    char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
    'Y', 'V' },
00435               new double[20, 20] {
00436         { -1.1354111867880292, 0.025870671727558636, 0.021258361408223507, 0.044966767685612 26,
    0.021151940281163652, 0.03559592643170215, 0.09802540056627916, 0.12580784375147247,
    0.008259863595157933, 0.00999488318102409, 0.03659089059741105, 0.05996215721773844,
    0.01858734734128384, 0.0086290232376018, 0.07022155771088084, 0.24995302845876177, 0.1380427166515966,
    0.0017360267088627171, 0.009057878082649915, 0.15169890215304813,  },
00437         { 0.05096702363658184, -0.9902438842087685, 0.026491061820913632, 0.008963197122538123,
    0.010878120120976895, 0.11892105714896123, 0.02719685430488389, 0.051919534535373714,
    0.05569441459136586, 0.009666295612820768, 0.0457641082002542, 0.3540715876461629,
    0.01421178089702265, 0.004210550446840853, 0.03316866012819069, 0.09077693891014615,
    0.03608142749171973, 0.017860360875955678, 0.014355533671911 73, 0.019045377046147838,  },
00438         { 0.04711167749403789, 0.029800021754982527, -1.4761789236926435, 0.3303709452632036,
    0.005462960616159395, 0.060474814909379 84, 0.058659672973167695, 0.09995219705581013,
    0.10310144608645852, 0.028652463727302693, 0.012094861109775404, 0.19928676177652271,
    0.004123580382381817, 0.003942083838382985, 0.009522708073960827, 0.29469970668349693,
    0.13211714496925678, 0.0011035622852153429, 0.040861759186482265, 0.014840555506666011,  },
00439         { 0.06828591157509498, 0.006909089542699483, 0.226382319038295 53, -1.0058412779653094,
    0.0006239249840549781, 0.024163560003791055, 0.3823637787284317, 0.07686575796612918,
    0.024253540927980724, 0.0020387835001942 07, 0.007798347636671813, 0.031749147271182 14,
    0.0021583886621177524, 0.001915673166467582, 0.020696407441166527, 0.0794738496853742,
    0.024396434423635458, 0.001991275559995653, 0.012255179048239709, 0.011519908803786914,  },
00440         { 0.09490196539650382, 0.024774065297941558, 0.011059978491039946, 0.0018433929866641786,
    -0.49532302419935553, 0.0038713652226125303, 0.001322322616098299, 0.027233323925 50304,
    0.006488243590595672, 0.008795507538746 39, 0.035337706733868 53, 0.004898382207458443,
    0.008123175217929567, 0.0163164927695339, 0.0053404603940086955, 0.10438172655082653,
    0.03338521102573777, 0.011003444371882 55, 0.020462221992322477, 0.07578403786803438,  },
00441         { 0.08395752449304077, 0.14237591176657988, 0.06436282383022025, 0.037530193415185 13,
    0.0020351596038226377, -1.4023387548390867, 0.3387225496005603, 0.029309341610100084,
    0.11190508042997918, 0.005889192889713301, 0.07995520871204236, 0.25770233446880264,
    0.032145957399464874, 0.004096168561193013, 0.045561736306503266, 0.07629344230591827,
    0.05583430930572718, 0.003310485836050631, 0.008567800353917012, 0.0227835339502657,  },
00442         { 0.14626068695265626, 0.020598049179270 61, 0.039493883292954945, 0.37568710385386667,
    0.000439745510285292, 0.21427611742531016, -1.257111487537639, 0.050414514957828435,
    0.014854927673510668, 0.006585968101205492, 0.014185493274263147, 0.17099633989863866,
    0.006555506956466206, 0.0033260155085525515, 0.0333085614068389, 0.052273098569976494,
    0.053545886713018596, 0.0024023683050871133, 0.007386082793355458, 0.04452113716455361,  },
00443         { 0.13090971375655758, 0.027422899836273246, 0.04693077400841045, 0.05266930984980068,
    0.006315966402268248, 0.012930368227352645, 0.035158534499975 56, -0.5057491780818103,
    0.006499657929126434, 0.001574185700212 0755, 0.005637276754876061, 0.02471610789994531,
    0.0036217925677535395, 0.0020468790524988626, 0.011889656120148239, 0.099499515735639 34,
    0.014697305104204883, 0.005171006588994236, 0.003898196776018 7003, 0.014160031271754 28,  },
00444         { 0.02928762028770396, 0.10024005265424024, 0.16495944410047816, 0.05663004701308376,
    0.005127590819911471, 0.16822931995187806, 0.035301468211007 53, 0.022148155111846 2,
    -1.0085628865968066, 0.007144039655446344, 0.045928802381322936, 0.058914581911146 61,
    0.008407322631387035, 0.027850238704916878, 0.033984295321816 99, 0.054885496610970495,
    0.03080301544484597, 0.004029123217092933, 0.14574177946895 75, 0.008950493098753481,  },
00445         { 0.017864807096055722, 0.008769990316654042, 0.02310914075067111, 0.002399674176679085,
    0.0035039388531466916, 0.004462899050317317, 0.007889532113049963, 0.00270403567608046 16,
    0.0036012498665890782, -1.2533192612174784, 0.2915914613866592, 0.021426034654471 57,
    0.0885677023866053, 0.04343207525887664, 0.004877936808716465, 0.023687324161655532,
    0.09489765628029291, 0.0032605531823541914, 0.015809286701090 47, 0.591463962497512 7,  },
00446         { 0.03676869017832784, 0.023342567084429 44, 0.005484124208196995, 0.005160221331516516,
    0.007914410027679094, 0.034063762496917525, 0.009553458866952599, 0.005443903037288344,
    0.013016046464044532, 0.16393035260316008, -0.7378782205897432, 0.01704088031890741,
    0.10097790671870785, 0.086709602561014 58, 0.020299060473896 31, 0.025563312184338 12,
    0.021256694937168644, 0.010209171449946 04, 0.014998372673478071, 0.136145682973773 16,  },
00447         { 0.08374194741204094, 0.25100092299321725, 0.12558722824289453, 0.029198358192822 54,
    0.0015247318815057268, 0.15258956530702983, 0.16005293362321688, 0.033172769845920 55,
    0.023204776910099468, 0.016741216077158262, 0.023683869237943283, -1.1429624377918817,
    0.019435691401668617, 0.0036417565742282135, 0.027184390256132007, 0.07171525737685298,
    0.090265232421435 69, 0.002110015226345 6985, 0.005014181838146751, 0.023097592973222 52,  },
00448         { 0.08256204867767036, 0.032042764827633095, 0.008264921421089172, 0.006313253911990309,
    0.008041996363208195, 0.06053828126173739, 0.0195155297899663, 0.015460461909997288,
    0.010531968466120388, 0.22009887163670735, 0.44635893288175915, 0.0618154782505777,
    -1.3437091202174194, 0.048808868363876 56, 0.008363275006810 68, 0.036624367142694 95,
    0.09866971706786477, 0.007913512325499689, 0.016120340258359945, 0.15566453065385616,  },
00449         { 0.019450356660083032, 0.0048175168089135845, 0.0040095257323236 14, 0.002843465125285 5015,
    0.008197241851056197, 0.0039145732720045874, 0.005024596801342159, 0.004433982329856833,
    0.017704499045621883, 0.05477165998810143, 0.1945037358856186, 0.005877749001224 82,
    0.02476860933339688, -0.7253348974230803, 0.007880747393608459, 0.040482233179616 73,
    0.011187518384505948, 0.023472336939219917, 0.2428482827643911, 0.049146266926909016,  },
00450         { 0.1329268794994748, 0.031870487863733026, 0.008133997587276306, 0.025798703149963 37,
    0.0022531808639589773, 0.036566491501643955, 0.042258029632247 79, 0.021629550301080065,
    0.018143012914245478, 0.005166041753536194, 0.038239579582548 53, 0.036846491374961 7,
    0.0035641476096533384, 0.006618260033879 2795, -0.615488321017 2385, 0.119628995503116 83,
    0.051763199010010 65, 0.002139170740181 9723, 0.008128931514919656, 0.023812449689815537,  },
00451         { 0.3114723827103921, 0.057418931782417866, 0.16570746116372745, 0.065214767745963 85,
    0.028990828260031572, 0.04030779379636031, 0.043656649619226245, 0.119156559449009 53,
    0.01928890305017274, 0.016514161860740863, 0.031701009094054 98, 0.063989231747878 81,
    0.010274678105550326, 0.022379978931120 08, 0.078750849552542 38, -1.414995944815485,
    0.2849233535059582, 0.008036824804019845, 0.029611342953450494, 0.017600236682867 12,  },
00452         { 0.19599772922396227, 0.026003971790559993, 0.084644343334441 28, 0.022809956637548038,
    0.010564931193714416, 0.033610839966296824, 0.050953576584587 72, 0.020054465792765177,
```

```
      0.01233444366890279, 0.07538282700619177, 0.03003503227751553, 0.09176820556664868,
      0.031539759608400324, 0.007047017879908512, 0.03882597299941695, 0.3246418073071975,
      -1.1738518192966378, 0.0017012088873392932, 0.01095471405490418, 0.10498101551633642,  },
00453         { 0.01045387136937 4774, 0.054592051136009775, 0.0029986019360413057, 0.0078961005879026,
      0.014768092593680887, 0.008451876826270943, 0.009695525562406397, 0.02992482961411037,
      0.0068425832275883025, 0.010984781663710872, 0.06117952033090722, 0.009097886852328098,
      0.010728203089867357, 0.06270629616599628, 0.0068049329204305335, 0.03883685991445591,
      0.0072150750026460705, -0.47432220705100175, 0.09351510834667694, 0.027630009910597178,  },
00454         { 0.0222447272158118, 0.017895275488070673, 0.04528130048432225, 0.01981895873824889,
      0.01120026789504408, 0.008920940182306032, 0.012156982788869633, 0.00920026246825594,
      0.1009423640150575, 0.021721623431716408, 0.036655536052068394, 0.008817285142275928,
      0.008912750961232759, 0.264587741702797, 0.010546096178238477, 0.05835762053579319,
      0.018948032335748287, 0.038138327649229746, -0.7381466702685061, 0.023800577003419147,  },
00455         { 0.18536210040995543, 0.011812627574535127, 0.008182572831350187, 0.0092693248904432,
      0.020639123533032152, 0.01180321364483485, 0.03645992488282914, 0.01662794434957646,
      0.003084425296401694, 0.40433951340286917, 0.1655530552458264, 0.020208748575353482,
      0.042821820281977445, 0.026641772069187292, 0.015370932983147012, 0.017258214807348946,
      0.090346582938766, 0.005606589965708167, 0.0118420087189333, -1.1032304964020754,  },
00456                 }, new double[20] {
00457 0.0866279, 0.043972, 0.0390894, 0.0570451, 0.0193078, 0.0367281, 0.0580589, 0.0832518, 0.0244313,
      0.048466, 0.086209, 0.0620286, 0.0195027, 0.0384319, 0.0457631, 0.0695179, 0.0610127, 0.0143859,
      0.0352742, 0.0708956,  });
00458
00459 /// <summary>
00460 /// Read an amino acid substitution matrix in PAML format.
00461 /// </summary>
00462 /// <param name="pamlMatrixReader">A <see cref="TextReader"/> that contains the matrix in PAML format
      to read.</param>
00463 /// <returns>An <see cref="ImmutableRateMatrix"/> corresponding to the matrix contained in the input
      file,
00464 /// normalised so that the average substitution rate is 1.</returns>
00465             public static ImmutableRateMatrix ParsePAMLAminoAcidMatrix(TextReader pamlMatrixReader)
00466             {
00467                 static double ReadNumber(TextReader sr)
00468                 {
00469                     StringBuilder sb = new StringBuilder();
00470
00471                     int c = sr.Read();
00472
00473                     while (c >= 0 && char.IsWhiteSpace((char)c))
00474                     {
00475                         c = sr.Read();
00476                     }
00477
00478                     while (c >= 0 && !char.IsWhiteSpace((char)c))
00479                     {
00480                         sb.Append((char)c);
00481                         c = sr.Read();
00482                     }
00483
00484                     if (sb.Length > 0)
00485                     {
00486                         return double.Parse(sb.ToString(),
      System.Globalization.CultureInfo.InvariantCulture);
00487                     }
00488                     else
00489                     {
00490                         return double.NaN;
00491                     }
00492                 }
00493
00494                 double[] rates = new double[190];
00495                 double[] equilibriumFrequencies = new double[20];
00496
00497                 for (int i = 0; i < rates.Length; i++)
00498                 {
00499                     rates[i] = ReadNumber(pamlMatrixReader);
00500                 }
00501
00502                 for (int i = 0; i < equilibriumFrequencies.Length; i++)
00503                 {
00504                     equilibriumFrequencies[i] = ReadNumber(pamlMatrixReader);
00505                 }
00506
00507                 double[,] rateMatrix = new double[20, 20];
00508
00509                 double total = 0;
00510
00511                 for (int i = 0; i < 20; i++)
00512                 {
00513                     for (int j = 0; j < 20; j++)
00514                     {
00515                         if (i != j)
00516                         {
00517                             int i2 = Math.Max(i, j) - 1;
00518                             int j2 = Math.Min(i, j);
00519
```

```
00520                             rateMatrix[i, j] = rates[i2 * (i2 + 1) / 2 + j2] *
      equilibriumFrequencies[j];
00521
00522                             total += rateMatrix[i, j];
00523                         }
00524                     }
00525                 }
00526
00527                 for (int i = 0; i < 20; i++)
00528                 {
00529                     for (int j = 0; j < 20; j++)
00530                     {
00531                         rateMatrix[i, j] *= 20.0 / total;
00532                     }
00533                 }
00534
00535                 return new ImmutableRateMatrix(new char[20] { 'A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G',
      'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W', 'Y', 'V' }, rateMatrix, equilibriumFrequencies);
00536             }
00537
00538 /// <summary>
00539 /// Read an amino acid substitution matrix in PAML format.
00540 /// </summary>
00541 /// <param name="fileName">The path to the file containing the matrix in PAML format to read.</param>
00542 /// <returns>An <see cref="ImmutableRateMatrix"/> corresponding to the matrix contained in the input
      file,
00543 /// normalised so that the average substitution rate is 1.</returns>
00544             public static ImmutableRateMatrix ParsePAMLAminoAcidMatrix(string fileName)
00545             {
00546                 using (StreamReader sr = new StreamReader(fileName))
00547                 {
00548                     return ParsePAMLAminoAcidMatrix(sr);
00549                 }
00550             }
00551         }
00552     }
00553 }
```

8.13 Sequence.cs

```
00001 using System;
00002 using System.Collections;
00003 using System.Collections.Generic;
00004 using System.Collections.Immutable;
00005 using System.Linq;
00006 using System.Runtime.Versioning;
00007 using System.Text;
00008
00009 namespace PhyloTree.SequenceSimulation
00010 {
00011 /// <summary>
00012 /// Represents a sequence of characters.
00013 /// </summary>
00014     public class Sequence :  IReadOnlyList<char>
00015     {
00016         internal readonly int[] intSequence;
00017         private string stringSequence;
00018
00019 /// <summary>
00020 /// The possible states that make up the sequence.  Note that some states may not actually be present
      in the sequence.
00021 /// </summary>
00022         public ImmutableArray<char> States { get; }
00023
00024 /// <summary>
00025 /// The conservation profile of the sequence.  If this is <see langword="null"/>, all positions in the
      sequence are equally conserved.
00026 /// </summary>
00027         public ImmutableArray<double>?  Conservation { get; }
00028
00029 /// <summary>
00030 /// The indel profile of the sequence.  If this is <see langword="null"/>, all positions in the
      sequence have equal
00031 /// probability of being affected by an indel event.
00032 /// </summary>
00033         public ImmutableArray<double>?  IndelProfile { get; }
00034
00035 /// <summary>
00036 /// Returns the sequence as a <see langword="string"/>.
00037 /// </summary>
00038         public string StringSequence
00039         {
00040             get
```

```
00041             {
00042                 return stringSequence;
00043             }
00044         }
00045 
00046 /// <summary>
00047 /// The length of the sequence.
00048 /// </summary>
00049         public int Length
00050         {
00051             get { return intSequence.Length; }
00052         }
00053 
00054 /// <inheritdoc/>
00055         public int Count => this.Length;
00056 
00057 /// <inheritdoc/>
00058         public char this[int index] => this.stringSequence[index];
00059 
00060 /// <summary>
00061 /// Creates a new <see cref="Sequence"/>.
00062 /// </summary>
00063 /// <param name="sequence">The sequence.</param>
00064         public Sequence(ReadOnlySpan<char> sequence)
00065         {
00066             List<char> states = new List<char>();
00067             Dictionary<char, int> characterIndices = new Dictionary<char, int>();
00068 
00069             intSequence = new int[sequence.Length];
00070 
00071             for (int i = 0; i < sequence.Length; i++)
00072             {
00073                 if (sequence[i] == '-')
00074                 {
00075                     intSequence[i] = -1;
00076                 }
00077                 else if (!characterIndices.TryGetValue(sequence[i], out intSequence[i]))
00078                 {
00079                     characterIndices[sequence[i]] = states.Count;
00080                     states.Add(sequence[i]);
00081                 }
00082             }
00083 
00084             this.States = ImmutableArray.Create<char>(states.ToArray());
00085             this.Conservation = null;
00086             this.IndelProfile = null;
00087             this.stringSequence = new string(sequence);
00088         }
00089 
00090 /// <summary>
00091 /// Creates a new <see cref="Sequence"/>.
00092 /// </summary>
00093 /// <param name="sequence">The sequence.</param>
00094 /// <param name="states">The possible states for the sequence.</param>
00095         public Sequence(ReadOnlySpan<char> sequence, IReadOnlyList<char> states)
00096         {
00097             Dictionary<char, int> characterIndices = new Dictionary<char, int>(from el in
      Enumerable.Range(0, states.Count) select new KeyValuePair<char, int>(states[el], el));
00098 
00099             intSequence = new int[sequence.Length];
00100 
00101             for (int i = 0; i < sequence.Length; i++)
00102             {
00103                 if (sequence[i] == '-')
00104                 {
00105                     intSequence[i] = -1;
00106                 }
00107                 else if (!characterIndices.TryGetValue(sequence[i], out intSequence[i]))
00108                 {
00109                     throw new ArgumentException("State " + sequence[i] + " is not a valid character
      state!");
00110                 }
00111             }
00112 
00113             this.States = ImmutableArray.Create<char>(states.ToArray());
00114             this.Conservation = null;
00115             this.IndelProfile = null;
00116             this.stringSequence = new string(sequence);
00117         }
00118 
00119 /// <summary>
00120 /// Creates a new <see cref="Sequence"/>.
00121 /// </summary>
00122 /// <param name="sequence">The sequence.</param>
00123 /// <param name="states">The possible states for the sequence.</param>
00124 /// <param name="conservation">The conservation profile for the sequence.</param>
00125         public Sequence(ReadOnlySpan<char> sequence, IReadOnlyList<char> states, IReadOnlyList<double>
```

```
conservation)
00126         {
00127             Dictionary<char, int> characterIndices = new Dictionary<char, int>(from el in
Enumerable.Range(0, states.Count) select new KeyValuePair<char, int>(states[el], el));
00128 
00129             intSequence = new int[sequence.Length];
00130 
00131             for (int i = 0; i < sequence.Length; i++)
00132             {
00133                 if (sequence[i] == '-')
00134                 {
00135                     intSequence[i] = -1;
00136                 }
00137                 else if (!characterIndices.TryGetValue(sequence[i], out intSequence[i]))
00138                 {
00139                     throw new ArgumentException("State " + sequence[i] + " is not a valid character
state!");
00140                 }
00141             }
00142 
00143             this.States = ImmutableArray.Create<char>(states.ToArray());
00144 
00145             if (conservation != null)
00146             {
00147                 this.Conservation = ImmutableArray.Create<double>(conservation.ToArray());
00148             }
00149 
00150             this.IndelProfile = null;
00151             this.stringSequence = new string(sequence);
00152         }
00153 
00154 /// <summary>
00155 /// Creates a new <see cref="Sequence"/>.
00156 /// </summary>
00157 /// <param name="sequence">The sequence.</param>
00158 /// <param name="states">The possible states for the sequence.</param>
00159 /// <param name="conservation">The conservation profile for the sequence.</param>
00160 /// <param name="indelProfile">The indel profile for the sequence.</param>
00161         public Sequence(ReadOnlySpan<char> sequence, IReadOnlyList<char> states, IReadOnlyList<double>
conservation, IReadOnlyList<double> indelProfile)
00162         {
00163             Dictionary<char, int> characterIndices = new Dictionary<char, int>(from el in
Enumerable.Range(0, states.Count) select new KeyValuePair<char, int>(states[el], el));
00164 
00165             intSequence = new int[sequence.Length];
00166 
00167             for (int i = 0; i < sequence.Length; i++)
00168             {
00169                 if (sequence[i] == '-')
00170                 {
00171                     intSequence[i] = -1;
00172                 }
00173                 else if (!characterIndices.TryGetValue(sequence[i], out intSequence[i]))
00174                 {
00175                     throw new ArgumentException("State " + sequence[i] + " is not a valid character
state!");
00176                 }
00177             }
00178 
00179             this.States = ImmutableArray.Create<char>(states.ToArray());
00180 
00181             if (conservation != null)
00182             {
00183                 this.Conservation = ImmutableArray.Create<double>(conservation.ToArray());
00184             }
00185 
00186             if (indelProfile != null)
00187             {
00188                 this.IndelProfile = ImmutableArray.Create<double>(indelProfile.ToArray());
00189             }
00190 
00191             this.stringSequence = new string(sequence);
00192         }
00193 
00194         internal Sequence(int[] intSequence, char[] states, double[] conservation = null, double[]
indelProfile = null)
00195         {
00196             this.intSequence = (int[])intSequence.Clone();
00197             this.States = ImmutableArray.Create<char>(states);
00198 
00199             if (conservation != null)
00200             {
00201                 this.Conservation = ImmutableArray.Create<double>(conservation);
00202             }
00203 
00204             if (indelProfile != null)
00205             {
```

```
00206                 this.IndelProfile = ImmutableArray.Create<double>(indelProfile);
00207             }
00208 
00209             this.stringSequence = SequenceSimulation.SequenceToString(intSequence, states);
00210         }
00211 
00212 /// <summary>
00213 /// Converts a <see cref="Sequence"/> to a <see langword="string"/>
00214 /// </summary>
00215 /// <param name="sequence">The <see cref="Sequence"/> to convert.</param>
00216         public static implicit operator string(Sequence sequence)
00217         {
00218             return sequence.stringSequence;
00219         }
00220 
00221 /// <inheritdoc/>
00222         public override string ToString()
00223         {
00224             return this.stringSequence;
00225         }
00226 
00227 /// <inheritdoc/>
00228         public IEnumerator<char> GetEnumerator()
00229         {
00230             return this.stringSequence.GetEnumerator();
00231         }
00232 
00233         IEnumerator IEnumerable.GetEnumerator()
00234         {
00235             return this.GetEnumerator();
00236         }
00237 
00238 
00239 /// <summary>
00240 /// Create a random sequence with the specified <paramref name="length"/>, using the states and
      equilibrium frequencies from the <paramref name="rateMatrix"/>.
00241 /// </summary>
00242 /// <param name="length">The length of the sequence.</param>
00243 /// <param name="rateMatrix">The rate matrix.</param>
00244 /// <returns>A random sequence.</returns>
00245         public static Sequence RandomSequence(int length, RateMatrix rateMatrix) => new
      Sequence(SequenceSimulation.RandomSequence(length, rateMatrix.GetEquilibriumFrequencies()),
      rateMatrix.GetStates());
00246 
00247 /// <summary>
00248 /// Create a random sequence with the specified <paramref name="length"/> containing all <paramref
      name="states"/> with equal probability.
00249 /// </summary>
00250 /// <param name="length">The length of the sequence.</param>
00251 /// <param name="states">The character states to use for the sequence.</param>
00252 /// <returns>A random sequence</returns>
00253         public static Sequence RandomSequence(int length, IReadOnlyList<char> states) => new
      Sequence(SequenceSimulation.RandomSequence(length, Enumerable.Repeat(1.0, states.Count).ToArray()),
      states.ToArray());
00254 
00255 /// <summary>
00256 /// Create a random sequence with the specified <paramref name="length"/> containing all <paramref
      name="states"/> with equal probability.
00257 /// </summary>
00258 /// <param name="length">The length of the sequence.</param>
00259 /// <param name="states">The character states to use for the sequence.</param>
00260 /// <returns>A random sequence</returns>
00261         public static Sequence RandomSequence(int length, char[] states) => new
      Sequence(SequenceSimulation.RandomSequence(length, Enumerable.Repeat(1.0, states.Length).ToArray()),
      states);
00262 
00263 /// <summary>
00264 /// Create a random sequence with the specified <paramref name="length"/> containing each state with
      probabilities given by <paramref name="stateFrequencies"/>.
00265 /// </summary>
00266 /// <param name="length">The length of the sequence.</param>
00267 /// <param name="states">The character states to use for the sequence.</param>
00268 /// <param name="stateFrequencies">The frequency for each state.</param>
00269 /// <returns>A random sequence</returns>
00270         public static Sequence RandomSequence(int length, IReadOnlyList<char> states,
      IReadOnlyList<double> stateFrequencies) => new Sequence(SequenceSimulation.RandomSequence(length,
      stateFrequencies.ToArray()), states.ToArray());
00271 
00272 /// <summary>
00273 /// Create a random sequence with the specified <paramref name="length"/> containing each state with
      probabilities given by <paramref name="stateFrequencies"/>.
00274 /// </summary>
00275 /// <param name="length">The length of the sequence.</param>
00276 /// <param name="states">The character states to use for the sequence.</param>
00277 /// <param name="stateFrequencies">The frequency for each state.</param>
00278 /// <returns>A random sequence</returns>
00279         public static Sequence RandomSequence(int length, char[] states, double[] stateFrequencies) =>
```

```
    new Sequence(SequenceSimulation.RandomSequence(length, stateFrequencies), states);
00280
00281 /// <summary>
00282 /// Simulates the evolution of the <see cref="Sequence"/> over the specified amount of <paramref
    name="time"/>, using the specified rate matrix.
00283 /// No indels are allowed to happen.
00284 /// </summary>
00285 /// <param name="rateMatrix">The rate matrix to simulate sequence evolution.</param>
00286 /// <param name="time">The length of time over which the sequence evolves.</param>
00287 /// <returns>The evolved sequence.</returns>
00288         public Sequence Evolve(RateMatrix rateMatrix, double time)
00289         {
00290             (int[] newSequence, int[][] _, double[] positionRates, double[] gapProfile) =
    SequenceSimulation.Evolve(time, this.intSequence, rateMatrix.GetMatrix(),
    rateMatrix.GetEquilibriumFrequencies(), rateMatrix.GetExponential(), positionRates:
    this.Conservation?.ToArray());
00291
00292             return new Sequence(newSequence, rateMatrix.States.ToArray(), positionRates, gapProfile);
00293         }
00294
00295 /// <summary>
00296 /// Simulates the evolution of the <see cref="Sequence"/> over the specified amount of <paramref
    name="time"/>, using the specified rate matrix.
00297 /// Insertions and deletions happen according to the specified <paramref name="indelModel"/>.
00298 /// </summary>
00299 /// <param name="rateMatrix">The rate matrix to simulate sequence evolution.</param>
00300 /// <param name="indelModel">The insertion/deletion model.</param>
00301 /// <param name="time">The length of time over which the sequence evolves.</param>
00302 /// <returns>The evolved sequence.</returns>
00303         public Sequence Evolve(RateMatrix rateMatrix, IndelModel indelModel, double time)
00304         {
00305             (int[] newSequence, int[][] _, double[] positionRates, double[] gapProfile) =
    SequenceSimulation.Evolve(time, this.intSequence, rateMatrix.GetMatrix(),
    rateMatrix.GetEquilibriumFrequencies(), rateMatrix.GetExponential(), new double[] {
    indelModel.InsertionRate, indelModel.DeletionRate }, indelModel.InsertionSizeDistribution,
    indelModel.DeletionSizeDistribution, this.Conservation?.ToArray(), this.IndelProfile?.ToArray());
00306
00307             return new Sequence(newSequence, rateMatrix.States.ToArray(), positionRates, gapProfile);
00308         }
00309
00310 /// <summary>
00311 /// Simulates the evolution of the <see cref="Sequence"/> over the specified amount of <paramref
    name="time"/>, using the specified rate matrix.
00312 /// Insertions and deletions happen according to the specified <paramref name="indelModel"/>.
00313 /// </summary>
00314 /// <param name="rateMatrix">The rate matrix to simulate sequence evolution.</param>
00315 /// <param name="indelModel">The insertion/deletion model.</param>
00316 /// <param name="time">The length of time over which the sequence evolves.</param>
00317 /// <param name="insertions">An array containing all the insertion events that have occurred during
    the evolution of the sequence (useful to
00318 /// map the evolved sequence back onto the ancestral sequence).</param>
00319 /// <returns>The evolved sequence and an array containing all the insertion events that happened
    during its evolution.</returns>
00320         public Sequence Evolve(RateMatrix rateMatrix, IndelModel indelModel, double time, out
    Insertion[] insertions)
00321         {
00322             (int[] newSequence, int[][] insertionEvents, double[] positionRates, double[] gapProfile)
    = SequenceSimulation.Evolve(time, this.intSequence, rateMatrix.GetMatrix(),
    rateMatrix.GetEquilibriumFrequencies(), rateMatrix.GetExponential(), new double[] {
    indelModel.InsertionRate, indelModel.DeletionRate }, indelModel.InsertionSizeDistribution,
    indelModel.DeletionSizeDistribution, this.Conservation?.ToArray(), this.IndelProfile?.ToArray());
00323
00324             if (insertionEvents != null)
00325             {
00326                 insertions = new Insertion[insertionEvents.Length];
00327                 for (int i = 0; i < insertionEvents.Length; i++)
00328                 {
00329                     insertions[i] = new Insertion(insertionEvents[i][0], insertionEvents[i][1]);
00330                 }
00331             }
00332             else
00333             {
00334                 insertions = null;
00335             }
00336
00337             return new Sequence(newSequence, rateMatrix.States.ToArray(), positionRates, gapProfile);
00338         }
00339
00340 /// <summary>
00341 /// Simulates the evolution of the sequence over a phylogenetic <paramref name="tree"/>, with the
    specified rate matrix, <paramref name="scale"/> factor,
00342 /// and insertion/deletion model.
00343 /// </summary>
00344 /// <param name="tree">The tree over which the sequence evolves.  This is assumed to be rooted (i.e.,
    the ancestral sequence is placed at the root of the tree).</param>
00345 /// <param name="rateMatrix">The rate matrix that governs the evolution of the sequence.</param>
00346 /// <param name="scale">A scaling factor.  If this is different from 1, the effect is the same as
```

```
      multiplying the branch lengths of the tree or the rate matrix by
00347 /// the supplied value.</param>
00348 /// <param name="indelModel">The model for insertions/deletions.  If this is null, no
      insertions/deletions are allowed to happen.</param>
00349 /// <returns>A <see cref="Dictionary{String, Sequence}"/> containing entries for all the nodes in the
      tree that have names.  For each entry, the key is a <see langword="string"/>
00350 /// containing the <see cref="TreeNode.Name"/> of the node, and the value is a <see cref="Sequence"/>
      containing the sequence.  The sequences are all aligned.</returns>
00351         public Dictionary<string, Sequence> Evolve(TreeNode tree, RateMatrix rateMatrix, double scale
      = 1, IndelModel indelModel = null)
00352         {
00353             return tree.SimulateSequences(this, rateMatrix, scale, indelModel);
00354         }
00355
00356 /// <summary>
00357 /// Simulates the evolution of the sequence over a phylogenetic <paramref name="tree"/>, with the
      specified rate matrix, <paramref name="scale"/> factor,
00358 /// and insertion/deletion model.
00359 /// </summary>
00360 /// <param name="tree">The tree over which the sequence evolves.  This is assumed to be rooted (i.e.,
      the ancestral sequence is placed at the root of the tree).</param>
00361 /// <param name="rateMatrix">The rate matrix that governs the evolution of the sequence.</param>
00362 /// <param name="scale">A scaling factor.  If this is different from 1, the effect is the same as
      multiplying the branch lengths of the tree or the rate matrix by
00363 /// the supplied value.</param>
00364 /// <param name="indelModel">The model for insertions/deletions.  If this is null, no
      insertions/deletions are allowed to happen.</param>
00365 /// <returns>A <see cref="Dictionary{String, Sequence}"/> containing entries for all the nodes in the
      tree.  For each entry, the key is a <see langword="string"/>
00366 /// containing the <see cref="TreeNode.Id"/> of the node, and the value is a <see cref="Sequence"/>
      containing the sequence.  The sequences are all aligned.</returns>
00367         public Dictionary<string, Sequence> EvolveAll(TreeNode tree, RateMatrix rateMatrix, double
      scale = 1, IndelModel indelModel = null)
00368         {
00369             return tree.SimulateAllSequences(this, rateMatrix, scale, indelModel);
00370         }
00371
00372     }
00373 }
```

8.14 SequenceSimulation.cs

```
00001 using MathNet.Numerics.LinearAlgebra;
00002 using System;
00003 using System.Collections.Generic;
00004 using System.Linq;
00005 using PhyloTree.TreeBuilding;
00006 using MathNet.Numerics.Distributions;
00007
00008 /// <summary>
00009 /// Contains classes and methods that can be used to simulate sequence evolution.
00010 /// </summary>
00011 namespace PhyloTree.SequenceSimulation
00012 {
00013 /// <summary>
00014 /// Contains methods to simulate sequence evolution.
00015 /// </summary>
00016     public static partial class SequenceSimulation
00017     {
00018 /// <summary>
00019 /// The random number generator used to simulate sequence evolution.  If you change this, please
      ensure that it is thread-safe.
00020 /// </summary>
00021         public static Random RandomNumberGenerator { get; set; } = new ThreadSafeRandom();
00022
00023 /// <summary>
00024 /// Generate a random sequence of the specified length using the provided state frequencies.
00025 /// </summary>
00026 /// <param name="length">The length of the sequence.</param>
00027 /// <param name="stateFrequencies">The frequencies for the character states.</param>
00028 /// <returns>An <see cref="T:int[]"/> array where each element represents a character in the
      sequence.</returns>
00029         internal static int[] RandomSequence(int length, double[] stateFrequencies)
00030         {
00031             int[] samples = MathNet.Numerics.Distributions.Categorical.Samples(RandomNumberGenerator,
      stateFrequencies).Take(length).ToArray();
00032
00033             return samples;
00034         }
00035
00036 /// <summary>
00037 /// Converts a sequence stored as an <see cref="T:int[]"/> array into a <see cref="string"/>.
00038 /// </summary>
```

```
00039 /// <param name="sequence">The sequence as an <see cref="T:int[]"/> array.</param>
00040 /// <param name="states">The character states.</param>
00041 /// <returns>A <see cref="string"/> corresponding to the <paramref name="sequence"/> where each
      element has been replaced by the corresponding state.</returns>
00042         internal static string SequenceToString(int[] sequence, char[] states)
00043         {
00044             string tbr = new string('\0', sequence.Length);
00045 
00046             unsafe
00047             {
00048                 fixed (char* sequencePtr = tbr)
00049                 fixed (int* sequenceIntPtr = sequence)
00050                 fixed (char* statePtr = states)
00051                 {
00052                     for (int i = 0; i < sequence.Length; i++)
00053                     {
00054                         if (sequenceIntPtr[i] >= 0)
00055                         {
00056                             sequencePtr[i] = statePtr[sequenceIntPtr[i]];
00057                         }
00058                         else
00059                         {
00060                             sequencePtr[i] = '-';
00061                         }
00062                     }
00063                 }
00064             }
00065 
00066             return tbr;
00067         }
00068 
00069 /// <summary>
00070 /// Simulates the evolution of a sequence along a branch.
00071 /// </summary>
00072 /// <param name="branchLength">The length of the branch.</param>
00073 /// <param name="ancestralSequence">The ancestral sequence that is evolving along the branch.</param>
00074 /// <param name="rateMatrix">The rate matrix according to which the sequence is evolving.</param>
00075 /// <param name="equilibriumFrequencies">The equilibrium frequencies for the rate matrix (used to
      sample new states for insertions).</param>
00076 /// <param name="cachedExponential">The cached exponential of the rate matrix.</param>
00077 /// <param name="indelRates">The insertion and deletion rate.</param>
00078 /// <param name="insertionSizeDistribution">The size distribution for insertions.</param>
00079 /// <param name="deletionSizeDistribution">The size distribution for deletions.</param>
00080 /// <param name="positionRates">The site-specific evolutionary rates.</param>
00081 /// <param name="gapProfile">The site-specific indel probabilities.</param>
00082 /// <returns>The evolved sequence, an array containing all insertions that have happened during its
      evolution, the updated site-specific
00083 /// evolutionary rates (modified to have the same length of the sequence that has evolved), and the
      updated site-specific indel probabilities
00084 /// (again, modified to have the same length of the sequence that has evolved).</returns>
00085         internal static (int[] sequence, int[][] insertions, double[] positionRates, double[]
      gapProfile) Evolve(double branchLength, int[] ancestralSequence, Matrix<double> rateMatrix, double[]
      equilibriumFrequencies, MatrixExponential cachedExponential = null, double[] indelRates = null,
      IDiscreteDistribution insertionSizeDistribution = null, IDiscreteDistribution deletionSizeDistribution
      = null, double[] positionRates = null, double[] gapProfile = null)
00086         {
00087             int[] newSequence = new int[ancestralSequence.Length];
00088 
00089             if (positionRates == null)
00090             {
00091                 Matrix<double> matrixExp = rateMatrix.FastExponential(branchLength,
      cachedExponential).Result;
00092 
00093                 double[][] newStateProbabilities = new double[rateMatrix.RowCount][];
00094 
00095                 for (int i = 0; i < rateMatrix.RowCount; i++)
00096                 {
00097                     Vector<double> initialState = Vector<double>.Build.Dense(rateMatrix.RowCount);
00098                     initialState[i] = 1;
00099 
00100                     newStateProbabilities[i] = (initialState * matrixExp).AsArray();
00101                 }
00102 
00103                 unsafe
00104                 {
00105                     fixed (int* newSequencePtr = newSequence)
00106                     fixed (int* ancestralSequencePtr = ancestralSequence)
00107                     {
00108                         int[] counts = new int[rateMatrix.RowCount];
00109                         int[] indices = new int[rateMatrix.RowCount];
00110                         int[][] samples = new int[rateMatrix.RowCount][];
00111 
00112                         for (int i = 0; i < ancestralSequence.Length; i++)
00113                         {
00114                             if (ancestralSequencePtr[i] >= 0)
00115                             {
00116                                 counts[ancestralSequencePtr[i]]++;
```

```
00117                           }
00118                       }
00119
00120                       for (int i = 0; i < counts.Length; i++)
00121                       {
00122                           samples[i] = new int[counts[i]];
00123                           MathNet.Numerics.Distributions.Categorical.Samples(RandomNumberGenerator,
      samples[i], newStateProbabilities[i]);
00124                       }
00125
00126
00127                       for (int i = 0; i < ancestralSequence.Length; i++)
00128                       {
00129                           if (ancestralSequencePtr[i] >= 0)
00130                           {
00131                               int oldState = ancestralSequencePtr[i];
00132                               newSequencePtr[i] = samples[oldState][indices[oldState]];
00133                               indices[oldState]++;
00134                           }
00135                           else
00136                           {
00137                               newSequencePtr[i] = ancestralSequencePtr[i];
00138                           }
00139                       }
00140                   }
00141               }
00142           }
00143           else
00144           {
00145               unsafe
00146               {
00147                   fixed (int* newSequencePtr = newSequence)
00148                   fixed (int* ancestralSequencePtr = ancestralSequence)
00149                   {
00150                       Vector<double> initialState = Vector<double>.Build.Dense(rateMatrix.RowCount);
00151
00152                       for (int i = 0; i < ancestralSequence.Length; i++)
00153                       {
00154                           if (ancestralSequencePtr[i] >= 0)
00155                           {
00156                               Matrix<double> matrixExp = rateMatrix.FastExponential(branchLength *
      positionRates[i], cachedExponential).Result;
00157
00158                               for (int j = 0; j < rateMatrix.RowCount; j++)
00159                               {
00160                                   initialState[j] = 0;
00161                               }
00162                               initialState[ancestralSequencePtr[i]] = 1;
00163
00164                               double[] newStateProbabilities = (initialState * matrixExp).AsArray();
00165
00166                               newSequencePtr[i] =
      MathNet.Numerics.Distributions.Categorical.Sample(RandomNumberGenerator, newStateProbabilities);
00167                           }
00168                           else
00169                           {
00170                               newSequencePtr[i] = ancestralSequencePtr[i];
00171                           }
00172                       }
00173                   }
00174               }
00175           }
00176
00177           int[][] insertions;
00178
00179           if (indelRates != null && indelRates[0] + indelRates[1] > 0)
00180           {
00181               List<int[]> insertionList = new List<int[]>();
00182
00183               double indelRate = rateMatrix.Trace() * (indelRates[0] + indelRates[1]);
00184
00185               double time = Exponential.Sample(RandomNumberGenerator, -indelRate);
00186
00187               while (time < branchLength)
00188               {
00189                   int type = Categorical.Sample(RandomNumberGenerator, indelRates);
00190
00191                   if (type == 0)
00192                   {
00193                       int length = insertionSizeDistribution.Sample();
00194
00195                       int start;
00196
00197                       if (gapProfile == null)
00198                       {
00199                           start = RandomNumberGenerator.Next(0, newSequence.Length - length);
00200                       }
```

```
00201                       else
00202                       {
00203                           start = Categorical.Sample(gapProfile);
00204                       }
00205 
00206                       //Console.WriteLine("Insertion:  {0} {1}", start, length);
00207 
00208                       int[] newNewSequence = new int[newSequence.Length + length];
00209 
00210                       for (int i = 0; i < start; i++)
00211                       {
00212                           newNewSequence[i] = newSequence[i];
00213                       }
00214 
00215                       for (int i = start; i < newSequence.Length; i++)
00216                       {
00217                           newNewSequence[i + length] = newSequence[i];
00218                       }
00219 
00220                       int[] insertion = RandomSequence(length, equilibriumFrequencies);
00221 
00222                       for (int i = 0; i < length; i++)
00223                       {
00224                           newNewSequence[i + start] = insertion[i];
00225                       }
00226 
00227                       newSequence = newNewSequence;
00228 
00229                       if (gapProfile != null)
00230                       {
00231                           double[] newGapProfile = new double[gapProfile.Length + length];
00232                           for (int i = 0; i < start; i++)
00233                           {
00234                               newGapProfile[i] = gapProfile[i];
00235                           }
00236 
00237                           for (int i = start; i < gapProfile.Length; i++)
00238                           {
00239                               newGapProfile[i + length] = gapProfile[i];
00240                           }
00241 
00242                           for (int i = 0; i < length; i++)
00243                           {
00244                               newGapProfile[i + start] = gapProfile[start];
00245                           }
00246 
00247                           gapProfile = newGapProfile;
00248                       }
00249 
00250                       if (positionRates != null)
00251                       {
00252                           double[] newPositionRates = new double[positionRates.Length + length];
00253                           for (int i = 0; i < start; i++)
00254                           {
00255                               newPositionRates[i] = positionRates[i];
00256                           }
00257 
00258                           for (int i = start; i < positionRates.Length; i++)
00259                           {
00260                               newPositionRates[i + length] = positionRates[i];
00261                           }
00262 
00263                           for (int i = 0; i < length; i++)
00264                           {
00265                               newPositionRates[i + start] = positionRates[start];
00266                           }
00267 
00268                           positionRates = newPositionRates;
00269                       }
00270 
00271                       insertionList.Add(new int[] { start, length });
00272                   }
00273                   else if (type == 1)
00274                   {
00275                       int length = deletionSizeDistribution.Sample();
00276 
00277                       int start;
00278 
00279                       if (gapProfile == null)
00280                       {
00281                           start = RandomNumberGenerator.Next(0, newSequence.Length - length);
00282                       }
00283                       else
00284                       {
00285                           double[] startWeights = new double[newSequence.Length - length];
00286 
00287                           for (int i = 0; i < newSequence.Length - length; i++)
```

```
00288                               {
00289                                   for (int j = 0; j < length; j++)
00290                                   {
00291                                       startWeights[i] += gapProfile[i + j];
00292                                   }
00293                               }
00294 
00295                               start = Categorical.Sample(startWeights);
00296                           }
00297 
00298                           for (int i = 0; i < length; i++)
00299                           {
00300                               newSequence[start + i] = -1;
00301                           }
00302                       }
00303 
00304                       time += Exponential.Sample(RandomNumberGenerator, -indelRate);
00305                   }
00306 
00307                   insertions = insertionList.ToArray();
00308               }
00309               else
00310               {
00311                   insertions = new int[0][];
00312               }
00313 
00314 
00315               return (newSequence, insertions, positionRates, gapProfile);
00316           }
00317 
00318 /// <summary>
00319 /// Applies the specified insertion by inserting gaps in the specified sequence.
00320 /// </summary>
00321 /// <param name="sequence">The sequence on which the insertion should be applied.</param>
00322 /// <param name="insertion">The insertion (first element:  start, second element:  length).</param>
00323 /// <returns>A sequence where gaps (-1) have been inserted in the position corresponding to the
      insertion.</returns>
00324           private static int[] ApplyInsertion(int[] sequence, int[] insertion)
00325           {
00326               int[] tbr = new int[sequence.Length + insertion[1]];
00327 
00328               for (int i = 0; i < insertion[0]; i++)
00329               {
00330                   tbr[i] = sequence[i];
00331               }
00332 
00333               for (int i = insertion[0]; i < sequence.Length; i++)
00334               {
00335                   tbr[i + insertion[1]] = sequence[i];
00336               }
00337 
00338               for (int i = 0; i < insertion[1]; i++)
00339               {
00340                   tbr[insertion[0] + i] = -1;
00341               }
00342 
00343               return tbr;
00344           }
00345       }
00346 }
```

8.15 SequenceSimulation.public.cs

```
00001 using MathNet.Numerics.LinearAlgebra;
00002 using PhyloTree.TreeBuilding;
00003 using System;
00004 using System.Collections.Generic;
00005 using System.Linq;
00006 
00007 namespace PhyloTree.SequenceSimulation
00008 {
00009     partial class SequenceSimulation
00010     {
00011 /// <summary>
00012 /// Simulates the evolution of the specified ancestral sequence over a phylogenetic <paramref
      name="tree"/>, with the specified rate matrix, <paramref name="scale"/> factor,
00013 /// and insertion/deletion model.
00014 /// </summary>
00015 /// <param name="tree">The tree over which the sequence evolves.  This is assumed to be rooted (i.e.,
      the ancestral sequence is placed at the root of the tree).</param>
00016 /// <param name="ancestralSequence">The ancestral sequence whose evolution is being simulated.</param>
00017 /// <param name="rateMatrix">The rate matrix that governs the evolution of the sequence.</param>
00018 /// <param name="scale">A scaling factor.  If this is different from 1, the effect is the same as
      multiplying the branch lengths of the tree or the rate matrix by
```

```
00019 /// the supplied value.</param>
00020 /// <param name="indelModel">The model for insertions/deletions.  If this is null, no
      insertions/deletions are allowed to happen.</param>
00021 /// <returns>A <see cref="Dictionary{String, Sequence}"/> containing entries for all the nodes in the
      tree that have names.  For each entry, the key is a <see langword="string"/>
00022 /// containing the <see cref="TreeNode.Name"/> of the node, and the value is a <see cref="Sequence"/>
      containing the sequence.  The sequences are all aligned.</returns>
00023         public static Dictionary<string, Sequence> SimulateSequences(this TreeNode tree, Sequence
      ancestralSequence, RateMatrix rateMatrix, double scale = 1, IndelModel indelModel = null)
00024         {
00025             Matrix<double> transitionRateMatrix = rateMatrix.GetMatrix();
00026             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00027             char[] states = rateMatrix.GetStates();
00028 
00029             Matrix<double> mRateMatrix = transitionRateMatrix * scale;
00030 
00031             int[] ancestralSequenceInt = ancestralSequence.intSequence;
00032 
00033             List<TreeNode> nodes = tree.GetChildrenRecursive();
00034             int[][] sequences = new int[nodes.Count][];
00035 
00036             Dictionary<string, (int, int)> indices = new Dictionary<string, (int, int)>(nodes.Count);
00037 
00038             for (int i = 0; i < nodes.Count; i++)
00039             {
00040                 indices[nodes[i].Id] = (i, nodes[i].Children.Count);
00041             }
00042 
00043             MatrixExponential cachedExponential = mRateMatrix.FastExponential(1);
00044 
00045             sequences[0] = ancestralSequenceInt;
00046 
00047             double[] indelRates = indelModel != null ?  new double[] { indelModel.InsertionRate,
      indelModel.DeletionRate } :  null;
00048 
00049             double[] positionRates = ancestralSequence.Conservation?.ToArray();
00050             double[] gapProfile = ancestralSequence.IndelProfile?.ToArray();
00051 
00052 
00053             for (int i = 1; i < nodes.Count; i++)
00054             {
00055                 int[][] insertions;
00056 
00057                 (sequences[i], insertions, positionRates, gapProfile) = Evolve(nodes[i].Length,
      sequences[indices[nodes[i].Parent.Id].Item1], mRateMatrix, equilibriumFrequencies, cachedExponential,
      indelRates, indelModel?.InsertionSizeDistribution, indelModel?.DeletionSizeDistribution,
      positionRates, gapProfile);
00058 
00059                 if (insertions.Length > 0)
00060                 {
00061                     for (int j = 0; j < i; j++)
00062                     {
00063                         for (int k = 0; k < insertions.Length; k++)
00064                         {
00065                             sequences[j] = ApplyInsertion(sequences[j], insertions[k]);
00066                         }
00067                     }
00068                 }
00069             }
00070 
00071             Dictionary<string, Sequence> leafSequences = new Dictionary<string, Sequence>();
00072 
00073             foreach (TreeNode leaf in tree.GetChildrenRecursiveLazy())
00074             {
00075                 if (!string.IsNullOrEmpty(leaf.Name))
00076                 {
00077                     leafSequences[leaf.Name] = new Sequence(sequences[indices[leaf.Id].Item1], states,
      positionRates, gapProfile);
00078                 }
00079             }
00080 
00081             return leafSequences;
00082         }
00083 
00084 /// <summary>
00085 /// Simulates the evolution of a random ancestral sequence with the specified length over a
      phylogenetic <paramref name="tree"/>, with the specified rate matrix,
00086 /// <paramref name="scale"/> factor, and insertion/deletion model.
00087 /// </summary>
00088 /// <param name="tree">The tree over which the sequence evolves.  This is assumed to be rooted (i.e.,
      the ancestral sequence is placed at the root of the tree).</param>
00089 /// <param name="ancestralSequenceLength">The length of the ancestral sequence whose evolution is
      being simulated.  Note that if insertions/deletions are allowed to happen,
00090 /// the final length of the (aligned) sequences may differ from this.</param>
00091 /// <param name="rateMatrix">The rate matrix that governs the evolution of the sequence.</param>
00092 /// <param name="scale">A scaling factor.  If this is different from 1, the effect is the same as
      multiplying the branch lengths of the tree or the rate matrix by
```

```
00093 /// the supplied value.</param>
00094 /// <param name="indelModel">The model for insertions/deletions.  If this is null, no
      insertions/deletions are allowed to happen.</param>
00095 /// <returns>A <see cref="Dictionary{String, Sequence}"/> containing entries for all the nodes in the
      tree that have names.  For each entry, the key is a <see langword="string"/>
00096 /// containing the <see cref="TreeNode.Name"/> of the node, and the value is a <see cref="Sequence"/>
      containing the sequence.  The sequences are all aligned.</returns>
00097         public static Dictionary<string, Sequence> SimulateSequences(this TreeNode tree, int
      ancestralSequenceLength, RateMatrix rateMatrix, double scale = 1, IndelModel indelModel = null)
00098         {
00099             Sequence ancestralSequence = Sequence.RandomSequence(ancestralSequenceLength, rateMatrix);
00100
00101             return SimulateSequences(tree, ancestralSequence, rateMatrix, scale, indelModel);
00102         }
00103
00104 /// <summary>
00105 /// Simulates the evolution of the specified ancestral sequence over a phylogenetic <paramref
      name="tree"/>, with the specified rate matrix, <paramref name="scale"/> factor,
00106 /// and insertion/deletion model.
00107 /// </summary>
00108 /// <param name="tree">The tree over which the sequence evolves.  This is assumed to be rooted (i.e.,
      the ancestral sequence is placed at the root of the tree).</param>
00109 /// <param name="ancestralSequence">The ancestral sequence whose evolution is being simulated.</param>
00110 /// <param name="rateMatrix">The rate matrix that governs the evolution of the sequence.</param>
00111 /// <param name="scale">A scaling factor.  If this is different from 1, the effect is the same as
      multiplying the branch lengths of the tree or the rate matrix by
00112 /// the supplied value.</param>
00113 /// <param name="indelModel">The model for insertions/deletions.  If this is null, no
      insertions/deletions are allowed to happen.</param>
00114 /// <returns>A <see cref="Dictionary{String, Sequence}"/> containing entries for all the nodes in the
      tree.  For each entry, the key is a <see langword="string"/>
00115 /// containing the <see cref="TreeNode.Id"/> of the node, and the value is a <see cref="Sequence"/>
      containing the sequence.  The sequences are all aligned.</returns>
00116         public static Dictionary<string, Sequence> SimulateAllSequences(this TreeNode tree, Sequence
      ancestralSequence, RateMatrix rateMatrix, double scale = 1, IndelModel indelModel = null)
00117         {
00118             Matrix<double> transitionRateMatrix = rateMatrix.GetMatrix();
00119             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00120             char[] states = rateMatrix.GetStates();
00121
00122             Matrix<double> mRateMatrix = transitionRateMatrix * scale;
00123
00124             int[] ancestralSequenceInt = ancestralSequence.intSequence;
00125
00126             List<TreeNode> nodes = tree.GetChildrenRecursive();
00127             int[][] sequences = new int[nodes.Count][];
00128
00129             Dictionary<string, (int, int)> indices = new Dictionary<string, (int, int)>(nodes.Count);
00130
00131             for (int i = 0; i < nodes.Count; i++)
00132             {
00133                 indices[nodes[i].Id] = (i, nodes[i].Children.Count);
00134             }
00135
00136             MatrixExponential cachedExponential = mRateMatrix.FastExponential(1);
00137
00138             sequences[0] = ancestralSequenceInt;
00139
00140             double[] indelRates = indelModel != null ?  new double[] { indelModel.InsertionRate,
      indelModel.DeletionRate } :  null;
00141
00142             double[] positionRates = ancestralSequence.Conservation?.ToArray();
00143             double[] gapProfile = ancestralSequence.IndelProfile?.ToArray();
00144
00145
00146             for (int i = 1; i < nodes.Count; i++)
00147             {
00148                 int[][] insertions;
00149
00150                 (sequences[i], insertions, positionRates, gapProfile) = Evolve(nodes[i].Length,
      sequences[indices[nodes[i].Parent.Id].Item1], mRateMatrix, equilibriumFrequencies, cachedExponential,
      indelRates, indelModel?.InsertionSizeDistribution, indelModel?.DeletionSizeDistribution,
      positionRates, gapProfile);
00151
00152                 if (insertions.Length > 0)
00153                 {
00154                     for (int j = 0; j < i; j++)
00155                     {
00156                         for (int k = 0; k < insertions.Length; k++)
00157                         {
00158                             sequences[j] = ApplyInsertion(sequences[j], insertions[k]);
00159                         }
00160                     }
00161                 }
00162             }
00163
00164             Dictionary<string, Sequence> leafSequences = new Dictionary<string,
```

```
Sequence>(nodes.Count);
00165
00166             foreach (TreeNode node in nodes)
00167             {
00168                 leafSequences[node.Id] = new Sequence(sequences[indices[node.Id].Item1], states,
positionRates, gapProfile);
00169             }
00170
00171             return leafSequences;
00172         }
00173
00174 /// <summary>
00175 /// Simulates the evolution of a random ancestral sequence with the specified length over a
phylogenetic <paramref name="tree"/>, with the specified rate matrix,
00176 /// <paramref name="scale"/> factor, and insertion/deletion model.
00177 /// </summary>
00178 /// <param name="tree">The tree over which the sequence evolves.  This is assumed to be rooted (i.e.,
the ancestral sequence is placed at the root of the tree).</param>
00179 /// <param name="ancestralSequenceLength">The length of the ancestral sequence whose evolution is
being simulated.  Note that if insertions/deletions are allowed to happen,
00180 /// the final length of the (aligned) sequences may differ from this.</param>
00181 /// <param name="rateMatrix">The rate matrix that governs the evolution of the sequence.</param>
00182 /// <param name="scale">A scaling factor.  If this is different from 1, the effect is the same as
multiplying the branch lengths of the tree or the rate matrix by
00183 /// the supplied value.</param>
00184 /// <param name="indelModel">The model for insertions/deletions.  If this is null, no
insertions/deletions are allowed to happen.</param>
00185 /// <returns>A <see cref="Dictionary{String, Sequence}"/> containing entries for all the nodes in the
tree.  For each entry, the key is a <see langword="string"/>
00186 /// containing the <see cref="TreeNode.Id"/> of the node, and the value is a <see cref="Sequence"/>
containing the sequence.  The sequences are all aligned.</returns>
00187         public static Dictionary<string, Sequence> SimulateAllSequences(this TreeNode tree, int
ancestralSequenceLength, RateMatrix rateMatrix, double scale = 1, IndelModel indelModel = null)
00188         {
00189             Sequence ancestralSequence = Sequence.RandomSequence(ancestralSequenceLength, rateMatrix);
00190
00191             return SimulateAllSequences(tree, ancestralSequence, rateMatrix, scale, indelModel);
00192         }
00193
00194 /// <summary>
00195 /// Represents a function that can be evaluated to return the scaling factor to use in order to obtain
the specified average <paramref name="conservation" />.
00196 /// </summary>
00197 /// <param name="conservation">The average conservation whose corresponding scaling factor will be
returned.</param>
00198 /// <returns>The scaling factor that will produce the speicfied average <paramref name="conservation"
/>.</returns>
00199         public delegate double GetScale(double conservation);
00200
00201 /// <summary>
00202 /// Returns a method that can be evaluated to determine the scaling factor that, when applied to a
sequence simulation done using the specified <paramref name="tree"/>
00203 /// and rate matrix, will produce at the tips a sequence alignment with the specified (average)
percent identity.
00204 /// </summary>
00205 /// <param name="tree">The <see cref="TreeNode"/> on which the sequence simulations will be
performed.</param>
00206 /// <param name="rateMatrix">The rate matrix that will be used for the sequence simulations.</param>
00207 /// <param name="minRate">Minimum rate value to test.</param>
00208 /// <param name="maxRate">Maximum rate value to test.</param>
00209 /// <returns>A method that can be evaluated to determine the scaling factor that, when applied to a
sequence simulation done using the specified <paramref name="tree"/>
00210 /// and rate matrix, will produce at the tips a sequence alignment with the specified (average)
percent identity.</returns>
00211         public static GetScale ConservationToScale(TreeNode tree, RateMatrix rateMatrix, double
minRate = 1e-5, double maxRate = 1e4)
00212         {
00213             Matrix<double> transitionRateMatrix = rateMatrix.GetMatrix();
00214             double[] equilibriumFrequencies = rateMatrix.GetEquilibriumFrequencies();
00215
00216             int steps = 101;
00217
00218             double logMinRate = Math.Log(minRate);
00219             double logMaxRate = Math.Log(maxRate);
00220
00221             double[][] data = new double[steps][];
00222
00223             List<TreeNode> nodes = tree.GetChildrenRecursive();
00224
00225             for (int i = 0; i < steps; i++)
00226             {
00227                 double rate = Math.Exp(logMinRate + (logMaxRate - logMinRate) * i / (steps - 1));
00228                 Dictionary<string, Sequence> sequences = SimulateAllSequences(tree, 100, rateMatrix,
rate);
00229
00230                 int[][] counts = new int[100][];
00231
```

```
00232                 for (int j = 0; j < 100; j++)
00233                 {
00234                     counts[j] = new int[rateMatrix.States.Length];
00235                 }
00236 
00237                 int totalSeqs = 0;
00238 
00239                 foreach (TreeNode node in nodes)
00240                 {
00241                     if (node.Children.Count == 0)
00242                     {
00243                         totalSeqs++;
00244 
00245                         for (int j = 0; j < sequences[node.Id].Length; j++)
00246                         {
00247                             if (sequences[node.Id].intSequence[j] >= 0)
00248                             {
00249                                 counts[j][sequences[node.Id].intSequence[j]]++;
00250                             }
00251                         }
00252                     }
00253                 }
00254 
00255                 double averageConservation = (from el in counts select (double)el.Max() /
      totalSeqs).Average();
00256 
00257                 if (i > 0)
00258                 {
00259                     data[i] = new double[] { rate, Math.Min(averageConservation, data[i - 1][1]) };
00260                 }
00261                 else
00262                 {
00263                     data[i] = new double[] { rate, averageConservation };
00264                 }
00265             }
00266 
00267             return value =>
00268             {
00269                 if (value > data[0][1])
00270                 {
00271                     if (data[1][1] - data[0][1] != 0)
00272                     {
00273                         return data[0][0] + (data[0][0] - data[1][0]) * (data[0][1] - value) /
      (data[1][1] - data[0][1]);
00274                     }
00275                     else
00276                     {
00277                         return data[0][0];
00278                     }
00279                 }
00280                 if (value < data[data.Length - 1][1])
00281                 {
00282                     if (data[data.Length - 1][1] - data[data.Length - 2][1] != 0)
00283                     {
00284                         return data[data.Length - 1][0] + (data[data.Length - 1][0] - data[data.Length
      - 2][0]) * (value - data[data.Length - 1][1]) / (data[data.Length - 1][1] - data[data.Length - 2][1]);
00285                     }
00286                     else
00287                     {
00288                         return data[data.Length - 1][0];
00289                     }
00290                 }
00291 
00292                 for (int i = 0; i < data.Length - 1; i++)
00293                 {
00294                     if (data[i][1] > value && data[i + 1][1] <= value)
00295                     {
00296                         return data[i][0] + (data[i + 1][0] - data[i][0]) * (value - data[i][1]) /
      (data[i + 1][1] - data[i][1]);
00297                     }
00298                 }
00299 
00300                 return double.NaN;
00301             };
00302         }
00303 
00304 
00305 /// <summary>
00306 /// Converts a sequence alignment where the sequences are stored as <see cref="Sequence"/>s into an
      alignment where the sequences are stored as <see langword="string"/>s.
00307 /// </summary>
00308 /// <param name="alignment">The alignment to convert.</param>
00309 /// <returns>A <see cref="Dictionary{String, String}"/> where both keys and values are
      string.</returns>
00310         public static Dictionary<string, string> ToStringAlignment(this Dictionary<string, Sequence>
      alignment)
00311         {
```

```
00312             return new Dictionary<string, string>(from el in alignment select new KeyValuePair<string,
      string>(el.Key, el.Value));
00313         }
00314     }
00315 }
```

8.16 BirthDeathTree.cs

```
00001 using MathNet.Numerics.Distributions;
00002 using System;
00003 using System.Collections.Generic;
00004 using System.Linq;
00005 using System.Threading;
00006
00007 namespace PhyloTree.TreeBuilding
00008 {
00009 /// <summary>
00010 /// Contains methods to simulate birth-death trees.
00011 /// </summary>
00012     public static class BirthDeathTree
00013     {
00014 /// <summary>
00015 /// Simulate an unlabelled birth-death tree, stopping when the age of the tree reaches a certain
      value.
00016 /// </summary>
00017 /// <param name="treeAge">The final age of the tree.  Note that the actual age of the tree may be
      smaller than this;
00018 /// this can happen either if <paramref name="keepDeadLineages"/> is <see langword="false"/> and one
      of the two clades
00019 /// descending from the root node goes exinct, or if <paramref name="keepDeadLineages"/> is <see
      langword="true"/> and
00020 /// all the clades go extinct.  If all the clades go extinct and <paramref name="keepDeadLineages"/>
      is <see langword="false"/>,
00021 /// this method will return <see langword="null"/>.</param>
00022 /// <param name="birthRate">The birth rate of the tree.</param>
00023 /// <param name="deathRate">The death rate of the tree.</param>
00024 /// <param name="keepDeadLineages">If this is <see langword="true"/>, dead lineages are kept in the
      tree.  If this is
00025 /// <see langword="false"/>, they are pruned from the tree.</param>
00026 /// <param name="cancellationToken">A <see cref="CancellationToken"/> used to cancel the simulation if
      it takes too long.</param>
00027 /// <returns>A <see cref="TreeNode"/> object containing the unlabelled birth-death tree, or <see
      langword="null"/> if all
00028 /// the lineages went extinct and <paramref name="keepDeadLineages"/> is <see
      langword="false"/>.</returns>
00029         public static TreeNode UnlabelledTree(double treeAge, double birthRate, double deathRate = 0,
      bool keepDeadLineages = false, CancellationToken cancellationToken = default)
00030         {
00031             List<TreeNode> aliveLineages = new List<TreeNode>();
00032
00033             TreeNode root = new TreeNode(null);
00034             TreeNode child1 = new TreeNode(root) { Length = 0 };
00035             TreeNode child2 = new TreeNode(root) { Length = 0 };
00036             root.Children.Add(child1);
00037             root.Children.Add(child2);
00038
00039             aliveLineages.Add(child1);
00040             aliveLineages.Add(child2);
00041
00042             double time = 0;
00043
00044             while (time < treeAge)
00045             {
00046                 cancellationToken.ThrowIfCancellationRequested();
00047
00048                 double timeToNextEvent = Exponential.Sample(RandomTree.RandomNumberGenerator,
      aliveLineages.Count * (birthRate + deathRate));
00049
00050                 if (timeToNextEvent < treeAge - time)
00051                 {
00052                     foreach (TreeNode lineage in aliveLineages)
00053                     {
00054                         lineage.Length += timeToNextEvent;
00055                     }
00056
00057                     bool isBirth = ContinuousUniform.Sample(RandomTree.RandomNumberGenerator, 0, 1) <
      birthRate / (birthRate + deathRate);
00058
00059                     if (isBirth)
00060                     {
00061                         int parent = DiscreteUniform.Sample(RandomTree.RandomNumberGenerator, 0,
      aliveLineages.Count - 1);
00062
```

```
00063                           TreeNode newChild1 = new TreeNode(aliveLineages[parent]) { Length = 0 };
00064                           TreeNode newChild2 = new TreeNode(aliveLineages[parent]) { Length = 0 };
00065
00066                           aliveLineages[parent].Children.Add(newChild1);
00067                           aliveLineages[parent].Children.Add(newChild2);
00068
00069                           aliveLineages.RemoveAt(parent);
00070                           aliveLineages.Add(newChild1);
00071                           aliveLineages.Add(newChild2);
00072                       }
00073                       else
00074                       {
00075                           int dead = DiscreteUniform.Sample(RandomTree.RandomNumberGenerator, 0,
      aliveLineages.Count - 1);
00076
00077                           if (!keepDeadLineages)
00078                           {
00079                               if (aliveLineages.Count == 1)
00080                               {
00081                                   return null;
00082                               }
00083                               else
00084                               {
00085                                   root = root.Prune(aliveLineages[dead], false);
00086                               }
00087                           }
00088
00089                           aliveLineages.RemoveAt(dead);
00090
00091                           if (aliveLineages.Count == 0)
00092                           {
00093                               return root;
00094                           }
00095                       }
00096                   }
00097                   else
00098                   {
00099                       foreach (TreeNode lineage in aliveLineages)
00100                       {
00101                           lineage.Length += treeAge - time;
00102                       }
00103                   }
00104
00105                   time += timeToNextEvent;
00106               }
00107
00108               return root;
00109           }
00110
00111 /// <summary>
00112 /// Simulate a labelled birth-death tree, stopping when the age of the tree reaches a certain value.
00113 /// </summary>
00114 /// <param name="treeAge">The final age of the tree.  Note that the actual age of the tree may be
      smaller than this;
00115 /// this can happen either if <paramref name="keepDeadLineages"/> is <see langword="false"/> and one
      of the two clades
00116 /// descending from the root node goes exinct, or if <paramref name="keepDeadLineages"/> is <see
      langword="true"/> and
00117 /// all the clades go extinct.  If all the clades go extinct and <paramref name="keepDeadLineages"/>
      is <see langword="false"/>,
00118 /// this method will return <see langword="null"/>.</param>
00119 /// <param name="birthRate">The birth rate of the tree.</param>
00120 /// <param name="deathRate">The death rate of the tree.</param>
00121 /// <param name="keepDeadLineages">If this is <see langword="true"/>, dead lineages are kept in the
      tree.  If this is
00122 /// <see langword="false"/>, they are pruned from the tree.</param>
00123 /// <param name="cancellationToken">A <see cref="CancellationToken"/> used to cancel the simulation if
      it takes too long.</param>
00124 /// <returns>A <see cref="TreeNode"/> object containing the labelled birth-death tree, or <see
      langword="null"/> if all
00125 /// the lineages went extinct and <paramref name="keepDeadLineages"/> is <see
      langword="false"/>.</returns>
00126           public static TreeNode LabelledTree(double treeAge, double birthRate, double deathRate = 0,
      bool keepDeadLineages = false, CancellationToken cancellationToken = default)
00127           {
00128               TreeNode unlabelledTree = UnlabelledTree(treeAge, birthRate, deathRate, keepDeadLineages,
      cancellationToken);
00129
00130               if (unlabelledTree == null)
00131               {
00132                   return null;
00133               }
00134
00135               List<TreeNode> leaves = unlabelledTree.GetLeaves();
00136
00137               List<int> labels = Enumerable.Range(1, leaves.Count).ToList();
00138
```

```
00139             for (int i = 0; i < leaves.Count; i++)
00140             {
00141                 int label = RandomTree.RandomNumberGenerator.Next(0, labels.Count);
00142 
00143                 leaves[i].Name = "t" + labels[label].ToString();
00144                 labels.RemoveAt(label);
00145             }
00146 
00147             return unlabelledTree;
00148         }
00149 
00150 /// <summary>
00151 /// Simulate an unlabelled birth-death tree, stopping when the number of lineages that are alive in
      the tree reaches a certain value.
00152 /// </summary>
00153 /// <param name="leafCount">The final number of lineages that are alive in the tree.  Note that, if
      <paramref name="keepDeadLineages"/>
00154 /// is <see langword="true"/>, the actual number of leaves in the tree may be larger than this,
      because the leaves corresponding
00155 /// to dead lineages are kept.  If all the lineages go extinct before the target number of lineages is
      reaced, the method will return
00156 /// a smaller tree if <paramref name="keepDeadLineages"/> is <see langword="true"/>, or <see
      langword="null"/> if it is <see langword="false"/>.</param>
00157 /// <param name="birthRate">The birth rate of the tree.</param>
00158 /// <param name="deathRate">The death rate of the tree.</param>
00159 /// <param name="keepDeadLineages">If this is <see langword="true"/>, dead lineages are kept in the
      tree.  If this is
00160 /// <see langword="false"/>, they are pruned from the tree.</param>
00161 /// <param name="cancellationToken">A <see cref="CancellationToken"/> used to cancel the simulation if
      it takes too long.</param>
00162 /// <returns>A <see cref="TreeNode"/> object containing the unlabelled birth-death tree, or <see
      langword="null"/> if all
00163 /// the lineages went extinct and <paramref name="keepDeadLineages"/> is <see
      langword="false"/>.</returns>
00164         public static TreeNode UnlabelledTree(int leafCount, double birthRate, double deathRate = 0,
      bool keepDeadLineages = false, CancellationToken cancellationToken = default)
00165         {
00166             return UnlabelledTree(leafCount, birthRate, deathRate, keepDeadLineages, out _,
      cancellationToken);
00167         }
00168 
00169 /// <summary>
00170 /// Simulate an unlabelled birth-death tree, stopping when the number of lineages that are alive in
      the tree reaches a certain value.
00171 /// </summary>
00172 /// <param name="leafCount">The final number of lineages that are alive in the tree.  Note that, if
      <paramref name="keepDeadLineages"/>
00173 /// is <see langword="true"/>, the actual number of leaves in the tree may be larger than this,
      because the leaves corresponding
00174 /// to dead lineages are kept.  If all the lineages go extinct before the target number of lineages is
      reaced, the method will return
00175 /// a smaller tree if <paramref name="keepDeadLineages"/> is <see langword="true"/>, or <see
      langword="null"/> if it is <see langword="false"/>.</param>
00176 /// <param name="birthRate">The birth rate of the tree.</param>
00177 /// <param name="deathRate">The death rate of the tree.</param>
00178 /// <param name="keepDeadLineages">If this is <see langword="true"/>, dead lineages are kept in the
      tree.  If this is
00179 /// <see langword="false"/>, they are pruned from the tree.</param>
00180 /// <param name="aliveLineages">The lineages that are alive at the end of the tree.</param>
00181 /// <param name="cancellationToken">A <see cref="CancellationToken"/> used to cancel the simulation if
      it takes too long.</param>
00182 /// <returns>A <see cref="TreeNode"/> object containing the unlabelled birth-death tree, or <see
      langword="null"/> if all
00183 /// the lineages went extinct and <paramref name="keepDeadLineages"/> is <see
      langword="false"/>.</returns>
00184         private static TreeNode UnlabelledTree(int leafCount, double birthRate, double deathRate, bool
      keepDeadLineages, out List<TreeNode> aliveLineages, CancellationToken cancellationToken)
00185         {
00186             List<TreeNode> leaves = new List<TreeNode>(leafCount);
00187 
00188             for (int i = 0; i < leafCount; i++)
00189             {
00190                 leaves.Add(new TreeNode(null) { Length = 0 });
00191             }
00192 
00193             aliveLineages = new List<TreeNode>(leaves);
00194 
00195             List<TreeNode> deadLeaves = new List<TreeNode>();
00196 
00197             while (leaves.Count > 1)
00198             {
00199                 cancellationToken.ThrowIfCancellationRequested();
00200 
00201                 double timeToNextEvent = Exponential.Sample(RandomTree.RandomNumberGenerator,
      leaves.Count * (birthRate + deathRate));
00202 
00203                 foreach (TreeNode leaf in leaves)
```

```
00204                 {
00205                     leaf.Length += timeToNextEvent;
00206                 }
00207 
00208                 bool isBirth = ContinuousUniform.Sample(RandomTree.RandomNumberGenerator, 0, 1) <
      birthRate / (birthRate + deathRate);
00209 
00210                 if (isBirth)
00211                 {
00212 
00213                     int index1 = RandomTree.RandomNumberGenerator.Next(0, leaves.Count);
00214                     TreeNode leaf1 = leaves[index1];
00215                     leaves.RemoveAt(index1);
00216 
00217                     int index2 = RandomTree.RandomNumberGenerator.Next(0, leaves.Count);
00218                     TreeNode leaf2 = leaves[index2];
00219                     leaves.RemoveAt(index2);
00220 
00221                     TreeNode newNode = new TreeNode(null) { Length = 0 };
00222                     newNode.Children.Add(leaf1);
00223                     newNode.Children.Add(leaf2);
00224                     leaf1.Parent = newNode;
00225                     leaf2.Parent = newNode;
00226 
00227                     leaves.Add(newNode);
00228                 }
00229                 else
00230                 {
00231                     TreeNode newLeaf = new TreeNode(null) { Length = 0 };
00232                     leaves.Add(newLeaf);
00233                     deadLeaves.Add(newLeaf);
00234                 }
00235             }
00236 
00237             leaves[0].Length = double.NaN;
00238 
00239             if (!keepDeadLineages)
00240             {
00241                 foreach (TreeNode leaf in deadLeaves)
00242                 {
00243                     leaves[0] = leaves[0].Prune(leaf, false);
00244                 }
00245             }
00246 
00247             return leaves[0];
00248         }
00249 
00250 /// <summary>
00251 /// Simulate a labelled birth-death tree, stopping when the number of lineages that are alive in the
      tree reaches a certain value.
00252 /// </summary>
00253 /// <param name="leafNames">The names for the terminal nodes of the tree.  Note that, if <paramref
      name="keepDeadLineages"/>
00254 /// is <see langword="true"/>, the actual number of leaves in the tree may be larger than this,
      because the leaves corresponding
00255 /// to dead lineages are kept (their names will be empty).  If all the lineages go extinct before the
      target number of lineages
00256 /// is reaced, the method will return a smaller tree (without any leaf names) if <paramref
      name="keepDeadLineages"/> is <see langword="true"/>,
00257 /// or <see langword="null"/> if it is <see langword="false"/>.</param>
00258 /// <param name="birthRate">The birth rate of the tree.</param>
00259 /// <param name="deathRate">The death rate of the tree.</param>
00260 /// <param name="keepDeadLineages">If this is <see langword="true"/>, dead lineages are kept in the
      tree.  If this is
00261 /// <see langword="false"/>, they are pruned from the tree.</param>
00262 /// <param name="constraint">A tree to constrain the sampling.  The tree produced by this method will
      be compatible with this tree.  The constraint tree can be multifurcating.
00263 /// Please note that, as the constraint is applied at every step while growing the tree, this will
      bias the sampled topology distribution.</param>
00264 /// <param name="cancellationToken">A <see cref="CancellationToken"/> used to cancel the simulation if
      it takes too long.</param>
00265 /// <returns>A <see cref="TreeNode"/> object containing the unlabelled birth-death tree, or <see
      langword="null"/> if all
00266 /// the lineages went extinct and <paramref name="keepDeadLineages"/> is <see
      langword="false"/>.</returns>
00267         public static TreeNode LabelledTree(IReadOnlyList<string> leafNames, double birthRate, double
      deathRate = 0, bool keepDeadLineages = false, TreeNode constraint = null, CancellationToken
      cancellationToken = default)
00268         {
00269             if (constraint == null)
00270             {
00271                 int leafCount = leafNames.Count;
00272 
00273                 List<TreeNode> leaves = new List<TreeNode>(leafCount);
00274 
00275                 for (int i = 0; i < leafCount; i++)
00276                 {
```

```
00277                   leaves.Add(new TreeNode(null) { Length = 0, Name = leafNames[i] });
00278               }
00279               List<TreeNode> deadLeaves = new List<TreeNode>();
00280
00281               while (leaves.Count > 1)
00282               {
00283                   cancellationToken.ThrowIfCancellationRequested();
00284
00285                   double timeToNextEvent = Exponential.Sample(RandomTree.RandomNumberGenerator,
      leaves.Count * (birthRate + deathRate));
00286
00287                   foreach (TreeNode leaf in leaves)
00288                   {
00289                       leaf.Length += timeToNextEvent;
00290                   }
00291
00292                   bool isBirth = ContinuousUniform.Sample(RandomTree.RandomNumberGenerator, 0, 1) <
      birthRate / (birthRate + deathRate);
00293
00294                   if (isBirth)
00295                   {
00296
00297                       int index1 = RandomTree.RandomNumberGenerator.Next(0, leaves.Count);
00298                       TreeNode leaf1 = leaves[index1];
00299                       leaves.RemoveAt(index1);
00300
00301                       int index2 = RandomTree.RandomNumberGenerator.Next(0, leaves.Count);
00302                       TreeNode leaf2 = leaves[index2];
00303                       leaves.RemoveAt(index2);
00304
00305                       TreeNode newNode = new TreeNode(null) { Length = 0 };
00306                       newNode.Children.Add(leaf1);
00307                       newNode.Children.Add(leaf2);
00308                       leaf1.Parent = newNode;
00309                       leaf2.Parent = newNode;
00310
00311                       leaves.Add(newNode);
00312                   }
00313                   else
00314                   {
00315                       TreeNode newLeaf = new TreeNode(null) { Length = 0 };
00316                       leaves.Add(newLeaf);
00317                       deadLeaves.Add(newLeaf);
00318                   }
00319               }
00320
00321               leaves[0].Length = double.NaN;
00322
00323               if (!keepDeadLineages)
00324               {
00325                   foreach (TreeNode leaf in deadLeaves)
00326                   {
00327                       leaves[0] = leaves[0].Prune(leaf, false);
00328                   }
00329               }
00330
00331               return leaves[0];
00332           }
00333           else
00334           {
00335               constraint = constraint.Clone();
00336
00337               List<string> availableNames = leafNames.ToList();
00338
00339               Dictionary<string, int> sequenceIndices = new Dictionary<string, int>();
00340               List<TreeNode> toBePruned = new List<TreeNode>();
00341
00342               foreach (TreeNode leaf in constraint.GetLeaves())
00343               {
00344                   if (!string.IsNullOrEmpty(leaf.Name))
00345                   {
00346                       int index = availableNames.IndexOf(leaf.Name);
00347                       if (index >= 0)
00348                       {
00349                           sequenceIndices[leaf.Name] = index;
00350                       }
00351                       else
00352                       {
00353                           toBePruned.Add(leaf);
00354                       }
00355                   }
00356                   else
00357                   {
00358                       toBePruned.Add(leaf);
00359                   }
00360               }
00361
```

```
00362                 for (int i = 0; i < toBePruned.Count; i++)
00363                 {
00364                     constraint = constraint.Prune(toBePruned[i], false);
00365                 }
00366 
00367                 if (constraint == null || constraint.GetLeaves().Count == 1)
00368                 {
00369                     return LabelledTree(leafNames, birthRate, deathRate, keepDeadLineages, null,
      cancellationToken);
00370                 }
00371 
00372                 List<int[][]> splits = NeighborJoining.GetSplits(constraint, sequenceIndices);
00373 
00374                 List<TreeNode> leaves = new List<TreeNode>(leafNames.Count);
00375                 List<HashSet<int» underlyingLeaves = new List<HashSet<int»();
00376                 HashSet<int> allLeaves = new HashSet<int>();
00377 
00378                 for (int i = 0; i < leafNames.Count; i++)
00379                 {
00380                     leaves.Add(new TreeNode(null) { Length = 0, Name = leafNames[i] });
00381 
00382                     if (sequenceIndices.TryGetValue(leafNames[i], out int index))
00383                     {
00384                         underlyingLeaves.Add(new HashSet<int>() { index });
00385                         allLeaves.Add(index);
00386                     }
00387                     else
00388                     {
00389                         underlyingLeaves.Add(new HashSet<int>());
00390                     }
00391                 }
00392 
00393                 allLeaves.Add(-1);
00394 
00395                 int leafCount = leafNames.Count;
00396 
00397                 List<TreeNode> deadLeaves = new List<TreeNode>();
00398 
00399                 while (leaves.Count > 1)
00400                 {
00401                     cancellationToken.ThrowIfCancellationRequested();
00402 
00403                     double timeToNextEvent = Exponential.Sample(RandomTree.RandomNumberGenerator,
      leaves.Count * (birthRate + deathRate));
00404 
00405                     foreach (TreeNode leaf in leaves)
00406                     {
00407                         leaf.Length += timeToNextEvent;
00408                     }
00409 
00410                     bool isBirth = ContinuousUniform.Sample(RandomTree.RandomNumberGenerator, 0, 1) <
      birthRate / (birthRate + deathRate);
00411 
00412                     if (isBirth)
00413                     {
00414                         int index1, index2;
00415                         HashSet<int> potentialSplitLeft, potentialSplitRight;
00416 
00417                         List<(int, int)> availablePairs = (from el in Enumerable.Range(0,
      leaves.Count) select (from el2 in Enumerable.Range(0, el) select (el, el2))).Aggregate(new List<(int,
      int)>(), (a, b) => { a.AddRange(b); return a; });
00418 
00419                         do
00420                         {
00421                             (index1, index2) = availablePairs.Sample();
00422 
00423                             potentialSplitRight = new HashSet<int>(underlyingLeaves[index1]);
00424                             potentialSplitRight.UnionWith(underlyingLeaves[index2]);
00425 
00426                             potentialSplitLeft = new HashSet<int>(allLeaves);
00427                             potentialSplitLeft.ExceptWith(potentialSplitRight);
00428 
00429                         } while (!NeighborJoining.IsCompatible(potentialSplitLeft,
      potentialSplitRight, splits));
00430 
00431 
00432                         TreeNode leaf1 = leaves[index1];
00433                         TreeNode leaf2 = leaves[index2];
00434 
00435                         underlyingLeaves[Math.Min(index1,
      index2)].UnionWith(underlyingLeaves[Math.Max(index1, index2)]);
00436                         underlyingLeaves.RemoveAt(Math.Max(index1, index2));
00437 
00438                         TreeNode newNode = new TreeNode(null) { Length = 0 };
00439                         newNode.Children.Add(leaf1);
00440                         newNode.Children.Add(leaf2);
00441                         leaf1.Parent = newNode;
```

```
00442                           leaf2.Parent = newNode;
00443
00444                           leaves.RemoveAt(Math.Max(index1, index2));
00445                           leaves[Math.Min(index1, index2)] = newNode;
00446                       }
00447                       else
00448                       {
00449                           TreeNode newLeaf = new TreeNode(null) { Length = 0 };
00450                           leaves.Add(newLeaf);
00451                           deadLeaves.Add(newLeaf);
00452                           underlyingLeaves.Add(new HashSet<int>());
00453                       }
00454                   }
00455
00456                   leaves[0].Length = double.NaN;
00457
00458                   if (!keepDeadLineages)
00459                   {
00460                       foreach (TreeNode leaf in deadLeaves)
00461                       {
00462                           leaves[0] = leaves[0].Prune(leaf, false);
00463                       }
00464                   }
00465
00466                   return leaves[0];
00467               }
00468           }
00469
00470 /// <summary>
00471 /// Simulate a labelled birth-death tree, stopping when the number of lineages that are alive in the
      tree reaches a certain value.
00472 /// </summary>
00473 /// <param name="leafCount">The final number of lineages that are alive in the tree (their names will
      be in the form <c>t1, t2, ..., tN</c>,
00474 /// where <c>N</c> is <paramref name="leafCount"/>).  Note that, if <paramref
      name="keepDeadLineages"/>
00475 /// is <see langword="true"/>, the actual number of leaves in the tree may be larger than this,
      because the leaves corresponding
00476 /// to dead lineages are kept (their names will be empty).  If all the lineages go extinct before the
      target number of lineages
00477 /// is reaced, the method will return a smaller tree (without any leaf names) if <paramref
      name="keepDeadLineages"/> is <see langword="true"/>,
00478 /// or <see langword="null"/> if it is <see langword="false"/>.</param>
00479 /// <param name="birthRate">The birth rate of the tree.</param>
00480 /// <param name="deathRate">The death rate of the tree.</param>
00481 /// <param name="keepDeadLineages">If this is <see langword="true"/>, dead lineages are kept in the
      tree.  If this is
00482 /// <see langword="false"/>, they are pruned from the tree.</param>
00483 /// <param name="constraint">A tree to constrain the sampling.  The tree produced by this method will
      be compatible with this tree.  The constraint tree can be multifurcating.
00484 /// Please note that, as the constraint is applied at every step while growing the tree, this will
      bias the sampled topology distribution.</param>
00485 /// <param name="cancellationToken">A <see cref="CancellationToken"/> used to cancel the simulation if
      it takes too long.</param>
00486 /// <returns>A <see cref="TreeNode"/> object containing the unlabelled birth-death tree, or <see
      langword="null"/> if all
00487 /// the lineages went extinct and <paramref name="keepDeadLineages"/> is <see
      langword="false"/>.</returns>
00488
00489           public static TreeNode LabelledTree(int leafCount, double birthRate, double deathRate = 0,
      bool keepDeadLineages = false, TreeNode constraint = null, CancellationToken cancellationToken =
      default)
00490           {
00491               string[] leafNames = new string[leafCount];
00492
00493               for (int i = 0; i < leafCount; i++)
00494               {
00495                   leafNames[i] = "t" + (i + 1).ToString();
00496               }
00497
00498               return LabelledTree(leafNames, birthRate, deathRate, keepDeadLineages, constraint,
      cancellationToken);
00499           }
00500       }
00501 }
```

8.17 CoalescentTree.cs

```
00001 using MathNet.Numerics.Distributions;
00002 using System;
00003 using System.Collections.Generic;
00004 using System.Linq;
00005 using System.Text;
```

```
00006
00007 namespace PhyloTree.TreeBuilding
00008 {
00009 /// <summary>
00010 /// Contains methods to simulate coalescent trees.
00011 /// </summary>
00012     public static class CoalescentTree
00013     {
00014 /// <summary>
00015 /// Simulate an unlabelled coalescent tree.
00016 /// </summary>
00017 /// <param name="leafCount">The number of terminal nodes in the tree.</param>
00018 /// <returns>A <see cref="TreeNode"/> object containing the unlabelled coalescent tree.</returns>
00019         public static TreeNode UnlabelledTree(int leafCount)
00020         {
00021             string[] leafNames = new string[leafCount];
00022
00023             for (int i = 0; i < leafNames.Length; i++)
00024             {
00025                 leafNames[i] = "";
00026             }
00027
00028             return LabelledTree(leafNames);
00029         }
00030
00031 /// <summary>
00032 /// Simulate a labelled coalescent tree with the supplied tip labels.
00033 /// </summary>
00034 /// <param name="leafNames">The labels for the terminal nodes of the tree.</param>
00035 /// <param name="constraint">A tree to constrain the sampling.  The tree produced by this method will
      be compatible with this tree.  The constraint tree can be multifurcating.
00036 /// Please note that, as the constraint is applied at every step while growing the tree, this will
      bias the sampled topology distribution.</param>
00037 /// <returns>A <see cref="TreeNode"/> object containing the labelled coalescent tree.</returns>
00038         public static TreeNode LabelledTree(IReadOnlyList<string> leafNames, TreeNode constraint =
      null)
00039         {
00040             if (constraint == null)
00041             {
00042                 int leafCount = leafNames.Count;
00043
00044                 double[] coalescenceTimes = new double[leafCount - 1];
00045
00046                 for (int k = leafCount; k > 1; k--)
00047                 {
00048                     coalescenceTimes[leafCount - k] = Exponential.Sample(k * (k - 1) * 0.25);
00049                 }
00050
00051                 List<TreeNode> leaves = new List<TreeNode>(leafCount);
00052
00053                 for (int i = 0; i < leafCount; i++)
00054                 {
00055                     leaves.Add(new TreeNode(null) { Length = 0, Name = leafNames[i] });
00056                 }
00057
00058                 for (int i = 0; i < coalescenceTimes.Length; i++)
00059                 {
00060                     foreach (TreeNode leaf in leaves)
00061                     {
00062                         leaf.Length += coalescenceTimes[i];
00063                     }
00064
00065                     int index1 = RandomTree.RandomNumberGenerator.Next(0, leaves.Count);
00066                     TreeNode leaf1 = leaves[index1];
00067                     leaves.RemoveAt(index1);
00068
00069                     int index2 = RandomTree.RandomNumberGenerator.Next(0, leaves.Count);
00070                     TreeNode leaf2 = leaves[index2];
00071                     leaves.RemoveAt(index2);
00072
00073                     TreeNode newNode = new TreeNode(null) { Length = 0 };
00074                     newNode.Children.Add(leaf1);
00075                     newNode.Children.Add(leaf2);
00076                     leaf1.Parent = newNode;
00077                     leaf2.Parent = newNode;
00078
00079                     leaves.Add(newNode);
00080                 }
00081
00082                 leaves[0].Length = double.NaN;
00083
00084                 return leaves[0];
00085             }
00086             else
00087             {
00088                 constraint = constraint.Clone();
00089
```

```
00090                 List<string> availableNames = leafNames.ToList();
00091
00092                 Dictionary<string, int> sequenceIndices = new Dictionary<string, int>();
00093                 List<TreeNode> toBePruned = new List<TreeNode>();
00094
00095                 foreach (TreeNode leaf in constraint.GetLeaves())
00096                 {
00097                     if (!string.IsNullOrEmpty(leaf.Name))
00098                     {
00099                         int index = availableNames.IndexOf(leaf.Name);
00100                         if (index >= 0)
00101                         {
00102                             sequenceIndices[leaf.Name] = index;
00103                         }
00104                         else
00105                         {
00106                             toBePruned.Add(leaf);
00107                         }
00108                     }
00109                     else
00110                     {
00111                         toBePruned.Add(leaf);
00112                     }
00113                 }
00114
00115                 for (int i = 0; i < toBePruned.Count; i++)
00116                 {
00117                     constraint = constraint.Prune(toBePruned[i], false);
00118                 }
00119
00120                 if (constraint == null || constraint.GetLeaves().Count == 1)
00121                 {
00122                     return LabelledTree(leafNames, null);
00123                 }
00124
00125                 List<int[][]> splits = NeighborJoining.GetSplits(constraint, sequenceIndices);
00126
00127                 List<TreeNode> leaves = new List<TreeNode>(leafNames.Count);
00128                 List<HashSet<int» underlyingLeaves = new List<HashSet<int»();
00129                 HashSet<int> allLeaves = new HashSet<int>();
00130
00131                 for (int i = 0; i < leafNames.Count; i++)
00132                 {
00133                     leaves.Add(new TreeNode(null) { Length = 0, Name = leafNames[i] });
00134
00135                     if (sequenceIndices.TryGetValue(leafNames[i], out int index))
00136                     {
00137                         underlyingLeaves.Add(new HashSet<int>() { index });
00138                         allLeaves.Add(index);
00139                     }
00140                     else
00141                     {
00142                         underlyingLeaves.Add(new HashSet<int>());
00143                     }
00144                 }
00145
00146                 allLeaves.Add(-1);
00147
00148                 int leafCount = leafNames.Count;
00149
00150                 double[] coalescenceTimes = new double[leafCount - 1];
00151
00152                 for (int k = leafCount; k > 1; k--)
00153                 {
00154                     coalescenceTimes[leafCount - k] = Exponential.Sample(k * (k - 1) * 0.25);
00155                 }
00156
00157                 for (int i = 0; i < coalescenceTimes.Length; i++)
00158                 {
00159                     foreach (TreeNode leaf in leaves)
00160                     {
00161                         leaf.Length += coalescenceTimes[i];
00162                     }
00163
00164
00165                     int index1, index2;
00166                     HashSet<int> potentialSplitLeft, potentialSplitRight;
00167
00168                     List<(int, int)> availablePairs = (from el in Enumerable.Range(0, leaves.Count)
      select (from el2 in Enumerable.Range(0, el) select (el, el2))).Aggregate(new List<(int, int)>(), (a,
      b) => { a.AddRange(b); return a; });
00169
00170                     do
00171                     {
00172                         (index1, index2) = availablePairs.Sample();
00173
00174                         potentialSplitRight = new HashSet<int>(underlyingLeaves[index1]);
```

```
00175                         potentialSplitRight.UnionWith(underlyingLeaves[index2]);
00176
00177                         potentialSplitLeft = new HashSet<int>(allLeaves);
00178                         potentialSplitLeft.ExceptWith(potentialSplitRight);
00179
00180                     } while (!NeighborJoining.IsCompatible(potentialSplitLeft, potentialSplitRight,
      splits));
00181
00182
00183                     TreeNode leaf1 = leaves[index1];
00184                     TreeNode leaf2 = leaves[index2];
00185
00186                     underlyingLeaves[Math.Min(index1,
      index2)].UnionWith(underlyingLeaves[Math.Max(index1, index2)]);
00187                     underlyingLeaves.RemoveAt(Math.Max(index1, index2));
00188
00189                     TreeNode newNode = new TreeNode(null) { Length = 0 };
00190                     newNode.Children.Add(leaf1);
00191                     newNode.Children.Add(leaf2);
00192                     leaf1.Parent = newNode;
00193                     leaf2.Parent = newNode;
00194
00195                     leaves.RemoveAt(Math.Max(index1, index2));
00196                     leaves[Math.Min(index1, index2)] = newNode;
00197                 }
00198
00199                 leaves[0].Length = double.NaN;
00200
00201                 return leaves[0];
00202             }
00203         }
00204
00205 /// <summary>
00206 /// Simulate a labelled coalescent tree with the specified number of terminal nodes.
00207 /// </summary>
00208 /// <param name="leafCount">The number of terminal nodes in the tree.  Their names will be in the form
      <c>t1, t2, ..., tN</c>, where <c>N</c> is <paramref name="leafCount"/>.</param>
00209 /// <param name="constraint">A tree to constrain the sampling.  The tree produced by this method will
      be compatible with this tree.  The constraint tree can be multifurcating.
00210 /// Please note that, as the constraint is applied at every step while growing the tree, this will
      bias the sampled topology distribution.</param>
00211 /// <returns>A <see cref="TreeNode"/> object containing the labelled coalescent tree.</returns>
00212         public static TreeNode LabelledTree(int leafCount, TreeNode constraint = null)
00213         {
00214             string[] leafNames = new string[leafCount];
00215
00216             for (int i = 0; i < leafNames.Length; i++)
00217             {
00218                 leafNames[i] = "t" + (i + 1).ToString();
00219             }
00220
00221             return LabelledTree(leafNames, constraint);
00222         }
00223     }
00224 }
```

8.18 DistanceMatrix.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Linq;
00004 using System.Threading.Tasks;
00005
00006 namespace PhyloTree.TreeBuilding
00007 {
00008 /// <summary>
00009 /// The sequence evolution model used to compute the distance matrix from the alignment.
00010 /// </summary>
00011     public enum EvolutionModel
00012     {
00013 /// <summary>
00014 /// Normalised Hamming distance (proportion of different nucleotides/amino acids).
00015 /// </summary>
00016         Hamming = 0,
00017
00018 /// <summary>
00019 /// Jukes-Cantor model (assuming equal nucleotide/amino acid frequencies and substitution rates).
00020 /// </summary>
00021         JukesCantor = 1,
00022
00023 /// <summary>
00024 /// For DNA alignments, this represents the Kimura 1980 model (assuming equal base frequencies, and
      unequal transition/transversion rates).
```

```
00025 /// For protein alignments, this represents the Kimura 1983 model that approximates PAM distances based only on the fraction of differing amino acids.
00026 /// </summary>
00027         Kimura = 2,
00028
00029 /// <summary>
00030 /// Only applicable for DNA alignments.  This represents the GTR distance, computed as in Waddell &amp;amp; Steel, 1997 (doi:  10.1006/mpev.1997.0452).
00031 /// Note that since this process involves matrix diagonalisation, it is about an order of magnitude slower than other distance metrics.
00032 /// </summary>
00033         GTR = 3,
00034
00035 /// <summary>
00036 /// Only applicable for Protein alignments.  This represents a distance computed using the Scoredist algorithm with the BLOSUM62 matrix, as in
00037 /// Sonnhammer &amp;amp; Hollich, 2005 (doi:  10.1186/1471-2105-6-108).  Note that in this case U, O and J are treated as gaps.
00038 /// </summary>
00039         BLOSUM62 = 4
00040     }
00041
00042 /// <summary>
00043 /// The kind of sequences in the alignment.
00044 /// </summary>
00045     public enum AlignmentType
00046     {
00047 /// <summary>
00048 /// DNA sequences.
00049 /// </summary>
00050         DNA = 0,
00051
00052 /// <summary>
00053 /// Protein sequences.
00054 /// </summary>
00055         Protein = 1,
00056
00057 /// <summary>
00058 /// The kind of sequences will be determined based on the first sequence.
00059 /// </summary>
00060         Autodetect = 2
00061     }
00062
00063 /// <summary>
00064 /// Contains methods to compute distance matrices.
00065 /// </summary>
00066     public static partial class DistanceMatrix
00067     {
00068         private static readonly byte[][] DNAMatchMatrixJCK80;
00069         private static readonly uint[][] DNAMatchMatrixGTR;
00070         private static readonly byte[][] ProteinMatchMatrixJCK83;
00071         private static readonly byte[][] ProteinMatchMatrixBLOSUM62;
00072
00073         static DistanceMatrix()
00074         {
00075             DNAMatchMatrixJCK80 = new byte[256][];
00076             BuildDNAMatchMatrixJCK80();
00077
00078             DNAMatchMatrixGTR = new uint[256][];
00079             BuildDNAMatchMatrixGTR();
00080
00081             ProteinMatchMatrixJCK83 = new byte[784][];
00082             BuildProteinMatchMatrixJCK83();
00083
00084             ProteinMatchMatrixBLOSUM62 = new byte[784][];
00085             BuildProteinMatchMatrixBLOSUM62();
00086         }
00087
00088 /// <summary>
00089 /// Post-process a distance matrix removing invalid entries and replacing them with twice the maximum distance in the matrix.
00090 /// </summary>
00091 /// <param name="distMat">The matrix to post-process.</param>
00092         private static void PostProcessDistanceMatrix(float[][] distMat)
00093         {
00094             List<(int, int)> indicesToAdjust = new List<(int, int)>();
00095
00096             float maxDist = float.MinValue;
00097
00098             for (int i = 0; i < distMat.Length; i++)
00099             {
00100                 for (int j = 0; j < distMat[i].Length; j++)
00101                 {
00102                     if (!float.IsFinite(distMat[i][j]) || distMat[i][j] < 0)
00103                     {
00104                         indicesToAdjust.Add((i, j));
00105                     }
```

```
00106                     else
00107                     {
00108                         maxDist = Math.Max(distMat[i][j], maxDist);
00109                     }
00110                 }
00111             }
00112 
00113             for (int i = 0; i < indicesToAdjust.Count; i++)
00114             {
00115                 distMat[indicesToAdjust[i].Item1][indicesToAdjust[i].Item2] = 2 * maxDist;
00116             }
00117         }
00118 
00119 /// <summary>
00120 /// Build a distance matrix from aligned DNA sequences stored as a <see cref="T:byte[]"/> array where each <see cref="byte"/> corresponds to three positions.
00121 /// </summary>
00122 /// <param name="sequences">The aligned sequences.</param>
00123 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00124 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime decide.</param>
00125 /// <param name="progressCallback">A method used to report progress.</param>
00126 /// <returns>A <see cref="T:float[][]"/> jagged array containing the lower-triangular distance matrix.</returns>
00127         public static float[][] BuildFromAlignment(IReadOnlyList<byte[]> sequences, EvolutionModel evolutionModel = EvolutionModel.Kimura, int numCores = -1, Action<double> progressCallback = null)
00128         {
00129             float[][] allocatedMatrix = new float[sequences.Count][];
00130 
00131             for (int i = 0; i < sequences.Count; i++)
00132             {
00133                 allocatedMatrix[i] = new float[i];
00134             }
00135 
00136             BuildFromAlignment(allocatedMatrix, sequences, evolutionModel, numCores, progressCallback);
00137 
00138             return allocatedMatrix;
00139         }
00140 
00141 /// <summary>
00142 /// Build a distance matrix from aligned protein sequences stored as a <see cref="T:ushort[]"/> array where each <see cref="ushort"/> corresponds to two positions.
00143 /// </summary>
00144 /// <param name="sequences">The aligned sequences.</param>
00145 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00146 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime decide.</param>
00147 /// <param name="progressCallback">A method used to report progress.</param>
00148 /// <returns>A <see cref="T:float[][]"/> jagged array containing the lower-triangular distance matrix.</returns>
00149         public static float[][] BuildFromAlignment(IReadOnlyList<ushort[]> sequences, EvolutionModel evolutionModel = EvolutionModel.Kimura, int numCores = -1, Action<double> progressCallback = null)
00150         {
00151             float[][] allocatedMatrix = new float[sequences.Count][];
00152 
00153             for (int i = 0; i < sequences.Count; i++)
00154             {
00155                 allocatedMatrix[i] = new float[i];
00156             }
00157 
00158             BuildFromAlignment(allocatedMatrix, sequences, evolutionModel, numCores, progressCallback);
00159 
00160             return allocatedMatrix;
00161         }
00162 
00163 /// <summary>
00164 /// Build a distance matrix from aligned DNA sequences stored as a <see cref="T:byte[]"/> array where each <see cref="byte"/> corresponds to three positions.
00165 /// </summary>
00166 /// <param name="allocatedMatrix">A pre-allocated <see cref="T:float[][]"/> jagged array that will contain the lower-triangular distance matrix.</param>
00167 /// <param name="sequences">The aligned sequences.</param>
00168 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00169 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime decide.</param>
00170 /// <param name="progressCallback">A method used to report progress.</param>
00171         public static void BuildFromAlignment(float[][] allocatedMatrix, IReadOnlyList<byte[]> sequences, EvolutionModel evolutionModel = EvolutionModel.Kimura, int numCores = -1, Action<double> progressCallback = null)
00172         {
00173             if (numCores <= 0)
00174             {
00175                 numCores = -1;
00176             }
00177 
```

```
00178             switch (evolutionModel)
00179             {
00180                 case EvolutionModel.Hamming:
00181                     ComputeDistanceMatrixHamming(sequences, numCores, progressCallback,
      allocatedMatrix);
00182                     break;
00183 
00184                 case EvolutionModel.JukesCantor:
00185                     ComputeDistanceMatrixJC(sequences, numCores, progressCallback, allocatedMatrix);
00186                     break;
00187 
00188                 case EvolutionModel.Kimura:
00189                     ComputeDistanceMatrixK80(sequences, numCores, progressCallback, allocatedMatrix);
00190                     break;
00191 
00192                 case EvolutionModel.GTR:
00193                     ComputeDistanceMatrixGTR(sequences, numCores, progressCallback, allocatedMatrix);
00194                     break;
00195 
00196                 default:
00197                     throw new ArgumentException(evolutionModel.ToString() + "is not a valid
      evolutionary model for DNA sequences!", nameof(evolutionModel));
00198             }
00199 
00200             PostProcessDistanceMatrix(allocatedMatrix);
00201         }
00202 
00203 /// <summary>
00204 /// Build a distance matrix from aligned protein sequences stored as a <see cref="T:ushort[]"/> array
      where each <see cref="ushort"/> corresponds to two positions.
00205 /// </summary>
00206 /// <param name="allocatedMatrix">A pre-allocated <see cref="T:float[][]"/> jagged array that will
      contain the lower-triangular distance matrix.</param>
00207 /// <param name="sequences">The aligned sequences.</param>
00208 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00209 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
      decide.</param>
00210 /// <param name="progressCallback">A method used to report progress.</param>
00211         public static void BuildFromAlignment(float[][] allocatedMatrix, IReadOnlyList<ushort[]>
      sequences, EvolutionModel evolutionModel = EvolutionModel.Kimura, int numCores = 0, Action<double>
      progressCallback = null)
00212         {
00213             if (numCores <= 0)
00214             {
00215                 numCores = -1;
00216             }
00217 
00218             switch (evolutionModel)
00219             {
00220                 case EvolutionModel.Hamming:
00221                     ComputeDistanceMatrixHamming(sequences, numCores, progressCallback,
      allocatedMatrix);
00222                     break;
00223 
00224                 case EvolutionModel.JukesCantor:
00225                     ComputeDistanceMatrixJC(sequences, numCores, progressCallback, allocatedMatrix);
00226                     break;
00227 
00228                 case EvolutionModel.Kimura:
00229                     ComputeDistanceMatrixK83(sequences, numCores, progressCallback, allocatedMatrix);
00230                     break;
00231 
00232                 case EvolutionModel.BLOSUM62:
00233                     ComputeDistanceMatrixBLOSUM62(sequences, numCores, progressCallback,
      allocatedMatrix);
00234                     break;
00235 
00236                 default:
00237                     throw new ArgumentException(evolutionModel.ToString() + "is not a valid
      evolutionary model for amino acid sequences!", nameof(evolutionModel));
00238             }
00239 
00240             PostProcessDistanceMatrix(allocatedMatrix);
00241         }
00242 
00243 /// <summary>
00244 /// Build a distance matrix from aligned DNA or protein sequences.
00245 /// </summary>
00246 /// <param name="allocatedMatrix">A pre-allocated <see cref="T:float[][]"/> jagged array that will
      contain the lower-triangular distance matrix.</param>
00247 /// <param name="sequences">The aligned sequences.</param>
00248 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00249 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00250 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
      decide.</param>
00251 /// <param name="progressCallback">A method used to report progress.</param>
00252         public static void BuildFromAlignment(float[][] allocatedMatrix, IReadOnlyList<string>
```

```
      sequences, AlignmentType alignmentType = AlignmentType.Autodetect, EvolutionModel evolutionModel =
      EvolutionModel.Kimura, int numCores = 0, Action<double> progressCallback = null)
00253         {
00254             if (alignmentType == AlignmentType.Autodetect)
00255             {
00256                 alignmentType = AlignmentType.DNA;
00257 
00258                 foreach (string seq in sequences)
00259                 {
00260                     foreach (char c in seq)
00261                     {
00262                         if (!(c < 65 ||
00263                               c == 'A' ||
00264                               c == 'a' ||
00265                               c == 'C' ||
00266                               c == 'c' ||
00267                               c == 'G' ||
00268                               c == 'g' ||
00269                               c == 'T' ||
00270                               c == 't' ||
00271                               c == 'U' ||
00272                               c == 'u' ||
00273                               c == 'N' ||
00274                               c == 'n'))
00275                         {
00276                             alignmentType = AlignmentType.Protein;
00277                             break;
00278                         }
00279                     }
00280 
00281                     break;
00282                 }
00283             }
00284 
00285             int length = sequences[0].Length;
00286 
00287             if (alignmentType == AlignmentType.DNA)
00288             {
00289                 List<byte[]> sequenceBytes = new List<byte[]>(sequences.Count);
00290 
00291                 for (int i = 0; i < sequences.Count; i++)
00292                 {
00293                     if (length != sequences[i].Length)
00294                     {
00295                         throw new ArgumentOutOfRangeException("Not all of the sequences have the same
      length!", nameof(sequences));
00296                     }
00297 
00298 
00299                     sequenceBytes.Add(ConvertDNASequence(sequences[i]));
00300                 }
00301 
00302                 BuildFromAlignment(allocatedMatrix, sequenceBytes, evolutionModel, numCores,
      progressCallback);
00303             }
00304             else if (alignmentType == AlignmentType.Protein)
00305             {
00306                 List<ushort[]> sequenceUShorts = new List<ushort[]>(sequences.Count);
00307 
00308                 for (int i = 0; i < sequences.Count; i++)
00309                 {
00310                     if (length != sequences[i].Length)
00311                     {
00312                         throw new ArgumentOutOfRangeException("Not all of the sequences have the same
      length!", nameof(sequences));
00313                     }
00314 
00315 
00316                     sequenceUShorts.Add(ConvertProteinSequence(sequences[i]));
00317                 }
00318 
00319                 BuildFromAlignment(allocatedMatrix, sequenceUShorts, evolutionModel, numCores,
      progressCallback);
00320             }
00321         }
00322 
00323 /// <summary>
00324 /// Build a distance matrix from aligned DNA or protein sequences.
00325 /// </summary>
00326 /// <param name="sequences">The aligned sequences.</param>
00327 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00328 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00329 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
      decide.</param>
00330 /// <param name="progressCallback">A method used to report progress.</param>
00331 /// <returns>A <see cref="T:float[][]"/> jagged array containing the lower-triangular distance
      matrix.</returns>
```

```
00332         public static float[][] BuildFromAlignment(IReadOnlyList<string> sequences, AlignmentType
      alignmentType = AlignmentType.Autodetect, EvolutionModel evolutionModel = EvolutionModel.Kimura, int
      numCores = 0, Action<double> progressCallback = null)
00333         {
00334             float[][] allocatedMatrix = new float[sequences.Count][];
00335 
00336             for (int i = 0; i < sequences.Count; i++)
00337             {
00338                 allocatedMatrix[i] = new float[i];
00339             }
00340 
00341             BuildFromAlignment(allocatedMatrix, sequences, alignmentType, evolutionModel, numCores,
      progressCallback);
00342 
00343             return allocatedMatrix;
00344         }
00345 
00346 /// <summary>
00347 /// Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
00348 /// </summary>
00349 /// <param name="allocatedMatrix">A pre-allocated <see cref="T:float[][]"/> jagged array that will
      contain the lower-triangular distance matrix.</param>
00350 /// <param name="sequences">The aligned sequences.  These will be resampled randomly.</param>
00351 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00352 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00353 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
      decide.</param>
00354 /// <param name="progressCallback">A method used to report progress.</param>
00355         public static void BootstrapReplicateFromAlignment(float[][] allocatedMatrix,
      IReadOnlyList<string> sequences, AlignmentType alignmentType = AlignmentType.Autodetect,
      EvolutionModel evolutionModel = EvolutionModel.Kimura, int numCores = 0, Action<double>
      progressCallback = null)
00356         {
00357             if (alignmentType == AlignmentType.Autodetect)
00358             {
00359                 alignmentType = AlignmentType.DNA;
00360 
00361                 foreach (string seq in sequences)
00362                 {
00363                     foreach (char c in seq)
00364                     {
00365                         if (!(c < 65 ||
00366                               c == 'A' ||
00367                               c == 'a' ||
00368                               c == 'C' ||
00369                               c == 'c' ||
00370                               c == 'G' ||
00371                               c == 'g' ||
00372                               c == 'T' ||
00373                               c == 't' ||
00374                               c == 'U' ||
00375                               c == 'u' ||
00376                               c == 'N' ||
00377                               c == 'n'))
00378                         {
00379                             alignmentType = AlignmentType.Protein;
00380                             break;
00381                         }
00382                     }
00383 
00384                     break;
00385                 }
00386             }
00387 
00388             if (alignmentType == AlignmentType.DNA)
00389             {
00390                 List<byte[]> sequenceBytes = BootstrapDNASequences(sequences);
00391 
00392                 BuildFromAlignment(allocatedMatrix, sequenceBytes, evolutionModel, numCores,
      progressCallback);
00393             }
00394             else if (alignmentType == AlignmentType.Protein)
00395             {
00396                 List<ushort[]> sequenceUShorts = BootstrapProteinSequences(sequences);
00397 
00398                 BuildFromAlignment(allocatedMatrix, sequenceUShorts, evolutionModel, numCores,
      progressCallback);
00399             }
00400         }
00401 
00402 /// <summary>
00403 /// Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
00404 /// </summary>
00405 /// <param name="sequences">The aligned sequences.  These will be resampled randomly.</param>
00406 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00407 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00408 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
```

```
decide.</param>
00409 /// <param name="progressCallback">A method used to report progress.</param>
00410 /// <returns>A <see cref="T:float[][]"/> jagged array containing the lower-triangular distance
matrix.</returns>
00411         public static float[][] BootstrapReplicateFromAlignment(IReadOnlyList<string> sequences,
AlignmentType alignmentType = AlignmentType.Autodetect, EvolutionModel evolutionModel =
EvolutionModel.Kimura, int numCores = 0, Action<double> progressCallback = null)
00412         {
00413             float[][] allocatedMatrix = new float[sequences.Count][];
00414 
00415             for (int i = 0; i < sequences.Count; i++)
00416             {
00417                 allocatedMatrix[i] = new float[i];
00418             }
00419 
00420             BootstrapReplicateFromAlignment(allocatedMatrix, sequences, alignmentType, evolutionModel,
numCores, progressCallback);
00421 
00422             return allocatedMatrix;
00423         }
00424 
00425 /// <summary>
00426 /// Build a distance matrix from aligned DNA or protein sequences.
00427 /// </summary>
00428 /// <param name="alignment">The aligned sequences.</param>
00429 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00430 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00431 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
decide.</param>
00432 /// <param name="progressCallback">A method used to report progress.</param>
00433 /// <returns>A <see cref="T:float[][]"/> jagged array containing the lower-triangular distance
matrix.</returns>
00434         public static float[][] BuildFromAlignment(Dictionary<string, string> alignment, AlignmentType
alignmentType = AlignmentType.Autodetect, EvolutionModel evolutionModel = EvolutionModel.Kimura, int
numCores = 0, Action<double> progressCallback = null)
00435         {
00436             return BuildFromAlignment(alignment.Values.ToList(), alignmentType, evolutionModel,
numCores, progressCallback);
00437         }
00438 
00439 /// <summary>
00440 /// Build a distance matrix from aligned DNA or protein sequences.
00441 /// </summary>
00442 /// <param name="allocatedMatrix">A pre-allocated <see cref="T:float[][]"/> jagged array that will
contain the lower-triangular distance matrix.</param>
00443 /// <param name="alignment">The aligned sequences.</param>
00444 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00445 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00446 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
decide.</param>
00447 /// <param name="progressCallback">A method used to report progress.</param>
00448         public static void BuildFromAlignment(float[][] allocatedMatrix, Dictionary<string, string>
alignment, AlignmentType alignmentType = AlignmentType.Autodetect, EvolutionModel evolutionModel =
EvolutionModel.Kimura, int numCores = 0, Action<double> progressCallback = null)
00449         {
00450             BuildFromAlignment(allocatedMatrix, alignment.Values.ToList(), alignmentType,
evolutionModel, numCores, progressCallback);
00451         }
00452 
00453 /// <summary>
00454 /// Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
00455 /// </summary>
00456 /// <param name="alignment">The aligned sequences.  These will be resampled randomly.</param>
00457 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00458 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00459 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
decide.</param>
00460 /// <param name="progressCallback">A method used to report progress.</param>
00461 /// <returns>A <see cref="T:float[][]"/> jagged array containing the lower-triangular distance
matrix.</returns>
00462         public static float[][] BootstrapReplicateFromAlignment(Dictionary<string, string> alignment,
AlignmentType alignmentType = AlignmentType.Autodetect, EvolutionModel evolutionModel =
EvolutionModel.Kimura, int numCores = 0, Action<double> progressCallback = null)
00463         {
00464             return BootstrapReplicateFromAlignment(alignment.Values.ToList(), alignmentType,
evolutionModel, numCores, progressCallback);
00465         }
00466 
00467 /// <summary>
00468 /// Build a bootstrap replicate of a distance matrix from aligned DNA or protein sequences.
00469 /// </summary>
00470 /// <param name="allocatedMatrix">A pre-allocated <see cref="T:float[][]"/> jagged array that will
contain the lower-triangular distance matrix.</param>
00471 /// <param name="alignment">The aligned sequences.  These will be resampled randomly.</param>
00472 /// <param name="alignmentType">The type of sequences in the alignment.</param>
00473 /// <param name="evolutionModel">The evolutionary model to use to compute the distance matrix.</param>
00474 /// <param name="numCores">The maximum number of threads to use, or -1 to let the runtime
```

```
decide.</param>
00475 /// <param name="progressCallback">A method used to report progress.</param>
00476         public static void BootstrapReplicateFromAlignment(float[][] allocatedMatrix,
Dictionary<string, string> alignment, AlignmentType alignmentType = AlignmentType.Autodetect,
EvolutionModel evolutionModel = EvolutionModel.Kimura, int numCores = 0, Action<double>
progressCallback = null)
00477         {
00478             BootstrapReplicateFromAlignment(allocatedMatrix, alignment.Values.ToList(), alignmentType,
evolutionModel, numCores, progressCallback);
00479         }
00480
00481 /// <summary>
00482 /// Convert DNA sequences into <see cref="T:byte[]"/> arrays in which each byte corresponds to 3
positions.
00483 /// </summary>
00484 /// <param name="sequences">The sequences to convert.</param>
00485 /// <returns>An <see cref="T:IEnumerable{byte[]}"/> that, when enumerated, will contain the converted
sequences.</returns>
00486         public static IEnumerable<byte[]> ConvertDNASequences(IEnumerable<string> sequences)
00487         {
00488             foreach (string sequence in sequences)
00489             {
00490                 yield return ConvertDNASequence(sequence);
00491             }
00492         }
00493
00494 /// <summary>
00495 /// Convert DNA sequences into <see cref="T:byte[]"/> arrays in which each <see cref="byte"/>
corresponds to 3 positions.
00496 /// </summary>
00497 /// <param name="sequences">The sequences to convert.</param>
00498 /// <returns>A <see cref="T:List{byte[]}"/> that contains the converted sequences.</returns>
00499         public static List<byte[]> ConvertDNASequences(IReadOnlyList<string> sequences)
00500         {
00501             return ConvertDNASequences((IEnumerable<string>)sequences).ToList();
00502         }
00503
00504 /// <summary>
00505 /// Resample a sequence taking the specified positions.
00506 /// </summary>
00507 /// <param name="sequence">The sequence to resample.</param>
00508 /// <param name="sampledPositions">The columns to resample.</param>
00509 /// <returns>A <see cref="string"/> containing the resampled sequence.</returns>
00510         private static unsafe string BootstrapSequence(string sequence, int[] sampledPositions)
00511         {
00512             string tbr = new string('\0', sequence.Length);
00513
00514             fixed (char* sequencePtr = sequence)
00515             fixed (char* tbrPtr = tbr)
00516             fixed (int* valuesPtr = sampledPositions)
00517             {
00518                 for (int i = 0; i < sampledPositions.Length; i++)
00519                 {
00520                     tbrPtr[i] = sequencePtr[valuesPtr[i]];
00521                 }
00522             }
00523
00524             return tbr;
00525         }
00526
00527 /// <summary>
00528 /// Computes a bootstrap replicate of a DNA sequence alignment.
00529 /// </summary>
00530 /// <param name="sequences">The aligned DNA sequences.</param>
00531 /// <returns>An <see cref="T:IEnumerable{byte[]}"/> that, when enumerated, will contain the
bootstrapped sequences, converted into <see cref="T:byte[]"/> arrays where each byte corresponds to 3
positions.</returns>
00532 /// <exception cref="ArgumentOutOfRangeException">Thrown if not all of the sequences have the same
length.</exception>
00533         public static IEnumerable<byte[]> BootstrapDNASequences(IEnumerable<string> sequences)
00534         {
00535             int[] values = null;
00536             int length = -1;
00537
00538             foreach (string sequence in sequences)
00539             {
00540                 if (values == null)
00541                 {
00542                     length = sequence.Length;
00543                     values = new int[sequence.Length];
00544                     MathNet.Numerics.Distributions.DiscreteUniform.Samples(values, 0, sequence.Length
- 1);
00545                 }
00546
00547                 if (length != sequence.Length)
00548                 {
00549                     throw new ArgumentOutOfRangeException("Not all of the sequences have the same
```

```
length!", nameof(sequences));
00550                 }
00551
00552                 yield return ConvertDNASequence(BootstrapSequence(sequence, values));
00553             }
00554         }
00555
00556 /// <summary>
00557 /// Computes a bootstrap replicate of a DNA sequence alignment.
00558 /// </summary>
00559 /// <param name="sequences">The aligned DNA sequences.</param>
00560 /// <returns>An <see cref="T:List{byte[]}"/> that contains the bootstrapped sequences, converted into
<see cref="T:byte[]"/> arrays where each <see cref="byte"/> corresponds to 3 positions.</returns>
00561 /// <exception cref="ArgumentOutOfRangeException">Thrown if not all of the sequences have the same
length.</exception>
00562         public static List<byte[]> BootstrapDNASequences(IReadOnlyList<string> sequences)
00563         {
00564             return BootstrapDNASequences((IEnumerable<string>)sequences).ToList();
00565         }
00566
00567 /// <summary>
00568 /// Convert protein sequences into <see cref="T:ushort[]"/> arrays in which each <see cref="ushort"/>
corresponds to 2 positions.
00569 /// </summary>
00570 /// <param name="sequences">The sequences to convert.</param>
00571 /// <returns>An <see cref="T:IEnumerable{ushort[]}"/> that, when enumerated, will contain the
converted sequences.</returns>
00572         public static IEnumerable<ushort[]> ConvertProteinSequences(IEnumerable<string> sequences)
00573         {
00574             foreach (string sequence in sequences)
00575             {
00576                 yield return ConvertProteinSequence(sequence);
00577             }
00578         }
00579
00580 /// <summary>
00581 /// Convert protein sequences into <see cref="T:ushort[]"/> arrays in which each <see cref="ushort"/>
corresponds to 2 positions.
00582 /// </summary>
00583 /// <param name="sequences">The sequences to convert.</param>
00584 /// <returns>A <see cref="T:List{ushort[]}"/> that contains the converted sequences.</returns>
00585         public static List<ushort[]> ConvertProteinSequences(IReadOnlyList<string> sequences)
00586         {
00587             return ConvertProteinSequences((IEnumerable<string>)sequences).ToList();
00588         }
00589
00590 /// <summary>
00591 /// Computes a bootstrap replicate of a protein sequence alignment.
00592 /// </summary>
00593 /// <param name="sequences">The aligned protein sequences.</param>
00594 /// <returns>An <see cref="T:IEnumerable{ushort[]}"/> that, when enumerated, will contain the
bootstrapped sequences, converted into <see cref="T:ushort[]"/> arrays where each <see cref="ushort"/>
corresponds to 2 positions.</returns>
00595 /// <exception cref="ArgumentOutOfRangeException">Thrown if not all of the sequences have the same
length.</exception>
00596         public static IEnumerable<ushort[]> BootstrapProteinSequences(IEnumerable<string> sequences)
00597         {
00598             int[] values = null;
00599             int length = -1;
00600
00601             foreach (string sequence in sequences)
00602             {
00603                 if (values == null)
00604                 {
00605                     values = new int[sequence.Length];
00606                     MathNet.Numerics.Distributions.DiscreteUniform.Samples(values, 0, sequence.Length
- 1);
00607                     length = sequence.Length;
00608                 }
00609
00610                 if (length != sequence.Length)
00611                 {
00612                     throw new ArgumentOutOfRangeException("Not all of the sequences have the same
length!", nameof(sequences));
00613                 }
00614
00615                 yield return ConvertProteinSequence(BootstrapSequence(sequence, values));
00616             }
00617         }
00618
00619 /// <summary>
00620 /// Computes a bootstrap replicate of a protein sequence alignment.
00621 /// </summary>
00622 /// <param name="sequences">The aligned protein sequences.</param>
00623 /// <returns>A <see cref="T:List{ushort[]}"/> that contains the bootstrapped sequences, converted into
<see cref="T:ushort[]"/> arrays where each <see cref="ushort"/> corresponds to 2 positions.</returns>
00624 /// <exception cref="ArgumentOutOfRangeException">Thrown if not all of the sequences have the same
```

```
length.</exception>
00625         public static List<ushort[]> BootstrapProteinSequences(IReadOnlyList<string> sequences)
00626         {
00627             return BootstrapProteinSequences((IEnumerable<string>)sequences).ToList();
00628         }
00629     }
00630 }
```

8.19 DistanceMatrix.DNA.cs

```
00001 using MathNet.Numerics.LinearAlgebra;
00002 using MathNet.Numerics.LinearAlgebra.Factorization;
00003 using System;
00004 using System.Collections.Generic;
00005 using System.Threading.Tasks;
00006
00007 namespace PhyloTree.TreeBuilding
00008 {
00009     public static partial class DistanceMatrix
00010     {
00011 /// <summary>
00012 /// Converts a DNA sequence stored as a string into a <see cref="T:byte[]"/> array.  Each byte
      contains 3 nucleotide position.
00013 /// The allowed symbols are ACTGUN?- (uppercase and lowercase).  All other characters are treated as
      gaps (<c>?</c> is equivalent to <c>N</c>).
00014 /// If the sequence length is not a multiple of 3, it is padded with gaps.
00015 /// </summary>
00016 /// <param name="sequence">The sequence to convert.</param>
00017 /// <returns>A <see cref="T:byte[]"/> array representing the sequence.</returns>
00018         private static byte[] ConvertDNASequence(string sequence)
00019         {
00020             byte[] tbr = new byte[(sequence.Length + 2) / 3];
00021
00022             int length = sequence.Length;
00023
00024             int currIndex = 0;
00025             int currPosition = 0;
00026
00027             unsafe
00028             {
00029                 fixed (byte* byteString = tbr)
00030                 fixed (char* charString = sequence)
00031                 {
00032                     for (int i = 0; i < sequence.Length; i++)
00033                     {
00034                         switch (charString[i])
00035                         {
00036                             case 'A':
00037                             case 'a':
00038                                 byteString[currIndex] += (byte)((currPosition == 0 ?  1 :
      currPosition == 1 ?  6 :  36) * 1);
00039                                 break;
00040
00041                             case 'C':
00042                             case 'c':
00043                                 byteString[currIndex] += (byte)((currPosition == 0 ?  1 :
      currPosition == 1 ?  6 :  36) * 2);
00044                                 break;
00045
00046                             case 'G':
00047                             case 'g':
00048                                 byteString[currIndex] += (byte)((currPosition == 0 ?  1 :
      currPosition == 1 ?  6 :  36) * 3);
00049                                 break;
00050
00051                             case 'T':
00052                             case 't':
00053                             case 'U':
00054                             case 'u':
00055                                 byteString[currIndex] += (byte)((currPosition == 0 ?  1 :
      currPosition == 1 ?  6 :  36) * 4);
00056                                 break;
00057
00058                             case 'N':
00059                             case 'n':
00060                             case '?':
00061                                 byteString[currIndex] += (byte)((currPosition == 0 ?  1 :
      currPosition == 1 ?  6 :  36) * 5);
00062                                 break;
00063
00064                             default:
00065                                 break;
00066                         }
```

```
00067
00068                           currPosition = (currPosition + 1) % 3;
00069
00070                           if (currPosition == 0)
00071                           {
00072                               currIndex++;
00073                           }
00074                       }
00075                   }
00076               }
00077
00078               return tbr;
00079           }
00080
00081 /// <summary>
00082 /// Builds the match matrix used for the JC and K80 evolutionary models.
00083 /// </summary>
00084           private static void BuildDNAMatchMatrixJCK80()
00085           {
00086               for (int x = 0; x < 256; x++)
00087               {
00088                   DNAMatchMatrixJCK80[x] = new byte[256];
00089                   for (int y = 0; y < 256; y++)
00090                   {
00091                       // Every byte contains 3 positions.  We cache all the possible comparisons between
      bytes (i.e., between 3 positions).
00092                       // Here, we store the number of matches, the number of mismatches, the number of
      transitions and the number of transversions.
00093                       // Each can have a value between 0 and 3, thus 2 bits for each and a total of 8
      bits (1 byte).
00094                       // The matrix should be small enough that this code doesn't need to be very
      optimised.
00095                       byte x1 = (byte)(x % 6);
00096                       byte y1 = (byte)(y % 6);
00097
00098                       byte x2 = (byte)((x - x1) % 36 / 6);
00099                       byte y2 = (byte)((y - y1) % 36 / 6);
00100
00101                       byte x3 = (byte)((x - x1 - x2 * 6) % 216 / 36);
00102                       byte y3 = (byte)((y - y1 - y2 * 6) % 216 / 36);
00103
00104                       int match = 0;
00105                       int mismatch = 0;
00106                       int transitions = 0;
00107                       int transversions = 0;
00108
00109                       // Same letter, no gap.
00110                       if (x1 == y1 && x1 != 0)
00111                       {
00112                           match++;
00113                       }
00114                       // One is N, the other is not a gap.
00115                       else if ((x1 != 0 && y1 == 5) || (x1 == 5 && y1 != 0))
00116                       {
00117                           match++;
00118                       }
00119                       // Different letter, and neither is a gap.
00120                       else if (x1 != y1 && x1 != 0 && y1 != 0)
00121                       {
00122                           mismatch++;
00123                       }
00124
00125                       // Purine transition.
00126                       if ((x1 == 1 && y1 == 3) || (x1 == 3 && y1 == 1) ||
00127                           // Pyrimidine transition.
00128                           (x1 == 2 && y1 == 4) || (x1 == 4 && y1 == 2))
00129                       {
00130                           transitions++;
00131                       }
00132
00133                       // Transversion from a purine.
00134                       if (((x1 == 1 || x1 == 3) && (y1 == 2 || y1 == 4)) ||
00135                           // Transversion from a pyrimidine.
00136                           ((x1 == 2 || x1 == 4) && (y1 == 1 || y1 == 3)))
00137                       {
00138                           transversions++;
00139                       }
00140
00141                       // Same letter, no gap.
00142                       if (x2 == y2 && x2 != 0)
00143                       {
00144                           match++;
00145                       }
00146                       // One is N, the other is not a gap.
00147                       else if ((x2 != 0 && y2 == 5) || (x2 == 5 && y2 != 0))
00148                       {
00149                           match++;
```

```
00150                     }
00151                     // Different letter, and neither is a gap.
00152                     else if (x2 != y2 && x2 != 0 && y2 != 0)
00153                     {
00154                         mismatch++;
00155                     }
00156 
00157                     // Purine transition.
00158                     if ((x2 == 1 && y2 == 3) || (x2 == 3 && y2 == 1) ||
00159                         // Pyrimidine transition.
00160                         (x2 == 2 && y2 == 4) || (x2 == 4 && y2 == 2))
00161                     {
00162                         transitions++;
00163                     }
00164 
00165                     // Transversion from a purine.
00166                     if (((x2 == 1 || x2 == 3) && (y2 == 2 || y2 == 4)) ||
00167                         // Transversion from a pyrimidine.
00168                         ((x2 == 2 || x2 == 4) && (y2 == 1 || y2 == 3)))
00169                     {
00170                         transversions++;
00171                     }
00172 
00173                     // Same letter, no gap.
00174                     if (x3 == y3 && x3 != 0)
00175                     {
00176                         match++;
00177                     }
00178                     // One is N, the other is not a gap.
00179                     else if ((x3 != 0 && y3 == 5) || (x3 == 5 && y3 != 0))
00180                     {
00181                         match++;
00182                     }
00183                     // Different letter, and neither is a gap.
00184                     else if (x3 != y3 && x3 != 0 && y3 != 0)
00185                     {
00186                         mismatch++;
00187                     }
00188 
00189                     // Purine transition.
00190                     if ((x3 == 1 && y3 == 3) || (x3 == 3 && y3 == 1) ||
00191                         // Pyrimidine transition.
00192                         (x3 == 2 && y3 == 4) || (x3 == 4 && y3 == 2))
00193                     {
00194                         transitions++;
00195                     }
00196 
00197                     // Transversion from a purine.
00198                     if (((x3 == 1 || x3 == 3) && (y3 == 2 || y3 == 4)) ||
00199                         // Transversion from a pyrimidine.
00200                         ((x3 == 2 || x3 == 4) && (y3 == 1 || y3 == 3)))
00201                     {
00202                         transversions++;
00203                     }
00204 
00205                     DNAMatchMatrixJCK80[x][y] = (byte)(match + 4 * mismatch + 16 * transitions + 64 *
    transversions);
00206                 }
00207             }
00208         }
00209 
00210 
00211 /// <summary>
00212 /// Builds the match matrix used for the GTR evolutionary model.
00213 /// </summary>
00214         private static void BuildDNAMatchMatrixGTR()
00215         {
00216             for (int x = 0; x < 256; x++)
00217             {
00218                 DNAMatchMatrixGTR[x] = new uint[256];
00219                 for (int y = 0; y < 256; y++)
00220                 {
00221                     // Every byte contains 3 positions.  We cache all the possible comparisons between
    bytes (i.e., between 3 positions).
00222                     // Here, we store the number of matches (AA, CC, GG, TT) and the number of state
    changes (AC/CA, AG/GA, AT/TA, CG/GC,
00223                     // CT/TC, GT/TG). Each can have a value between 0 and 3, thus 2 bits for each and
    a total of 20 bits (3 bytes, stored
00224                     // as a uint taking up 4 bytes).  The matrix should be small enough that this code
    doesn't need to be very optimised.
00225                     byte x1 = (byte)(x % 6);
00226                     byte y1 = (byte)(y % 6);
00227 
00228                     byte comp1 = (byte)(x1 | (y1 << 3));
00229 
00230                     byte x2 = (byte)((x - x1) % 36 / 6);
00231                     byte y2 = (byte)((y - y1) % 36 / 6);
```

```
00232
00233                   byte comp2 = (byte)(x2 | (y2 << 3));
00234
00235                   byte x3 = (byte)((x - x1 - x2 * 6) % 216 / 36);
00236                   byte y3 = (byte)((y - y1 - y2 * 6) % 216 / 36);
00237
00238                   byte comp3 = (byte)(x3 | (y3 << 3));
00239
00240                   int aa = 0;
00241                   int cc = 0;
00242                   int gg = 0;
00243                   int tt = 0;
00244
00245                   int ac = 0;
00246                   int ag = 0;
00247                   int at = 0;
00248                   int cg = 0;
00249                   int ct = 0;
00250                   int gt = 0;
00251
00252                   switch (comp1)
00253                   {
00254                       // AA
00255                       case 0b001001:
00256                       // AN
00257                       case 0b101001:
00258                       // NA
00259                       case 0b001101:
00260                           aa++;
00261                           break;
00262
00263                       // AC
00264                       case 0b010001:
00265                       // CA
00266                       case 0b001010:
00267                           ac++;
00268                           break;
00269
00270                       // AG
00271                       case 0b011001:
00272                       // GA
00273                       case 0b001011:
00274                           ag++;
00275                           break;
00276
00277                       // AT
00278                       case 0b100001:
00279                       // TA
00280                       case 0b001100:
00281                           at++;
00282                           break;
00283
00284
00285                       // CC
00286                       case 0b010010:
00287                       // CN
00288                       case 0b101010:
00289                       // NC
00290                       case 0b010101:
00291                           cc++;
00292                           break;
00293
00294                       // CG
00295                       case 0b011010:
00296                       // GC
00297                       case 0b010011:
00298                           cg++;
00299                           break;
00300
00301                       // CT
00302                       case 0b100010:
00303                       // TC
00304                       case 0b010100:
00305                           ct++;
00306                           break;
00307
00308                       // GG
00309                       case 0b011011:
00310                       // GN
00311                       case 0b101011:
00312                       // NG
00313                       case 0b011101:
00314                           gg++;
00315                           break;
00316
00317                       // GT
00318                       case 0b100011:
```

```
00319                       // TG
00320                       case 0b011100:
00321                           gt++;
00322                           break;
00323
00324                       // TT
00325                       case 0b100100:
00326                       // TN
00327                       case 0b101100:
00328                       // NT
00329                       case 0b100101:
00330                           tt++;
00331                           break;
00332                   }
00333
00334                   switch (comp2)
00335                   {
00336                       // AA
00337                       case 0b001001:
00338                       // AN
00339                       case 0b101001:
00340                       // NA
00341                       case 0b001101:
00342                           aa++;
00343                           break;
00344
00345                       // AC
00346                       case 0b010001:
00347                       // CA
00348                       case 0b001010:
00349                           ac++;
00350                           break;
00351
00352                       // AG
00353                       case 0b011001:
00354                       // GA
00355                       case 0b001011:
00356                           ag++;
00357                           break;
00358
00359                       // AT
00360                       case 0b100001:
00361                       // TA
00362                       case 0b001100:
00363                           at++;
00364                           break;
00365
00366
00367                       // CC
00368                       case 0b010010:
00369                       // CN
00370                       case 0b101010:
00371                       // NC
00372                       case 0b010101:
00373                           cc++;
00374                           break;
00375
00376                       // CG
00377                       case 0b011010:
00378                       // GC
00379                       case 0b010011:
00380                           cg++;
00381                           break;
00382
00383                       // CT
00384                       case 0b100010:
00385                       // TC
00386                       case 0b010100:
00387                           ct++;
00388                           break;
00389
00390                       // GG
00391                       case 0b011011:
00392                       // GN
00393                       case 0b101011:
00394                       // NG
00395                       case 0b011101:
00396                           gg++;
00397                           break;
00398
00399                       // GT
00400                       case 0b100011:
00401                       // TG
00402                       case 0b011100:
00403                           gt++;
00404                           break;
00405
```

```
00406                       // TT
00407                       case 0b100100:
00408                       // TN
00409                       case 0b101100:
00410                       // NT
00411                       case 0b100101:
00412                           tt++;
00413                           break;
00414                   }
00415
00416                   switch (comp3)
00417                   {
00418                       // AA
00419                       case 0b001001:
00420                       // AN
00421                       case 0b101001:
00422                       // NA
00423                       case 0b001101:
00424                           aa++;
00425                           break;
00426
00427                       // AC
00428                       case 0b010001:
00429                       // CA
00430                       case 0b001010:
00431                           ac++;
00432                           break;
00433
00434                       // AG
00435                       case 0b011001:
00436                       // GA
00437                       case 0b001011:
00438                           ag++;
00439                           break;
00440
00441                       // AT
00442                       case 0b100001:
00443                       // TA
00444                       case 0b001100:
00445                           at++;
00446                           break;
00447
00448
00449                       // CC
00450                       case 0b010010:
00451                       // CN
00452                       case 0b101010:
00453                       // NC
00454                       case 0b010101:
00455                           cc++;
00456                           break;
00457
00458                       // CG
00459                       case 0b011010:
00460                       // GC
00461                       case 0b010011:
00462                           cg++;
00463                           break;
00464
00465                       // CT
00466                       case 0b100010:
00467                       // TC
00468                       case 0b010100:
00469                           ct++;
00470                           break;
00471
00472                       // GG
00473                       case 0b011011:
00474                       // GN
00475                       case 0b101011:
00476                       // NG
00477                       case 0b011101:
00478                           gg++;
00479                           break;
00480
00481                       // GT
00482                       case 0b100011:
00483                       // TG
00484                       case 0b011100:
00485                           gt++;
00486                           break;
00487
00488                       // TT
00489                       case 0b100100:
00490                       // TN
00491                       case 0b101100:
00492                       // NT
```

```
00493                         case 0b100101:
00494                             tt++;
00495                             break;
00496                     }
00497 
00498                     DNAMatchMatrixGTR[x][y] = (uint)(aa + (cc « 2) + (gg « 4) + (tt « 6) + (ac « 8) +
      (ag « 10) + (at « 12) + (cg « 14) + (ct « 16) + (gt « 18));
00499                 }
00500             }
00501         }
00502 
00503 
00504 /// <summary>
00505 /// Compares two DNA sequences using the Hamming distance.
00506 /// </summary>
00507 /// <param name="sequence1">The first sequence.</param>
00508 /// <param name="sequence2">The second sequence.</param>
00509 /// <returns>The (normalised) Hamming distance between the two sequences.</returns>
00510         private static float CompareDNASequencesHamming(byte[] sequence1, byte[] sequence2)
00511         {
00512             int length = sequence1.Length;
00513 
00514             int match = 0;
00515             int mismatch = 0;
00516 
00517             unsafe
00518             {
00519                 fixed (byte* seq1 = sequence1)
00520                 fixed (byte* seq2 = sequence2)
00521                 {
00522                     for (int i = 0; i < length; i++)
00523                     {
00524                         byte result = DNAMatchMatrixJCK80[seq1[i]][seq2[i]];
00525 
00526                         match += result & 3;
00527                         mismatch += (result » 2) & 3;
00528                     }
00529                 }
00530             }
00531 
00532             return mismatch / (float)(match + mismatch);
00533         }
00534 
00535 /// <summary>
00536 /// Compares a DNA sequence with two other sequences using the Hamming distance.  Faster than
00537 /// calling <see cref="CompareDNASequencesHamming(byte[], byte[])"/> twice.
00538 /// </summary>
00539 /// <param name="sequence1">The first sequence.</param>
00540 /// <param name="sequence2">The second sequence.</param>
00541 /// <param name="sequence3">The third sequence.</param>
00542 /// <param name="dist12">When the method returns, this variable will contain the (normalised) Hamming
      distance between <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00543 /// <param name="dist13">When the method returns, this variable will contain the (normalised) Hamming
      distance between <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00544         private static void CompareDNASequencesHamming(byte[] sequence1, byte[] sequence2, byte[]
      sequence3, out float dist12, out float dist13)
00545         {
00546             int length = sequence1.Length;
00547 
00548             int match12 = 0;
00549             int mismatch12 = 0;
00550 
00551             int match13 = 0;
00552             int mismatch13 = 0;
00553 
00554             unsafe
00555             {
00556                 fixed (byte* seq1 = sequence1)
00557                 fixed (byte* seq2 = sequence2)
00558                 fixed (byte* seq3 = sequence3)
00559                 {
00560                     for (int i = 0; i < length; i++)
00561                     {
00562                         int i1 = seq1[i];
00563 
00564                         byte result12 = DNAMatchMatrixJCK80[i1][seq2[i]];
00565 
00566                         match12 += result12 & 3;
00567                         mismatch12 += (result12 » 2) & 3;
00568 
00569                         byte result13 = DNAMatchMatrixJCK80[i1][seq3[i]];
00570 
00571                         match13 += result13 & 3;
00572                         mismatch13 += (result13 » 2) & 3;
00573                     }
00574                 }
00575             }
```

```
00576
00577             dist12 = mismatch12 / (float)(match12 + mismatch12);
00578             dist13 = mismatch13 / (float)(match13 + mismatch13);
00579         }
00580
00581 /// <summary>
00582 /// Computes a distance matrix between DNA sequences using the Hamming distance.
00583 /// </summary>
00584 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00585 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00586 /// <param name="progressCallback">A method used to report progress.</param>
00587 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
      matrix that will contain the distances between the sequences.</param>
00588         private static void ComputeDistanceMatrixHamming(IReadOnlyList<byte[]> sequences, int
      numCores, Action<double> progressCallback, float[][] matrix)
00589         {
00590             long progress = 0;
00591             double total = sequences.Count * 0.5 * (sequences.Count - 1);
00592             object progressLock = new object();
00593
00594             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
      }, i =>
00595             {
00596                 for (int j = 0; j < i - 1; j += 2)
00597                 {
00598                     CompareDNASequencesHamming(sequences[i], sequences[j], sequences[j + 1], out
      matrix[i][j], out matrix[i][j + 1]);
00599                 }
00600
00601                 for (int j = i - 1; j < i; j++)
00602                 {
00603                     matrix[i][j] = CompareDNASequencesHamming(sequences[i], sequences[j]);
00604                 }
00605
00606                 if (progressCallback != null)
00607                 {
00608                     lock (progressLock)
00609                     {
00610                         progress += i;
00611                         progressCallback((double)progress / total);
00612                     }
00613                 }
00614             });
00615         }
00616
00617 /// <summary>
00618 /// Compares two DNA sequences using the JC model of evolution.
00619 /// </summary>
00620 /// <param name="sequence1">The first sequence.</param>
00621 /// <param name="sequence2">The second sequence.</param>
00622 /// <returns>The JC distance between the two sequences.</returns>
00623         private static float CompareDNASequencesJC(byte[] sequence1, byte[] sequence2)
00624         {
00625             int length = sequence1.Length;
00626
00627             int match = 0;
00628             int mismatch = 0;
00629
00630             unsafe
00631             {
00632                 fixed (byte* seq1 = sequence1)
00633                 fixed (byte* seq2 = sequence2)
00634                 {
00635                     for (int i = 0; i < length; i++)
00636                     {
00637                         byte result = DNAMatchMatrixJCK80[seq1[i]][seq2[i]];
00638
00639                         match += result & 3;
00640                         mismatch += (result » 2) & 3;
00641                     }
00642                 }
00643             }
00644
00645             return (float)(-0.75 * Math.Log(1 - 1.3333333333333333333333333333333333333333333333333333 *
      mismatch / (match + mismatch)));
00646         }
00647
00648 /// <summary>
00649 /// Compares a DNA sequence with two other sequences using the JC model of evolution.  Faster than
00650 /// calling <see cref="CompareDNASequencesJC(byte[], byte[])"/> twice.
00651 /// </summary>
00652 /// <param name="sequence1">The first sequence.</param>
00653 /// <param name="sequence2">The second sequence.</param>
00654 /// <param name="sequence3">The third sequence.</param>
00655 /// <param name="dist12">When the method returns, this variable will contain the JC distance between
      <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00656 /// <param name="dist13">When the method returns, this variable will contain the JC distance between
```

```
    <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00657         private static void CompareDNASequencesJC(byte[] sequence1, byte[] sequence2, byte[]
    sequence3, out float dist12, out float dist13)
00658         {
00659             int length = sequence1.Length;
00660 
00661             int match12 = 0;
00662             int mismatch12 = 0;
00663 
00664             int match13 = 0;
00665             int mismatch13 = 0;
00666 
00667             unsafe
00668             {
00669                 fixed (byte* seq1 = sequence1)
00670                 fixed (byte* seq2 = sequence2)
00671                 fixed (byte* seq3 = sequence3)
00672                 {
00673                     for (int i = 0; i < length; i++)
00674                     {
00675                         int i1 = seq1[i];
00676 
00677                         byte result12 = DNAMatchMatrixJCK80[i1][seq2[i]];
00678 
00679                         match12 += result12 & 3;
00680                         mismatch12 += (result12 » 2) & 3;
00681 
00682                         byte result13 = DNAMatchMatrixJCK80[i1][seq3[i]];
00683 
00684                         match13 += result13 & 3;
00685                         mismatch13 += (result13 » 2) & 3;
00686                     }
00687                 }
00688             }
00689 
00690             dist12 = (float)(-0.75 * Math.Log(1 - 1.333333333333333333333333333333333333333333333333333333
    * mismatch12 / (match12 + mismatch12)));
00691             dist13 = (float)(-0.75 * Math.Log(1 - 1.333333333333333333333333333333333333333333333333333333
    * mismatch13 / (match13 + mismatch13)));
00692         }
00693 
00694 /// <summary>
00695 /// Computes a distance matrix between DNA sequences using the JC model of evolution.
00696 /// </summary>
00697 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00698 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00699 /// <param name="progressCallback">A method used to report progress.</param>
00700 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
    matrix that will contain the distances between the sequences.</param>
00701         private static void ComputeDistanceMatrixJC(IReadOnlyList<byte[]> sequences, int numCores,
    Action<double> progressCallback, float[][] matrix)
00702         {
00703             long progress = 0;
00704             double total = sequences.Count * 0.5 * (sequences.Count - 1);
00705             object progressLock = new object();
00706 
00707             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
    }, i =>
00708             {
00709                 for (int j = 0; j < i - 1; j += 2)
00710                 {
00711                     CompareDNASequencesJC(sequences[i], sequences[j], sequences[j + 1], out
    matrix[i][j], out matrix[i][j + 1]);
00712                 }
00713 
00714                 for (int j = i - 1; j < i; j++)
00715                 {
00716                     matrix[i][j] = CompareDNASequencesJC(sequences[i], sequences[j]);
00717                 }
00718 
00719                 if (progressCallback != null)
00720                 {
00721                     lock (progressLock)
00722                     {
00723                         progress += i;
00724                         progressCallback((double)progress / total);
00725                     }
00726                 }
00727             });
00728         }
00729 
00730 
00731 /// <summary>
00732 /// Compares two DNA sequences using the K80 model of evolution.
00733 /// </summary>
00734 /// <param name="sequence1">The first sequence.</param>
00735 /// <param name="sequence2">The second sequence.</param>
```

```
00736 /// <returns>The K80 distance between the two sequences.</returns>
00737         private static float CompareDNASequencesK80(byte[] sequence1, byte[] sequence2)
00738         {
00739             int length = sequence1.Length;
00740 
00741             int match = 0;
00742             int transitions = 0;
00743             int transversions = 0;
00744 
00745             unsafe
00746             {
00747                 fixed (byte* seq1 = sequence1)
00748                 fixed (byte* seq2 = sequence2)
00749                 {
00750                     for (int i = 0; i < length; i++)
00751                     {
00752                         byte result = DNAMatchMatrixJCK80[seq1[i]][seq2[i]];
00753 
00754                         match += result & 3;
00755                         transitions += (result » 4) & 3;
00756                         transversions += (result » 6) & 3;
00757                     }
00758                 }
00759             }
00760 
00761             int total = transitions + transversions + match;
00762             double transversionProp = (double)transversions / total;
00763 
00764             return (float)(-0.5 * Math.Log((1 - 2.0 * transitions / total - transversionProp) *
    Math.Sqrt(1 - 2 * transversionProp)));
00765         }
00766 
00767 
00768 /// <summary>
00769 /// Compares a DNA sequence with two other sequences using the K80 model of evolution.  Faster than
00770 /// calling <see cref="CompareDNASequencesK80(byte[], byte[])"/> twice.
00771 /// </summary>
00772 /// <param name="sequence1">The first sequence.</param>
00773 /// <param name="sequence2">The second sequence.</param>
00774 /// <param name="sequence3">The third sequence.</param>
00775 /// <param name="dist12">When the method returns, this variable will contain the K80 distance between
    <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00776 /// <param name="dist13">When the method returns, this variable will contain the K80 distance between
    <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00777         private static void CompareDNASequencesK80(byte[] sequence1, byte[] sequence2, byte[]
    sequence3, out float dist12, out float dist13)
00778         {
00779             int length = sequence1.Length;
00780 
00781             int match12 = 0;
00782             int transitions12 = 0;
00783             int transversions12 = 0;
00784 
00785             int match13 = 0;
00786             int transitions13 = 0;
00787             int transversions13 = 0;
00788 
00789             unsafe
00790             {
00791                 fixed (byte* seq1 = sequence1)
00792                 fixed (byte* seq2 = sequence2)
00793                 fixed (byte* seq3 = sequence3)
00794                 {
00795                     for (int i = 0; i < length; i++)
00796                     {
00797                         int i1 = seq1[i];
00798 
00799                         byte result12 = DNAMatchMatrixJCK80[i1][seq2[i]];
00800 
00801                         match12 += result12 & 3;
00802                         transitions12 += (result12 » 4) & 3;
00803                         transversions12 += (result12 » 6) & 3;
00804 
00805                         byte result13 = DNAMatchMatrixJCK80[i1][seq3[i]];
00806 
00807                         match13 += result13 & 3;
00808                         transitions13 += (result13 » 4) & 3;
00809                         transversions13 += (result13 » 6) & 3;
00810                     }
00811                 }
00812             }
00813 
00814             int total12 = transitions12 + transversions12 + match12;
00815             double transversionProp12 = (double)transversions12 / total12;
00816 
00817             dist12 = (float)(-0.5 * Math.Log((1 - 2.0 * transitions12 / total12 - transversionProp12)
    * Math.Sqrt(1 - 2 * transversionProp12)));
```

```
00818
00819             int total13 = transitions13 + transversions13 + match13;
00820             double transversionProp13 = (double)transversions13 / total13;
00821
00822             dist13 = (float)(-0.5 * Math.Log((1 - 2.0 * transitions13 / total13 - transversionProp13)
      * Math.Sqrt(1 - 2 * transversionProp13)));
00823         }
00824
00825 /// <summary>
00826 /// Computes a distance matrix between DNA sequences using the K80 model of evolution.
00827 /// </summary>
00828 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00829 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00830 /// <param name="progressCallback">A method used to report progress.</param>
00831 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
      matrix that will contain the distances between the sequences.</param>
00832         private static void ComputeDistanceMatrixK80(IReadOnlyList<byte[]> sequences, int numCores,
      Action<double> progressCallback, float[][] matrix)
00833         {
00834             long progress = 0;
00835             double total = sequences.Count * 0.5 * (sequences.Count - 1);
00836             object progressLock = new object();
00837
00838             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
      }, i =>
00839             {
00840                 for (int j = 0; j < i - 1; j += 2)
00841                 {
00842                     CompareDNASequencesK80(sequences[i], sequences[j], sequences[j + 1], out
      matrix[i][j], out matrix[i][j + 1]);
00843                 }
00844
00845                 for (int j = i - 1; j < i; j++)
00846                 {
00847                     matrix[i][j] = CompareDNASequencesK80(sequences[i], sequences[j]);
00848                 }
00849
00850                 if (progressCallback != null)
00851                 {
00852                     lock (progressLock)
00853                     {
00854                         progress += i;
00855                         progressCallback((double)progress / total);
00856                     }
00857                 }
00858             });
00859         }
00860
00861 /// <summary>
00862 /// Compares two DNA sequences using the GTR model of evolution.
00863 /// </summary>
00864 /// <param name="sequence1">The first sequence.</param>
00865 /// <param name="sequence2">The second sequence.</param>
00866 /// <remarks>From Waddel &amp;amp; Steel, 1997.</remarks>
00867 /// <returns>The GTR distance between the two sequences.</returns>
00868         private static float CompareDNASequencesGTR(byte[] sequence1, byte[] sequence2)
00869         {
00870             int length = sequence1.Length;
00871
00872             uint aa = 0;
00873             uint cc = 0;
00874             uint gg = 0;
00875             uint tt = 0;
00876
00877             uint ac = 0;
00878             uint ag = 0;
00879             uint at = 0;
00880
00881             uint cg = 0;
00882             uint ct = 0;
00883
00884             uint gt = 0;
00885
00886             unsafe
00887             {
00888                 fixed (byte* seq1 = sequence1)
00889                 fixed (byte* seq2 = sequence2)
00890                 {
00891                     for (int i = 0; i < length; i++)
00892                     {
00893                         uint result = DNAMatchMatrixGTR[seq1[i]][seq2[i]];
00894
00895                         aa += result & 3;
00896                         cc += (result » 2) & 3;
00897                         gg += (result » 4) & 3;
00898                         tt += (result » 6) & 3;
00899
```

```
00900                           ac += (result » 8) & 3;
00901                           ag += (result » 10) & 3;
00902                           at += (result » 12) & 3;
00903 
00904                           cg += (result » 14) & 3;
00905                           ct += (result » 16) & 3;
00906 
00907                           gt += (result » 18) & 3;
00908                       }
00909                   }
00910               }
00911 
00912               double total = aa + cc + gg + tt + ac + ag + at + cg + ct + gt;
00913 
00914               Matrix<double> fSharp = Matrix<double>.Build.Dense(4, 4);
00915 
00916               fSharp[0, 0] = aa / total;
00917               fSharp[0, 1] = ac / total * 0.5;
00918               fSharp[0, 2] = ag / total * 0.5;
00919               fSharp[0, 3] = at / total * 0.5;
00920 
00921               fSharp[1, 0] = fSharp[0, 1];
00922               fSharp[1, 1] = cc / total;
00923               fSharp[1, 2] = cg / total * 0.5;
00924               fSharp[1, 3] = ct / total * 0.5;
00925 
00926               fSharp[2, 0] = fSharp[0, 2];
00927               fSharp[2, 1] = fSharp[1, 2];
00928               fSharp[2, 2] = gg / total;
00929               fSharp[2, 3] = gt / total * 0.5;
00930 
00931               fSharp[3, 0] = fSharp[0, 3];
00932               fSharp[3, 1] = fSharp[1, 3];
00933               fSharp[3, 2] = fSharp[2, 3];
00934               fSharp[3, 3] = tt / total;
00935 
00936               Matrix<double> Pi = Matrix<double>.Build.DiagonalOfDiagonalVector(fSharp.RowSums());
00937 
00938               Matrix<double> P = Pi.Inverse() * fSharp;
00939 
00940               Evd<double> P_EVD = P.Evd();
00941 
00942               double[] diagonal = new double[4];
00943 
00944               // Why -280?  No idea, but it produces results comparable to PAUP*.
00945               for (int i = 0; i < 4; i++)
00946               {
00947                   if (P_EVD.EigenValues[i].Imaginary == 0 && P_EVD.EigenValues[i].Real > 0)
00948                   {
00949                       diagonal[i] = Math.Max(-280, Math.Log(P_EVD.EigenValues[i].Real));
00950                   }
00951                   else
00952                   {
00953                       diagonal[i] = -280;
00954                   }
00955               }
00956 
00957               Matrix<double> logPsi = Matrix<double>.Build.DiagonalOfDiagonalArray(diagonal);
00958 
00959               return (float)(-(Pi * (P_EVD.EigenVectors * logPsi *
      P_EVD.EigenVectors.Inverse()))).Trace());
00960           }
00961 
00962 /// <summary>
00963 /// Compares a DNA sequence with two other sequences using the GTR model of evolution.  Faster than
00964 /// calling <see cref="CompareDNASequencesGTR(byte[], byte[])"/> twice.
00965 /// </summary>
00966 /// <param name="sequence1">The first sequence.</param>
00967 /// <param name="sequence2">The second sequence.</param>
00968 /// <param name="sequence3">The third sequence.</param>
00969 /// <param name="dist12">When the method returns, this variable will contain the GTR distance between
      <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00970 /// <param name="dist13">When the method returns, this variable will contain the GTR distance between
      <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00971           private static void CompareDNASequencesGTR(byte[] sequence1, byte[] sequence2, byte[]
      sequence3, out float dist12, out float dist13)
00972           {
00973               int length = sequence1.Length;
00974 
00975               uint aa12 = 0;
00976               uint cc12 = 0;
00977               uint gg12 = 0;
00978               uint tt12 = 0;
00979 
00980               uint ac12 = 0;
00981               uint ag12 = 0;
00982               uint at12 = 0;
```

```
00983
00984             uint cg12 = 0;
00985             uint ct12 = 0;
00986
00987             uint gt12 = 0;
00988
00989             uint aa13 = 0;
00990             uint cc13 = 0;
00991             uint gg13 = 0;
00992             uint tt13 = 0;
00993
00994             uint ac13 = 0;
00995             uint ag13 = 0;
00996             uint at13 = 0;
00997
00998             uint cg13 = 0;
00999             uint ct13 = 0;
01000
01001             uint gt13 = 0;
01002
01003             unsafe
01004             {
01005                 fixed (byte* seq1 = sequence1)
01006                 fixed (byte* seq2 = sequence2)
01007                 fixed (byte* seq3 = sequence3)
01008                 {
01009                     for (int i = 0; i < length; i++)
01010                     {
01011                         int i1 = seq1[i];
01012
01013                         uint result12 = DNAMatchMatrixGTR[i1][seq2[i]];
01014
01015                         aa12 += result12 & 3;
01016                         cc12 += (result12 » 2) & 3;
01017                         gg12 += (result12 » 4) & 3;
01018                         tt12 += (result12 » 6) & 3;
01019
01020                         ac12 += (result12 » 8) & 3;
01021                         ag12 += (result12 » 10) & 3;
01022                         at12 += (result12 » 12) & 3;
01023
01024                         cg12 += (result12 » 14) & 3;
01025                         ct12 += (result12 » 16) & 3;
01026
01027                         gt12 += (result12 » 18) & 3;
01028
01029                         uint result13 = DNAMatchMatrixGTR[i1][seq3[i]];
01030
01031                         aa13 += result13 & 3;
01032                         cc13 += (result13 » 2) & 3;
01033                         gg13 += (result13 » 4) & 3;
01034                         tt13 += (result13 » 6) & 3;
01035
01036                         ac13 += (result13 » 8) & 3;
01037                         ag13 += (result13 » 10) & 3;
01038                         at13 += (result13 » 12) & 3;
01039
01040                         cg13 += (result13 » 14) & 3;
01041                         ct13 += (result13 » 16) & 3;
01042
01043                         gt13 += (result13 » 18) & 3;
01044                     }
01045                 }
01046             }
01047
01048             double total12 = aa12 + cc12 + gg12 + tt12 + ac12 + ag12 + at12 + cg12 + ct12 + gt12;
01049
01050             Matrix<double> fSharp12 = Matrix<double>.Build.Dense(4, 4);
01051
01052             fSharp12[0, 0] = aa12 / total12;
01053             fSharp12[0, 1] = ac12 / total12 * 0.5;
01054             fSharp12[0, 2] = ag12 / total12 * 0.5;
01055             fSharp12[0, 3] = at12 / total12 * 0.5;
01056
01057             fSharp12[1, 0] = fSharp12[0, 1];
01058             fSharp12[1, 1] = cc12 / total12;
01059             fSharp12[1, 2] = cg12 / total12 * 0.5;
01060             fSharp12[1, 3] = ct12 / total12 * 0.5;
01061
01062             fSharp12[2, 0] = fSharp12[0, 2];
01063             fSharp12[2, 1] = fSharp12[1, 2];
01064             fSharp12[2, 2] = gg12 / total12;
01065             fSharp12[2, 3] = gt12 / total12 * 0.5;
01066
01067             fSharp12[3, 0] = fSharp12[0, 3];
01068             fSharp12[3, 1] = fSharp12[1, 3];
01069             fSharp12[3, 2] = fSharp12[2, 3];
```

```
01070             fSharp12[3, 3] = tt12 / total12;
01071
01072             Matrix<double> Pi12 = Matrix<double>.Build.DiagonalOfDiagonalVector(fSharp12.RowSums());
01073
01074             Matrix<double> P12 = Pi12.Inverse() * fSharp12;
01075
01076             Evd<double> P_EVD12 = P12.Evd();
01077
01078             double[] diagonal12 = new double[4];
01079
01080             // Why -280?  No idea, but it produces results comparable to PAUP*.
01081             for (int i = 0; i < 4; i++)
01082             {
01083                 if (P_EVD12.EigenValues[i].Imaginary == 0 && P_EVD12.EigenValues[i].Real > 0)
01084                 {
01085                     diagonal12[i] = Math.Max(-280, Math.Log(P_EVD12.EigenValues[i].Real));
01086                 }
01087                 else
01088                 {
01089                     diagonal12[i] = -280;
01090                 }
01091             }
01092
01093             Matrix<double> logPsi12 = Matrix<double>.Build.DiagonalOfDiagonalArray(diagonal12);
01094
01095             dist12 = (float)(-(Pi12 * (P_EVD12.EigenVectors * logPsi12 *
      P_EVD12.EigenVectors.Inverse()))).Trace());
01096
01097             double total13 = aa13 + cc13 + gg13 + tt13 + ac13 + ag13 + at13 + cg13 + ct13 + gt13;
01098
01099             Matrix<double> fSharp13 = Matrix<double>.Build.Dense(4, 4);
01100
01101             fSharp13[0, 0] = aa13 / total13;
01102             fSharp13[0, 1] = ac13 / total13 * 0.5;
01103             fSharp13[0, 2] = ag13 / total13 * 0.5;
01104             fSharp13[0, 3] = at13 / total13 * 0.5;
01105
01106             fSharp13[1, 0] = fSharp13[0, 1];
01107             fSharp13[1, 1] = cc13 / total13;
01108             fSharp13[1, 2] = cg13 / total13 * 0.5;
01109             fSharp13[1, 3] = ct13 / total13 * 0.5;
01110
01111             fSharp13[2, 0] = fSharp13[0, 2];
01112             fSharp13[2, 1] = fSharp13[1, 2];
01113             fSharp13[2, 2] = gg13 / total13;
01114             fSharp13[2, 3] = gt13 / total13 * 0.5;
01115
01116             fSharp13[3, 0] = fSharp13[0, 3];
01117             fSharp13[3, 1] = fSharp13[1, 3];
01118             fSharp13[3, 2] = fSharp13[2, 3];
01119             fSharp13[3, 3] = tt13 / total13;
01120
01121             Matrix<double> Pi13 = Matrix<double>.Build.DiagonalOfDiagonalVector(fSharp13.RowSums());
01122
01123             Matrix<double> P13 = Pi13.Inverse() * fSharp13;
01124
01125             Evd<double> P_EVD13 = P13.Evd();
01126
01127             double[] diagonal13 = new double[4];
01128
01129             // Why -280?  No idea, but it produces results comparable to PAUP*.
01130             for (int i = 0; i < 4; i++)
01131             {
01132                 if (P_EVD13.EigenValues[i].Imaginary == 0 && P_EVD13.EigenValues[i].Real > 0)
01133                 {
01134                     diagonal13[i] = Math.Max(-280, Math.Log(P_EVD13.EigenValues[i].Real));
01135                 }
01136                 else
01137                 {
01138                     diagonal13[i] = -280;
01139                 }
01140             }
01141
01142             Matrix<double> logPsi13 = Matrix<double>.Build.DiagonalOfDiagonalArray(diagonal13);
01143
01144             dist13 = (float)(-(Pi13 * (P_EVD13.EigenVectors * logPsi13 *
      P_EVD13.EigenVectors.Inverse()))).Trace());
01145         }
01146
01147 /// <summary>
01148 /// Computes a distance matrix between DNA sequences using the GTR model of evolution.
01149 /// </summary>
01150 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
01151 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
01152 /// <param name="progressCallback">A method used to report progress.</param>
01153 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
      matrix that will contain the distances between the sequences.</param>
```

```
01154         private static void ComputeDistanceMatrixGTR(IReadOnlyList<byte[]> sequences, int numCores,
      Action<double> progressCallback, float[][] matrix)
01155         {
01156             long progress = 0;
01157             double total = sequences.Count * 0.5 * (sequences.Count - 1);
01158             object progressLock = new object();
01159 
01160             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
      }, i =>
01161             {
01162                 for (int j = 0; j < i - 1; j += 2)
01163                 {
01164                     CompareDNASequencesGTR(sequences[i], sequences[j], sequences[j + 1], out
      matrix[i][j], out matrix[i][j + 1]);
01165                 }
01166 
01167                 for (int j = i - 1; j < i; j++)
01168                 {
01169                     matrix[i][j] = CompareDNASequencesGTR(sequences[i], sequences[j]);
01170                 }
01171 
01172                 if (progressCallback != null)
01173                 {
01174                     lock (progressLock)
01175                     {
01176                         progress += i;
01177                         progressCallback((double)progress / total);
01178                     }
01179                 }
01180             });
01181         }
01182     }
01183 }
01184 
01185 
```

8.20 DistanceMatrix.Protein.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Text;
00004 using System.Threading.Tasks;
00005 
00006 namespace PhyloTree.TreeBuilding
00007 {
00008     public static partial class DistanceMatrix
00009     {
00010         // From rapidNJ source code.
00011         private static readonly int[] DayhoffPAMs = { 195, 196, 197, 198, 199, 200, 200, 201, 202,
      203, 204, 205, 206, 207, 208, 209, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221,
      222, 223, 224, 226, 227, 228, 229, 230, 231, 232, 233, 234, 236, 237, 238, 239, 240, 241, 243, 244,
      245, 246, 248, 249, 250, 252, 253, 254, 255, 257, 258, 260, 261, 262, 264, 265, 267, 268, 270, 271,
      273, 274, 276, 277, 279, 281, 282, 284, 285, 287, 289, 291, 292, 294, 296, 298, 299, 301, 303, 305,
      307, 309, 311, 313, 315, 317, 319, 321, 323, 325, 328, 330, 332, 335, 337, 339, 342, 344, 347, 349,
      352, 354, 357, 360, 362, 365, 368, 371, 374, 377, 380, 383, 386, 389, 393, 396, 399, 403, 407, 410,
      414, 418, 422, 426, 430, 434, 438, 442, 447, 451, 456, 461, 466, 471, 476, 482, 487, 493, 498, 504,
      511, 517, 524, 531, 538, 545, 553, 560, 569, 577, 586, 595, 605, 615, 626, 637, 649, 661, 675, 688,
      703, 719, 736, 754, 775, 796, 819, 845, 874, 907, 945, 988 };
00012 
00013         private static readonly sbyte[,] BLOSUM62 = new sbyte[28, 28]
00014         {
00015             { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
00016             { 0, 4, 0, -2, -1, -2, 0, -2, -1, -1, -1, -1, -2, -1, -1, -1, 1, 0, 0, -3, -2, 0, 0, -2,
      -1, 0, 0, -4 },
00017             { 0, 0, 9, -3, -4, -2, -3, -3, -1, -3, -1, -1, -3, -3, -3, -3, -1, -1, -1, -2, -2, 0, 0,
      -3, -3, 0, -2, -4 },
00018             { 0, -2, -3, 6, 2, -3, -1, -1, -3, -1, -4, -3, 1, -1, 0, -2, 0, -1, -3, -4, -3, 0, 0, 4,
      1, 0, -1, -4 },
00019             { 0, -1, -4, 2, 5, -3, -2, 0, -3, 1, -3, -2, 0, -1, 2, 0, 0, -1, -2, -3, -2, 0, 0, 1, 4,
      0, -1, -4 },
00020             { 0, -2, -2, -3, -3, 6, -3, -1, 0, -3, 0, 0, -3, -4, -3, -3, -2, -2, -1, 1, 3, 0, 0, -3,
      -3, 0, -1, -4 },
00021             { 0, 0, -3, -1, -2, -3, 6, -2, -4, -2, -4, -3, 0, -2, -2, -2, 0, -2, -3, -2, -3, 0, 0, -1,
      -2, 0, -1, -4 },
00022             { 0, -2, -3, -1, 0, -1, -2, 8, -3, -1, -3, -2, 1, -2, 0, 0, -1, -2, -3, -2, 2, 0, 0, 0, 0,
      0, -1, -4 },
00023             { 0, -1, -1, -3, -3, 0, -4, -3, 4, -3, 2, 1, -3, -3, -3, -3, -2, -1, 3, -3, -1, 0, 0, -3,
      -3, 0, -1, -4 },
00024             { 0, -1, -3, -1, 1, -3, -2, -1, -3, 5, -2, -1, 0, -1, 1, 2, 0, -1, -2, -3, -2, 0, 0, 0, 1,
      0, -1, -4 },
00025             { 0, -1, -1, -4, -3, 0, -4, -3, 2, -2, 4, 2, -3, -3, -2, -2, -2, -1, 1, -2, -1, 0, 0, -4,
      -3, 0, -1, -4 },
00026             { 0, -1, -1, -3, -2, 0, -3, -2, 1, -1, 2, 5, -2, -2, 0, -1, -1, -1, 1, -1, -1, 0, 0, -3,
      -1, 0, -1, -4 },
```

```
00027             { 0, -2, -3, 1, 0, -3, 0, 1, -3, 0, -3, -2, 6, -2, 0, 0, 1, 0, -3, -4, -2, 0, 0, 3, 0, 0,
      -1, -4 },
00028             { 0, -1, -3, -1, -1, -4, -2, -2, -3, -1, -3, -2, -2, 7, -1, -2, -1, -1, -2, -4, -3, 0, 0,
      -2, -1, 0, -2, -4 },
00029             { 0, -1, -3, 0, 2, -3, -2, 0, -3, 1, -2, 0, 0, -1, 5, 1, 0, -1, -2, -2, -1, 0, 0, 0, 3, 0,
      -1, -4 },
00030             { 0, -1, -3, -2, 0, -3, -2, 0, -3, 2, -2, -1, 0, -2, 1, 5, -1, -1, -3, -3, -2, 0, 0, -1,
      0, 0, -1, -4 },
00031             { 0, 1, -1, 0, 0, -2, 0, -1, -2, 0, -2, -1, 1, -1, 0, -1, 4, 1, -2, -3, -2, 0, 0, 0, 0, 0,
      0, -4 },
00032             { 0, 0, -1, -1, -1, -2, -2, -2, -1, -1, -1, -1, 0, -1, -1, -1, 1, 5, 0, -2, -2, 0, 0, -1,
      -1, 0, 0, -4 },
00033             { 0, 0, -1, -3, -2, -1, -3, -3, 3, -2, 1, 1, -3, -2, -2, -3, -2, 0, 4, -3, -1, 0, 0, -3,
      -2, 0, -1, -4 },
00034             { 0, -3, -2, -4, -3, 1, -2, -2, -3, -3, -2, -1, -4, -4, -2, -3, -3, -2, -3, 11, 2, 0, 0,
      -4, -3, 0, -2, -4 },
00035             { 0, -2, -2, -3, -2, 3, -3, 2, -1, -2, -1, -1, -2, -3, -1, -2, -2, -2, -1, 2, 7, 0, 0, -3,
      -2, 0, -1, -4 },
00036             { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
00037             { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
00038             { 0, -2, -3, 4, 1, -3, -1, 0, -3, 0, -4, -3, 3, -2, 0, -1, 0, -1, -3, -4, -3, 0, 0, 4, 1,
      0, -1, -4 },
00039             { 0, -1, -3, 1, 4, -3, -2, 0, -3, 1, -3, -1, 0, -1, 3, 0, 0, -1, -2, -3, -2, 0, 0, 1, 4,
      0, -1, -4 },
00040             { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
00041             { 0, 0, -2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -2, -1, -1, 0, 0, -1, -2, -1, 0, 0,
      -1, -1, 0, -1, -4 },
00042             { 0, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, -4, 0, 0,
      -4, -4, 0, -4, 1 },
00043         };
00044
00045 /// <summary>
00046 /// Converts a protein sequence stored as a string into a <see cref="T:ushort[]"/> array.  Each ushort
      contains 2 amino acid positions.
00047 /// The allowed symbols are the usual 20 1-letter amino acid abbreviations, plus U for Sec, O for Pyl,
      B for Asn or Asp, Z for
00048 /// Gln or Glu, J for Ile or Leu, X for any amino acid, * for stop codons and - for gaps (uppercase
      and lowercase).  All other
00049 /// characters are treated as gaps.  If the sequence length is not a multiple of 2, it is padded with
      gaps.
00050 /// </summary>
00051 /// <param name="sequence">The sequence to convert.</param>
00052 /// <returns>A <see cref="T:ushort[]"/> array representing the sequence.</returns>
00053         public static ushort[] ConvertProteinSequence(string sequence)
00054         {
00055             ushort[] tbr = new ushort[(sequence.Length + 1) / 2];
00056
00057             int length = sequence.Length;
00058
00059             int currIndex = 0;
00060             int currPosition = 0;
00061
00062             unsafe
00063             {
00064                 fixed (ushort* ushortString = tbr)
00065                 fixed (char* charString = sequence)
00066                 {
00067                     for (int i = 0; i < sequence.Length; i++)
00068                     {
00069                         switch (charString[i])
00070                         {
00071                             case 'A':
00072                             case 'a':
00073                                 ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      1);
00074                                 break;
00075
00076                             case 'C':
00077                             case 'c':
00078                                 ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      2);
00079                                 break;
00080
00081                             case 'D':
00082                             case 'd':
00083                                 ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      3);
00084                                 break;
00085
00086                             case 'E':
00087                             case 'e':
00088                                 ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      4);
00089                                 break;
00090
00091                             case 'F':
00092                             case 'f':
```

```
00093                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      5);
00094                               break;
00095
00096                           case 'G':
00097                           case 'g':
00098                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      6);
00099                               break;
00100
00101                           case 'H':
00102                           case 'h':
00103                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      7);
00104                               break;
00105
00106                           case 'I':
00107                           case 'i':
00108                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      8);
00109                               break;
00110
00111                           case 'K':
00112                           case 'k':
00113                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      9);
00114                               break;
00115
00116                           case 'L':
00117                           case 'l':
00118                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      10);
00119                               break;
00120
00121                           case 'M':
00122                           case 'm':
00123                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      11);
00124                               break;
00125
00126                           case 'N':
00127                           case 'n':
00128                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      12);
00129                               break;
00130
00131                           case 'P':
00132                           case 'p':
00133                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      13);
00134                               break;
00135
00136                           case 'Q':
00137                           case 'q':
00138                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      14);
00139                               break;
00140
00141                           case 'R':
00142                           case 'r':
00143                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      15);
00144                               break;
00145
00146                           case 'S':
00147                           case 's':
00148                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      16);
00149                               break;
00150
00151                           case 'T':
00152                           case 't':
00153                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      17);
00154                               break;
00155
00156                           case 'V':
00157                           case 'v':
00158                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      18);
00159                               break;
00160
00161                           case 'W':
00162                           case 'w':
00163                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      19);
00164                               break;
```

```
00165
00166                           case 'Y':
00167                           case 'y':
00168                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      20);
00169                               break;
00170
00171                           // Selenocysteine
00172                           case 'U':
00173                           case 'u':
00174                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      21);
00175                               break;
00176
00177                           // Pyrrolysine
00178                           case 'O':
00179                           case 'o':
00180                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      22);
00181                               break;
00182
00183                           // Asparagine or aspartate
00184                           case 'B':
00185                           case 'b':
00186                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      23);
00187                               break;
00188
00189                           // Glutamine or glutamate
00190                           case 'Z':
00191                           case 'z':
00192                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      24);
00193                               break;
00194
00195                           // Leucine or isoleucine
00196                           case 'J':
00197                           case 'j':
00198                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      25);
00199                               break;
00200
00201                           // Any amino acid
00202                           case 'X':
00203                           case 'x':
00204                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      26);
00205                               break;
00206
00207                           // Stop codon
00208                           case '*':
00209                               ushortString[currIndex] += (ushort)((currPosition == 0 ?  1 :  28) *
      27);
00210                               break;
00211
00212                           default:
00213                               break;
00214                       }
00215
00216                       currPosition = (currPosition + 1) % 2;
00217
00218                       if (currPosition == 0)
00219                       {
00220                           currIndex++;
00221                       }
00222                   }
00223               }
00224           }
00225
00226           return tbr;
00227       }
00228
00229 /// <summary>
00230 /// Builds the match matrix used for the JC and K83 evolutionary models.
00231 /// </summary>
00232       private static void BuildProteinMatchMatrixJCK83()
00233       {
00234           for (int x = 0; x < 784; x++)
00235           {
00236               ProteinMatchMatrixJCK83[x] = new byte[784];
00237               for (int y = 0; y < 784; y++)
00238               {
00239                   // Every ushort contains 2 positions.  We cache all the possible comparisons
      between ushorts (i.e., between 2 positions).
00240                   // Here, we store the number of matches and the number of mismatches.  Each can
      have a value between 0 and 2, thus 2 bits
00241                   // for each and a total of 4 bits (stored as 1 byte).  The matrix should be small
```

```
    enough that this code doesn't need to be
00242                   // very optimised.
00243                   byte x1 = (byte)(x % 28);
00244                   byte y1 = (byte)(y % 28);
00245
00246                   byte x2 = (byte)(x / 28);
00247                   byte y2 = (byte)(y / 28);
00248
00249                   int match = 0;
00250                   int mismatch = 0;
00251
00252                   // Same letter, no gap.
00253                   if (x1 == y1 && x1 != 0)
00254                   {
00255                       match++;
00256                   }
00257                   // One is X, the other is not a gap.
00258                   else if ((x1 != 0 && y1 == 26) || (x1 == 26 && y1 != 0))
00259                   {
00260                       match++;
00261                   }
00262                   // One is B, the other is N or D.
00263                   else if (((x1 == 12 || x1 == 3) && y1 == 23) || ((y1 == 12 || y1 == 3) && x1 ==
    23))
00264                   {
00265                       match++;
00266                   }
00267                   // One is Z, the other is Q or E.
00268                   else if (((x1 == 14 || x1 == 4) && y1 == 24) || ((y1 == 14 || y1 == 4) && x1 ==
    24))
00269                   {
00270                       match++;
00271                   }
00272                   // One is J, the other is I or L.
00273                   else if (((x1 == 10 || x1 == 8) && y1 == 25) || ((y1 == 10 || y1 == 8) && x1 ==
    25))
00274                   {
00275                       match++;
00276                   }
00277                   // Different letter, and neither is a gap.
00278                   else if (x1 != y1 && x1 != 0 && y1 != 0)
00279                   {
00280                       mismatch++;
00281                   }
00282
00283                   // Same letter, no gap.
00284                   if (x2 == y2 && x2 != 0)
00285                   {
00286                       match++;
00287                   }
00288                   // One is X, the other is not a gap.
00289                   else if ((x2 != 0 && y2 == 26) || (x2 == 26 && y2 != 0))
00290                   {
00291                       match++;
00292                   }
00293                   // One is B, the other is N or D.
00294                   else if (((x2 == 12 || x2 == 3) && y2 == 23) || ((y2 == 12 || y2 == 3) && x2 ==
    23))
00295                   {
00296                       match++;
00297                   }
00298                   // One is Z, the other is Q or E.
00299                   else if (((x2 == 14 || x2 == 4) && y2 == 24) || ((y2 == 14 || y2 == 4) && x2 ==
    24))
00300                   {
00301                       match++;
00302                   }
00303                   // One is J, the other is I or L.
00304                   else if (((x2 == 10 || x2 == 8) && y2 == 25) || ((y2 == 10 || y2 == 8) && x2 ==
    25))
00305                   {
00306                       match++;
00307                   }
00308                   // Different letter, and neither is a gap.
00309                   else if (x2 != y2 && x2 != 0 && y2 != 0)
00310                   {
00311                       mismatch++;
00312                   }
00313
00314                   ProteinMatchMatrixJCK83[x][y] = (byte)(match + 4 * mismatch);
00315               }
00316           }
00317       }
00318
00319 /// <summary>
00320 /// Builds the match matrix used for the BLOSUM62 evolutionary model.
00321 /// </summary>
```

```
00322         private static void BuildProteinMatchMatrixBLOSUM62()
00323         {
00324             for (int x = 0; x < 784; x++)
00325             {
00326                 ProteinMatchMatrixBLOSUM62[x] = new byte[784];
00327                 for (int y = 0; y < 784; y++)
00328                 {
00329                     // Every ushort contains 2 positions.  We cache all the possible comparisons
      between ushorts (i.e., between 2 positions).
00330                     // Here, we store the score corresponding to these comparisons, as well as the
      number of non-gap positions.  Scores in
00331                     // the BLOSUM62 matrix range from -4 to 11, thus the comparison between two
      positions can score from -8 to 22.  We translate
00332                     // this to a value between 0 and 30, which takes up 5 bits.
00333                     // The matrix should be small enough that this code doesn't
00334                     // need to be very optimised.
00335                     byte x1 = (byte)(x % 28);
00336                     byte y1 = (byte)(y % 28);
00337
00338                     byte x2 = (byte)(x / 28);
00339                     byte y2 = (byte)(y / 28);
00340
00341                     byte score = (byte)(BLOSUM62[x1, y1] + BLOSUM62[x2, y2] + 8);
00342
00343                     byte length = 0;
00344
00345                     // Ignore gaps, U, O and J.
00346                     if ((x1 != 0 && x1 != 21 && x1 != 22 && x1 != 25) &&
00347                         (y1 != 0 && y1 != 21 && y1 != 22 && y1 != 25))
00348                     {
00349                         length++;
00350                     }
00351
00352                     // Ignore gaps, U, O and J.
00353                     if ((x2 != 0 && x2 != 21 && x2 != 22 && x2 != 25) &&
00354                         (y2 != 0 && y2 != 21 && y2 != 22 && y2 != 25))
00355                     {
00356                         length++;
00357                     }
00358
00359                     ProteinMatchMatrixBLOSUM62[x][y] = (byte)(score + (length << 5));
00360                 }
00361             }
00362         }
00363
00364 /// <summary>
00365 /// Compares two protein sequences using the Hamming distance.
00366 /// </summary>
00367 /// <param name="sequence1">The first sequence.</param>
00368 /// <param name="sequence2">The second sequence.</param>
00369 /// <returns>The (normalised) Hamming distance between the two sequences.</returns>
00370         private static float CompareProteinSequencesHamming(ushort[] sequence1, ushort[] sequence2)
00371         {
00372             int length = sequence1.Length;
00373
00374             int match = 0;
00375             int mismatch = 0;
00376
00377             unsafe
00378             {
00379                 fixed (ushort* seq1 = sequence1)
00380                 fixed (ushort* seq2 = sequence2)
00381                 {
00382                     for (int i = 0; i < length; i++)
00383                     {
00384                         byte result = ProteinMatchMatrixJCK83[seq1[i]][seq2[i]];
00385
00386                         match += result & 3;
00387                         mismatch += (result >> 2) & 3;
00388                     }
00389                 }
00390             }
00391
00392             return (float)mismatch / (match + mismatch);
00393         }
00394
00395
00396 /// <summary>
00397 /// Compares a protein sequence with two other sequences using the JC model of evolution.  Faster than
00398 /// calling <see cref="CompareProteinSequencesHamming(ushort[], ushort[])"/> twice.
00399 /// </summary>
00400 /// <param name="sequence1">The first sequence.</param>
00401 /// <param name="sequence2">The second sequence.</param>
00402 /// <param name="sequence3">The third sequence.</param>
00403 /// <param name="dist12">When the method returns, this variable will contain the JC distance between
      <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00404 /// <param name="dist13">When the method returns, this variable will contain the JC distance between
```

```
    <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00405         private static void CompareProteinSequencesHamming(ushort[] sequence1, ushort[] sequence2,
    ushort[] sequence3, out float dist12, out float dist13)
00406         {
00407             int length = sequence1.Length;
00408 
00409             int match12 = 0;
00410             int mismatch12 = 0;
00411 
00412             int match13 = 0;
00413             int mismatch13 = 0;
00414 
00415             unsafe
00416             {
00417                 fixed (ushort* seq1 = sequence1)
00418                 fixed (ushort* seq2 = sequence2)
00419                 fixed (ushort* seq3 = sequence3)
00420                 {
00421                     for (int i = 0; i < length; i++)
00422                     {
00423                         int i1 = seq1[i];
00424 
00425                         byte result12 = ProteinMatchMatrixJCK83[i1][seq2[i]];
00426 
00427                         match12 += result12 & 3;
00428                         mismatch12 += (result12 » 2) & 3;
00429 
00430                         byte result13 = ProteinMatchMatrixJCK83[i1][seq3[i]];
00431 
00432                         match13 += result13 & 3;
00433                         mismatch13 += (result13 » 2) & 3;
00434                     }
00435                 }
00436             }
00437 
00438             dist12 = (float)mismatch12 / (match12 + mismatch12);
00439             dist13 = (float)mismatch13 / (match13 + mismatch13);
00440         }
00441 
00442 /// <summary>
00443 /// Computes a distance matrix between protein sequences using the Hamming distance.
00444 /// </summary>
00445 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00446 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00447 /// <param name="progressCallback">A method used to report progress.</param>
00448 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
    matrix that will contain the distances between the sequences.</param>
00449         private static void ComputeDistanceMatrixHamming(IReadOnlyList<ushort[]> sequences, int
    numCores, Action<double> progressCallback, float[][] matrix)
00450         {
00451             long progress = 0;
00452             double total = sequences.Count * 0.5 * (sequences.Count - 1);
00453             object progressLock = new object();
00454 
00455             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
    }, i =>
00456             {
00457                 for (int j = 0; j < i - 1; j += 2)
00458                 {
00459                     CompareProteinSequencesHamming(sequences[i], sequences[j], sequences[j + 1], out
    matrix[i][j], out matrix[i][j + 1]);
00460                 }
00461 
00462                 for (int j = i - 1; j < i; j++)
00463                 {
00464                     matrix[i][j] = CompareProteinSequencesHamming(sequences[i], sequences[j]);
00465                 }
00466 
00467                 if (progressCallback != null)
00468                 {
00469                     lock (progressLock)
00470                     {
00471                         progress += i;
00472                         progressCallback((double)progress / total);
00473                     }
00474                 }
00475             });
00476         }
00477 
00478 /// <summary>
00479 /// Compares two protein sequences using the JC model of evolution.
00480 /// </summary>
00481 /// <param name="sequence1">The first sequence.</param>
00482 /// <param name="sequence2">The second sequence.</param>
00483 /// <returns>The JC distance between the two sequences.</returns>
00484         private static float CompareProteinSequencesJC(ushort[] sequence1, ushort[] sequence2)
00485         {
```

```
00486             int length = sequence1.Length;
00487
00488             int match = 0;
00489             int mismatch = 0;
00490
00491             unsafe
00492             {
00493                 fixed (ushort* seq1 = sequence1)
00494                 fixed (ushort* seq2 = sequence2)
00495                 {
00496                     for (int i = 0; i < length; i++)
00497                     {
00498                         byte result = ProteinMatchMatrixJCK83[seq1[i]][seq2[i]];
00499
00500                         match += result & 3;
00501                         mismatch += (result » 2) & 3;
00502                     }
00503                 }
00504             }
00505
00506             return (float)(-0.95 * Math.Log(1 - 1.052631578947368 * mismatch / (match + mismatch)));
00507         }
00508
00509
00510 /// <summary>
00511 /// Compares a protein sequence with two other sequences using the JC model of evolution.  Faster than
00512 /// calling <see cref="CompareProteinSequencesJC(ushort[], ushort[])"/> twice.
00513 /// </summary>
00514 /// <param name="sequence1">The first sequence.</param>
00515 /// <param name="sequence2">The second sequence.</param>
00516 /// <param name="sequence3">The third sequence.</param>
00517 /// <param name="dist12">When the method returns, this variable will contain the JC distance between
      <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00518 /// <param name="dist13">When the method returns, this variable will contain the JC distance between
      <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00519         private static void CompareProteinSequencesJC(ushort[] sequence1, ushort[] sequence2, ushort[]
      sequence3, out float dist12, out float dist13)
00520         {
00521             int length = sequence1.Length;
00522
00523             int match12 = 0;
00524             int mismatch12 = 0;
00525
00526             int match13 = 0;
00527             int mismatch13 = 0;
00528
00529             unsafe
00530             {
00531                 fixed (ushort* seq1 = sequence1)
00532                 fixed (ushort* seq2 = sequence2)
00533                 fixed (ushort* seq3 = sequence3)
00534                 {
00535                     for (int i = 0; i < length; i++)
00536                     {
00537                         int i1 = seq1[i];
00538
00539                         byte result12 = ProteinMatchMatrixJCK83[i1][seq2[i]];
00540
00541                         match12 += result12 & 3;
00542                         mismatch12 += (result12 » 2) & 3;
00543
00544                         byte result13 = ProteinMatchMatrixJCK83[i1][seq3[i]];
00545
00546                         match13 += result13 & 3;
00547                         mismatch13 += (result13 » 2) & 3;
00548                     }
00549                 }
00550             }
00551
00552             dist12 = (float)(-0.95 * Math.Log(1 - 1.052631578947368 * mismatch12 / (match12 +
      mismatch12)));
00553             dist13 = (float)(-0.95 * Math.Log(1 - 1.052631578947368 * mismatch13 / (match13 +
      mismatch13)));
00554         }
00555
00556 /// <summary>
00557 /// Computes a distance matrix between protein sequences using the JC model of evolution.
00558 /// </summary>
00559 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00560 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00561 /// <param name="progressCallback">A method used to report progress.</param>
00562 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
      matrix that will contain the distances between the sequences.</param>
00563         private static void ComputeDistanceMatrixJC(IReadOnlyList<ushort[]> sequences, int numCores,
      Action<double> progressCallback, float[][] matrix)
00564         {
00565             long progress = 0;
```

```
00566             double total = sequences.Count * 0.5 * (sequences.Count - 1);
00567             object progressLock = new object();
00568 
00569             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
      }, i =>
00570             {
00571                 for (int j = 0; j < i - 1; j += 2)
00572                 {
00573                     CompareProteinSequencesJC(sequences[i], sequences[j], sequences[j + 1], out
      matrix[i][j], out matrix[i][j + 1]);
00574                 }
00575 
00576                 for (int j = i - 1; j < i; j++)
00577                 {
00578                     matrix[i][j] = CompareProteinSequencesJC(sequences[i], sequences[j]);
00579                 }
00580 
00581                 if (progressCallback != null)
00582                 {
00583                     lock (progressLock)
00584                     {
00585                         progress += i;
00586                         progressCallback((double)progress / total);
00587                     }
00588                 }
00589             });
00590         }
00591 
00592 
00593 /// <summary>
00594 /// Compares two protein sequences using the K83 formula approximating the Dayhoff model of evolution.
00595 /// </summary>
00596 /// <param name="sequence1">The first sequence.</param>
00597 /// <param name="sequence2">The second sequence.</param>
00598 /// <returns>The K83 distance between the two sequences.</returns>
00599         private static float CompareProteinSequencesK83(ushort[] sequence1, ushort[] sequence2)
00600         {
00601             int length = sequence1.Length;
00602 
00603             int match = 0;
00604             int mismatch = 0;
00605 
00606             unsafe
00607             {
00608                 fixed (ushort* seq1 = sequence1)
00609                 fixed (ushort* seq2 = sequence2)
00610                 {
00611                     for (int i = 0; i < length; i++)
00612                     {
00613                         byte result = ProteinMatchMatrixJCK83[seq1[i]][seq2[i]];
00614 
00615                         match += result & 3;
00616                         mismatch += (result » 2) & 3;
00617                     }
00618                 }
00619             }
00620 
00621             double dist = (double)mismatch / (match + mismatch);
00622 
00623             if (dist < 0.75)
00624             {
00625                 return (float)-Math.Log(1 - dist - 0.2 * dist * dist);
00626             }
00627             else if (dist <= 0.93)
00628             {
00629                 return DayhoffPAMs[(int)((dist * 1000) - 750)] * 0.01f;
00630             }
00631             else
00632             {
00633                 return 10;
00634             }
00635         }
00636 
00637 
00638 /// <summary>
00639 /// Compares a protein sequence with two other sequences using the K83 formula approximating the
      Dayhoff model of evolution.  Faster than
00640 /// calling <see cref="CompareProteinSequencesK83(ushort[], ushort[])"/> twice.
00641 /// </summary>
00642 /// <param name="sequence1">The first sequence.</param>
00643 /// <param name="sequence2">The second sequence.</param>
00644 /// <param name="sequence3">The third sequence.</param>
00645 /// <param name="dist12">When the method returns, this variable will contain the K83 distance between
      <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00646 /// <param name="dist13">When the method returns, this variable will contain the K83 distance between
      <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00647         private static void CompareProteinSequencesK83(ushort[] sequence1, ushort[] sequence2,
```

```
      ushort[] sequence3, out float dist12, out float dist13)
00648         {
00649             int length = sequence1.Length;
00650 
00651             int match12 = 0;
00652             int mismatch12 = 0;
00653 
00654             int match13 = 0;
00655             int mismatch13 = 0;
00656 
00657             unsafe
00658             {
00659                 fixed (ushort* seq1 = sequence1)
00660                 fixed (ushort* seq2 = sequence2)
00661                 fixed (ushort* seq3 = sequence3)
00662                 {
00663                     for (int i = 0; i < length; i++)
00664                     {
00665                         int i1 = seq1[i];
00666 
00667                         byte result12 = ProteinMatchMatrixJCK83[i1][seq2[i]];
00668 
00669                         match12 += result12 & 3;
00670                         mismatch12 += (result12 » 2) & 3;
00671 
00672                         byte result13 = ProteinMatchMatrixJCK83[i1][seq3[i]];
00673 
00674                         match13 += result13 & 3;
00675                         mismatch13 += (result13 » 2) & 3;
00676                     }
00677                 }
00678             }
00679 
00680             double tmpDist12 = (double)mismatch12 / (match12 + mismatch12);
00681 
00682             if (tmpDist12 < 0.75)
00683             {
00684                 dist12 = (float)-Math.Log(1 - tmpDist12 - 0.2 * tmpDist12 * tmpDist12);
00685             }
00686             else if (tmpDist12 <= 0.93)
00687             {
00688                 dist12 = DayhoffPAMs[(int)((tmpDist12 * 1000) - 750)] * 0.01f;
00689             }
00690             else
00691             {
00692                 dist12 = 10;
00693             }
00694 
00695             double tmpDist13 = (double)mismatch13 / (match13 + mismatch13);
00696 
00697             if (tmpDist13 < 0.75)
00698             {
00699                 dist13 = (float)-Math.Log(1 - tmpDist13 - 0.2 * tmpDist13 * tmpDist13);
00700             }
00701             else if (tmpDist13 <= 0.93)
00702             {
00703                 dist13 = DayhoffPAMs[(int)((tmpDist13 * 1000) - 750)] * 0.01f;
00704             }
00705             else
00706             {
00707                 dist13 = 10;
00708             }
00709         }
00710 
00711 /// <summary>
00712 /// Computes a distance matrix between protein sequences using the K83 formula approximating the
      Dayhoff model of evolution.
00713 /// </summary>
00714 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00715 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00716 /// <param name="progressCallback">A method used to report progress.</param>
00717 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
      matrix that will contain the distances between the sequences.</param>
00718         private static void ComputeDistanceMatrixK83(IReadOnlyList<ushort[]> sequences, int numCores,
      Action<double> progressCallback, float[][] matrix)
00719         {
00720             long progress = 0;
00721             double total = sequences.Count * 0.5 * (sequences.Count - 1);
00722             object progressLock = new object();
00723 
00724             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
      }, i =>
00725             {
00726                 for (int j = 0; j < i - 1; j += 2)
00727                 {
00728                     CompareProteinSequencesK83(sequences[i], sequences[j], sequences[j + 1], out
      matrix[i][j], out matrix[i][j + 1]);
```

```
00729                 }
00730
00731                 for (int j = i - 1; j < i; j++)
00732                 {
00733                     matrix[i][j] = CompareProteinSequencesK83(sequences[i], sequences[j]);
00734                 }
00735
00736                 if (progressCallback != null)
00737                 {
00738                     lock (progressLock)
00739                     {
00740                         progress += i;
00741                         progressCallback((double)progress / total);
00742                     }
00743                 }
00744             });
00745         }
00746
00747 /// <summary>
00748 /// Computes the BLOSUM62 score of a sequence with itself.
00749 /// </summary>
00750 /// <param name="sequence">The sequence whose score will be computed.</param>
00751 /// <returns>The BLOSUM62 score of a sequence with itself.</returns>
00752         private static int SelfScore(ushort[] sequence)
00753         {
00754             int length = sequence.Length;
00755             int score = 0;
00756
00757             unsafe
00758             {
00759                 fixed (ushort* seq1 = sequence)
00760                 {
00761                     for (int i = 0; i < length; i++)
00762                     {
00763                         score += (ProteinMatchMatrixBLOSUM62[seq1[i]][seq1[i]] & 31) - 8;
00764                     }
00765                 }
00766             }
00767
00768             return score;
00769         }
00770
00771 /// <summary>
00772 /// Compares two protein sequences using the Scoredist correction with the BLOSUM62 matrix.
00773 /// </summary>
00774 /// <param name="sequence1">The first sequence.</param>
00775 /// <param name="sequence2">The second sequence.</param>
00776 /// <param name="selfScore1">The score obtained by comparing <paramref name="sequence1"/> with
      itself.</param>
00777 /// <param name="selfScore2">The score obtained by comparing <paramref name="sequence2"/> with
      itself.</param>
00778 /// <returns>The distance between the two sequences.</returns>
00779         public static float CompareProteinSequencesBLOSUM62(ushort[] sequence1, ushort[] sequence2,
      int selfScore1, int selfScore2)
00780         {
00781             int length = sequence1.Length;
00782
00783             int score = 0;
00784             int seqLength = 0;
00785
00786             unsafe
00787             {
00788                 fixed (ushort* seq1 = sequence1)
00789                 fixed (ushort* seq2 = sequence2)
00790                 {
00791                     for (int i = 0; i < length; i++)
00792                     {
00793                         byte result = ProteinMatchMatrixBLOSUM62[seq1[i]][seq2[i]];
00794                         score += (result & 31) - 8;
00795                         seqLength += result >> 5;
00796                     }
00797                 }
00798             }
00799
00800             double expectedScore = seqLength * -0.5209;
00801
00802             double sigmaN = score - expectedScore;
00803
00804             if (sigmaN <= 0)
00805             {
00806                 return 300;
00807             }
00808
00809             double sigmaUN = (selfScore1 + selfScore2) * 0.5 - expectedScore;
00810
00811             return Math.Min(300, (float)(-Math.Log(sigmaN / sigmaUN) * 133.70));
00812         }
```

```
00813
00814 /// <summary>
00815 /// Compares a protein sequence with two other sequences using the Scoredist correction with the
      BLOSUM62 matrix.  Faster than
00816 /// calling <see cref="CompareProteinSequencesBLOSUM62(ushort[], ushort[], int, int)"/> twice.
00817 /// </summary>
00818 /// <param name="sequence1">The first sequence.</param>
00819 /// <param name="sequence2">The second sequence.</param>
00820 /// <param name="sequence3">The third sequence.</param>
00821 /// <param name="selfScore1">The score obtained by comparing <paramref name="sequence1"/> with
      itself.</param>
00822 /// <param name="selfScore2">The score obtained by comparing <paramref name="sequence2"/> with
      itself.</param>
00823 /// <param name="selfScore3">The score obtained by comparing <paramref name="sequence3"/> with
      itself.</param>
00824 /// <param name="dist12">When the method returns, this variable will contain the distance between
      <paramref name="sequence1"/> and <paramref name="sequence2"/>.</param>
00825 /// <param name="dist13">When the method returns, this variable will contain the distance between
      <paramref name="sequence1"/> and <paramref name="sequence3"/>.</param>
00826         private static void CompareProteinSequencesBLOSUM62(ushort[] sequence1, ushort[] sequence2,
      ushort[] sequence3, int selfScore1, int selfScore2, int selfScore3, out float dist12, out float
      dist13)
00827         {
00828             int length = sequence1.Length;
00829
00830             int score12 = 0;
00831             int score13 = 0;
00832
00833             int seqLength12 = 0;
00834             int seqLength13 = 0;
00835
00836             unsafe
00837             {
00838                 fixed (ushort* seq1 = sequence1)
00839                 fixed (ushort* seq2 = sequence2)
00840                 fixed (ushort* seq3 = sequence3)
00841                 {
00842                     for (int i = 0; i < length; i++)
00843                     {
00844                         byte result12 = ProteinMatchMatrixBLOSUM62[seq1[i]][seq2[i]];
00845                         byte result13 = ProteinMatchMatrixBLOSUM62[seq1[i]][seq3[i]];
00846
00847                         score12 += (result12 & 31) - 8;
00848                         score13 += (result13 & 31) - 8;
00849
00850                         seqLength12 += result12 >> 5;
00851                         seqLength13 += result13 >> 5;
00852                     }
00853                 }
00854             }
00855
00856             double expectedScore12 = seqLength12 * -0.5209;
00857             double expectedScore13 = seqLength13 * -0.5209;
00858
00859             double sigmaN12 = score12 - expectedScore12;
00860
00861             if (sigmaN12 <= 0)
00862             {
00863                 dist12 = 300;
00864             }
00865             else
00866             {
00867                 double sigmaUN12 = (selfScore1 + selfScore2) * 0.5 - expectedScore12;
00868                 dist12 = Math.Min(300, (float)(-Math.Log(sigmaN12 / sigmaUN12) * 133.70));
00869             }
00870
00871             double sigmaN13 = score13 - expectedScore13;
00872
00873             if (sigmaN13 <= 0)
00874             {
00875                 dist13 = 300;
00876             }
00877             else
00878             {
00879                 double sigmaUN13 = (selfScore1 + selfScore3) * 0.5 - expectedScore13;
00880                 dist13 = Math.Min(300, (float)(-Math.Log(sigmaN13 / sigmaUN13) * 133.70));
00881             }
00882         }
00883
00884 /// <summary>
00885 /// Computes a distance matrix between protein sequences using the Scoredist correction with the
      BLOSUM62 matrix.
00886 /// </summary>
00887 /// <param name="sequences">The sequences whose distance matrix will be computed.</param>
00888 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00889 /// <param name="progressCallback">A method used to report progress.</param>
00890 /// <param name="matrix">A pre-allocated lower triangular <see cref="T:float[][]"/> jagged array
```

```
matrix that will contain the distances between the sequences.</param>
00891         private static void ComputeDistanceMatrixBLOSUM62(IReadOnlyList<ushort[]> sequences, int
numCores, Action<double> progressCallback, float[][] matrix)
00892         {
00893             int[] selfScores = new int[sequences.Count];
00894 
00895             long progress = 0;
00896             double total = sequences.Count * 0.5 * (sequences.Count - 1) + sequences.Count;
00897             object progressLock = new object();
00898 
00899             Parallel.For(0, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
}, i =>
00900             {
00901                 selfScores[i] = SelfScore(sequences[i]);
00902             });
00903 
00904             if (progressCallback != null)
00905             {
00906                 lock (progressLock)
00907                 {
00908                     progress += sequences.Count;
00909                     progressCallback((double)progress / total);
00910                 }
00911             }
00912 
00913             Parallel.For(1, sequences.Count, new ParallelOptions() { MaxDegreeOfParallelism = numCores
}, i =>
00914             {
00915                 for (int j = 0; j < i - 1; j += 2)
00916                 {
00917                     CompareProteinSequencesBLOSUM62(sequences[i], sequences[j], sequences[j + 1],
selfScores[i], selfScores[j], selfScores[j + 1], out matrix[i][j], out matrix[i][j + 1]);
00918                 }
00919 
00920                 for (int j = i - 1; j < i; j++)
00921                 {
00922                     matrix[i][j] = CompareProteinSequencesBLOSUM62(sequences[i], sequences[j],
selfScores[i], selfScores[j]);
00923                 }
00924 
00925                 if (progressCallback != null)
00926                 {
00927                     lock (progressLock)
00928                     {
00929                         progress += i;
00930                         progressCallback((double)progress / total);
00931                     }
00932                 }
00933             });
00934         }
00935     }
00936 }
00937 
```

8.21 MatrixExponential.cs

```
00001 using MathNet.Numerics;
00002 using MathNet.Numerics.LinearAlgebra;
00003 using MathNet.Numerics.LinearAlgebra.Double;
00004 using MathNet.Numerics.LinearAlgebra.Factorization;
00005 using System;
00006 using System.Collections.Generic;
00007 using System.Numerics;
00008 using System.Text;
00009 
00010 namespace PhyloTree.TreeBuilding
00011 {
00012     internal class MatrixExponential
00013     {
00014         public Matrix<double> Result;
00015         public bool Exact;
00016         public Matrix<Complex> P;
00017         public Matrix<Complex> PInv;
00018         public Matrix<Complex> D;
00019 
00020         public MatrixExponential(Matrix<double> result, Matrix<Complex> p, Matrix<Complex> pInv,
Matrix<Complex> d)
00021         {
00022             Result = result;
00023             P = p;
00024             PInv = pInv;
00025             Exact = p != null;
00026             D = d;
```

```
00027         }
00028     }
00029
00030     internal static class MatrixExtensions
00031     {
00032         static bool ForcePade = false;
00033
00034         public static MatrixExponential FastExponential(this Matrix<double> mat, double t,
      MatrixExponential cachedResult = null)
00035         {
00036             if (ForcePade)
00037             {
00038                 return new MatrixExponential((mat * t).PadeExponential().PointwiseAbs(), null, null,
      null);
00039             }
00040
00041             if (cachedResult == null)
00042             {
00043                 try
00044                 {
00045                     Matrix<Complex> m = mat.ToComplex();
00046                     Evd<Complex> evd = m.Evd();
00047
00048                     HashSet<Complex> eigenValues = new HashSet<Complex>();
00049
00050                     for (int i = 0; i < evd.EigenValues.Count; i++)
00051                     {
00052                         if (Math.Abs(evd.EigenValues[i].Real) < 1e-5)
00053                         {
00054                             evd.EigenValues[i] = new Complex(0, evd.EigenValues[i].Imaginary);
00055                         }
00056
00057                         if (Math.Abs(evd.EigenValues[i].Imaginary) < 1e-5)
00058                         {
00059                             evd.EigenValues[i] = new Complex(evd.EigenValues[i].Real, 0);
00060                         }
00061
00062                         eigenValues.Add(evd.EigenValues[i]);
00063                     }
00064
00065                     if (eigenValues.Count == m.ColumnCount)
00066                     {
00067                         //Diagonalizable
00068
00069                         Matrix<Complex> inv = evd.EigenVectors.Inverse();
00070                         Matrix<Complex> eig = evd.EigenVectors;
00071                         Matrix<Complex> diag =
      Matrix<Complex>.Build.DenseOfDiagonalVector(evd.EigenValues);
00072
00073                         return new MatrixExponential((eig * (diag * t).DiagonalExp() *
      inv).Real().PointwiseAbs(), eig, inv, diag);
00074                     }
00075                     else
00076                     {
00077                         //Might not diagonalizable:  fallback to Padé approximation [note:  this
      happens "almost never"]
00078                         return new MatrixExponential((mat * t).PadeExponential().PointwiseAbs(), null,
      null, null);
00079                     }
00080                 }
00081                 catch
00082                 {
00083                     //Error during diagonalization:  fallback to Padé approximation
00084                     return new MatrixExponential((mat * t).PadeExponential().PointwiseAbs(), null,
      null, null);
00085                 }
00086             }
00087             else
00088             {
00089                 if (cachedResult.Exact)
00090                 {
00091                     return new MatrixExponential((cachedResult.P * (cachedResult.D * t).DiagonalExp()
      * cachedResult.PInv).Real().PointwiseAbs(), cachedResult.P, cachedResult.PInv, cachedResult.D);
00092                 }
00093                 else
00094                 {
00095                     return new MatrixExponential((mat * t).PadeExponential().PointwiseAbs(), null,
      null, null);
00096                 }
00097             }
00098         }
00099
00100         static Matrix<Complex> DiagonalExp(this Matrix<Complex> m)
00101         {
00102             Matrix<Complex> tbr = Matrix<Complex>.Build.DenseOfMatrix(m);
00103
00104             for (int i = 0; i < m.ColumnCount; i++)
```

```
00105             {
00106                 tbr[i, i] = tbr[i, i].Exp();
00107             }
00108 
00109             return tbr;
00110         }
00111 
00112 
00113         public static void TimesLogVectorAndAdd(this Matrix<double> mat, double[] logVector, double[]
      addToVector)
00114         {
00115             int maxInd = logVector.MaxInd();
00116 
00117             for (int i = 0; i < mat.RowCount; i++)
00118             {
00119                 double toBeAdded = logVector[maxInd] + Math.Log(mat[i, maxInd]);
00120 
00121                 double log1pArg = 0;
00122 
00123                 for (var j = 0; j < mat.ColumnCount; j++)
00124                 {
00125                     if (j != maxInd)
00126                     {
00127                         log1pArg += mat[i, j] / mat[i, maxInd] * Math.Exp(logVector[j] -
      logVector[maxInd]);
00128                     }
00129                 }
00130 
00131                 if (!double.IsNaN(log1pArg))
00132                 {
00133                     toBeAdded += Log1p(log1pArg);
00134                     addToVector[i] += toBeAdded;
00135                     if (addToVector[i] > 0)
00136                     {
00137                         addToVector[i] = double.NaN;
00138                     }
00139                 }
00140                 else
00141                 {
00142                     double logArg = 0;
00143                     for (var j = 0; j < mat.ColumnCount; j++)
00144                     {
00145                         logArg += mat[i, j] * Math.Exp(logVector[j]);
00146                     }
00147 
00148                     addToVector[i] += Math.Log(logArg);
00149 
00150                     if (addToVector[i] > 0)
00151                     {
00152                         addToVector[i] = double.NaN;
00153                     }
00154                 }
00155             }
00156         }
00157 
00158         private static int MaxInd(this double[] arr)
00159         {
00160             int tbr = 0;
00161 
00162             for (int i = 0; i < arr.Length; i++)
00163             {
00164                 if (arr[i] > arr[tbr])
00165                 {
00166                     tbr = i;
00167                 }
00168             }
00169 
00170             return tbr;
00171         }
00172 
00173         private static double Log1p(double x)
00174         {
00175             if (x <= -1)
00176             {
00177                 return double.NegativeInfinity;
00178             }
00179             else if (Math.Abs(x) > 0.0001)
00180             {
00181                 return Math.Log(1 + x);
00182             }
00183             else
00184             {
00185                 return (1 - 0.5 * x) * x;
00186             }
00187         }
00188 
00189         //Adapted from
```

```
    https://github.com/horribleheffalump/MatrixExponential/blob/master/MatrixExponential.cs
00190         public static Matrix<double> PadeExponential(this Matrix<double> m)
00191         {
00192             Matrix<double> exp_m = null;
00193 
00194             // last hope:  Padé approximation method
00195             // details could be found in
00196             // M.Arioli, B.Codenotti, C.Fassino The Padé method for computing the matrix exponential
    // Linear Algebra and its Applications, 1996, V. 240, P. 111-130
00197             // https://www.sciencedirect.com/science/article/pii/0024379594001901
00198 
00199             //Assume that matrix is not diagonalizable (otherwise, FastExponential would have succeded
00200 
00201             int p = 5; // order of Padé
00202 
00203             // high matrix norm may result in high roundoff erroros,
00204             // so first we have to find normalizing coefficient such that || m / norm_coeff || < 0.5
00205             // to reduce the following computations we set it norm_coeff = 2^k
00206 
00207             double k = 0;
00208             double mNorm = m.L1Norm();
00209             if (mNorm > 0.5)
00210             {
00211                 k = Math.Ceiling(Math.Log(mNorm) / Math.Log(2.0));
00212                 m = m / Math.Pow(2.0, k);
00213             }
00214 
00215             Matrix<double> N = DenseMatrix.CreateIdentity(m.RowCount);
00216             Matrix<double> D = DenseMatrix.CreateIdentity(m.RowCount);
00217             Matrix<double> m_pow_j = m;
00218 
00219             int q = p; // here we use simmetric approximation, but in general p may not be equal to q.
00220             for (int j = 1; j <= Math.Max(p, q); j++)
00221             {
00222                 if (j > 1)
00223                     m_pow_j = m_pow_j * m;
00224                 if (j <= p)
00225                     N = N + SpecialFunctions.Factorial(p + q - j) * SpecialFunctions.Factorial(p) /
    SpecialFunctions.Factorial(p + q) / SpecialFunctions.Factorial(j) / SpecialFunctions.Factorial(p - j)
    * m_pow_j;
00226                 if (j <= q)
00227                     D = D + Math.Pow(-1.0, j) * SpecialFunctions.Factorial(p + q - j) *
    SpecialFunctions.Factorial(q) / SpecialFunctions.Factorial(p + q) / SpecialFunctions.Factorial(j) /
    SpecialFunctions.Factorial(q - j) * m_pow_j;
00228             }
00229 
00230             // calculate inv(D)*N with LU decomposition
00231             exp_m = D.LU().Solve(N);
00232 
00233             // denormalize if need
00234             if (k > 0)
00235             {
00236                 for (int i = 0; i < k; i++)
00237                 {
00238                     exp_m = exp_m * exp_m;
00239                 }
00240             }
00241 
00242             return exp_m;
00243         }
00244     }
00245 }
```

8.22 NeighborJoining.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Linq;
00004 using System.Runtime.InteropServices;
00005 using System.Threading;
00006 using System.Threading.Tasks;
00007 
00008 namespace PhyloTree.TreeBuilding
00009 {
00010 /// <summary>
00011 /// Contains methods to compute neighbour-joining trees.
00012 /// </summary>
00013     public static class NeighborJoining
00014     {
00015 /// <summary>
00016 /// Builds a neighbour-joining tree using data from a sequence alignment.  This method first computes
    a distance matrix from the sequence alignment, and then uses the distance matrix to compute the tree.
00017 /// </summary>
```

```
00018 /// <param name="alignment">The sequence alignment.</param>
00019 /// <param name="evolutionModel">The evolutionary model to use when computing the distance
      matrix.</param>
00020 /// <param name="bootstrapReplicates">The number of bootstrap replicates to perform.</param>
00021 /// <param name="alignmentType">The type of sequence alignment (DNA, protein, or autodetect).</param>
00022 /// <param name="constraint">An optional tree to constrain the search.  The tree produced by this
      method will be compatible with this tree.  The constraint tree can be multifurcating.</param>
00023 /// <param name="allowNegativeBranches">If this is <see langword="true"/> negative branches produced
      by the neighbour-joining algorithm are left untouched; otherwise, their (absolute) length is added to
      the
00024 /// sibling branch, and the negative length is set to 0.</param>
00025 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00026 /// <param name="progressCallback">A method used to report progress.</param>
00027 /// <returns>The neighbour-joining tree built from the supplied <paramref
      name="alignment"/>.</returns>
00028         public static TreeNode BuildTree(Dictionary<string, string> alignment, EvolutionModel
      evolutionModel = EvolutionModel.Kimura, int bootstrapReplicates = 0, AlignmentType alignmentType =
      AlignmentType.Autodetect, TreeNode constraint = null, bool allowNegativeBranches = true, int numCores
      = -1, Action<double> progressCallback = null)
00029         {
00030             List<string> sequenceNames = alignment.Keys.ToList();
00031             List<string> sequences = alignment.Values.ToList();
00032 
00033             if (bootstrapReplicates == 0)
00034             {
00035                 Action<double> distMatProgress = null;
00036                 Action<double> treeProgress = null;
00037 
00038                 if (progressCallback != null)
00039                 {
00040                     distMatProgress = x => progressCallback(x * 0.5);
00041                     treeProgress = x => progressCallback(0.5 + x * 0.5);
00042                 }
00043 
00044                 float[][] distMat = DistanceMatrix.BuildFromAlignment(sequences, alignmentType,
      evolutionModel, numCores, distMatProgress);
00045 
00046                 return BuildTree(distMat, sequenceNames, constraint, copyMatrix:  false,
      allowNegativeBranches:  allowNegativeBranches, numCores:  numCores, progressCallback:  treeProgress);
00047             }
00048             else
00049             {
00050                 float[][] distMat = DistanceMatrix.BuildFromAlignment(sequences, alignmentType,
      evolutionModel, numCores);
00051 
00052                 TreeNode initialTree = BuildTree(distMat, sequenceNames, constraint, copyMatrix:
      false, allowNegativeBranches:  allowNegativeBranches, numCores:  numCores);
00053 
00054                 progressCallback?.Invoke(1.0 / (bootstrapReplicates + 1));
00055 
00056                 Dictionary<string, int> sequenceIndices = new Dictionary<string,
      int>(sequenceNames.Count);
00057                 for (int i = 0; i < sequenceNames.Count; i++)
00058                 {
00059                     sequenceIndices[sequenceNames[i]] = i;
00060                 }
00061 
00062                 List<int[][]> splits = GetSplits(initialTree, sequenceIndices);
00063                 List<TreeNode> nodes = initialTree.GetChildrenRecursive();
00064                 int[] supports = new int[splits.Count];
00065                 object supportLock = new object();
00066 
00067                 (HashSet<int>, HashSet<int>)[] setSplits = new (HashSet<int>,
      HashSet<int>)[splits.Count];
00068 
00069                 for (int i = 0; i < splits.Count; i++)
00070                 {
00071                     setSplits[i] = (new HashSet<int>(splits[i][0]), new HashSet<int>(splits[i][1]));
00072                 }
00073 
00074                 int completed = 0;
00075                 object progressLock = new object();
00076 
00077                 ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = numCores };
00078 
00079                 Parallel.For(0, bootstrapReplicates, opt, i =>
00080                 {
00081                     float[][] currDistMat = DistanceMatrix.BootstrapReplicateFromAlignment(sequences,
      alignmentType, evolutionModel);
00082 
00083                     TreeNode tree = BuildTree(currDistMat, sequenceNames, constraint, copyMatrix:
      false, allowNegativeBranches:  allowNegativeBranches, numCores:  1);
00084 
00085                     currDistMat = null;
00086 
00087                     List<int[][]> newSplits = GetSplits(tree, sequenceIndices);
00088 
```

```
00089                     List<int> supported = new List<int>(nodes.Count);
00090
00091                     Parallel.For(0, splits.Count, opt, j =>
00092                     {
00093                         if (nodes[j].Children.Count == 0 || IsCompatible(setSplits[j].Item1,
      setSplits[j].Item2, newSplits))
00094                         {
00095                             Interlocked.Increment(ref supports[j]);
00096                         }
00097                     });
00098
00099                     if (progressCallback != null)
00100                     {
00101                         lock (progressLock)
00102                         {
00103                             completed++;
00104                             progressCallback?.Invoke((double)completed / (bootstrapReplicates + 1));
00105                         }
00106                     }
00107                 });
00108
00109                 for (int i = 0; i < nodes.Count; i++)
00110                 {
00111                     nodes[i].Support = (double)supports[i] / bootstrapReplicates;
00112                 }
00113
00114                 return initialTree;
00115             }
00116         }
00117
00118 /// <summary>
00119 /// Builds a neighbour-joining tree using data from a distance matrix.
00120 /// </summary>
00121 /// <param name="distanceMatrix">The distance matrix containg distances between the taxa.  This can be
      a lower triangular matrix or a full matrix; values above the diagonal will not be used.</param>
00122 /// <param name="sequenceNames">The names of the taxa.  The indices of this list should correspond to
      the rows and columns of the <paramref name="distanceMatrix"/>.</param>
00123 /// <param name="constraint">An optional tree to constrain the search.  The tree produced by this
      method will be compatible with this tree.  The constraint tree can be multifurcating.</param>
00124 /// <param name="copyMatrix">If this is <see langword="true"/>, the matrix is copied before using it
      to compute the tree.  If this is <see langword="false"/>, the matrix is not copied.  Copying the
      matrix
00125 /// increases the memory used by the method, but note that if the matrix is not copied, it will be
      modified in-place!</param>
00126 /// <param name="allowNegativeBranches">If this is <see langword="true"/> negative branches produced
      by the neighbour-joining algorithm are left untouched; otherwise, their (absolute) length is added to
      the
00127 /// sibling branch, and the negative length is set to 0.</param>
00128 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00129 /// <param name="progressCallback">A method used to report progress.</param>
00130 /// <returns>The neighbour-joining tree built from the supplied <paramref
      name="distanceMatrix"/>.</returns>
00131         public static TreeNode BuildTree(float[][] distanceMatrix, IReadOnlyList<string>
      sequenceNames, TreeNode constraint = null, bool copyMatrix = true, bool allowNegativeBranches = true,
      int numCores = -1, Action<double> progressCallback = null)
00132         {
00133             if (copyMatrix)
00134             {
00135                 float[][] newDistMat = new float[distanceMatrix.Length][];
00136
00137                 for (int i = 0; i < distanceMatrix.Length; i++)
00138                 {
00139                     newDistMat[i] = new float[i];
00140
00141                     for (int j = 0; j < i; j++)
00142                     {
00143                         newDistMat[i][j] = distanceMatrix[i][j];
00144                     }
00145                 }
00146
00147                 distanceMatrix = newDistMat;
00148             }
00149
00150             unsafe
00151             {
00152                 IntPtr[] rows = new IntPtr[distanceMatrix.Length];
00153
00154                 GCHandle[] handles = new GCHandle[distanceMatrix.Length];
00155
00156                 for (int i = 0; i < distanceMatrix.Length; i++)
00157                 {
00158                     handles[i] = GCHandle.Alloc(distanceMatrix[i], GCHandleType.Pinned);
00159                     rows[i] = handles[i].AddrOfPinnedObject();
00160                 }
00161
00162                 fixed (IntPtr* rowsPointer = rows)
00163                 {
```

```
00164                     float** rowsFloatPointer = (float**)rowsPointer;
00165
00166                     TreeNode tree;
00167
00168                     if (constraint == null)
00169                     {
00170                         tree = BuildTree(rowsFloatPointer, sequenceNames, numCores,
      allowNegativeBranches, progressCallback);
00171                     }
00172                     else
00173                     {
00174                         tree = BuildTreeWithConstraint(rowsFloatPointer, sequenceNames, constraint,
      numCores, allowNegativeBranches, progressCallback);
00175                     }
00176
00177                     for (int i = 0; i < distanceMatrix.Length; i++)
00178                     {
00179                         handles[i].Free();
00180                     }
00181
00182                     return tree;
00183                 }
00184             }
00185         }
00186
00187 /// <summary>
00188 /// Builds a neighbour-joining tree using data from a distance matrix.
00189 /// </summary>
00190 /// <param name="distanceMatrix">The distance matrix containg distances between the taxa.  This can be
      a lower triangular matrix or a full matrix; values above the diagonal will not be used.</param>
00191 /// <param name="sequenceNames">The names of the taxa.  The indices of this list should correspond to
      the rows and columns of the <paramref name="distanceMatrix"/>.</param>
00192 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00193 /// <param name="allowNegativeBranches">If this is <see langword="true"/> negative branches produced
      by the neighbour-joining algorithm are left untouched; otherwise, their (absolute) length is added to
      the
00194 /// sibling branch, and the negative length is set to 0.</param>
00195 /// <param name="progressCallback">A method used to report progress.</param>
00196 /// <returns>The neighbour-joining tree built from the supplied <paramref
      name="distanceMatrix"/>.</returns>
00197         private static unsafe TreeNode BuildTree(float** distanceMatrix, IReadOnlyList<string>
      sequenceNames, int numCores, bool allowNegativeBranches, Action<double> progressCallback)
00198         {
00199             List<TreeNode> currentLeaves = new List<TreeNode>(sequenceNames.Count);
00200             List<int> correspondences = new List<int>(sequenceNames.Count);
00201
00202             fixed (double* sums = new double[sequenceNames.Count])
00203             {
00204                 IntPtr sumsPtr = (IntPtr)sums;
00205
00206                 for (int i = 0; i < sequenceNames.Count; i++)
00207                 {
00208                     currentLeaves.Add(new TreeNode(null) { Name = sequenceNames[i] });
00209                     correspondences.Add(i);
00210
00211                     sums[i] = 0;
00212                     for (int k = 0; k < sequenceNames.Count; k++)
00213                     {
00214                         if (k != i)
00215                         {
00216                             sums[i] += distanceMatrix[Math.Max(i, k)][Math.Min(i, k)];
00217                         }
00218                     }
00219                 }
00220
00221                 double totalToProcess = ((double)sequenceNames.Count * (sequenceNames.Count + 1) * (2
      * sequenceNames.Count + 1)) / 12 + 0.25 * sequenceNames.Count * (sequenceNames.Count + 1) - 4;
00222                 long processed = 0;
00223
00224                 ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = numCores };
00225
00226                 while (currentLeaves.Count > 1)
00227                 {
00228                     long newStep = (long)currentLeaves.Count * (currentLeaves.Count + 1) / 2;
00229                     progressCallback?.Invoke(processed / totalToProcess);
00230
00231                     int minI = -1;
00232                     int minJ = -1;
00233
00234                     double minDist = double.MaxValue;
00235                     double minSumI = double.NaN;
00236                     double minSumJ = double.NaN;
00237
00238                     double[] minDists = new double[currentLeaves.Count];
00239                     int[] minJs = new int[currentLeaves.Count];
00240
00241                     Parallel.For(0, currentLeaves.Count, opt, i =>
```

```
00242                     {
00243                         double* mySums = (double*)sumsPtr;
00244 
00245                         double myMinDist = double.MaxValue;
00246                         int myMinJ = -1;
00247 
00248                         int realI = correspondences[i];
00249                         double sumI = mySums[realI];
00250 
00251                         for (int j = 0; j < i; j++)
00252                         {
00253                             int realJ = correspondences[j];
00254 
00255                             double dist = distanceMatrix[realI][realJ] * (currentLeaves.Count - 2);
00256                             double sumJ = mySums[realJ];
00257 
00258                             dist -= sumI + sumJ;
00259 
00260                             if (dist < myMinDist)
00261                             {
00262                                 myMinDist = dist;
00263                                 myMinJ = j;
00264                             }
00265                         }
00266 
00267                         minDists[i] = myMinDist;
00268                         minJs[i] = myMinJ;
00269                     });
00270 
00271                     for (int i = 0; i < currentLeaves.Count; i++)
00272                     {
00273                         if (minDists[i] < minDist)
00274                         {
00275                             minDist = minDists[i];
00276                             minI = i;
00277                             minSumI = sums[correspondences[i]];
00278                             minJ = minJs[i];
00279                             minSumJ = sums[correspondences[minJ]];
00280                         }
00281                     }
00282 
00283                     minDist = distanceMatrix[correspondences[minI]][correspondences[minJ]];
00284 
00285                     TreeNode newNode = new TreeNode(null);
00286                     newNode.Children.Add(currentLeaves[minI]);
00287                     newNode.Children.Add(currentLeaves[minJ]);
00288                     currentLeaves[minI].Parent = newNode;
00289                     currentLeaves[minJ].Parent = newNode;
00290 
00291                     double lengthI = 0.5F * minDist + (minSumI - minSumJ) / (2 * (currentLeaves.Count
    - 2));
00292 
00293                     if (!allowNegativeBranches)
00294                     {
00295                         if (lengthI >= 0 && minDist >= lengthI)
00296                         {
00297                             currentLeaves[minI].Length = lengthI;
00298                             currentLeaves[minJ].Length = minDist - lengthI;
00299                         }
00300                         else if (lengthI < 0 && minDist >= lengthI)
00301                         {
00302                             currentLeaves[minI].Length = 0;
00303                             currentLeaves[minJ].Length = minDist - lengthI * 2;
00304                         }
00305                         else if (lengthI >= 0 && minDist < lengthI)
00306                         {
00307                             currentLeaves[minI].Length = lengthI * 2 - minDist;
00308                             currentLeaves[minJ].Length = 0;
00309                         }
00310                         else if (lengthI < 0 && minDist < lengthI)
00311                         {
00312                             currentLeaves[minI].Length = -lengthI;
00313                             currentLeaves[minJ].Length = lengthI - minDist;
00314                         }
00315                     }
00316                     else
00317                     {
00318                         currentLeaves[minI].Length = lengthI;
00319                         currentLeaves[minJ].Length = minDist - lengthI;
00320                     }
00321 
00322                     int correspI = correspondences[minI];
00323                     int correspJ = correspondences[minJ];
00324 
00325                     currentLeaves.RemoveAt(Math.Min(minI, minJ));
00326                     correspondences.RemoveAt(Math.Min(minI, minJ));
00327 
```

```
00328                     currentLeaves[Math.Max(minI, minJ) - 1] = newNode;
00329                     correspondences[Math.Max(minI, minJ) - 1] = correspI;
00330 
00331                     sums[correspI] = 0;
00332 
00333                     for (int i = 0; i < currentLeaves.Count; i++)
00334                     {
00335                         if (correspondences[i] != correspI)
00336                         {
00337                             float distI = distanceMatrix[Math.Max(correspI,
      correspondences[i])][Math.Min(correspI, correspondences[i])];
00338                             float distJ = distanceMatrix[Math.Max(correspJ,
      correspondences[i])][Math.Min(correspJ, correspondences[i])];
00339                             double newDist = (distI + distJ - minDist) * 0.5F;
00340 
00341                             distanceMatrix[Math.Max(correspI, correspondences[i])][Math.Min(correspI,
      correspondences[i])] = (float)newDist;
00342 
00343                             sums[correspondences[i]] -= distI + distJ - newDist;
00344                             sums[correspI] += newDist;
00345                         }
00346                     }
00347 
00348                     if (currentLeaves.Count == 2)
00349                     {
00350                         float branchLength = distanceMatrix[Math.Max(correspondences[0],
      correspondences[1])][Math.Min(correspondences[0], correspondences[1])];
00351 
00352                         if (currentLeaves[1].Children.Count > 0)
00353                         {
00354                             if (!allowNegativeBranches)
00355                             {
00356                                 currentLeaves[0].Length = Math.Abs(branchLength);
00357                             }
00358                             else
00359                             {
00360                                 currentLeaves[0].Length = branchLength;
00361                             }
00362 
00363                             currentLeaves[1].Children.Add(currentLeaves[0]);
00364                             currentLeaves[0].Parent = currentLeaves[1];
00365                             currentLeaves.RemoveAt(0);
00366                             correspondences.RemoveAt(0);
00367                         }
00368                         else
00369                         {
00370                             if (!allowNegativeBranches)
00371                             {
00372                                 currentLeaves[1].Length = Math.Abs(branchLength);
00373                             }
00374                             else
00375                             {
00376                                 currentLeaves[1].Length = branchLength;
00377                             }
00378 
00379                             currentLeaves[0].Children.Add(currentLeaves[1]);
00380                             currentLeaves[1].Parent = currentLeaves[0];
00381                             currentLeaves.RemoveAt(1);
00382                             correspondences.RemoveAt(1);
00383                         }
00384                     }
00385 
00386                     processed += newStep;
00387                 }
00388             }
00389 
00390             progressCallback?.Invoke(1);
00391 
00392             return currentLeaves[0];
00393         }
00394 
00395 /// <summary>
00396 /// Determines whether two splits are compatible.
00397 /// </summary>
00398 /// <param name="leftSide">The elements of the left side of the first split.</param>
00399 /// <param name="rightSide">The elements of the right side of the first split.</param>
00400 /// <param name="otherSplit">The second split.</param>
00401 /// <returns><see langword="true"/> if the splits are compatible, <see langword="false"/>
      otherwise.</returns>
00402         private static bool AreCompatible(ISet<int> leftSide, ISet<int> rightSide, int[][] otherSplit)
00403         {
00404             return !leftSide.Overlaps(otherSplit[0]) || !leftSide.Overlaps(otherSplit[1]) ||
      !rightSide.Overlaps(otherSplit[0]) || !rightSide.Overlaps(otherSplit[1]);
00405         }
00406 
00407 /// <summary>
00408 /// Determines whether a split is compatible with a set of splits.
```

```
00409 /// </summary>
00410 /// <param name="leftSide">The elements of the left side of the first split.</param>
00411 /// <param name="rightSide">The elements of the right side of the first split.</param>
00412 /// <param name="otherSplits">The set of splits.</param>
00413 /// <returns><see langword="true"/> if the split is compatible with all the other splits, <see
      langword="false"/> otherwise.</returns>
00414         internal static bool IsCompatible(ISet<int> leftSide, ISet<int> rightSide,
      IEnumerable<int[][]> otherSplits)
00415         {
00416             foreach (int[][] split in otherSplits)
00417             {
00418                 if (!AreCompatible(leftSide, rightSide, split))
00419                 {
00420                     return false;
00421                 }
00422             }
00423 
00424             return true;
00425         }
00426 
00427 /// <summary>
00428 /// Gets a list of all the splits in the tree.
00429 /// </summary>
00430 /// <param name="tree">The tree from which the splits will be obtained.</param>
00431 /// <param name="sequenceIndices">A dictionary relating each taxon to an integer index.</param>
00432 /// <returns>A list of all the splits in the tree.</returns>
00433         internal static List<int[][]> GetSplits(TreeNode tree, Dictionary<string, int>
      sequenceIndices)
00434         {
00435             List<int[][]> splits = new List<int[][]>();
00436 
00437             foreach (var v in tree.GetSplits())
00438             {
00439                 int[] leftSide = new int[v.side1.Count];
00440                 int[] rightSide = new int[v.side2.Count];
00441 
00442 
00443                 for (int i = 0; i < v.side1.Count; i++)
00444                 {
00445                     if (v.side1[i] != null)
00446                     {
00447                         leftSide[i] = sequenceIndices[v.side1[i].Name];
00448                     }
00449                     else
00450                     {
00451                         leftSide[i] = -1;
00452                     }
00453                 }
00454 
00455                 for (int i = 0; i < v.side2.Count; i++)
00456                 {
00457                     rightSide[i] = sequenceIndices[v.side2[i].Name];
00458                 }
00459 
00460                 splits.Add(new int[][] { leftSide, rightSide });
00461             }
00462 
00463             return splits;
00464         }
00465 
00466 /// <summary>
00467 /// Builds a neighbour-joining tree using data from a distance matrix, applying the specified
      constraint.
00468 /// </summary>
00469 /// <param name="distanceMatrix">The distance matrix containg distances between the taxa.  This can be
      a lower triangular matrix or a full matrix; values above the diagonal will not be used.</param>
00470 /// <param name="sequenceNames">The names of the taxa.  The indices of this list should correspond to
      the rows and columns of the <paramref name="distanceMatrix"/>.</param>
00471 /// <param name="constraint">A tree to constrain the search.  The tree produced by this method will be
      compatible with this tree.  The constraint tree can be multifurcating.</param>
00472 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00473 /// <param name="allowNegativeBranches">If this is <see langword="true"/> negative branches produced
      by the neighbour-joining algorithm are left untouched; otherwise, their (absolute) length is added to
      the
00474 /// sibling branch, and the negative length is set to 0.</param>
00475 /// <param name="progressCallback">A method used to report progress.</param>
00476 /// <returns>The neighbour-joining tree built from the supplied <paramref
      name="distanceMatrix"/>.</returns>
00477         private static unsafe TreeNode BuildTreeWithConstraint(float** distanceMatrix,
      IReadOnlyList<string> sequenceNames, TreeNode constraint, int numCores, bool allowNegativeBranches,
      Action<double> progressCallback)
00478         {
00479             Dictionary<string, int> sequenceIndices = new Dictionary<string,
      int>(sequenceNames.Count);
00480             for (int i = 0; i < sequenceNames.Count; i++)
00481             {
00482                 sequenceIndices[sequenceNames[i]] = i;
```

```
00483             }
00484
00485             constraint = constraint.GetUnrootedTree();
00486
00487             List<int[][]> splits = new List<int[][]>();
00488
00489             foreach (var v in constraint.GetSplits())
00490             {
00491                 if (v.side1.Count > 1 && v.side2.Count > 1)
00492                 {
00493                     int[] leftSide = new int[v.side1.Count];
00494                     int[] rightSide = new int[v.side2.Count];
00495
00496                     for (int i = 0; i < v.side1.Count; i++)
00497                     {
00498                         leftSide[i] = sequenceIndices[v.side1[i].Name];
00499                     }
00500
00501                     for (int i = 0; i < v.side2.Count; i++)
00502                     {
00503                         rightSide[i] = sequenceIndices[v.side2[i].Name];
00504                     }
00505
00506                     splits.Add(new int[][] { leftSide, rightSide });
00507                 }
00508             }
00509
00510             List<TreeNode> currentLeaves = new List<TreeNode>(sequenceNames.Count);
00511             List<int> correspondences = new List<int>(sequenceNames.Count);
00512             List<List<int» underlyingLeaves = new List<List<int»(sequenceNames.Count);
00513
00514             fixed (double* sums = new double[sequenceNames.Count])
00515             {
00516                 IntPtr sumsPtr = (IntPtr)sums;
00517
00518                 for (int i = 0; i < sequenceNames.Count; i++)
00519                 {
00520                     currentLeaves.Add(new TreeNode(null) { Name = sequenceNames[i] });
00521                     correspondences.Add(i);
00522                     underlyingLeaves.Add(new List<int>() { i });
00523
00524                     sums[i] = 0;
00525                     for (int k = 0; k < sequenceNames.Count; k++)
00526                     {
00527                         if (k != i)
00528                         {
00529                             sums[i] += distanceMatrix[Math.Max(i, k)][Math.Min(i, k)];
00530                         }
00531                     }
00532                 }
00533
00534                 double totalToProcess = ((double)sequenceNames.Count * (sequenceNames.Count + 1) * (2
    * sequenceNames.Count + 1)) / 12 + 0.25 * sequenceNames.Count * (sequenceNames.Count + 1) - 4;
00535                 long processed = 0;
00536
00537                 ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = numCores };
00538
00539                 while (currentLeaves.Count > 1)
00540                 {
00541                     long newStep = (long)currentLeaves.Count * (currentLeaves.Count + 1) / 2;
00542                     progressCallback?.Invoke(processed / totalToProcess);
00543
00544                     int minI = -1;
00545                     int minJ = -1;
00546
00547                     double minDist = double.MaxValue;
00548                     double minSumI = double.NaN;
00549                     double minSumJ = double.NaN;
00550
00551                     double[] minDists = new double[currentLeaves.Count];
00552                     int[] minJs = new int[currentLeaves.Count];
00553
00554                     Parallel.For(0, currentLeaves.Count, opt, i =>
00555                     {
00556                         double* mySums = (double*)sumsPtr;
00557
00558                         double myMinDist = double.MaxValue;
00559                         int myMinJ = -1;
00560
00561                         int realI = correspondences[i];
00562                         double sumI = mySums[realI];
00563
00564                         for (int j = 0; j < i; j++)
00565                         {
00566                             int realJ = correspondences[j];
00567
00568                             double dist = distanceMatrix[realI][realJ] * (currentLeaves.Count - 2);
```

```
00569                               double sumJ = mySums[realJ];
00570
00571                               dist -= sumI + sumJ;
00572
00573                               if (dist < myMinDist)
00574                               {
00575                                   HashSet<int> leftSide = new HashSet<int>(underlyingLeaves[i].Count +
      underlyingLeaves[j].Count);
00576                                   for (int k = 0; k < underlyingLeaves[i].Count; k++)
00577                                   {
00578                                       leftSide.Add(underlyingLeaves[i][k]);
00579                                   }
00580                                   for (int k = 0; k < underlyingLeaves[j].Count; k++)
00581                                   {
00582                                       leftSide.Add(underlyingLeaves[j][k]);
00583                                   }
00584
00585                                   HashSet<int> rightSide = new HashSet<int>(sequenceNames.Count);
00586
00587                                   for (int k = 0; k < sequenceNames.Count; k++)
00588                                   {
00589                                       if (!leftSide.Contains(k))
00590                                       {
00591                                           rightSide.Add(k);
00592                                       }
00593                                   }
00594
00595                                   if (IsCompatible(leftSide, rightSide, splits))
00596                                   {
00597                                       myMinDist = dist;
00598                                       myMinJ = j;
00599                                   }
00600                               }
00601                           }
00602
00603                           minDists[i] = myMinDist;
00604                           minJs[i] = myMinJ;
00605                       });
00606
00607                       for (int i = 0; i < currentLeaves.Count; i++)
00608                       {
00609                           if (minDists[i] < minDist)
00610                           {
00611                               minDist = minDists[i];
00612                               minI = i;
00613                               minSumI = sums[correspondences[i]];
00614                               minJ = minJs[i];
00615                               minSumJ = sums[correspondences[minJ]];
00616                           }
00617                       }
00618
00619                       minDist = distanceMatrix[correspondences[minI]][correspondences[minJ]];
00620
00621                       TreeNode newNode = new TreeNode(null);
00622                       newNode.Children.Add(currentLeaves[minI]);
00623                       newNode.Children.Add(currentLeaves[minJ]);
00624                       currentLeaves[minI].Parent = newNode;
00625                       currentLeaves[minJ].Parent = newNode;
00626
00627                       double lengthI = 0.5F * minDist + (minSumI - minSumJ) / (2 * (currentLeaves.Count
      - 2));
00628
00629                       if (!allowNegativeBranches)
00630                       {
00631                           if (lengthI >= 0 && minDist >= lengthI)
00632                           {
00633                               currentLeaves[minI].Length = lengthI;
00634                               currentLeaves[minJ].Length = minDist - lengthI;
00635                           }
00636                           else if (lengthI < 0 && minDist >= lengthI)
00637                           {
00638                               currentLeaves[minI].Length = 0;
00639                               currentLeaves[minJ].Length = minDist - lengthI * 2;
00640                           }
00641                           else if (lengthI >= 0 && minDist < lengthI)
00642                           {
00643                               currentLeaves[minI].Length = lengthI * 2 - minDist;
00644                               currentLeaves[minJ].Length = 0;
00645                           }
00646                           else if (lengthI < 0 && minDist < lengthI)
00647                           {
00648                               currentLeaves[minI].Length = -lengthI;
00649                               currentLeaves[minJ].Length = lengthI - minDist;
00650                           }
00651                       }
00652                       else
00653                       {
```

```
00654                         currentLeaves[minI].Length = lengthI;
00655                         currentLeaves[minJ].Length = minDist - lengthI;
00656                     }
00657 
00658                     int correspI = correspondences[minI];
00659                     int correspJ = correspondences[minJ];
00660 
00661                     underlyingLeaves[Math.Max(minI, minJ)].AddRange(underlyingLeaves[Math.Min(minI,
      minJ)]);
00662                     underlyingLeaves.RemoveAt(Math.Min(minI, minJ));
00663 
00664                     currentLeaves.RemoveAt(Math.Min(minI, minJ));
00665                     correspondences.RemoveAt(Math.Min(minI, minJ));
00666 
00667                     currentLeaves[Math.Max(minI, minJ) - 1] = newNode;
00668                     correspondences[Math.Max(minI, minJ) - 1] = correspI;
00669 
00670                     sums[correspI] = 0;
00671 
00672                     for (int i = 0; i < currentLeaves.Count; i++)
00673                     {
00674                         if (correspondences[i] != correspI)
00675                         {
00676                             float distI = distanceMatrix[Math.Max(correspI,
      correspondences[i])][Math.Min(correspI, correspondences[i])];
00677                             float distJ = distanceMatrix[Math.Max(correspJ,
      correspondences[i])][Math.Min(correspJ, correspondences[i])];
00678                             double newDist = (distI + distJ - minDist) * 0.5F;
00679 
00680                             distanceMatrix[Math.Max(correspI, correspondences[i])][Math.Min(correspI,
      correspondences[i])] = (float)newDist;
00681 
00682                             sums[correspondences[i]] -= distI + distJ - newDist;
00683                             sums[correspI] += newDist;
00684                         }
00685                     }
00686 
00687                     if (currentLeaves.Count == 2)
00688                     {
00689                         float branchLength = distanceMatrix[Math.Max(correspondences[0],
      correspondences[1])][Math.Min(correspondences[0], correspondences[1])];
00690 
00691                         if (currentLeaves[1].Children.Count > 0)
00692                         {
00693                             if (!allowNegativeBranches)
00694                             {
00695                                 currentLeaves[0].Length = Math.Abs(branchLength);
00696                             }
00697                             else
00698                             {
00699                                 currentLeaves[0].Length = branchLength;
00700                             }
00701 
00702                             currentLeaves[1].Children.Add(currentLeaves[0]);
00703                             currentLeaves[0].Parent = currentLeaves[1];
00704                             currentLeaves.RemoveAt(0);
00705                             correspondences.RemoveAt(0);
00706                         }
00707                         else
00708                         {
00709                             if (!allowNegativeBranches)
00710                             {
00711                                 currentLeaves[1].Length = Math.Abs(branchLength);
00712                             }
00713                             else
00714                             {
00715                                 currentLeaves[1].Length = branchLength;
00716                             }
00717 
00718                             currentLeaves[0].Children.Add(currentLeaves[1]);
00719                             currentLeaves[1].Parent = currentLeaves[0];
00720                             currentLeaves.RemoveAt(1);
00721                             correspondences.RemoveAt(1);
00722                         }
00723                     }
00724 
00725                     processed += newStep;
00726                 }
00727             }
00728 
00729             progressCallback?.Invoke(1);
00730 
00731             return currentLeaves[0];
00732         }
00733     }
00734 }
```

8.23 RandomTree.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Linq;
00004 using System.Text;
00005 using MathNet.Numerics.Distributions;
00006
00007 namespace PhyloTree.TreeBuilding
00008 {
00009 /// <summary>
00010 /// Contains methods to generate random trees.
00011 /// </summary>
00012     public static class RandomTree
00013     {
00014 /// <summary>
00015 /// Random number generator used for sampling.
00016 /// </summary>
00017         public static Random RandomNumberGenerator = new ThreadSafeRandom();
00018
00019 /// <summary>
00020 /// Samples a random unlabelled topology according to the specified model.
00021 /// </summary>
00022 /// <param name="leafCount">The number of terminal nodes in the topology.</param>
00023 /// <param name="model">The model to use for growing the tree.</param>
00024 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00025 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled topology (branch lengths
      will all be set to <see cref="double.NaN"/>).</returns>
00026 /// <exception cref="ArgumentException">Thrown if <paramref name="model"/> is neither <see
      cref="TreeNode.NullHypothesis.PDA"/> nor <see cref="TreeNode.NullHypothesis.YHK"/>.</exception>
00027         public static TreeNode UnlabelledTopology(int leafCount, TreeNode.NullHypothesis model =
      TreeNode.NullHypothesis.PDA, bool rooted = false)
00028         {
00029             if (model == TreeNode.NullHypothesis.YHK)
00030             {
00031                 TreeNode initialTree = new TreeNode(null);
00032                 initialTree.Children.Add(new TreeNode(initialTree));
00033                 initialTree.Children.Add(new TreeNode(initialTree));
00034
00035                 if (!rooted)
00036                 {
00037                     initialTree.Children.Add(new TreeNode(initialTree));
00038                 }
00039
00040                 List<TreeNode> leaves = new List<TreeNode>();
00041
00042                 leaves.AddRange(initialTree.Children);
00043
00044                 while (leaves.Count < leafCount)
00045                 {
00046                     int index = RandomNumberGenerator.Next(0, leaves.Count);
00047
00048                     TreeNode selectedLeaf = leaves[index];
00049
00050                     leaves.RemoveAt(index);
00051
00052                     selectedLeaf.Children.Add(new TreeNode(selectedLeaf));
00053                     selectedLeaf.Children.Add(new TreeNode(selectedLeaf));
00054                     leaves.AddRange(selectedLeaf.Children);
00055                 }
00056
00057                 return initialTree;
00058             }
00059             else if (model == TreeNode.NullHypothesis.PDA)
00060             {
00061                 TreeNode initialTree = new TreeNode(null);
00062                 initialTree.Children.Add(new TreeNode(initialTree));
00063                 initialTree.Children.Add(new TreeNode(initialTree));
00064
00065                 if (!rooted)
00066                 {
00067                     initialTree.Children.Add(new TreeNode(initialTree));
00068                 }
00069
00070                 List<TreeNode> leaves = new List<TreeNode>(initialTree.Children);
00071                 List<TreeNode> nodes;
00072
00073                 if (rooted)
00074                 {
00075                     nodes = initialTree.GetChildrenRecursive();
00076                 }
00077                 else
00078                 {
00079                     nodes = new List<TreeNode>(initialTree.Children);
00080                 }
00081
```

```
00082                 while (leaves.Count < leafCount)
00083                 {
00084                     int index = RandomNumberGenerator.Next(0, nodes.Count);
00085 
00086                     TreeNode selectedNode = nodes[index];
00087 
00088                     if (selectedNode.Children.Count == 0)
00089                     {
00090                         leaves.Remove(selectedNode);
00091 
00092                         selectedNode.Children.Add(new TreeNode(selectedNode));
00093                         selectedNode.Children.Add(new TreeNode(selectedNode));
00094                         leaves.AddRange(selectedNode.Children);
00095                         nodes.AddRange(selectedNode.Children);
00096                     }
00097                     else
00098                     {
00099                         if (selectedNode.Parent != null)
00100                         {
00101                             TreeNode newNode = new TreeNode(selectedNode.Parent);
00102                             selectedNode.Parent.Children.Add(newNode);
00103 
00104                             TreeNode newLeaf = new TreeNode(newNode);
00105                             newNode.Children.Add(newLeaf);
00106 
00107                             selectedNode.Parent.Children.Remove(selectedNode);
00108                             selectedNode.Parent = newNode;
00109                             newNode.Children.Add(selectedNode);
00110 
00111                             nodes.Add(newNode);
00112                             nodes.Add(newLeaf);
00113                             leaves.Add(newLeaf);
00114                         }
00115                         else
00116                         {
00117                             TreeNode newNode = new TreeNode(null);
00118                             TreeNode newLeaf = new TreeNode(newNode);
00119                             newNode.Children.Add(newLeaf);
00120 
00121                             selectedNode.Parent = newNode;
00122                             newNode.Children.Add(selectedNode);
00123 
00124                             nodes.Add(newNode);
00125                             nodes.Add(newLeaf);
00126                             leaves.Add(newLeaf);
00127 
00128                             initialTree = newNode;
00129                         }
00130 
00131                     }
00132                 }
00133 
00134                 return initialTree;
00135             }
00136             else
00137             {
00138                 throw new ArgumentException("Invalid tree model");
00139             }
00140         }
00141 
00142 /// <summary>
00143 /// Samples a random unlabelled tree according to the specified model, using branch lengths drawn from
      the supplied distribution.
00144 /// </summary>
00145 /// <param name="leafCount">The number of terminal nodes in the topology.</param>
00146 /// <param name="branchLengthDistribution">The continuous univariate distribution from which the
      branch lengths will be drawn.</param>
00147 /// <param name="model">The model to use for growing the tree.</param>
00148 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00149 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled tree, with branch
      lengths.</returns>
00150         public static TreeNode UnlabelledTree(int leafCount, IContinuousDistribution
      branchLengthDistribution, TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA, bool rooted =
      false)
00151         {
00152             TreeNode topology = UnlabelledTopology(leafCount, model, rooted);
00153 
00154             List<TreeNode> nodes = topology.GetChildrenRecursive();
00155 
00156             double[] branchLengths = new double[nodes.Count - 1];
00157 
00158             branchLengthDistribution.Samples(branchLengths);
00159 
00160             for (int i = 1; i < nodes.Count; i++)
00161             {
00162                 nodes[i].Length = branchLengths[i - 1];
```

```
00163             }
00164
00165             return topology;
00166         }
00167
00168 /// <summary>
00169 /// Samples a random unlabelled tree according to the specified model, using branch lengths drawn from
      a Uniform(0, 1) distribution.
00170 /// </summary>
00171 /// <param name="leafCount">The number of terminal nodes in the topology.</param>
00172 /// <param name="model">The model to use for growing the tree.</param>
00173 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00174 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled tree, with branch
      lengths.</returns>
00175         public static TreeNode UnlabelledTree(int leafCount, TreeNode.NullHypothesis model =
      TreeNode.NullHypothesis.PDA, bool rooted = false)
00176         {
00177             return UnlabelledTree(leafCount, new ContinuousUniform(0, 1, RandomNumberGenerator),
      model, rooted);
00178         }
00179
00180 /// <summary>
00181 /// Adds leaf names to the supplied tree, in a random order.
00182 /// </summary>
00183 /// <param name="tree">The tree on which the leaf names will be added.</param>
00184 /// <param name="leafNames">The leaf names to add.</param>
00185         private static void AddLeafNames(TreeNode tree, IReadOnlyList<string> leafNames)
00186         {
00187             List<TreeNode> leaves = tree.GetLeaves();
00188             List<string> leafNamesList = leafNames.ToList();
00189
00190             for (int i = 0; i < leaves.Count; i++)
00191             {
00192                 int index = RandomNumberGenerator.Next(0, leafNamesList.Count);
00193                 leaves[i].Name = leafNamesList[index];
00194                 leafNamesList.RemoveAt(index);
00195             }
00196         }
00197
00198 /// <summary>
00199 /// Takes a random element from a list and returns it, removing it from the list.
00200 /// </summary>
00201 /// <param name="elements">The list from which the element is sampled.</param>
00202 /// <returns>A random element selected from the list.</returns>
00203         internal static T Sample<T>(this IList<T> elements)
00204         {
00205             int index = RandomNumberGenerator.Next(0, elements.Count);
00206             T name = elements[index];
00207             elements.RemoveAt(index);
00208             return name;
00209         }
00210
00211 /// <summary>
00212 /// Randomly resolve all the polytomies in a tree.
00213 /// </summary>
00214 /// <param name="tree">The tree containing the polytomies to be resolved.</param>
00215 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree is supposed to be rooted or
      not.  If this is <see langword="true"/>, a trichotomy at the root node will be resolved.</param>
00216 /// <returns>The tree, where all the polytomies have been randomly resolved.  This is performed
      in-place, but the return value of this method should be used in case the root node has
      changed.</returns>
00217         public static TreeNode ResolvePolytomies(TreeNode tree, bool rooted = false)
00218         {
00219             bool found = true;
00220
00221             while (found)
00222             {
00223                 List<TreeNode> nodes = tree.GetChildrenRecursive();
00224                 found = false;
00225
00226                 for (int i = 0; i < nodes.Count; i++)
00227                 {
00228                     if (nodes[i].Children.Count > 2 && !(i == 0 && !rooted && nodes[i].Children.Count
      == 3))
00229                     {
00230                         TreeNode nodeToRemove = nodes[i].Children[RandomNumberGenerator.Next(0,
      nodes[i].Children.Count)];
00231                         nodes[i].Children.Remove(nodeToRemove);
00232
00233                         int newSiblingIndex = RandomNumberGenerator.Next(0, nodes[i].Children.Count +
      1);
00234
00235                         if (newSiblingIndex < nodes[i].Children.Count)
00236                         {
00237                             TreeNode newNode = new TreeNode(nodes[i]);
00238                             nodes[i].Children[newSiblingIndex].Parent = newNode;
```

```
00239                           newNode.Children.Add(nodes[i].Children[newSiblingIndex]);
00240                           nodeToRemove.Parent = newNode;
00241                           newNode.Children.Add(nodeToRemove);
00242                           nodes[i].Children[newSiblingIndex] = newNode;
00243                       }
00244                       else
00245                       {
00246                           if (nodes[i].Parent != null)
00247                           {
00248                               TreeNode newNode = new TreeNode(nodes[i].Parent);
00249                               newNode.Parent.Children.Add(newNode);
00250 
00251                               nodes[i].Parent.Children.Remove(nodes[i]);
00252                               nodes[i].Parent = newNode;
00253                               newNode.Children.Add(nodes[i]);
00254 
00255 
00256                               nodeToRemove.Parent = newNode;
00257                               newNode.Children.Add(nodeToRemove);
00258                           }
00259                           else
00260                           {
00261                               TreeNode newNode = new TreeNode(null);
00262 
00263                               nodes[i].Parent = newNode;
00264                               newNode.Children.Add(nodes[i]);
00265 
00266                               nodeToRemove.Parent = newNode;
00267                               newNode.Children.Add(nodeToRemove);
00268 
00269                               tree = newNode;
00270                           }
00271                       }
00272 
00273                       found = true;
00274                       break;
00275                   }
00276               }
00277           }
00278 
00279           return tree;
00280       }
00281 
00282 /// <summary>
00283 /// Samples a random labelled topology according to the specified model.
00284 /// </summary>
00285 /// <param name="leafNames">The labels for the terminal nodes of the topology.</param>
00286 /// <param name="model">The model to use for growing the tree.</param>
00287 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00288 /// <param name="constraint">A tree to constrain the sampling.  The topology produced by this method
      will be compatible with this tree.  The constraint tree can be multifurcating.
00289 /// Please note that, as the constraint is applied at every step while growing the topology, using a
      constraint with <see cref="TreeNode.NullHypothesis.YHK"/> will bias the sampled topology
      distribution.</param>
00290 /// <returns>A <see cref="TreeNode"/> object containing a random labelled topology (branch lengths
      will all be set to <see cref="double.NaN"/>).</returns>
00291 /// <exception cref="ArgumentException">Thrown if <paramref name="model"/> is neither <see
      cref="TreeNode.NullHypothesis.PDA"/> nor <see cref="TreeNode.NullHypothesis.YHK"/>.</exception>
00292       public static TreeNode LabelledTopology(IReadOnlyList<string> leafNames,
      TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA, bool rooted = false, TreeNode constraint
      = null)
00293       {
00294           if (constraint == null)
00295           {
00296               TreeNode topology = UnlabelledTopology(leafNames.Count, model, rooted);
00297 
00298               AddLeafNames(topology, leafNames);
00299 
00300               return topology;
00301           }
00302           else
00303           {
00304               constraint = constraint.Clone();
00305 
00306               List<string> availableNames = leafNames.ToList();
00307 
00308               Dictionary<string, int> sequenceIndices = new Dictionary<string, int>();
00309               List<TreeNode> toBePruned = new List<TreeNode>();
00310 
00311               foreach (TreeNode leaf in constraint.GetLeaves())
00312               {
00313                   if (!string.IsNullOrEmpty(leaf.Name))
00314                   {
00315                       int index = availableNames.IndexOf(leaf.Name);
00316                       if (index >= 0)
00317                       {
```

```
00318                           sequenceIndices[leaf.Name] = index;
00319                       }
00320                       else
00321                       {
00322                           toBePruned.Add(leaf);
00323                       }
00324                   }
00325                   else
00326                   {
00327                       toBePruned.Add(leaf);
00328                   }
00329               }
00330
00331               for (int i = 0; i < toBePruned.Count; i++)
00332               {
00333                   constraint = constraint.Prune(toBePruned[i], false);
00334               }
00335
00336               if (constraint == null || constraint.GetLeaves().Count == 1)
00337               {
00338                   return LabelledTopology(leafNames, model, rooted, null);
00339               }
00340
00341               if (model == TreeNode.NullHypothesis.YHK)
00342               {
00343                   List<int[][]> splits = NeighborJoining.GetSplits(constraint, sequenceIndices);
00344
00345                   List<TreeNode> leaves = new List<TreeNode>(leafNames.Count);
00346                   List<HashSet<int» underlyingLeaves = new List<HashSet<int»();
00347                   HashSet<int> allLeaves = new HashSet<int>();
00348
00349                   for (int i = 0; i < leafNames.Count; i++)
00350                   {
00351                       leaves.Add(new TreeNode(null) { Name = leafNames[i] });
00352
00353                       if (sequenceIndices.TryGetValue(leafNames[i], out int index))
00354                       {
00355                           underlyingLeaves.Add(new HashSet<int>() { index });
00356                           allLeaves.Add(index);
00357                       }
00358                       else
00359                       {
00360                           underlyingLeaves.Add(new HashSet<int>());
00361                       }
00362                   }
00363
00364                   if (rooted)
00365                   {
00366                       allLeaves.Add(-1);
00367                   }
00368
00369                   int target = rooted ?  1 :  3;
00370
00371                   while (leaves.Count > target)
00372                   {
00373                       int leaf1Index, leaf2Index;
00374                       HashSet<int> potentialSplitLeft, potentialSplitRight;
00375
00376                           List<(int, int)> availablePairs = (from el in Enumerable.Range(0,
    leaves.Count) select (from el2 in Enumerable.Range(0, el) select (el, el2))).Aggregate(new List<(int,
    int)>(), (a, b) => { a.AddRange(b); return a; });
00377
00378                           do
00379                           {
00380                               (leaf1Index, leaf2Index) = Sample(availablePairs);
00381
00382                               potentialSplitRight = new HashSet<int>(underlyingLeaves[leaf1Index]);
00383                               potentialSplitRight.UnionWith(underlyingLeaves[leaf2Index]);
00384
00385                               potentialSplitLeft = new HashSet<int>(allLeaves);
00386                               potentialSplitLeft.ExceptWith(potentialSplitRight);
00387
00388                           } while (!NeighborJoining.IsCompatible(potentialSplitLeft,
    potentialSplitRight, splits));
00389
00390                           TreeNode leaf1 = leaves[leaf1Index];
00391                           TreeNode leaf2 = leaves[leaf2Index];
00392
00393                           TreeNode newNode = new TreeNode(null);
00394                           newNode.Children.Add(leaf1);
00395                           newNode.Children.Add(leaf2);
00396                           leaf1.Parent = newNode;
00397                           leaf2.Parent = newNode;
00398
00399                           underlyingLeaves[Math.Min(leaf1Index,
    leaf2Index)].UnionWith(underlyingLeaves[Math.Max(leaf1Index, leaf2Index)]);
00400                           underlyingLeaves.RemoveAt(Math.Max(leaf1Index, leaf2Index));
```

```
00401
00402                       leaves.RemoveAt(Math.Max(leaf1Index, leaf2Index));
00403                       leaves[Math.Min(leaf1Index, leaf2Index)] = newNode;
00404                   }
00405
00406                   if (!rooted)
00407                   {
00408                       TreeNode root = new TreeNode(null);
00409
00410                       leaves[0].Parent = root;
00411                       leaves[1].Parent = root;
00412                       leaves[2].Parent = root;
00413
00414                       root.Children.Add(leaves[0]);
00415                       root.Children.Add(leaves[1]);
00416                       root.Children.Add(leaves[2]);
00417
00418                       return root;
00419                   }
00420                   else
00421                   {
00422                       return leaves[0];
00423                   }
00424               }
00425               else if (model == TreeNode.NullHypothesis.PDA)
00426               {
00427                   TreeNode initialTree = constraint.Clone();
00428
00429                   initialTree = ResolvePolytomies(initialTree, rooted);
00430
00431                   List<TreeNode> leaves = initialTree.GetLeaves();
00432
00433                   for (int i = 0; i < leaves.Count; i++)
00434                   {
00435                       availableNames.Remove(leaves[i].Name);
00436                   }
00437
00438                   List<TreeNode> nodes;
00439
00440                   if (rooted)
00441                   {
00442                       if (!initialTree.IsRooted())
00443                       {
00444                           initialTree = initialTree.GetRootedTree(initialTree.Children[0]);
00445                       }
00446
00447                       nodes = initialTree.GetChildrenRecursive();
00448                   }
00449                   else
00450                   {
00451                       if (initialTree.IsRooted())
00452                       {
00453                           initialTree = initialTree.GetUnrootedTree();
00454                       }
00455
00456                       nodes = initialTree.GetChildrenRecursive();
00457                       nodes.RemoveAt(0);
00458                   }
00459
00460
00461                   while (availableNames.Count > 0)
00462                   {
00463                       string name = Sample(availableNames);
00464
00465                       int index = RandomNumberGenerator.Next(0, nodes.Count);
00466
00467                       TreeNode selectedNode = nodes[index];
00468
00469                       if (selectedNode.Children.Count == 0)
00470                       {
00471                           leaves.Remove(selectedNode);
00472
00473                           selectedNode.Children.Add(new TreeNode(selectedNode) { Name =
    selectedNode.Name });
00474                           selectedNode.Children.Add(new TreeNode(selectedNode) { Name = name });
00475                           selectedNode.Name = "";
00476                           leaves.AddRange(selectedNode.Children);
00477                           nodes.AddRange(selectedNode.Children);
00478                       }
00479                       else
00480                       {
00481                           if (selectedNode.Parent != null)
00482                           {
00483                               TreeNode newNode = new TreeNode(selectedNode.Parent);
00484                               selectedNode.Parent.Children.Add(newNode);
00485
00486                               TreeNode newLeaf = new TreeNode(newNode) { Name = name };
```

```
00487                                 newNode.Children.Add(newLeaf);
00488
00489                                 selectedNode.Parent.Children.Remove(selectedNode);
00490                                 selectedNode.Parent = newNode;
00491                                 newNode.Children.Add(selectedNode);
00492
00493                                 nodes.Add(newNode);
00494                                 nodes.Add(newLeaf);
00495                                 leaves.Add(newLeaf);
00496                             }
00497                             else
00498                             {
00499                                 TreeNode newNode = new TreeNode(null);
00500                                 TreeNode newLeaf = new TreeNode(newNode) { Name = name };
00501                                 newNode.Children.Add(newLeaf);
00502
00503                                 selectedNode.Parent = newNode;
00504                                 newNode.Children.Add(selectedNode);
00505
00506                                 nodes.Add(newNode);
00507                                 nodes.Add(newLeaf);
00508                                 leaves.Add(newLeaf);
00509
00510                                 initialTree = newNode;
00511                             }
00512
00513                         }
00514                     }
00515
00516                     return initialTree;
00517                 }
00518                 else
00519                 {
00520                     throw new ArgumentException("Invalid tree model");
00521                 }
00522             }
00523         }
00524
00525 /// <summary>
00526 /// Samples a random labelled topology according to the specified model.
00527 /// </summary>
00528 /// <param name="leafCount">The number of terminal nodes in the topology.  Their names will be in the
      form <c>t1, t2, ..., tN</c>, where <c>N</c> is <paramref name="leafCount"/>.</param>
00529 /// <param name="model">The model to use for growing the tree.</param>
00530 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00531 /// <param name="constraint">A tree to constrain the sampling.  The topology produced by this method
      will be compatible with this tree.  The constraint tree can be multifurcating.
00532 /// Please note that, as the constraint is applied at every step while growing the topology, using a
      constraint with <see cref="TreeNode.NullHypothesis.YHK"/> will bias the sampled topology
      distribution.</param>
00533 /// <returns>A <see cref="TreeNode"/> object containing a random labelled topology (branch lengths
      will all be set to <see cref="double.NaN"/>).</returns>
00534 /// <exception cref="ArgumentException">Thrown if <paramref name="model"/> is neither <see
      cref="TreeNode.NullHypothesis.PDA"/> nor <see cref="TreeNode.NullHypothesis.YHK"/>.</exception>
00535         public static TreeNode LabelledTopology(int leafCount, TreeNode.NullHypothesis model =
      TreeNode.NullHypothesis.PDA, bool rooted = false, TreeNode constraint = null)
00536         {
00537             string[] leafNames = new string[leafCount];
00538
00539             for (int i = 0; i < leafCount; i++)
00540             {
00541                 leafNames[i] = "t" + (i + 1).ToString();
00542             }
00543
00544             return LabelledTopology(leafNames, model, rooted, constraint);
00545         }
00546
00547 /// <summary>
00548 /// Samples a random labelled tree according to the specified model, using branch lengths drawn from
      the supplied distribution.
00549 /// </summary>
00550 /// <param name="leafNames">The labels for the terminal nodes of the topology.</param>
00551 /// <param name="branchLengthDistribution">The continuous univariate distribution from which the
      branch lengths will be drawn.</param>
00552 /// <param name="model">The model to use for growing the tree.</param>
00553 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00554 /// <param name="constraint">A tree to constrain the sampling.  The topology produced by this method
      will be compatible with this tree.  The constraint tree can be multifurcating.
00555 /// Please note that, as the constraint is applied at every step while growing the topology, using a
      constraint with <see cref="TreeNode.NullHypothesis.YHK"/> will bias the sampled topology
      distribution.</param>
00556 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled tree, with branch
      lengths.</returns>
00557         public static TreeNode LabelledTree(IReadOnlyList<string> leafNames, IContinuousDistribution
      branchLengthDistribution, TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA, bool rooted =
```

```
      false, TreeNode constraint = null)
00558         {
00559             TreeNode topology = LabelledTopology(leafNames, model, rooted, constraint);
00560 
00561             List<TreeNode> nodes = topology.GetChildrenRecursive();
00562 
00563             double[] branchLengths = new double[nodes.Count - 1];
00564 
00565             branchLengthDistribution.Samples(branchLengths);
00566 
00567             for (int i = 1; i < nodes.Count; i++)
00568             {
00569                 nodes[i].Length = branchLengths[i - 1];
00570             }
00571 
00572             return topology;
00573         }
00574 
00575 /// <summary>
00576 /// Samples a random labelled tree according to the specified model, using branch lengths drawn from
      the supplied distribution.
00577 /// </summary>
00578 /// <param name="leafCount">The number of terminal nodes in the topology.  Their names will be in the
      form <c>t1, t2, ..., tN</c>, where <c>N</c> is <paramref name="leafCount"/>.</param>
00579 /// <param name="branchLengthDistribution">The continuous univariate distribution from which the
      branch lengths will be drawn.</param>
00580 /// <param name="model">The model to use for growing the tree.</param>
00581 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00582 /// <param name="constraint">A tree to constrain the sampling.  The topology produced by this method
      will be compatible with this tree.  The constraint tree can be multifurcating.
00583 /// Please note that, as the constraint is applied at every step while growing the topology, using a
      constraint with <see cref="TreeNode.NullHypothesis.YHK"/> will bias the sampled topology
      distribution.</param>
00584 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled tree, with branch
      lengths.</returns>
00585         public static TreeNode LabelledTree(int leafCount, IContinuousDistribution
      branchLengthDistribution, TreeNode.NullHypothesis model = TreeNode.NullHypothesis.PDA, bool rooted =
      false, TreeNode constraint = null)
00586         {
00587             string[] leafNames = new string[leafCount];
00588 
00589             for (int i = 0; i < leafCount; i++)
00590             {
00591                 leafNames[i] = "t" + (i + 1).ToString();
00592             }
00593 
00594             return LabelledTree(leafNames, branchLengthDistribution, model, rooted, constraint);
00595         }
00596 
00597 /// <summary>
00598 /// Samples a random labelled tree according to the specified model, using branch lengths drawn from a
      Uniform(0, 1) distribution.
00599 /// </summary>
00600 /// <param name="leafNames">The labels for the terminal nodes of the topology.</param>
00601 /// <param name="model">The model to use for growing the tree.</param>
00602 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00603 /// <param name="constraint">A tree to constrain the sampling.  The topology produced by this method
      will be compatible with this tree.  The constraint tree can be multifurcating.
00604 /// Please note that, as the constraint is applied at every step while growing the topology, using a
      constraint with <see cref="TreeNode.NullHypothesis.YHK"/> will bias the sampled topology
      distribution.</param>
00605 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled tree, with branch
      lengths.</returns>
00606         public static TreeNode LabelledTree(IReadOnlyList<string> leafNames, TreeNode.NullHypothesis
      model = TreeNode.NullHypothesis.PDA, bool rooted = false, TreeNode constraint = null)
00607         {
00608             return LabelledTree(leafNames, new ContinuousUniform(0, 1, RandomNumberGenerator), model,
      rooted, constraint);
00609         }
00610 
00611 /// <summary>
00612 /// Samples a random labelled tree according to the specified model, using branch lengths drawn from a
      Uniform(0, 1) distribution.
00613 /// </summary>
00614 /// <param name="leafCount">The number of terminal nodes in the topology.  Their names will be in the
      form <c>t1, t2, ..., tN</c>, where <c>N</c> is <paramref name="leafCount"/>.</param>
00615 /// <param name="model">The model to use for growing the tree.</param>
00616 /// <param name="rooted">A <see cref="bool"/> indicating whether the tree should be rooted or
      not.</param>
00617 /// <param name="constraint">A tree to constrain the sampling.  The topology produced by this method
      will be compatible with this tree.  The constraint tree can be multifurcating.
00618 /// Please note that, as the constraint is applied at every step while growing the topology, using a
      constraint with <see cref="TreeNode.NullHypothesis.YHK"/> will bias the sampled topology
      distribution.</param>
00619 /// <returns>A <see cref="TreeNode"/> object containing a random unlabelled tree, with branch
```

```
      lengths.</returns>
00620         public static TreeNode LabelledTree(int leafCount, TreeNode.NullHypothesis model =
      TreeNode.NullHypothesis.PDA, bool rooted = false, TreeNode constraint = null)
00621         {
00622             return LabelledTree(leafCount, new ContinuousUniform(0, 1, RandomNumberGenerator), model,
      rooted, constraint);
00623         }
00624     }
00625 }
```

8.24 ThreadSafeRandom.cs

```
00001 using System;
00002 using System.Security.Cryptography;
00003
00004 namespace PhyloTree.TreeBuilding
00005 {
00006 /// <summary>
00007 /// Represents a thread-safe random number generator.
00008 /// </summary>
00009 /// <remarks>Adapted from
      https://stackoverflow.com/questions/3049467/is-c-sharp-random-number-generator-thread-safe</remarks>
00010     public class ThreadSafeRandom :  Random
00011     {
00012         private static Random _globalRandom;
00013         private static object _globalLock = new object();
00014         [ThreadStatic] private static Random _local;
00015
00016         private bool _useGlobalRandom;
00017
00018 /// <summary>
00019 /// Initialise a new thread-safe random number generator with the specified seed.
00020 /// </summary>
00021 /// <param name="seed">A number used to generate a starting number for the pseudo-random
      sequence.</param>
00022         public ThreadSafeRandom(int seed)
00023         {
00024             lock (_globalLock)
00025             {
00026                 _globalRandom = new Random(seed);
00027                 _useGlobalRandom = true;
00028             }
00029         }
00030
00031 /// <summary>
00032 /// Initialise a new thread-safe random number generator.
00033 /// </summary>
00034         public ThreadSafeRandom()
00035         {
00036             _useGlobalRandom = false;
00037         }
00038
00039         private void InitialiseLocal()
00040         {
00041             if (_local == null)
00042             {
00043                 if (!_useGlobalRandom)
00044                 {
00045                     byte[] buffer = new byte[4];
00046                     RandomNumberGenerator.Create().GetBytes(buffer);
00047                     _local = new Random(BitConverter.ToInt32(buffer, 0));
00048                 }
00049                 else
00050                 {
00051                     lock (_globalLock)
00052                     {
00053                         _local = new Random(_globalRandom.Next());
00054                     }
00055                 }
00056             }
00057         }
00058
00059 /// <inheritdoc/>
00060         public override int Next()
00061         {
00062             InitialiseLocal();
00063             return _local.Next();
00064         }
00065
00066 /// <inheritdoc/>
00067         public override int Next(int maxValue)
00068         {
00069             InitialiseLocal();
```

```
00070             return _local.Next(maxValue);
00071         }
00072 
00073 /// <inheritdoc/>
00074         public override int Next(int minValue, int maxValue)
00075         {
00076             InitialiseLocal();
00077             return _local.Next(minValue, maxValue);
00078         }
00079 
00080 /// <inheritdoc/>
00081         public override double NextDouble()
00082         {
00083             InitialiseLocal();
00084             return _local.NextDouble();
00085         }
00086 
00087 /// <inheritdoc/>
00088         public override void NextBytes(byte[] buffer)
00089         {
00090             InitialiseLocal();
00091             _local.NextBytes(buffer);
00092         }
00093     }
00094 }
```

8.25 UPGMA.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Linq;
00004 using System.Runtime.InteropServices;
00005 using System.Threading;
00006 using System.Threading.Tasks;
00007 
00008 namespace PhyloTree.TreeBuilding
00009 {
00010 /// <summary>
00011 /// Contains methods to compute UPGMA trees.
00012 /// </summary>
00013     public static class UPGMA
00014     {
00015 /// <summary>
00016 /// Builds a UPGMA tree using data from a sequence alignment.  This method first computes a distance
      matrix from the sequence alignment, and then uses the distance matrix to compute the tree.
00017 /// </summary>
00018 /// <param name="alignment">The sequence alignment.</param>
00019 /// <param name="evolutionModel">The evolutionary model to use when computing the distance
      matrix.</param>
00020 /// <param name="bootstrapReplicates">The number of bootstrap replicates to perform.</param>
00021 /// <param name="alignmentType">The type of sequence alignment (DNA, protein, or autodetect).</param>
00022 /// <param name="constraint">An optional tree to constrain the search.  The tree produced by this
      method will be compatible with this tree.  The constraint tree can be multifurcating.</param>
00023 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00024 /// <param name="progressCallback">A method used to report progress.</param>
00025 /// <returns>The UPGMA tree built from the supplied <paramref name="alignment"/>.</returns>
00026         public static TreeNode BuildTree(Dictionary<string, string> alignment, EvolutionModel
      evolutionModel = EvolutionModel.Kimura, int bootstrapReplicates = 0, AlignmentType alignmentType =
      AlignmentType.Autodetect, TreeNode constraint = null, int numCores = -1, Action<double>
      progressCallback = null)
00027         {
00028             List<string> sequenceNames = alignment.Keys.ToList();
00029             List<string> sequences = alignment.Values.ToList();
00030 
00031             if (bootstrapReplicates == 0)
00032             {
00033                 Action<double> distMatProgress = null;
00034                 Action<double> treeProgress = null;
00035 
00036                 if (progressCallback != null)
00037                 {
00038                     distMatProgress = x => progressCallback(x * 0.5);
00039                     treeProgress = x => progressCallback(0.5 + x * 0.5);
00040                 }
00041 
00042                 float[][] distMat = DistanceMatrix.BuildFromAlignment(sequences, alignmentType,
      evolutionModel, numCores, distMatProgress);
00043 
00044                 return BuildTree(distMat, sequenceNames, constraint, copyMatrix:  false, numCores:
      numCores, progressCallback:  treeProgress);
00045             }
00046             else
00047             {
```

```
00048                 float[][] distMat = DistanceMatrix.BuildFromAlignment(sequences, alignmentType,
      evolutionModel, numCores);
00049
00050                 TreeNode initialTree = BuildTree(distMat, sequenceNames, constraint, copyMatrix:
      false, numCores:  numCores);
00051
00052                 progressCallback?.Invoke(1.0 / (bootstrapReplicates + 1));
00053
00054                 Dictionary<string, int> sequenceIndices = new Dictionary<string,
      int>(sequenceNames.Count);
00055                 for (int i = 0; i < sequenceNames.Count; i++)
00056                 {
00057                     sequenceIndices[sequenceNames[i]] = i;
00058                 }
00059
00060                 List<int[][]> splits = NeighborJoining.GetSplits(initialTree, sequenceIndices);
00061                 List<TreeNode> nodes = initialTree.GetChildrenRecursive();
00062                 int[] supports = new int[splits.Count];
00063                 object supportLock = new object();
00064
00065                 (HashSet<int>, HashSet<int>)[] setSplits = new (HashSet<int>,
      HashSet<int>)[splits.Count];
00066
00067                 for (int i = 0; i < splits.Count; i++)
00068                 {
00069                     setSplits[i] = (new HashSet<int>(splits[i][0]), new HashSet<int>(splits[i][1]));
00070                 }
00071
00072                 int completed = 0;
00073                 object progressLock = new object();
00074
00075                 ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = numCores };
00076
00077                 Parallel.For(0, bootstrapReplicates, opt, i =>
00078                 {
00079                     float[][] currDistMat = DistanceMatrix.BootstrapReplicateFromAlignment(sequences,
      alignmentType, evolutionModel);
00080
00081                     TreeNode tree = BuildTree(currDistMat, sequenceNames, constraint, copyMatrix:
      false, numCores:  1);
00082
00083                     currDistMat = null;
00084
00085                     List<int[][]> newSplits = NeighborJoining.GetSplits(tree, sequenceIndices);
00086
00087                     List<int> supported = new List<int>(nodes.Count);
00088
00089                     Parallel.For(0, splits.Count, opt, j =>
00090                     {
00091                         if (nodes[j].Children.Count == 0 ||
      NeighborJoining.IsCompatible(setSplits[j].Item1, setSplits[j].Item2, newSplits))
00092                         {
00093                             Interlocked.Increment(ref supports[j]);
00094                         }
00095                     });
00096
00097                     if (progressCallback != null)
00098                     {
00099                         lock (progressLock)
00100                         {
00101                             completed++;
00102                             progressCallback?.Invoke((double)completed / (bootstrapReplicates + 1));
00103                         }
00104                     }
00105                 });
00106
00107                 for (int i = 0; i < nodes.Count; i++)
00108                 {
00109                     nodes[i].Support = (double)supports[i] / bootstrapReplicates;
00110                 }
00111
00112                 return initialTree;
00113             }
00114         }
00115
00116 /// <summary>
00117 /// Builds a UPGMA tree using data from a distance matrix.
00118 /// </summary>
00119 /// <param name="distanceMatrix">The distance matrix containg distances between the taxa.  This can be
      a lower triangular matrix or a full matrix; values above the diagonal will not be used.</param>
00120 /// <param name="sequenceNames">The names of the taxa.  The indices of this list should correspond to
      the rows and columns of the <paramref name="distanceMatrix"/>.</param>
00121 /// <param name="constraint">An optional tree to constrain the search.  The tree produced by this
      method will be compatible with this tree.  The constraint tree can be multifurcating.</param>
00122 /// <param name="copyMatrix">If this is <see langword="true"/>, the matrix is copied before using it
      to compute the tree.  If this is <see langword="false"/>, the matrix is not copied.  Copying the
      matrix
```

```
00123 /// increases the memory used by the method, but note that if the matrix is not copied, it will be
      modified in-place!</param>
00124 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00125 /// <param name="progressCallback">A method used to report progress.</param>
00126 /// <returns>The UPGMA tree built from the supplied <paramref name="distanceMatrix"/>.</returns>
00127         public static TreeNode BuildTree(float[][] distanceMatrix, IReadOnlyList<string>
      sequenceNames, TreeNode constraint = null, bool copyMatrix = true, int numCores = -1, Action<double>
      progressCallback = null)
00128         {
00129             if (copyMatrix)
00130             {
00131                 float[][] newDistMat = new float[distanceMatrix.Length][];
00132 
00133                 for (int i = 0; i < distanceMatrix.Length; i++)
00134                 {
00135                     newDistMat[i] = new float[i];
00136 
00137                     for (int j = 0; j < i; j++)
00138                     {
00139                         newDistMat[i][j] = distanceMatrix[i][j];
00140                     }
00141                 }
00142 
00143                 distanceMatrix = newDistMat;
00144             }
00145 
00146             unsafe
00147             {
00148                 IntPtr[] rows = new IntPtr[distanceMatrix.Length];
00149 
00150                 GCHandle[] handles = new GCHandle[distanceMatrix.Length];
00151 
00152                 for (int i = 0; i < distanceMatrix.Length; i++)
00153                 {
00154                     handles[i] = GCHandle.Alloc(distanceMatrix[i], GCHandleType.Pinned);
00155                     rows[i] = handles[i].AddrOfPinnedObject();
00156                 }
00157 
00158                 fixed (IntPtr* rowsPointer = rows)
00159                 {
00160                     float** rowsFloatPointer = (float**)rowsPointer;
00161 
00162                     TreeNode tree;
00163 
00164                     if (constraint == null)
00165                     {
00166                         tree = BuildTree(rowsFloatPointer, sequenceNames, numCores, progressCallback);
00167                     }
00168                     else
00169                     {
00170                         tree = BuildTreeWithConstraint(rowsFloatPointer, sequenceNames, constraint,
      numCores, progressCallback);
00171                     }
00172 
00173                     for (int i = 0; i < distanceMatrix.Length; i++)
00174                     {
00175                         handles[i].Free();
00176                     }
00177 
00178                     return tree;
00179                 }
00180             }
00181         }
00182 
00183 
00184 /// <summary>
00185 /// Builds a UPGMA tree using data from a distance matrix.
00186 /// </summary>
00187 /// <param name="distanceMatrix">The distance matrix containg distances between the taxa.  This can be
      a lower triangular matrix or a full matrix; values above the diagonal will not be used.</param>
00188 /// <param name="sequenceNames">The names of the taxa.  The indices of this list should correspond to
      the rows and columns of the <paramref name="distanceMatrix"/>.</param>
00189 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00190 /// <param name="progressCallback">A method used to report progress.</param>
00191 /// <returns>The UPGMA tree built from the supplied <paramref name="distanceMatrix"/>.</returns>
00192         private static unsafe TreeNode BuildTree(float** distanceMatrix, IReadOnlyList<string>
      sequenceNames, int numCores, Action<double> progressCallback)
00193         {
00194             List<TreeNode> currentLeaves = new List<TreeNode>(sequenceNames.Count);
00195             List<int> correspondences = new List<int>(sequenceNames.Count);
00196             List<int> weights = new List<int>(sequenceNames.Count);
00197             List<double> downstreamLengths = new List<double>(sequenceNames.Count);
00198 
00199             for (int i = 0; i < sequenceNames.Count; i++)
00200             {
00201                 currentLeaves.Add(new TreeNode(null) { Name = sequenceNames[i] });
00202                 correspondences.Add(i);
```

```
00203                 downstreamLengths.Add(0);
00204                 weights.Add(1);
00205             }
00206 
00207             double totalToProcess = ((double)sequenceNames.Count * (sequenceNames.Count + 1) * (2 *
      sequenceNames.Count + 1)) / 12 - 0.25 * sequenceNames.Count * (sequenceNames.Count + 1);
00208             long processed = 0;
00209 
00210             ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = numCores };
00211 
00212             while (currentLeaves.Count > 1)
00213             {
00214                 long newStep = (long)currentLeaves.Count * (currentLeaves.Count - 1) / 2;
00215                 progressCallback?.Invoke(processed / totalToProcess);
00216 
00217                 int minI = -1;
00218                 int minJ = -1;
00219 
00220                 double minDist = double.MaxValue;
00221 
00222                 double[] minDists = new double[currentLeaves.Count];
00223                 int[] minJs = new int[currentLeaves.Count];
00224 
00225                 Parallel.For(0, currentLeaves.Count, opt, i =>
00226                 {
00227                     double myMinDist = double.MaxValue;
00228                     int myMinJ = -1;
00229 
00230                     for (int j = 0; j < i; j++)
00231                     {
00232                         int realI = correspondences[i];
00233                         int realJ = correspondences[j];
00234 
00235                         double dist = distanceMatrix[realI][realJ];
00236 
00237                         if (dist < myMinDist)
00238                         {
00239                             myMinDist = dist;
00240                             myMinJ = j;
00241                         }
00242                     }
00243 
00244                     minDists[i] = myMinDist;
00245                     minJs[i] = myMinJ;
00246                 });
00247 
00248                 for (int i = 0; i < currentLeaves.Count; i++)
00249                 {
00250                     if (minDists[i] < minDist)
00251                     {
00252                         minDist = minDists[i];
00253                         minI = i;
00254                         minJ = minJs[i];
00255                     }
00256                 }
00257 
00258                 minDist /= 2;
00259 
00260                 TreeNode newNode = new TreeNode(null);
00261                 newNode.Children.Add(currentLeaves[minI]);
00262                 newNode.Children.Add(currentLeaves[minJ]);
00263                 currentLeaves[minI].Parent = newNode;
00264                 currentLeaves[minJ].Parent = newNode;
00265                 currentLeaves[minI].Length = minDist - downstreamLengths[minI];
00266                 currentLeaves[minJ].Length = minDist - downstreamLengths[minJ];
00267 
00268                 int correspI = correspondences[minI];
00269                 int correspJ = correspondences[minJ];
00270 
00271                 int weightI = weights[minI];
00272                 int weightJ = weights[minJ];
00273                 int newWeight = weightI + weightJ;
00274 
00275                 currentLeaves.RemoveAt(Math.Min(minI, minJ));
00276                 correspondences.RemoveAt(Math.Min(minI, minJ));
00277                 downstreamLengths.RemoveAt(Math.Min(minI, minJ));
00278                 weights.RemoveAt(Math.Min(minI, minJ));
00279 
00280                 currentLeaves[Math.Max(minI, minJ) - 1] = newNode;
00281                 correspondences[Math.Max(minI, minJ) - 1] = correspI;
00282                 downstreamLengths[Math.Max(minI, minJ) - 1] = minDist;
00283                 weights[Math.Max(minI, minJ) - 1] = newWeight;
00284 
00285                 for (int i = 0; i < currentLeaves.Count; i++)
00286                 {
00287                     distanceMatrix[Math.Max(correspI, correspondences[i])][Math.Min(correspI,
      correspondences[i])] = (distanceMatrix[Math.Max(correspI, correspondences[i])][Math.Min(correspI,
```

```
    correspondences[i])] * weightI + distanceMatrix[Math.Max(correspJ,
    correspondences[i])][Math.Min(correspJ, correspondences[i])] * weightJ) / newWeight;
00288                 }
00289
00290                 processed += newStep;
00291             }
00292
00293             progressCallback?.Invoke(1);
00294
00295             return currentLeaves[0];
00296         }
00297
00298 /// <summary>
00299 /// Builds a UPGMA tree using data from a distance matrix, applying the specified constraint.
00300 /// </summary>
00301 /// <param name="distanceMatrix">The distance matrix containg distances between the taxa.  This can be
    a lower triangular matrix or a full matrix; values above the diagonal will not be used.</param>
00302 /// <param name="sequenceNames">The names of the taxa.  The indices of this list should correspond to
    the rows and columns of the <paramref name="distanceMatrix"/>.</param>
00303 /// <param name="constraint">A tree to constrain the search.  The tree produced by this method will be
    compatible with this tree.  The constraint tree can be multifurcating.</param>
00304 /// <param name="numCores">Maximum number of threads to use, or -1 to let the runtime decide.</param>
00305 /// <param name="progressCallback">A method used to report progress.</param>
00306 /// <returns>The UPGMA tree built from the supplied <paramref name="distanceMatrix"/>.</returns>
00307         private static unsafe TreeNode BuildTreeWithConstraint(float** distanceMatrix,
    IReadOnlyList<string> sequenceNames, TreeNode constraint, int numCores, Action<double>
    progressCallback)
00308         {
00309             Dictionary<string, int> sequenceIndices = new Dictionary<string,
    int>(sequenceNames.Count);
00310             for (int i = 0; i < sequenceNames.Count; i++)
00311             {
00312                 sequenceIndices[sequenceNames[i]] = i;
00313             }
00314
00315             constraint = constraint.GetUnrootedTree();
00316
00317             List<int[][]> splits = new List<int[][]>();
00318
00319             foreach (var v in constraint.GetSplits())
00320             {
00321                 if (v.side1.Count > 1 && v.side2.Count > 1)
00322                 {
00323                     int[] leftSide = new int[v.side1.Count];
00324                     int[] rightSide = new int[v.side2.Count];
00325
00326                     for (int i = 0; i < v.side1.Count; i++)
00327                     {
00328                         leftSide[i] = sequenceIndices[v.side1[i].Name];
00329                     }
00330
00331                     for (int i = 0; i < v.side2.Count; i++)
00332                     {
00333                         rightSide[i] = sequenceIndices[v.side2[i].Name];
00334                     }
00335
00336                     splits.Add(new int[][] { leftSide, rightSide });
00337                 }
00338             }
00339
00340             List<TreeNode> currentLeaves = new List<TreeNode>(sequenceNames.Count);
00341             List<int> correspondences = new List<int>(sequenceNames.Count);
00342             List<int> weights = new List<int>(sequenceNames.Count);
00343             List<double> downstreamLengths = new List<double>(sequenceNames.Count);
00344             List<List<int>» underlyingLeaves = new List<List<int»(sequenceNames.Count);
00345
00346             for (int i = 0; i < sequenceNames.Count; i++)
00347             {
00348                 currentLeaves.Add(new TreeNode(null) { Name = sequenceNames[i] });
00349                 correspondences.Add(i);
00350                 downstreamLengths.Add(0);
00351                 weights.Add(1);
00352                 underlyingLeaves.Add(new List<int>() { i });
00353             }
00354
00355             double totalToProcess = ((double)sequenceNames.Count * (sequenceNames.Count + 1) * (2 *
    sequenceNames.Count + 1)) / 12 - 0.25 * sequenceNames.Count * (sequenceNames.Count + 1);
00356             long processed = 0;
00357
00358             ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = numCores };
00359
00360             while (currentLeaves.Count > 1)
00361             {
00362                 long newStep = (long)currentLeaves.Count * (currentLeaves.Count - 1) / 2;
00363                 progressCallback?.Invoke(processed / totalToProcess);
00364
00365                 int minI = -1;
```

```
00366                 int minJ = -1;
00367 
00368                 double minDist = double.MaxValue;
00369 
00370                 double[] minDists = new double[currentLeaves.Count];
00371                 int[] minJs = new int[currentLeaves.Count];
00372 
00373                 Parallel.For(0, currentLeaves.Count, opt, i =>
00374                 {
00375                     double myMinDist = double.MaxValue;
00376                     int myMinJ = -1;
00377 
00378                     for (int j = 0; j < i; j++)
00379                     {
00380                         int realI = correspondences[i];
00381                         int realJ = correspondences[j];
00382 
00383                         double dist = distanceMatrix[realI][realJ];
00384 
00385                         if (dist < myMinDist)
00386                         {
00387                             HashSet<int> leftSide = new HashSet<int>(underlyingLeaves[i].Count +
      underlyingLeaves[j].Count);
00388                             for (int k = 0; k < underlyingLeaves[i].Count; k++)
00389                             {
00390                                 leftSide.Add(underlyingLeaves[i][k]);
00391                             }
00392                             for (int k = 0; k < underlyingLeaves[j].Count; k++)
00393                             {
00394                                 leftSide.Add(underlyingLeaves[j][k]);
00395                             }
00396 
00397                             HashSet<int> rightSide = new HashSet<int>(sequenceNames.Count);
00398 
00399                             for (int k = 0; k < sequenceNames.Count; k++)
00400                             {
00401                                 if (!leftSide.Contains(k))
00402                                 {
00403                                     rightSide.Add(k);
00404                                 }
00405                             }
00406 
00407                             if (NeighborJoining.IsCompatible(leftSide, rightSide, splits))
00408                             {
00409                                 myMinDist = dist;
00410                                 myMinJ = j;
00411                             }
00412                         }
00413                     }
00414 
00415                     minDists[i] = myMinDist;
00416                     minJs[i] = myMinJ;
00417                 });
00418 
00419                 for (int i = 0; i < currentLeaves.Count; i++)
00420                 {
00421                     if (minDists[i] < minDist)
00422                     {
00423                         minDist = minDists[i];
00424                         minI = i;
00425                         minJ = minJs[i];
00426                     }
00427                 }
00428 
00429                 minDist /= 2;
00430 
00431                 TreeNode newNode = new TreeNode(null);
00432                 newNode.Children.Add(currentLeaves[minI]);
00433                 newNode.Children.Add(currentLeaves[minJ]);
00434                 currentLeaves[minI].Parent = newNode;
00435                 currentLeaves[minJ].Parent = newNode;
00436 
00437                 // Prevent negative branch lengths.
00438                 if (downstreamLengths[minI] > minDist && downstreamLengths[minJ] <= minDist)
00439                 {
00440                     minDist += 2 * (downstreamLengths[minI] - minDist);
00441                 }
00442                 else if (downstreamLengths[minJ] > minDist && downstreamLengths[minI] <= minDist)
00443                 {
00444                     minDist += 2 * (downstreamLengths[minJ] - minDist);
00445                 }
00446                 else if(downstreamLengths[minI] > minDist && downstreamLengths[minJ] > minDist)
00447                 {
00448                     minDist += downstreamLengths[minI] + downstreamLengths[minJ] - 2 * minDist;
00449                 }
00450 
00451                 currentLeaves[minI].Length = minDist - downstreamLengths[minI];
```

```
00452                 currentLeaves[minJ].Length = minDist - downstreamLengths[minJ];
00453
00454                 int correspI = correspondences[minI];
00455                 int correspJ = correspondences[minJ];
00456
00457                 int weightI = weights[minI];
00458                 int weightJ = weights[minJ];
00459                 int newWeight = weightI + weightJ;
00460
00461                 underlyingLeaves[Math.Max(minI, minJ)].AddRange(underlyingLeaves[Math.Min(minI,
      minJ)]);
00462                 underlyingLeaves.RemoveAt(Math.Min(minI, minJ));
00463
00464                 currentLeaves.RemoveAt(Math.Min(minI, minJ));
00465                 correspondences.RemoveAt(Math.Min(minI, minJ));
00466                 downstreamLengths.RemoveAt(Math.Min(minI, minJ));
00467                 weights.RemoveAt(Math.Min(minI, minJ));
00468
00469                 currentLeaves[Math.Max(minI, minJ) - 1] = newNode;
00470                 correspondences[Math.Max(minI, minJ) - 1] = correspI;
00471                 downstreamLengths[Math.Max(minI, minJ) - 1] = minDist;
00472                 weights[Math.Max(minI, minJ) - 1] = newWeight;
00473
00474                 for (int i = 0; i < currentLeaves.Count; i++)
00475                 {
00476                     distanceMatrix[Math.Max(correspI, correspondences[i])][Math.Min(correspI,
      correspondences[i])] = (distanceMatrix[Math.Max(correspI, correspondences[i])][Math.Min(correspI,
      correspondences[i])] * weightI + distanceMatrix[Math.Max(correspJ,
      correspondences[i])][Math.Min(correspJ, correspondences[i])] * weightJ) / newWeight;
00477                 }
00478
00479                 processed += newStep;
00480             }
00481
00482             progressCallback?.Invoke(1);
00483
00484             return currentLeaves[0];
00485         }
00486     }
00487 }
```

8.26 TreeCollection.cs

```
00001 using System;
00002 using System.Collections;
00003 using System.Collections.Generic;
00004 using System.Diagnostics.Contracts;
00005 using System.IO;
00006 using PhyloTree.Formats;
00007
00008 namespace PhyloTree
00009 {
00010 /// <summary>
00011 /// Represents a collection of <see cref="TreeNode"/> objects.
00012 /// If the full representations of the <see cref="TreeNode"/> objects reside in memory, this offers
      the best performance at the expense of memory usage.
00013 /// Alternatively, the trees may be read on demand from a stream in binary format.  In this case,
      accessing any of the trees will require the tree to be parsed.  This reduces memory usage, but worsens
      performance.
00014 /// The internal storage model of the collection is transparent to consumers (except for the
      difference in performance/memory usage).
00015 /// </summary>
00016     public class TreeCollection :  IList<TreeNode>, IReadOnlyList<TreeNode>, IDisposable
00017     {
00018 /// <summary>
00019 /// A list containing the <see cref="TreeNode"/> objects, if they are stored in memory.
00020 /// </summary>
00021         private List<TreeNode> InternalStorage = null;
00022
00023 /// <summary>
00024 /// A stream containing the tree data in binary format, if this is the chosen storage model.  This can
      be either a <see cref="MemoryStream"/> or a <see cref="FileStream"/>.
00025 /// </summary>
00026         public Stream UnderlyingStream { get; private set; } = null;
00027
00028 /// <summary>
00029 /// A <see cref="BinaryReader"/> that reads the <see cref="UnderlyingStream"/>
00030 /// </summary>
00031         private BinaryReader UnderlyingReader = null;
00032
00033 /// <summary>
00034 /// If the trees are stored in binary format, this contains the addresses of the trees (i.e.  byte
      offsets from the start of the stream).
```

```
00035 /// </summary>
00036         private List<long> TreeAddresses = null;
00037 
00038 /// <summary>
00039 /// If the collection is manipulated when the trees are stored in the <see cref="UnderlyingStream"/>,
      entries in <see cref="TreeIndexCorrespondence"/> are used to keep track of which indices have had
      their meaning change.
00040 /// </summary>
00041         private List<int> TreeIndexCorrespondence = null;
00042 
00043 /// <summary>
00044 /// If the trees are stored in binary format, this determines whether there are global names that are
      used in parsing the trees.
00045 /// </summary>
00046         private readonly bool GlobalNames = false;
00047 
00048 /// <summary>
00049 /// If the trees are stored in binary format, this contains any global names that are used in parsing
      the trees.
00050 /// </summary>
00051         private IReadOnlyList<string> AllNames = null;
00052 
00053 /// <summary>
00054 /// If the trees are stored in binary format, this contains any global attributes that are used in
      parsing the trees.
00055 /// </summary>
00056         private IReadOnlyList<Formats.Attribute> AllAttributes = null;
00057 
00058 /// <summary>
00059 /// Describes the internal storage model of the collection.
00060 /// </summary>
00061         enum StorageTypes
00062         {
00063 /// <summary>
00064 /// The trees are stored in a <see cref="List"/>.
00065 /// </summary>
00066             List,
00067 
00068 /// <summary>
00069 /// The trees are stored in binary format in a <see cref="FileStream"/> or <see cref="MemoryStream"/>.
00070 /// </summary>
00071             Stream
00072         }
00073 
00074 /// <summary>
00075 /// Determines the internal storage model of the collection.
00076 /// </summary>
00077         private StorageTypes StorageType;
00078 
00079 /// <summary>
00080 /// If the trees are stored on disk in a temporary file, you should assign this property to the full
      path of the file.  The file will be deleted when the <see cref="TreeCollection"/> is <see
      cref="Dispose()"/>d.
00081 /// </summary>
00082         public string TemporaryFile { get; set; } = null;
00083 
00084 /// <summary>
00085 /// The number of trees in the collection.
00086 /// </summary>
00087         public int Count
00088         {
00089             get
00090             {
00091                 if (StorageType == StorageTypes.List)
00092                 {
00093                     return InternalStorage.Count;
00094                 }
00095                 else
00096                 {
00097                     return TreeIndexCorrespondence.Count;
00098                 }
00099             }
00100         }
00101 
00102 /// <summary>
00103 /// Determine whether the collection is read-only.  This is always <c>false</c> in the current
      implementation.
00104 /// </summary>
00105         public bool IsReadOnly => false;
00106 
00107 /// <summary>
00108 /// Obtains an element from the collection.
00109 /// </summary>
00110 /// <param name="index">The index of the element to extract.</param>
00111 /// <returns>The requested element from the collection.</returns>
00112         public TreeNode this[int index]
00113         {
```

```
00114             get
00115             {
00116                 if (StorageType == StorageTypes.List)
00117                 {
00118                     return InternalStorage[index];
00119                 }
00120                 else
00121                 {
00122                     int correspIndex = TreeIndexCorrespondence[index];
00123                     if (correspIndex >= 0)
00124                     {
00125                         UnderlyingStream.Seek(TreeAddresses[correspIndex], SeekOrigin.Begin);
00126                         return UnderlyingReader.ReadTree(GlobalNames, AllNames, AllAttributes);
00127                     }
00128                     else
00129                     {
00130                         return InternalStorage[-correspIndex - 1];
00131                     }
00132                 }
00133             }
00134 
00135             set
00136             {
00137                 if (StorageType == StorageTypes.List)
00138                 {
00139                     InternalStorage[index] = value;
00140                 }
00141                 else
00142                 {
00143                     int correspIndex = TreeIndexCorrespondence[index];
00144                     if (correspIndex >= 0)
00145                     {
00146                         InternalStorage.Add(value);
00147                         TreeIndexCorrespondence[index] = -InternalStorage.Count;
00148                     }
00149                     else
00150                     {
00151                         InternalStorage[-correspIndex - 1] = value;
00152                     }
00153                 }
00154             }
00155         }
00156 
00157 /// <summary>
00158 /// Adds an element to the collection.  This is stored in memory, even if the internal storage model
      of the collection is a <see cref="Stream"/>.
00159 /// </summary>
00160 /// <param name="item">The element to add.</param>
00161         public void Add(TreeNode item)
00162         {
00163             InternalStorage.Add(item);
00164             if (StorageType == StorageTypes.Stream)
00165             {
00166                 TreeIndexCorrespondence.Add(-InternalStorage.Count);
00167             }
00168         }
00169 
00170 /// <summary>
00171 /// Adds multiple elements to the collection.  These are stored in memory, even if the internal
      storage model of the collection is a <see cref="Stream"/>.
00172 /// </summary>
00173 /// <param name="items">The elements to add.</param>
00174         public void AddRange(IEnumerable<TreeNode> items)
00175         {
00176             Contract.Requires(items != null);
00177             foreach (TreeNode item in items)
00178             {
00179                 InternalStorage.Add(item);
00180                 if (StorageType == StorageTypes.Stream)
00181                 {
00182                     TreeIndexCorrespondence.Add(-InternalStorage.Count + 1);
00183                 }
00184             }
00185         }
00186 
00187 /// <summary>
00188 /// Get an <see cref="IEnumerator"/> over the collection.
00189 /// </summary>
00190 /// <returns>An <see cref="IEnumerator"/> over the collection.</returns>
00191         public IEnumerator<TreeNode> GetEnumerator()
00192         {
00193             if (StorageType == StorageTypes.List)
00194             {
00195                 return InternalStorage.GetEnumerator();
00196             }
00197             else
00198             {
```

```
00199                 TreeCollection coll = this;
00200
00201                 IEnumerable<TreeNode> GetEnumerable()
00202                 {
00203                     for (int i = 0; i < coll.Count; i++)
00204                     {
00205                         yield return coll[i];
00206                     }
00207                 };
00208
00209                 return GetEnumerable().GetEnumerator();
00210             }
00211         }
00212
00213 /// <summary>
00214 /// Get an <see cref="IEnumerator"/> over the collection.
00215 /// </summary>
00216 /// <returns>An <see cref="IEnumerator"/> over the collection.</returns>
00217         IEnumerator IEnumerable.GetEnumerator()
00218         {
00219             if (StorageType == StorageTypes.List)
00220             {
00221                 return InternalStorage.GetEnumerator();
00222             }
00223             else
00224             {
00225                 TreeCollection coll = this;
00226
00227                 IEnumerable<TreeNode> GetEnumerable()
00228                 {
00229                     for (int i = 0; i < coll.Count; i++)
00230                     {
00231                         yield return coll[i];
00232                     }
00233                 };
00234
00235                 return GetEnumerable().GetEnumerator();
00236             }
00237         }
00238
00239 /// <summary>
00240 /// Finds the index of the first occurrence of an element in the collection.
00241 /// </summary>
00242 /// <param name="item">The item to search for.</param>
00243 /// <returns>The index of the item in the collection.</returns>
00244         public int IndexOf(TreeNode item)
00245         {
00246             if (StorageType == StorageTypes.List)
00247             {
00248                 return InternalStorage.IndexOf(item);
00249             }
00250             else
00251             {
00252                 for (int i = 0; i < this.TreeIndexCorrespondence.Count; i++)
00253                 {
00254                     if (this.TreeIndexCorrespondence[i] < 0)
00255                     {
00256                         if (InternalStorage[-this.TreeIndexCorrespondence[i] - 1] == item)
00257                         {
00258                             return i;
00259                         }
00260                     }
00261                 }
00262                 return -1;
00263             }
00264         }
00265
00266 /// <summary>
00267 /// Inserts an element in the collection at the specified index.
00268 /// </summary>
00269 /// <param name="index">The index at which to insert the element.</param>
00270 /// <param name="item">The element to insert.</param>
00271         public void Insert(int index, TreeNode item)
00272         {
00273             if (StorageType == StorageTypes.List)
00274             {
00275                 InternalStorage.Insert(index, item);
00276             }
00277             else
00278             {
00279                 if (index < 0 || index >= this.Count)
00280                 {
00281                     throw new ArgumentOutOfRangeException(paramName:  nameof(index));
00282                 }
00283
00284                 InternalStorage.Add(item);
00285                 TreeIndexCorrespondence.Add(TreeIndexCorrespondence[^-1]);
```

```
00286                 for (int i = TreeIndexCorrespondence.Count - 2; i > index; i--)
00287                 {
00288                     TreeIndexCorrespondence[i] = TreeIndexCorrespondence[i - 1];
00289                 }
00290                 TreeIndexCorrespondence[index] = -InternalStorage.Count;
00291             }
00292         }
00293 
00294 /// <summary>
00295 /// Removes from the collection the element at the specified index.
00296 /// </summary>
00297 /// <param name="index"></param>
00298         public void RemoveAt(int index)
00299         {
00300             if (StorageType == StorageTypes.List)
00301             {
00302                 InternalStorage.RemoveAt(index);
00303             }
00304             else
00305             {
00306                 if (TreeIndexCorrespondence[index] >= 0)
00307                 {
00308                     TreeIndexCorrespondence.RemoveAt(index);
00309                 }
00310                 else
00311                 {
00312                     int underlyingIndex = -TreeIndexCorrespondence[index] - 1;
00313                     TreeIndexCorrespondence.RemoveAt(index);
00314                     InternalStorage.RemoveAt(underlyingIndex);
00315                     for (int i = 0; i < TreeIndexCorrespondence.Count; i++)
00316                     {
00317                         if (TreeIndexCorrespondence[i] < 0 && (-TreeIndexCorrespondence[i] - 1) >
    underlyingIndex)
00318                         {
00319                             TreeIndexCorrespondence[i]++;
00320                         }
00321                     }
00322                 }
00323             }
00324         }
00325 
00326 /// <summary>
00327 /// Removes all elements from the collection.  If the internal storage model is a <see
    cref="Stream"/>, it is disposed and the internal storage model is converted to a <see
    cref="List{TreeNode}"/>.
00328 /// </summary>
00329         public void Clear()
00330         {
00331             if (StorageType == StorageTypes.List)
00332             {
00333                 this.InternalStorage.Clear();
00334             }
00335             else
00336             {
00337                 this.InternalStorage.Clear();
00338                 UnderlyingReader.Dispose();
00339                 UnderlyingStream.Dispose();
00340                 UnderlyingReader = null;
00341                 UnderlyingStream = null;
00342                 TreeAddresses = null;
00343                 TreeIndexCorrespondence = null;
00344                 AllNames = null;
00345                 AllAttributes = null;
00346 
00347                 this.StorageType = StorageTypes.List;
00348             }
00349         }
00350 
00351 /// <summary>
00352 /// Determines whether the collection contains the specified element.
00353 /// </summary>
00354 /// <param name="item">The element to search for.</param>
00355 /// <returns><c>true</c> if the collection contains the specified element, <c>false</c>
    otherwise.</returns>
00356         public bool Contains(TreeNode item)
00357         {
00358             if (StorageType == StorageTypes.List)
00359             {
00360                 return InternalStorage.Contains(item);
00361             }
00362             else
00363             {
00364                 for (int i = 0; i < this.TreeIndexCorrespondence.Count; i++)
00365                 {
00366                     if (this.TreeIndexCorrespondence[i] > 0)
00367                     {
00368                         if (InternalStorage[-this.TreeIndexCorrespondence[i] - 1] == item)
```

```
00369                         {
00370                             return true;
00371                         }
00372                     }
00373                 }
00374                 return false;
00375             }
00376         }
00377 
00378 /// <summary>
00379 /// Copies the collection to an array.
00380 /// </summary>
00381 /// <param name="array">The array in which to copy the collection.</param>
00382 /// <param name="arrayIndex">The index at which to start the copy.</param>
00383         public void CopyTo(TreeNode[] array, int arrayIndex)
00384         {
00385             Contract.Requires(array != null);
00386             if (StorageType == StorageTypes.List)
00387             {
00388                 InternalStorage.CopyTo(array, arrayIndex);
00389             }
00390             else
00391             {
00392                 for (int i = 0; i < this.Count; i++)
00393                 {
00394                     array[arrayIndex + i] = this[i];
00395                 }
00396             }
00397         }
00398 
00399 /// <summary>
00400 /// Removes the specified element from the collection.
00401 /// </summary>
00402 /// <param name="item">The element to remove.</param>
00403 /// <returns><c>true</c> if the removal was successful (i.e.  the list contained the element in the
      first place), <c>false</c> otherwise.</returns>
00404         public bool Remove(TreeNode item)
00405         {
00406             if (StorageType == StorageTypes.List)
00407             {
00408                 return InternalStorage.Remove(item);
00409             }
00410             else
00411             {
00412                 int index = this.IndexOf(item);
00413                 if (index >= 0)
00414                 {
00415                     this.RemoveAt(index);
00416                     return true;
00417                 }
00418                 else
00419                 {
00420                     return false;
00421                 }
00422             }
00423         }
00424 
00425 /// <summary>
00426 /// Constructs a <see cref="TreeCollection"/> object from a <see cref="List{TreeNode}"/>.
00427 /// </summary>
00428 /// <param name="internalStorage">The <see cref="List{TreeNode}"/> containing the trees to store in
      the collection.  Note that this list is not copied, but used as-is.</param>
00429         public TreeCollection(List<TreeNode> internalStorage)
00430         {
00431             InternalStorage = internalStorage;
00432             StorageType = StorageTypes.List;
00433         }
00434 
00435 /// <summary>
00436 /// Constructs a <see cref="TreeCollection"/> object from a stream of trees in binary format.
00437 /// </summary>
00438 /// <param name="binaryTreeStream">The stream of trees in binary format to use.  The stream will be
      disposed when the <see cref="TreeCollection"/> is disposed.  It should not be disposed earlier by
      external code.</param>
00439         public TreeCollection(Stream binaryTreeStream)
00440         {
00441             UnderlyingStream = binaryTreeStream;
00442             UnderlyingReader = new BinaryReader(UnderlyingStream);
00443 
00444             BinaryTreeMetadata metadata = BinaryTree.ParseMetadata(binaryTreeStream, true,
      UnderlyingReader);
00445 
00446             TreeAddresses = new List<long>(metadata.TreeAddresses);
00447 
00448             GlobalNames = metadata.GlobalNames;
00449             AllNames = metadata.Names;
00450             AllAttributes = metadata.AllAttributes;
```

```
00451
00452             TreeIndexCorrespondence = new List<int>();
00453             for (int i = 0; i < TreeAddresses.Count; i++)
00454             {
00455                 TreeIndexCorrespondence.Add(i);
00456             }
00457             StorageType = StorageTypes.Stream;
00458         }
00459
00460 /// <summary>
00461 /// Determines whether the <see cref="TreeCollection"/> has already been disposed.
00462 /// </summary>
00463         private bool disposedValue = false;
00464
00465 /// <summary>
00466 /// Disposes the <see cref="TreeCollection"/>.
00467 /// </summary>
00468 /// <param name="disposing">Determines whether the method has been called by user code or by the
      destructor.</param>
00469         [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1031")]
00470         protected virtual void Dispose(bool disposing)
00471         {
00472             if (!disposedValue)
00473             {
00474                 if (disposing)
00475                 {
00476                     UnderlyingReader?.Dispose();
00477                     UnderlyingStream?.Dispose();
00478                 }
00479
00480                 InternalStorage = null;
00481                 UnderlyingReader = null;
00482                 UnderlyingStream = null;
00483                 TreeAddresses = null;
00484                 TreeIndexCorrespondence = null;
00485                 AllNames = null;
00486                 AllAttributes = null;
00487
00488                 if (!string.IsNullOrEmpty(TemporaryFile))
00489                 {
00490                     try
00491                     {
00492                         File.Delete(TemporaryFile);
00493                     }
00494                     catch { }
00495                     TemporaryFile = null;
00496                 }
00497
00498                 disposedValue = true;
00499             }
00500         }
00501
00502 /// <summary>
00503 /// Destructor.
00504 /// </summary>
00505         ~TreeCollection()
00506         {
00507             Dispose(false);
00508         }
00509
00510 /// <summary>
00511 /// Disposes the <see cref="TreeCollection"/>, the underlying <see cref="Stream"/> and <see
      cref="StreamReader"/>, and deletes the <see cref="TemporaryFile"/> (if applicable).
00512 /// </summary>
00513         public void Dispose()
00514         {
00515             Dispose(true);
00516             GC.SuppressFinalize(this);
00517         }
00518     }
00519
00520 }
```

8.27 TreeNode.Comparisons.cs

```
00001 using PhyloTree.Extensions;
00002 using PhyloTree.Formats;
00003 using System;
00004 using System.Collections.Concurrent;
00005 using System.Collections.Generic;
00006 using System.Diagnostics.CodeAnalysis;
00007 using System.Linq;
00008 using System.Runtime.CompilerServices;
```

```
00009 using System.Runtime.InteropServices;
00010 using System.Text;
00011 using System.Threading;
00012 using System.Threading.Tasks;
00013
00014 namespace PhyloTree
00015 {
00016     public partial class TreeNode
00017     {
00018 /// <summary>
00019 /// Computes the Robinson-Foulds distance between the current tree and another tree.
00020 /// </summary>
00021 /// <param name="otherTree">The other tree whose distance to the current tree is computed.</param>
00022 /// <param name="weighted">If this is <see langword="true" />, the distance is weighted based on the
      branch lengths; otherwise, it is unweighted.</param>
00023 /// <returns>The Robinson-Foulds distance between this tree and the <paramref
      name="otherTree"/>.</returns>
00024         public double RobinsonFouldsDistance(TreeNode otherTree, bool weighted)
00025         {
00026             return RobinsonFouldsDistance(this, otherTree, weighted);
00027         }
00028
00029 /// <summary>
00030 /// Computes the Robinson-Foulds distance between two trees.
00031 /// </summary>
00032 /// <param name="tree1">The first tree.</param>
00033 /// <param name="tree2">The second tree.</param>
00034 /// <param name="weighted">If this is <see langword="true" />, the distance is weighted based on the
      branch lengths; otherwise, it is unweighted.</param>
00035 /// <returns>The Robinson-Foulds distance between <paramref name="tree1"/> and <paramref
      name="tree2"/>.</returns>
00036         public static double RobinsonFouldsDistance(TreeNode tree1, TreeNode tree2, bool weighted)
00037         {
00038             if (tree1 == null)
00039             {
00040                 throw new ArgumentNullException(nameof(tree1), "The first tree cannot be null!");
00041             }
00042
00043             if (tree2 == null)
00044             {
00045                 throw new ArgumentNullException(nameof(tree2), "The second tree cannot be null!");
00046             }
00047
00048             double[,] distMat = new double[2, 2];
00049
00050             if (weighted)
00051             {
00052                 FillDistanceMatrix(new TreeNode[] { tree1, tree2 }, wRFDistances:  distMat,
      maxThreadCount:  1);
00053             }
00054             else
00055             {
00056                 FillDistanceMatrix(new TreeNode[] { tree1, tree2 }, RFDistances:  distMat,
      maxThreadCount:  1);
00057             }
00058
00059             return distMat[0, 1];
00060         }
00061
00062 /// <summary>
00063 /// Computes the edge-length distance between the current tree and another tree.
00064 /// </summary>
00065 /// <param name="otherTree">The other tree whose distance to the current tree is computed.</param>
00066 /// <returns>The edge-length distance between this tree and the <paramref
      name="otherTree"/>.</returns>
00067         public double EdgeLengthDistance(TreeNode otherTree)
00068         {
00069             return EdgeLengthDistance(this, otherTree);
00070         }
00071
00072 /// <summary>
00073 /// Computes the edge-length distance between two trees.
00074 /// </summary>
00075 /// <param name="tree1">The first tree.</param>
00076 /// <param name="tree2">The second tree.</param>
00077 /// <returns>The edge-length distance between <paramref name="tree1"/> and <paramref
      name="tree2"/>.</returns>
00078         public static double EdgeLengthDistance(TreeNode tree1, TreeNode tree2)
00079         {
00080             if (tree1 == null)
00081             {
00082                 throw new ArgumentNullException(nameof(tree1), "The first tree cannot be null!");
00083             }
00084
00085             if (tree2 == null)
00086             {
00087                 throw new ArgumentNullException(nameof(tree2), "The second tree cannot be null!");
```

```
00088             }
00089
00090             double[,] distMat = new double[2, 2];
00091
00092             FillDistanceMatrix(new TreeNode[] { tree1, tree2 }, ELDistances:  distMat, maxThreadCount:
      1);
00093             return distMat[0, 1];
00094         }
00095
00096 /// <summary>
00097 /// Defines ways of pruning trees during comparisons.
00098 /// </summary>
00099         public enum TreeComparisonPruningMode
00100         {
00101 /// <summary>
00102 /// Specifies that the subset of leaves shared between each pair of trees should be used.
00103 /// </summary>
00104             Pairwise,
00105
00106 /// <summary>
00107 /// Specifies that the subset of leaves common to all trees should be used.
00108 /// </summary>
00109             Global
00110         }
00111
00112 /// <summary>
00113 /// Computes a distance matrix containing the Robinson-Foulds distances between each pair of trees in
      a list.
00114 /// </summary>
00115 /// <param name="trees">The list of trees that should be compared.</param>
00116 /// <param name="weighted">If this is <see langword="true" />, the distance is weighted based on the
      branch lengths; otherwise, it is unweighted.</param>
00117 /// <param name="pruningMode">If this is <see cref="TreeComparisonPruningMode.Global"/>, all trees are
      pruned so that they only contain the subset of leaves that are present in all trees.  If this is <see
      cref="TreeComparisonPruningMode.Pairwise"/>, during each comparisons the two trees are pruned so that
      they contain the subset of leaves that are in common between both of them.</param>
00118 /// <param name="maxThreadCount">The maximum number of threads to use for parallelised steps.</param>
00119 /// <param name="progress">An <see cref="IProgress{T}"/> for progress reporting.</param>
00120 /// <returns>A square <see langword="double"/>[,] matrix containing the requested distances between
      the trees.</returns>
00121         public static double[,] RobinsonFouldsDistances(IReadOnlyList<TreeNode> trees, bool weighted,
      TreeComparisonPruningMode pruningMode = TreeComparisonPruningMode.Pairwise, int maxThreadCount = -1,
      IProgress<double> progress = null)
00122         {
00123             double[,] distMat = new double[trees.Count, trees.Count];
00124
00125             if (weighted)
00126             {
00127                 FillDistanceMatrix(trees, comparePairwise:  pruningMode ==
      TreeComparisonPruningMode.Pairwise, wRFDistances:  distMat, maxThreadCount:  maxThreadCount, progress:
      progress);
00128             }
00129             else
00130             {
00131                 FillDistanceMatrix(trees, comparePairwise:  pruningMode ==
      TreeComparisonPruningMode.Pairwise, RFDistances:  distMat, maxThreadCount:  maxThreadCount, progress:
      progress);
00132             }
00133
00134             return distMat;
00135         }
00136
00137 /// <summary>
00138 /// Computes a distance matrix containing the edge-length distances between each pair of trees in a
      list.
00139 /// </summary>
00140 /// <param name="trees">The list of trees that should be compared.</param>
00141 /// <param name="pruningMode">If this is <see cref="TreeComparisonPruningMode.Global"/>, all trees are
      pruned so that they only contain the subset of leaves that are present in all trees.  If this is <see
      cref="TreeComparisonPruningMode.Pairwise"/>, during each comparisons the two trees are pruned so that
      they contain the subset of leaves that are in common between both of them.</param>
00142 /// <param name="maxThreadCount">The maximum number of threads to use for parallelised steps.</param>
00143 /// <param name="progress">An <see cref="IProgress{T}"/> for progress reporting.</param>
00144 /// <returns>A square <see langword="double"/>[,] matrix containing the requested distances between
      the trees.</returns>
00145         public static double[,] EdgeLengthDistances(IReadOnlyList<TreeNode> trees,
      TreeComparisonPruningMode pruningMode = TreeComparisonPruningMode.Pairwise, int maxThreadCount = -1,
      IProgress<double> progress = null)
00146         {
00147             double[,] distMat = new double[trees.Count, trees.Count];
00148
00149             FillDistanceMatrix(trees, comparePairwise:  pruningMode ==
      TreeComparisonPruningMode.Pairwise, ELDistances:  distMat, maxThreadCount:  maxThreadCount, progress:
      progress);
00150
00151             return distMat;
00152         }
```

```
00153
00154 /// <summary>
00155 /// Computes two distance matrices containing the unweighted and weighted Robinson-Foulds distances
      between each pair of trees in a list.  Much faster than computing the two distance matrices
      separately.
00156 /// </summary>
00157 /// <param name="trees">The list of trees that should be compared.</param>
00158 /// <param name="RFDistances">When this method returns, this variable will contain the computed
      Robinson-Foulds distances between the trees.</param>
00159 /// <param name="weightedRFDistances">When this method returns, this variable will contain the
      computed weighted Robinson-Foulds distances between the trees.</param>
00160 /// <param name="pruningMode">If this is <see cref="TreeComparisonPruningMode.Global"/>, all trees are
      pruned so that they only contain the subset of leaves that are present in all trees.  If this is <see
      cref="TreeComparisonPruningMode.Pairwise"/>, during each comparisons the two trees are pruned so that
      they contain the subset of leaves that are in common between both of them.</param>
00161 /// <param name="maxThreadCount">The maximum number of threads to use for parallelised steps.</param>
00162 /// <param name="progress">An <see cref="IProgress{T}"/> for progress reporting.</param>
00163 /// <returns>A square <see langword="double"/>[,] matrix containing the requested distances between
      the trees.</returns>
00164         public static void RobinsonFouldsDistances(IReadOnlyList<TreeNode> trees, out double[,]
      RFDistances, out double[,] weightedRFDistances, TreeComparisonPruningMode pruningMode =
      TreeComparisonPruningMode.Pairwise, int maxThreadCount = -1, IProgress<double> progress = null)
00165         {
00166             RFDistances = new double[trees.Count, trees.Count];
00167             weightedRFDistances = new double[trees.Count, trees.Count];
00168
00169             FillDistanceMatrix(trees, comparePairwise:  pruningMode ==
      TreeComparisonPruningMode.Pairwise, RFDistances:  RFDistances, wRFDistances:  weightedRFDistances,
      maxThreadCount:  maxThreadCount, progress:  progress);
00170         }
00171
00172 /// <summary>
00173 /// Computes three distance matrices containing the unweighted and weighted Robinson-Foulds distances
      and the edge-length distances between each pair of trees in a list.  Much faster than computing the
      three distance matrices separately.
00174 /// </summary>
00175 /// <param name="trees">The list of trees that should be compared.</param>
00176 /// <param name="RFDistances">When this method returns, this variable will contain the computed
      Robinson-Foulds distances between the trees.</param>
00177 /// <param name="weightedRFDistances">When this method returns, this variable will contain the
      computed weighted Robinson-Foulds distances between the trees.</param>
00178 /// <param name="ELDistances">When this method returns, this variable will contain the computed
      edge-length distances between the trees.</param>
00179 /// <param name="pruningMode">If this is <see cref="TreeComparisonPruningMode.Global"/>, all trees are
      pruned so that they only contain the subset of leaves that are present in all trees.  If this is <see
      cref="TreeComparisonPruningMode.Pairwise"/>, during each comparisons the two trees are pruned so that
      they contain the subset of leaves that are in common between both of them.</param>
00180 /// <param name="maxThreadCount">The maximum number of threads to use for parallelised steps.</param>
00181 /// <param name="progress">An <see cref="IProgress{T}"/> for progress reporting.</param>
00182 /// <returns>A square <see langword="double"/>[,] matrix containing the requested distances between
      the trees.</returns>
00183         public static void TreeDistances(IReadOnlyList<TreeNode> trees, out double[,] RFDistances, out
      double[,] weightedRFDistances, out double[,] ELDistances, TreeComparisonPruningMode pruningMode =
      TreeComparisonPruningMode.Pairwise, int maxThreadCount = -1, IProgress<double> progress = null)
00184         {
00185             RFDistances = new double[trees.Count, trees.Count];
00186             weightedRFDistances = new double[trees.Count, trees.Count];
00187             ELDistances = new double[trees.Count, trees.Count];
00188
00189             FillDistanceMatrix(trees, comparePairwise:  pruningMode ==
      TreeComparisonPruningMode.Pairwise, RFDistances:  RFDistances, wRFDistances:  weightedRFDistances,
      ELDistances:  ELDistances, maxThreadCount:  maxThreadCount, progress:  progress);
00190         }
00191
00192
00193 /// <summary>
00194 /// Fills distance matrices with the Robinson-Foulds and weighted Robinson-Foulds distances between
      the trees.
00195 /// </summary>
00196 /// <param name="trees">The trees to be compared.</param>
00197 /// <param name="comparePairwise">If this is <see langword="false"/>, only leaves that are in common
      to all trees are used.  If this is <see langword="true" />, for each pair of trees, the leaves that
      are in common between them are used.</param>
00198 /// <param name="RFDistances">The matrix to be filled with Robinson-Foulds distances, or <see
      langword="null"/>.</param>
00199 /// <param name="wRFDistances">The matrix to be filled with weighted Robinson-Foulds distances, or
      <see langword="null"/>.</param>
00200 /// <param name="ELDistances">The matrix to be filled with edge-length distances, or <see
      langword="null"/>.</param>
00201 /// <param name="maxThreadCount">The maximum number of threads to use for parallelised steps.</param>
00202 /// <param name="progress">An <see cref="IProgress{T}"/> for progress reporting.</param>
00203 /// <exception cref="ArgumentException">Thrown if at least one of the trees has a tip without a
      name.</exception>
00204         private static void FillDistanceMatrix(IReadOnlyList<TreeNode> trees, bool comparePairwise =
      true, double[,] RFDistances = null, double[,] wRFDistances = null, double[,] ELDistances = null, int
      maxThreadCount = -1, IProgress<double> progress = null)
00205         {
```

```
00206             if (maxThreadCount <= 0)
00207             {
00208                 maxThreadCount = Environment.ProcessorCount;
00209             }
00210 
00211             Dictionary<string, int> leaves = new Dictionary<string, int>();
00212             List<int> splitCounts = new List<int>(trees.Count);
00213 
00214             List<double>[] splitLengths = new List<double>[trees.Count];
00215 
00216             int maxSplitCount = 0;
00217             int totalSplitCounts = 0;
00218 
00219             for (int i = 0; i < trees.Count; i++)
00220             {
00221                 TreeNode t = trees[i];
00222                 splitLengths[i] = new List<double>();
00223                 int splitCount = 0;
00224                 foreach (TreeNode node in t.GetChildrenRecursiveLazy())
00225                 {
00226                     if (node.Children.Count == 0)
00227                     {
00228                         if (!string.IsNullOrEmpty(node.Name))
00229                         {
00230                             leaves[node.Name] = 0;
00231                         }
00232                         else
00233                         {
00234                             throw new ArgumentException("At least one of the trees contains a tip
      without a Name!");
00235                         }
00236                     }
00237                     else
00238                     {
00239                         splitCount++;
00240                         splitLengths[i].Add(node.Length);
00241                     }
00242                 }
00243                 splitCounts.Add(splitCount);
00244                 maxSplitCount = Math.Max(splitCount, maxSplitCount);
00245                 totalSplitCounts += splitCount;
00246             }
00247 
00248             List<string> leafNames = new List<string>(leaves.Count);
00249 
00250             foreach (KeyValuePair<string, int> kvp in leaves)
00251             {
00252                 leafNames.Add(kvp.Key);
00253                 leaves[kvp.Key] = leafNames.Count - 1;
00254             }
00255 
00256             int[] splitOffsets = new int[splitCounts.Count];
00257 
00258             for (int i = 1; i < splitCounts.Count; i++)
00259             {
00260                 splitOffsets[i] = splitOffsets[i - 1] + splitCounts[i - 1];
00261             }
00262 
00263             double[][] leafLengths = new double[trees.Count][];
00264 
00265             // Each split consists of 1 bits for each leaf, determining whether the leaf is on one
      side of the split (0) or the other (1).
00266             int splitSize = (int)Math.Ceiling(leaves.Count / 8.0);
00267 
00268             // Array used to store whether a certain split has already been analysed or not.
00269             int foundsSize = (int)Math.Ceiling(maxSplitCount / 8.0);
00270 
00271             // Each tree mask is the same size as a single split.
00272             int totalMemoryNeeded = splitSize * (totalSplitCounts + trees.Count + maxThreadCount +
      leaves.Count) + foundsSize * maxThreadCount;
00273 
00274             IntPtr memoryIntPtr = Marshal.AllocHGlobal(totalMemoryNeeded);
00275 
00276             try
00277             {
00278                 IntPtr comparisonMasksIntPtr = memoryIntPtr;
00279                 IntPtr foundsIntPtr = IntPtr.Add(comparisonMasksIntPtr, splitSize * maxThreadCount);
00280                 IntPtr masksIntPtr = IntPtr.Add(foundsIntPtr, foundsSize * maxThreadCount);
00281                 IntPtr splitsIntPtr = IntPtr.Add(masksIntPtr, splitSize * trees.Count);
00282                 IntPtr leafSplitsIntPtr = IntPtr.Add(splitsIntPtr, splitSize * totalSplitCounts);
00283 
00284                 unsafe
00285                 {
00286                     Unsafe.InitBlockUnaligned((byte*)memoryIntPtr, 0, (uint)totalMemoryNeeded);
00287 
00288                     byte* masks = (byte*)masksIntPtr;
00289                     byte* splits = (byte*)splitsIntPtr;
```

```
00290                     byte* leafSplits = (byte*)leafSplitsIntPtr;
00291
00292                     for (int i = 0; i < leaves.Count; i++)
00293                     {
00294                         leafSplits[splitSize * i + i / 8] |= (byte)(0b1 << (i % 8));
00295                     }
00296
00297                     for (int i = 0; i < trees.Count; i++)
00298                     {
00299                         leafLengths[i] = new double[leaves.Count];
00300
00301                         byte* currMask = masks + i * splitSize;
00302                         byte* currSplits = splits + splitSize * splitOffsets[i];
00303
00304                         foreach (TreeNode node in trees[i].GetChildrenRecursiveLazy())
00305                         {
00306                             if (node.Children.Count == 0)
00307                             {
00308                                 int index = leaves[node.Name];
00309
00310                                 int byteIndex = index / 8;
00311                                 int bitIndex = index % 8;
00312
00313                                 currMask[byteIndex] |= (byte)(0b1 << bitIndex);
00314
00315                                 leafLengths[i][index] = node.Length;
00316                             }
00317                         }
00318
00319                         int currSplitIndex = 0;
00320
00321                         foreach (TreeNode node in trees[i].GetChildrenRecursiveLazy())
00322                         {
00323                             if (node.Children.Count > 0)
00324                             {
00325                                 foreach (TreeNode n in node.GetChildrenRecursiveLazy())
00326                                 {
00327                                     if (n.Children.Count == 0)
00328                                     {
00329                                         int index = leaves[n.Name];
00330
00331                                         int byteIndex = currSplitIndex * splitSize + index / 8;
00332                                         int bitIndex = index % 8;
00333
00334                                         currSplits[byteIndex] |= (byte)(0b1 << bitIndex);
00335                                     }
00336                                 }
00337
00338                                 currSplitIndex++;
00339                             }
00340                         }
00341                     }
00342
00343                     int progressCount = 0;
00344                     object progressLock = new object();
00345
00346                     ThreadIndexer threadIndexer = new ThreadIndexer(maxThreadCount);
00347
00348                     byte* comparisonMasks = (byte*)comparisonMasksIntPtr;
00349                     byte* founds = (byte*)foundsIntPtr;
00350
00351                     if (!comparePairwise)
00352                     {
00353                         Unsafe.InitBlockUnaligned(comparisonMasks, 0b11111111, (uint)splitSize);
00354
00355                         for (int i = 0; i < trees.Count; i++)
00356                         {
00357                             for (int j = 0; j < splitSize; j++)
00358                             {
00359                                 comparisonMasks[j] &= masks[i * splitSize + j];
00360                             }
00361                         }
00362                     }
00363
00364                     int comparisonCounts = trees.Count * (trees.Count - 1) / 2;
00365
00366                     (int, int) getIndices(int index)
00367                     {
00368                         int i = trees.Count - 2 - (int)Math.Floor(Math.Sqrt(-8 * index + 4 *
      trees.Count * (trees.Count - 1) - 7) / 2 - 0.5);
00369                         int j = index + i + 1 - trees.Count * (trees.Count - 1) / 2 + (trees.Count -
      i) * (trees.Count - i - 1) / 2;
00370
00371                         return (i, j);
00372                     };
00373
00374                     HashSet<SplitPointer>[] alreadyCheckedSplits1 = new
```

```
HashSet<SplitPointer>[maxThreadCount];
00375                     HashSet<SplitPointer>[] alreadyCheckedSplits2 = new
HashSet<SplitPointer>[maxThreadCount];
00376
00377                     Dictionary<SplitPointer, double>[] splitLengths1 = null;
00378                     Dictionary<SplitPointer, double>[] splitLengths2 = null;
00379
00380                     if (ELDistances != null)
00381                     {
00382                         splitLengths1 = new Dictionary<SplitPointer, double>[maxThreadCount];
00383                         splitLengths2 = new Dictionary<SplitPointer, double>[maxThreadCount];
00384                     }
00385
00386                     for (int i = 0; i < maxThreadCount; i++)
00387                     {
00388                         alreadyCheckedSplits1[i] = new HashSet<SplitPointer>(maxSplitCount);
00389                         alreadyCheckedSplits2[i] = new HashSet<SplitPointer>(maxSplitCount);
00390                         if (ELDistances != null)
00391                         {
00392                             splitLengths1[i] = new Dictionary<SplitPointer, double>(maxSplitCount);
00393                             splitLengths2[i] = new Dictionary<SplitPointer, double>(maxSplitCount);
00394                         }
00395                     }
00396
00397                     Parallel.For(0, comparisonCounts, new ParallelOptions() { MaxDegreeOfParallelism =
maxThreadCount }, index =>
00398                     {
00399                         int threadIndex = threadIndexer.GetIndex();
00400
00401                         (int i, int j) = getIndices(index);
00402
00403                         int dist = 0;
00404                         double wDist = 0;
00405
00406                         byte* currMask;
00407
00408                         if (!comparePairwise)
00409                         {
00410                             currMask = comparisonMasks;
00411                         }
00412                         else
00413                         {
00414                             currMask = comparisonMasks + threadIndex * splitSize;
00415
00416                             Unsafe.CopyBlockUnaligned(currMask, masks + i * splitSize,
(uint)splitSize);
00417
00418                             for (int k = 0; k < splitSize; k++)
00419                             {
00420                                 currMask[k] &= masks[j * splitSize + k];
00421                             }
00422                         }
00423
00424                         byte* currFounds = founds + foundsSize * threadIndex;
00425
00426                         Unsafe.InitBlockUnaligned(currFounds, 0, (uint)foundsSize);
00427
00428                         alreadyCheckedSplits1[threadIndex].Clear();
00429                         alreadyCheckedSplits2[threadIndex].Clear();
00430
00431                         if (RFDistances != null || wRFDistances != null)
00432                         {
00433                             for (int k = 0; k < splitCounts[i]; k++)
00434                             {
00435                                 if (CheckIfMoreThanOne(splits + (splitOffsets[i] + k) * splitSize,
currMask, splitSize))
00436                                 {
00437                                     if (RFDistances != null)
00438                                     {
00439                                         SplitPointer split = new SplitPointer(splits +
(splitOffsets[i] + k) * splitSize, currMask, splitSize);
00440
00441                                         if (alreadyCheckedSplits1[threadIndex].Add(split))
00442                                         {
00443                                             bool found = false;
00444
00445                                             for (int l = 0; l < splitCounts[j]; l++)
00446                                             {
00447                                                 if (CompareSplits(splits + (splitOffsets[i] + k) *
splitSize, splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize))
00448                                                 {
00449                                                     found = true;
00450                                                     currFounds[l / 8] |= (byte)(0b1 << (l % 8));
00451                                                     alreadyCheckedSplits2[threadIndex].Add(new
SplitPointer(splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize));
00452                                                     break;
00453                                                 }
```

```
00454                                           }
00455
00456                                           if (!found)
00457                                           {
00458                                               dist++;
00459                                               wDist += splitLengths[i][k];
00460                                           }
00461                                       }
00462                                       else if (wRFDistances != null)
00463                                       {
00464                                           bool found = false;
00465
00466                                           for (int l = 0; l < splitCounts[j]; l++)
00467                                           {
00468                                               if (CompareSplits(splits + (splitOffsets[i] + k) *
      splitSize, splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize))
00469                                               {
00470                                                   found = true;
00471                                                   currFounds[l / 8] |= (byte)(0b1 << (l % 8));
00472                                                   break;
00473                                               }
00474                                           }
00475
00476                                           if (!found)
00477                                           {
00478                                               wDist += splitLengths[i][k];
00479                                           }
00480                                       }
00481                                   }
00482                                   else if (wRFDistances != null)
00483                                   {
00484                                       bool found = false;
00485
00486                                       for (int l = 0; l < splitCounts[j]; l++)
00487                                       {
00488                                           if (CompareSplits(splits + (splitOffsets[i] + k) *
      splitSize, splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize))
00489                                           {
00490                                               found = true;
00491                                               currFounds[l / 8] |= (byte)(0b1 << (l % 8));
00492                                               break;
00493                                           }
00494                                       }
00495
00496                                       if (!found)
00497                                       {
00498                                           wDist += splitLengths[i][k];
00499                                       }
00500                                   }
00501                               }
00502                           }
00503
00504                           for (int l = 0; l < splitCounts[j]; l++)
00505                           {
00506                               if ((currFounds[l / 8] & (byte)(0b1 << (l % 8))) == 0)
00507                               {
00508                                   if (CheckIfMoreThanOne(splits + (splitOffsets[j] + l) * splitSize,
      currMask, splitSize))
00509                                   {
00510                                       if (RFDistances != null)
00511                                       {
00512                                           SplitPointer split = new SplitPointer(splits +
      (splitOffsets[j] + l) * splitSize, currMask, splitSize);
00513
00514                                           if (alreadyCheckedSplits2[threadIndex].Add(split))
00515                                           {
00516                                               bool found = false;
00517
00518                                               for (int k = 0; k < splitCounts[i]; k++)
00519                                               {
00520                                                   if (CompareSplits(splits + (splitOffsets[i] + k) *
      splitSize, splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize))
00521                                                   {
00522                                                       found = true;
00523                                                       break;
00524                                                   }
00525                                               }
00526
00527                                               if (!found)
00528                                               {
00529                                                   dist++;
00530                                                   wDist += splitLengths[j][l];
00531                                               }
00532                                           }
00533                                           else if (wRFDistances != null)
00534                                           {
00535                                               bool found = false;
```

```
00536
00537                                                   for (int k = 0; k < splitCounts[i]; k++)
00538                                                   {
00539                                                       if (CompareSplits(splits + (splitOffsets[i] + k) *
      splitSize, splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize))
00540                                                       {
00541                                                           found = true;
00542                                                           break;
00543                                                       }
00544                                                   }
00545
00546                                                   if (!found)
00547                                                   {
00548                                                       wDist += splitLengths[j][l];
00549                                                   }
00550                                               }
00551                                           }
00552                                           else if (wRFDistances != null)
00553                                           {
00554                                               bool found = false;
00555
00556                                               for (int k = 0; k < splitCounts[i]; k++)
00557                                               {
00558                                                   if (CompareSplits(splits + (splitOffsets[i] + k) *
      splitSize, splits + (splitOffsets[j] + l) * splitSize, currMask, splitSize))
00559                                                   {
00560                                                       found = true;
00561                                                       break;
00562                                                   }
00563                                               }
00564
00565                                               if (!found)
00566                                               {
00567                                                   wDist += splitLengths[j][l];
00568                                               }
00569                                           }
00570                                       }
00571                                   }
00572                               }
00573                           }
00574
00575                           if (ELDistances != null)
00576                           {
00577                               splitLengths1[threadIndex].Clear();
00578                               splitLengths2[threadIndex].Clear();
00579
00580                               for (int k = 0; k < splitCounts[i]; k++)
00581                               {
00582                                   SplitPointer split = new SplitPointer(splits + (splitOffsets[i] + k) *
      splitSize, currMask, splitSize);
00583
00584                                   if (splitLengths1[threadIndex].TryGetValue(split, out double val))
00585                                   {
00586                                       splitLengths1[threadIndex][split] = val + splitLengths[i][k];
00587                                   }
00588                                   else
00589                                   {
00590                                       splitLengths1[threadIndex][split] = splitLengths[i][k];
00591                                   }
00592                               }
00593
00594                               for (int l = 0; l < splitCounts[j]; l++)
00595                               {
00596                                   SplitPointer split = new SplitPointer(splits + (splitOffsets[j] + l) *
      splitSize, currMask, splitSize);
00597
00598                                   if (splitLengths2[threadIndex].TryGetValue(split, out double val))
00599                                   {
00600                                       splitLengths2[threadIndex][split] = val + splitLengths[j][l];
00601                                   }
00602                                   else
00603                                   {
00604                                       splitLengths2[threadIndex][split] = splitLengths[j][l];
00605                                   }
00606                               }
00607
00608                               for (int k = 0; k < leaves.Count; k++)
00609                               {
00610                                   if ((currMask[k / 8] & (byte)(0b1 << (k % 8))) != 0 &&
      !double.IsNaN(leafLengths[i][k]) && !double.IsNaN(leafLengths[j][k]))
00611                                   {
00612                                       SplitPointer split = new SplitPointer(leafSplits + k * splitSize,
      currMask, splitSize);
00613
00614                                       if (splitLengths1[threadIndex].TryGetValue(split, out double val))
00615                                       {
00616                                           splitLengths1[threadIndex][split] = val + leafLengths[i][k];
```

```
00617                                   }
00618                                   else
00619                                   {
00620                                       splitLengths1[threadIndex][split] = leafLengths[i][k];
00621                                   }
00622
00623                                   if (splitLengths2[threadIndex].TryGetValue(split, out double
      val2))
00624                                   {
00625                                       splitLengths2[threadIndex][split] = val2 + leafLengths[j][k];
00626                                   }
00627                                   else
00628                                   {
00629                                       splitLengths2[threadIndex][split] = leafLengths[j][k];
00630                                   }
00631                               }
00632                           }
00633
00634                           double totDist = 0;
00635
00636                           foreach (KeyValuePair<SplitPointer, double> kvp in
      splitLengths1[threadIndex])
00637                           {
00638                               if (!double.IsNaN(kvp.Value) &&
      splitLengths2[threadIndex].TryGetValue(kvp.Key, out double val) && !double.IsNaN(val))
00639                               {
00640                                   totDist += (kvp.Value - val) * (kvp.Value - val);
00641                               }
00642                           }
00643
00644                           ELDistances[i, j] = Math.Sqrt(totDist);
00645                           ELDistances[j, i] = ELDistances[i, j];
00646                       }
00647
00648                       if (RFDistances != null)
00649                       {
00650                           RFDistances[i, j] = dist;
00651                           RFDistances[j, i] = dist;
00652                       }
00653
00654                       if (wRFDistances != null)
00655                       {
00656                           wRFDistances[i, j] = wDist;
00657                           wRFDistances[j, i] = wDist;
00658                       }
00659
00660                       threadIndexer.ReturnIndex(threadIndex);
00661
00662                       if (progress != null)
00663                       {
00664                           lock (progressLock)
00665                           {
00666                               progressCount++;
00667
00668                               double currProg = (double)progressCount / comparisonCounts;
00669
00670                               _ = Task.Run(() => progress?.Report(currProg));
00671                           }
00672                       }
00673                   });
00674               }
00675           }
00676           finally
00677           {
00678               Marshal.FreeHGlobal(memoryIntPtr);
00679           }
00680       }
00681
00682 /// <summary>
00683 /// Compares two splits to determine whether they are actually the same split.  This is not the same
      as the splits being compatible.
00684 /// </summary>
00685 /// <param name="split1">A pointer to the first byte of the first split.</param>
00686 /// <param name="split2">A pointer to the first byte of the first split.</param>
00687 /// <param name="mask">A pointer to the first byte of the mask.</param>
00688 /// <param name="splitSize">The size in bytes of the splits and mask.</param>
00689 /// <returns></returns>
00690       [MethodImpl(MethodImplOptions.AggressiveInlining)]
00691       private unsafe static bool CompareSplits(byte* split1, byte* split2, byte* mask, int
      splitSize)
00692       {
00693           bool simpleEqual = true;
00694           bool xorEqual = true;
00695
00696           for (int i = 0; i < splitSize; i++)
00697           {
00698               byte split1El = (byte)(split1[i] & mask[i]);
```

```
00699                 byte split2El = (byte)(split2[i] & mask[i]);
00700
00701                 if (split1El != split2El)
00702                 {
00703                     simpleEqual = false;
00704                 }
00705
00706                 if ((split1El ^ split2El) != mask[i])
00707                 {
00708                     xorEqual = false;
00709                 }
00710
00711                 if (!simpleEqual && !xorEqual)
00712                 {
00713                     return false;
00714                 }
00715             }
00716
00717             return true;
00718         }
00719
00720 /// <summary>
00721 /// Checks whether a <paramref name="split"/> contains more than one taxa on each side, when masked
    with the specified <paramref name="mask"/>.
00722 /// </summary>
00723 /// <param name="split">A pointer to the first byte in the split.</param>
00724 /// <param name="mask">A pointer to the first byte in the mask.</param>
00725 /// <param name="splitSize">The size of the split and mask in bytes.</param>
00726 /// <returns><see langword="true" /> if the <paramref name="split"/> contains at least two taxa on
    each side.</returns>
00727         private unsafe static bool CheckIfMoreThanOne(byte* split, byte* mask, int splitSize)
00728         {
00729             bool step1 = false;
00730
00731             {
00732                 bool oneFound = false;
00733
00734                 for (int i = 0; i < splitSize; i++)
00735                 {
00736                     byte b = (byte)(split[i] & mask[i]);
00737
00738                     if (b != 0)
00739                     {
00740                         if ((b & (b - 1)) == 0)
00741                         {
00742                             if (oneFound)
00743                             {
00744                                 step1 = true;
00745                                 break;
00746                             }
00747                             else
00748                             {
00749                                 oneFound = true;
00750                             }
00751                         }
00752                         else
00753                         {
00754                             step1 = true;
00755                             break;
00756                         }
00757                     }
00758                 }
00759             }
00760
00761             if (!step1)
00762             {
00763                 return false;
00764             }
00765             else
00766             {
00767                 bool step2 = false;
00768
00769                 bool oneFound = false;
00770
00771                 for (int i = 0; i < splitSize; i++)
00772                 {
00773                     byte b = (byte)(~split[i] & mask[i]);
00774
00775                     if (b != 0)
00776                     {
00777                         if ((b & (b - 1)) == 0)
00778                         {
00779                             if (oneFound)
00780                             {
00781                                 step2 = true;
00782                                 break;
00783                             }
```

```
00784                               else
00785                               {
00786                                   oneFound = true;
00787                               }
00788                           }
00789                           else
00790                           {
00791                               step2 = true;
00792                               break;
00793                           }
00794                       }
00795                   }
00796
00797                   return step2;
00798               }
00799           }
00800
00801 /// <summary>
00802 /// A wrapper for a pointer to a split in unmanaged memory, for use in <see cref="HashSet{T}"/>s.
00803 /// </summary>
00804           internal unsafe struct SplitPointer :  IEquatable<SplitPointer>
00805           {
00806               public byte* Split;
00807               public byte* Mask;
00808               public int Length;
00809               private int hashCode;
00810
00811               public SplitPointer(byte* split, byte* mask, int length)
00812               {
00813                   this.Split = split;
00814                   this.Mask = mask;
00815                   this.Length = length;
00816
00817                   hashCode = 0;
00818                   int hashCode2 = 0;
00819
00820                   for (int i = 0; i < length; i++)
00821                   {
00822                       hashCode ^= (split[i] & mask[i]) « ((i % 9) * 3);
00823
00824                       hashCode2 ^= ((~split[i]) & mask[i]) « ((i % 9) * 3);
00825                   }
00826
00827                   hashCode |= hashCode2;
00828               }
00829
00830               public bool Equals(SplitPointer other)
00831               {
00832 #if DEBUG
00833                   if (this.Mask != other.Mask || this.Length != other.Length)
00834                   {
00835                       throw new Exception("Invalid split comparison:  mask or length are different!");
00836                   }
00837 #endif
00838
00839                   if (this.Split == other.Split)
00840                   {
00841                       return true;
00842                   }
00843
00844                   return CompareSplits(this.Split, other.Split, this.Mask, this.Length);
00845               }
00846
00847               public bool NotEquals(SplitPointer other)
00848               {
00849 #if DEBUG
00850                   if (this.Mask != other.Mask || this.Length != other.Length)
00851                   {
00852                       throw new Exception("Invalid split comparison:  mask or length are different!");
00853                   }
00854 #endif
00855
00856                   if (this.Split == other.Split)
00857                   {
00858                       return false;
00859                   }
00860
00861                   return !CompareSplits(this.Split, other.Split, this.Mask, this.Length);
00862               }
00863
00864               public static bool operator ==(SplitPointer left, SplitPointer right)
00865               {
00866                   return left.Equals(right);
00867               }
00868
00869               public static bool operator !=(SplitPointer left, SplitPointer right)
00870               {
```

```
00871                 return left.NotEquals(right);
00872             }
00873 
00874             public override bool Equals([NotNullWhen(true)] object obj)
00875             {
00876                 if (obj is SplitPointer spt)
00877                 {
00878                     return this.Equals(spt);
00879                 }
00880                 else
00881                 {
00882                     return false;
00883                 }
00884             }
00885 
00886             public override int GetHashCode()
00887             {
00888                 return this.hashCode;
00889             }
00890         }
00891 
00892 /// <summary>
00893 /// Assigns a unique index to each thread.
00894 /// </summary>
00895         internal class ThreadIndexer
00896         {
00897 /// <summary>
00898 /// The maximum number of concurrent threads.
00899 /// </summary>
00900             public int MaxThreads { get; }
00901 
00902 /// <summary>
00903 /// Contains the available indices.
00904 /// </summary>
00905             private ConcurrentBag<int> threadIndices;
00906 
00907 /// <summary>
00908 /// Ensures that no more than the maximum number of indices are given out.
00909 /// </summary>
00910             private SemaphoreSlim semaphore;
00911 
00912 /// <summary>
00913 /// Creates a new <see cref="ThreadIndexer"/> instance.
00914 /// </summary>
00915 /// <param name="maxThreads">The maximum number of threads.</param>
00916             public ThreadIndexer(int maxThreads)
00917             {
00918                 this.MaxThreads = maxThreads;
00919 
00920                 threadIndices = new ConcurrentBag<int>(Enumerable.Range(0, maxThreads));
00921                 semaphore = new SemaphoreSlim(maxThreads, maxThreads);
00922             }
00923 
00924 /// <summary>
00925 /// Gets one of the available indices.  No guarantee about the order.
00926 /// </summary>
00927 /// <returns>A unique index that is not in use by any other thread.</returns>
00928 /// <exception cref="Exception">Thrown if an error occurs while retrieving the thread index.  This
      should never happen.</exception>
00929             public int GetIndex()
00930             {
00931                 semaphore.Wait();
00932                 if (threadIndices.TryTake(out int tbr))
00933                 {
00934                     return tbr;
00935                 }
00936                 else
00937                 {
00938                     throw new Exception("An error occurred while trying to access the thread index!");
00939                 }
00940             }
00941 
00942 /// <summary>
00943 /// Gives back an index so that it can be used by other threads.
00944 /// </summary>
00945 /// <param name="index">The index that will become available again.</param>
00946             public void ReturnIndex(int index)
00947             {
00948                 threadIndices.Add(index);
00949                 semaphore.Release();
00950             }
00951         }
00952 
00953 
00954     }
00955 }
```

8.28 TreeNode.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.Linq;
00005 using PhyloTree.Extensions;
00006 using PhyloTree.Formats;
00007 using System.Diagnostics.CodeAnalysis;
00008 using System.Security;
00009 using System.Threading;
00010
00011 [assembly:  SuppressMessage("Globalization", "CA1303")]
00012
00013 /// <summary>
00014 /// Contains classes and methods to read, write and manipulate phylogenetic trees.
00015 /// </summary>
00016 namespace PhyloTree
00017 {
00018 /// <summary>
00019 /// Represents a split induced by a branch in a tree.
00020 /// </summary>
00021     internal class Split
00022     {
00023 /// <summary>
00024 /// The name of the split.  It consists of a series of comma-separated tip names, optionally followed
      by a vertical bar | and by another series of comma-separated tip names.
00025 /// E.g.  "A,B|C,D"
00026 /// </summary>
00027         public string Name { get; }
00028
00029 /// <summary>
00030 /// The length of the split (representing either the age of the node that induced it or the length of
      the branch).
00031 /// </summary>
00032         public double Length { get; }
00033
00034 /// <summary>
00035 /// The support value for the split.
00036 /// </summary>
00037         public double Support { get; }
00038
00039 /// <summary>
00040 /// Determines whether the <see cref="Length"/> property contains the age of the node that induced the
      split or the length of the branch.
00041 /// </summary>
00042         public LengthTypes LengthType { get; }
00043
00044
00045 /// <summary>
00046 /// Determines whether the <see cref="Length"/> of the split contains the age of the node that induced
      the split or the length of the branch.
00047 /// </summary>
00048         public enum LengthTypes
00049         {
00050 /// <summary>
00051 /// The <see cref="Split.Length"/> of the split contains the length of the branch that induced it.
00052 /// </summary>
00053             Length,
00054 /// <summary>
00055 /// The <see cref="Split.Length"/> of the split contains the age of the node that induced it.
00056 /// </summary>
00057             Age
00058         }
00059
00060 /// <summary>
00061 /// Creates a <see cref="Split"/> object.
00062 /// </summary>
00063 /// <param name="name">The name of the split.</param>
00064 /// <param name="length">The length of the split.</param>
00065 /// <param name="lengthType">Determines whether <paramref name="length"/> contains the age of the node
      that induced the split or the length of the branch.</param>
00066 /// <param name="support">The support value of the split.</param>
00067         public Split(string name, double length, LengthTypes lengthType, double support)
00068         {
00069             this.Name = name;
00070             this.Length = length;
00071             this.Support = support;
00072             this.LengthType = lengthType;
00073         }
00074
00075 /// <summary>
00076 /// Determines whether two <see cref="Split"/>s are compatible with each other.
00077 /// </summary>
00078 /// <param name="s1">The first <see cref="Split"/> to compare.</param>
00079 /// <param name="s2">The second <see cref="Split"/> to compare.</param>
00080 /// <returns><c>true</c> if the two <see cref="Split"/>s are compatible with each other, <c>false</c>
```

```
    if they are not.</returns>
00081         public static bool AreCompatible(Split s1, Split s2)
00082         {
00083             if (!s1.Name.Contains("|", StringComparison.OrdinalIgnoreCase) && !s2.Name.Contains("|",
    StringComparison.OrdinalIgnoreCase))
00084             {
00085                 string[] leaves1 = s1.Name.Split(',');
00086                 string[] leaves2 = s2.Name.Split(',');
00087 
00088                 return !leaves1.ContainsAny(leaves2) || leaves1.ContainsAll(leaves2) ||
    leaves2.ContainsAll(leaves1);
00089             }
00090             else
00091             {
00092                 string[][] leaves1 = (from el in s1.Name.Split('|') select el.Split(',')).ToArray();
00093                 string[][] leaves2 = (from el in s2.Name.Split('|') select el.Split(',')).ToArray();
00094 
00095                 if (leaves1.Length == 2 && leaves2.Length == 2)
00096                 {
00097                     return !leaves1[0].Intersect(leaves2[0]).Any() ||
    !leaves1[0].Intersect(leaves2[1]).Any() || !leaves1[1].Intersect(leaves2[0]).Any() ||
    !leaves1[1].Intersect(leaves2[1]).Any();
00098                 }
00099                 else if (leaves1.Length == 1 && leaves2.Length == 2)
00100                 {
00101                     return (!leaves1[0].ContainsAny(leaves2[0]) || leaves1[0].ContainsAll(leaves2[0])
    || leaves2[0].ContainsAll(leaves1[0])) && (!leaves1[0].ContainsAny(leaves2[1]) ||
    leaves1[0].ContainsAll(leaves2[1]) || leaves2[1].ContainsAll(leaves1[0]));
00102                 }
00103                 else if (leaves1.Length == 2 && leaves2.Length == 1)
00104                 {
00105                     return (!leaves2[0].ContainsAny(leaves1[0]) || leaves2[0].ContainsAll(leaves1[0])
    || leaves1[0].ContainsAll(leaves2[0])) && (!leaves2[0].ContainsAny(leaves1[1]) ||
    leaves2[0].ContainsAll(leaves1[1]) || leaves1[1].ContainsAll(leaves2[0]));
00106                 }
00107                 else
00108                 {
00109                     throw new NotImplementedException();
00110                 }
00111             }
00112         }
00113 
00114 /// <summary>
00115 /// Gets the children on the left of the split.
00116 /// </summary>
00117 /// <returns>The children on the left of the split.</returns>
00118         public IEnumerable<string> GetChildrenLeft()
00119         {
00120             if (this.Name.Contains('|', StringComparison.OrdinalIgnoreCase))
00121             {
00122                 return this.Name.Split('|')[0].Split(',');
00123             }
00124             else
00125             {
00126                 return this.Name.Split(',');
00127             }
00128         }
00129 
00130 /// <summary>
00131 /// Gets the children on the right of the split.
00132 /// </summary>
00133 /// <returns>The children on the right of the split.</returns>
00134         public IEnumerable<string> GetChildrenRight()
00135         {
00136             if (this.Name.Contains('|', StringComparison.OrdinalIgnoreCase))
00137             {
00138                 return this.Name.Split('|')[1].Split(',');
00139             }
00140             else
00141             {
00142                 return this.Name.Split(',');
00143             }
00144         }
00145 
00146 /// <summary>
00147 /// Determines whether multiple <see cref="Split"/>s are compatible with each other.
00148 /// </summary>
00149 /// <param name="splits">The <see cref="Split"/>s to compare.</param>
00150 /// <returns><c>true</c> if all the <see cref="Split"/>s are compatible with each other, <c>false</c>
    if the are not.</returns>
00151         public bool IsCompatible(IEnumerable<Split> splits)
00152         {
00153             foreach (Split split in splits)
00154             {
00155                 if (!AreCompatible(split, this))
00156                 {
00157                     return false;
```

```
00158                 }
00159             }
00160             return true;
00161         }
00162 
00163 
00164 /// <summary>
00165 /// Builds a rooted or unrooted tree starting from a collection of compatible <see cref="Split"/>s.
00166 /// </summary>
00167 /// <param name="splits">The <see cref="Split"/>s to use in building the tree.  This method assumes
      that they are all compatible with each other.</param>
00168 /// <param name="rooted">Whether to build a rooted or an unrooted tree.</param>
00169 /// <returns>A <see cref="TreeNode"/> containing the tree represented by the <see
      cref="Split"/>s.</returns>
00170         public static TreeNode BuildTree(IEnumerable<Split> splits, bool rooted)
00171         {
00172             List<TreeNode> nodes = new List<TreeNode>();
00173 
00174             List<Split> allSplits = splits.ToList();
00175 
00176             HashSet<string> addedTips = new HashSet<string>();
00177 
00178             string[][] splitChildrenLeft = new string[allSplits.Count][];
00179             string[][] splitChildrenRight = new string[allSplits.Count][];
00180 
00181             bool clockLike = true;
00182 
00183             for (int i = 0; i < allSplits.Count; i++)
00184             {
00185                 if (allSplits[i].LengthType != Split.LengthTypes.Age)
00186                 {
00187                     clockLike = false;
00188                 }
00189 
00190                 splitChildrenLeft[i] = allSplits[i].GetChildrenLeft().ToArray();
00191                 splitChildrenRight[i] = allSplits[i].GetChildrenRight().ToArray();
00192                 if (splitChildrenLeft[i].Length == 1)
00193                 {
00194                     if (addedTips.Add(splitChildrenLeft[i][0]))
00195                     {
00196                         TreeNode child = new TreeNode(null) { Name = splitChildrenLeft[i][0], Support
      = allSplits[i].Support, Length = allSplits[i].Length };
00197                         nodes.Add(child);
00198                     }
00199                 }
00200                 if (splitChildrenRight[i].Length == 1)
00201                 {
00202                     if (addedTips.Add(splitChildrenRight[i][0]))
00203                     {
00204                         TreeNode child = new TreeNode(null) { Name = splitChildrenRight[i][0], Support
      = allSplits[i].Support, Length = allSplits[i].Length };
00205                         nodes.Add(child);
00206                     }
00207                 }
00208 
00209                 if (!rooted && splitChildrenLeft[i].Length > splitChildrenRight[i].Length)
00210                 {
00211                     string[] temp = splitChildrenLeft[i];
00212                     splitChildrenLeft[i] = splitChildrenRight[i];
00213                     splitChildrenRight[i] = temp;
00214                 }
00215             }
00216 
00217             int maxCoalescence = nodes.Count;
00218 
00219             for (int i = 2; i <= maxCoalescence; i++)
00220             {
00221                 Coalesce(splitChildrenLeft, nodes, allSplits, i);
00222             }
00223 
00224 
00225             if (clockLike)
00226             {
00227                 nodes = nodes[0].GetChildrenRecursive();
00228 
00229                 for (int i = nodes.Count - 1; i >= 0; i--)
00230                 {
00231                     if (nodes[i].Parent != null)
00232                     {
00233                         nodes[i].Length = nodes[i].Parent.Length - nodes[i].Length;
00234                     }
00235                     else
00236                     {
00237                         nodes[i].Length = double.NaN;
00238                     }
00239                 }
00240             }
```

```
00241
00242             if (nodes[0].Children.Count < 3 && !rooted)
00243             {
00244                 nodes[0] = nodes[0].GetUnrootedTree();
00245             }
00246
00247             return nodes[0];
00248         }
00249
00250 /// <summary>
00251 /// Coalesces nodes as commanded by the supplies list of <see cref="Split"/>s.
00252 /// </summary>
00253 /// <param name="splitChildrenLeft">The tips specified on the left side of each split.</param>
00254 /// <param name="nodes">The list of <see cref="TreeNode"/>s that will be coalesced.</param>
00255 /// <param name="allSplits">The list of <see cref="Split"/>s describing the tree.</param>
00256 /// <param name="level">The level at which to coalesce.  This method should be invoked multiple times,
      with <paramref name="level"/> increasing from 2 up to the number of tips in the tree
      (inclusive).</param>
00257         private static void Coalesce(string[][] splitChildrenLeft, List<TreeNode> nodes, List<Split>
      allSplits, int level)
00258         {
00259             for (int i = 0; i < allSplits.Count; i++)
00260             {
00261                 if (splitChildrenLeft[i].Length == level)
00262                 {
00263                     List<TreeNode> currChildren = new List<TreeNode>();
00264                     for (int j = 0; j < nodes.Count; j++)
00265                     {
00266                         if (splitChildrenLeft[i].ContainsAll(nodes[j].GetLeafNames()))
00267                         {
00268                             currChildren.Add(nodes[j]);
00269                         }
00270                     }
00271                     if (currChildren.Count > 1)
00272                     {
00273                         TreeNode parent = new TreeNode(null) { Support = allSplits[i].Support, Length
      = allSplits[i].Length };
00274                         parent.Children.AddRange(currChildren);
00275
00276                         foreach (TreeNode node in currChildren)
00277                         {
00278                             nodes.Remove(node);
00279                             node.Parent = parent;
00280                         }
00281
00282                         nodes.Add(parent);
00283                     }
00284                     else if (currChildren.Count > 0)
00285                     {
00286                         currChildren[0].Support = Math.Min(currChildren[0].Support,
      allSplits[i].Support);
00287
00288                         if (allSplits[i].LengthType == LengthTypes.Length)
00289                         {
00290                             currChildren[0].Length = currChildren[0].Length + allSplits[i].Length;
00291                         }
00292                         else
00293                         {
00294                             currChildren[0].Length = allSplits[i].Length;
00295                         }
00296                     }
00297                 }
00298             }
00299         }
00300     }
00301
00302 /// <summary>
00303 /// Represents a node in a tree (or a whole tree).
00304 /// </summary>
00305     [Serializable]
00306     public partial class TreeNode
00307     {
00308 /// <summary>
00309 /// The parent node of this node.  This will be <c>null</c> for the root node.
00310 /// </summary>
00311         public TreeNode Parent { get; set; }
00312
00313 /// <summary>
00314 /// The child nodes of this node.  This will be empty (but initialised) for leaf nodes.
00315 /// </summary>
00316         public List<TreeNode> Children { get; }
00317
00318 /// <summary>
00319 /// The attributes of this node.  Attributes <see cref="Name"/>, <see cref="Length"/> and <see
      cref="Support"/> are always included.  See the respective properties for default values.
00320 /// </summary>
00321         public AttributeDictionary Attributes { get; } = new AttributeDictionary();
```

```
00322
00323 /// <summary>
00324 /// The length of the branch leading to this node.  This is <c>double.NaN</c> for branches whose
      length is not specified (e.g.  the root node).
00325 /// </summary>
00326         public double Length
00327         {
00328             get
00329             {
00330                 return Attributes.Length;
00331             }
00332             set
00333             {
00334                 Attributes.Length = value;
00335             }
00336         }
00337
00338 /// <summary>
00339 /// The support value of this node.  This is <c>double.NaN</c> for branches whose support is not
      specified.  The interpretation of the support value depends on how the tree was built.
00340 /// </summary>
00341         public double Support
00342         {
00343             get
00344             {
00345                 return Attributes.Support;
00346             }
00347             set
00348             {
00349                 Attributes.Support = value;
00350             }
00351         }
00352
00353 /// <summary>
00354 /// The name of this node (e.g.  the species name for leaf nodes).  Default is <c>""</c>.
00355 /// </summary>
00356         public string Name
00357         {
00358             get
00359             {
00360                 return Attributes.Name;
00361             }
00362             set
00363             {
00364                 Attributes.Name = value;
00365             }
00366         }
00367
00368
00369 /// <summary>
00370 /// A univocal identifier for the node.
00371 /// </summary>
00372         public string Id { get; private set; }
00373
00374 /// <summary>
00375 /// Creates a new <see cref="TreeNode"/> object.
00376 /// </summary>
00377 /// <param name="parent">The parent node of this node.  For the root node, this should be
      <c>null</c>.</param>
00378         public TreeNode(TreeNode parent)
00379         {
00380             Parent = parent;
00381             Id = Guid.NewGuid().ToString();
00382             Children = new List<TreeNode>();
00383         }
00384
00385 /// <summary>
00386 /// Checks whether the node belongs to a rooted tree.
00387 /// </summary>
00388 /// <returns><c>true</c> if the node belongs to a rooted tree, <c>false</c> otherwise.</returns>
00389         public bool IsRooted()
00390         {
00391             return this.GetRootNode().Children.Count < 3;
00392         }
00393
00394 /// <summary>
00395 /// Get an unrooted version of the tree.
00396 /// </summary>
00397 /// <returns>A <see cref="TreeNode"/> containing the unrooted tree, having at least 3
      children.</returns>
00398         public TreeNode GetUnrootedTree()
00399         {
00400             //A tree is unrooted if the root node has at least 3 children
00401             if (this.Children.Count >= 3)
00402             {
00403                 //If the tree is already unrooted, just return a clone
00404                 return this.Clone();
```

```
00405             }
00406             else
00407             {
00408                 //At this point, assume that the root node has 2 children
00409 
00410                 //If the second child of the root node is not a leaf node (i.e.  it has at least 2 children), we can take the first child of the root node and graft it onto the second child; the second child will now have 3 children and will be the root node of the unrooted tree
00411                 if (this.Children[1].Children.Count >= 2)
00412                 {
00413                     TreeNode child1 = this.Children[1].Clone();
00414                     TreeNode child0 = this.Children[0].Clone();
00415                     child0.Parent = child1;
00416                     child0.Length += child1.Length;
00417                     child1.Children.Add(child0);
00418                     child1.Parent = null;
00419                     child1.Length = double.NaN;
00420                     child1.Name = this.Name;
00421                     return child1;
00422                 }
00423                 else
00424                 {
00425                     //If the second child of the root node is a leaf node, then the first child must not be a leaf node; thus we do the same as before, but swapping the two children
00426                     TreeNode child0 = this.Children[1].Clone();
00427                     TreeNode child1 = this.Children[0].Clone();
00428                     child0.Parent = child1;
00429                     child0.Length += child1.Length;
00430                     child1.Children.Add(child0);
00431                     child1.Parent = null;
00432                     child1.Length = double.NaN;
00433                     child1.Name = this.Name;
00434                     return child1;
00435                 }
00436             }
00437         }
00438 
00439 /// <summary>
00440 /// Get a version of the tree that is rooted at the specified point of the branch leading to the <paramref name="outgroup"/>.
00441 /// </summary>
00442 /// <param name="outgroup">The outgroup to be used when rooting the tree.</param>
00443 /// <param name="position">The (relative) position on the branch connecting the outgroup to the rest of the tree on which to place the root.</param>
00444 /// <returns>A <see cref="TreeNode"/> containing the rooted tree.</returns>
00445         public TreeNode GetRootedTree(TreeNode outgroup, double position = 0.5)
00446         {
00447             if (outgroup != null && outgroup.Parent != null)
00448             {
00449                 TreeNode subject;
00450 
00451                 if (this.Children.Count < 3)
00452                 {
00453                     subject = this.GetUnrootedTree();
00454                 }
00455                 else
00456                 {
00457                     subject = this.Clone();
00458                 }
00459 
00460                 outgroup = subject.GetNodeFromId(outgroup.Id);
00461 
00462                 if (outgroup != null && outgroup.Parent != null)
00463                 {
00464 
00465                     position = outgroup.Length * position;
00466 
00467                     TreeNode tbr = new TreeNode(null);
00468 
00469                     TreeNode outGroup2 = outgroup.Clone();
00470                     outGroup2.Parent = tbr;
00471                     outGroup2.Length = position;
00472                     tbr.Children.Add(outGroup2);
00473 
00474                     TreeNode otherChild = outgroup.Parent.Invert(outgroup);
00475                     otherChild.Parent = tbr;
00476                     otherChild.Length = outgroup.Length - position;
00477                     tbr.Children.Add(otherChild);
00478 
00479                     foreach (KeyValuePair<string, object> attribute in this.Attributes)
00480                     {
00481                         tbr.Attributes[attribute.Key] = attribute.Value;
00482                     }
00483 
00484                     tbr.Name = this.Name;
00485 
00486                     return tbr;
```

```
00487                 }
00488                 else
00489                 {
00490                     return this.Clone();
00491                 }
00492             }
00493             else
00494             {
00495                 return this.Clone();
00496             }
00497         }
00498 
00499         internal TreeNode Invert(TreeNode ignoredChild)
00500         {
00501             if (this.Children.Count < 2)
00502             {
00503                 return this.Clone();
00504             }
00505             else
00506             {
00507                 TreeNode nd = new TreeNode(null);
00508                 foreach (TreeNode chd in this.Children)
00509                 {
00510                     if (chd != ignoredChild)
00511                     {
00512                         TreeNode chd2 = chd.Clone();
00513                         chd2.Parent = nd;
00514                         nd.Children.Add(chd2);
00515                     }
00516                 }
00517 
00518                 if (this.Parent != null)
00519                 {
00520                     TreeNode prnt = this.Parent.Invert(this);
00521                     prnt.Parent = nd;
00522 
00523                     foreach (KeyValuePair<string, object> attribute in this.Attributes)
00524                     {
00525                         prnt.Attributes[attribute.Key] = attribute.Value;
00526                     }
00527 
00528                     prnt.Name = this.Name;
00529                     prnt.Length = this.Length;
00530                     prnt.Support = this.Support;
00531 
00532                     nd.Children.Add(prnt);
00533                 }
00534                 else
00535                 {
00536                     foreach (KeyValuePair<string, object> attribute in this.Attributes)
00537                     {
00538                         nd.Attributes[attribute.Key] = attribute.Value;
00539                     }
00540 
00541                     nd.Name = this.Name;
00542                     nd.Length = this.Length;
00543                     nd.Support = this.Support;
00544                 }
00545 
00546                 return nd;
00547             }
00548         }
00549 
00550 /// <summary>
00551 /// Recursively clone a <see cref="TreeNode"/> object.
00552 /// </summary>
00553 /// <returns>The cloned <see cref="TreeNode"/></returns>
00554         public TreeNode Clone()
00555         {
00556             TreeNode nd = new TreeNode(this.Parent)
00557             {
00558                 Id = this.Id,
00559                 Name = this.Name
00560             };
00561 
00562             foreach (TreeNode nd2 in this.Children)
00563             {
00564                 TreeNode nd22 = nd2.Clone();
00565                 nd22.Parent = nd;
00566                 nd.Children.Add(nd22);
00567             }
00568 
00569             nd.Length = this.Length;
00570             nd.Support = this.Support;
00571 
00572             foreach (KeyValuePair<string, object> kvp in this.Attributes)
00573             {
```

```
00574                 nd.Attributes[kvp.Key] = kvp.Value;
00575             }
00576 
00577             return nd;
00578         }
00579 
00580 /// <summary>
00581 /// Recursively get all the nodes that descend from this node.
00582 /// </summary>
00583 /// <returns>A <see cref="List{T}"/> of <see cref="TreeNode"/> objects, containing the nodes that
      descend from this node.</returns>
00584         public List<TreeNode> GetChildrenRecursive()
00585         {
00586             List<TreeNode> tbr = new List<TreeNode>
00587             {
00588                 this
00589             };
00590 
00591             for (int i = 0; i < this.Children.Count; i++)
00592             {
00593                 tbr.AddRange(this.Children[i].GetChildrenRecursive());
00594             }
00595             return tbr;
00596         }
00597 
00598 /// <summary>
00599 /// Lazily recursively get all the nodes that descend from this node.
00600 /// </summary>
00601 /// <returns>An <see cref="IEnumerable{T}"/> of <see cref="TreeNode"/> objects, containing the nodes
      that descend from this node.</returns>
00602         public IEnumerable<TreeNode> GetChildrenRecursiveLazy()
00603         {
00604             yield return this;
00605 
00606             for (int i = 0; i < this.Children.Count; i++)
00607             {
00608                 foreach (TreeNode t in this.Children[i].GetChildrenRecursiveLazy())
00609                 {
00610                     yield return t;
00611                 }
00612             }
00613         }
00614 
00615 /// <summary>
00616 /// Get all the leaves that descend (directly or indirectly) from this node.
00617 /// </summary>
00618 /// <returns>A <see cref="List{T}"/> of <see cref="TreeNode"/> objects, containing the leaves that
      descend from this node.</returns>
00619         public List<TreeNode> GetLeaves()
00620         {
00621             List<TreeNode> tbr = new List<TreeNode>();
00622 
00623             if (this.Children.Count == 0)
00624             {
00625                 tbr.Add(this);
00626             }
00627 
00628             for (int i = 0; i < this.Children.Count; i++)
00629             {
00630                 tbr.AddRange(this.Children[i].GetLeaves());
00631             }
00632             return tbr;
00633         }
00634 
00635 /// <summary>
00636 /// Get the names of all the leaves that descend (directly or indirectly) from this node.
00637 /// </summary>
00638 /// <returns>A <see cref="List{T}"/> of <see cref="string"/>s, containing the names of the leaves that
      descend from this node.</returns>
00639         public List<string> GetLeafNames()
00640         {
00641             List<string> tbr = new List<string>();
00642 
00643             if (this.Children.Count == 0 && !string.IsNullOrEmpty(this.Name))
00644             {
00645                 tbr.Add(this.Name);
00646             }
00647 
00648             for (int i = 0; i < this.Children.Count; i++)
00649             {
00650                 tbr.AddRange(this.Children[i].GetLeafNames());
00651             }
00652             return tbr;
00653         }
00654 
00655 /// <summary>
00656 /// Get the names of all the named nodes that descend (directly or indirectly) from this node.
```

```
00657 /// </summary>
00658 /// <returns>A <see cref="List{T}"/> of <see cref="string"/>s, containing the names of the named nodes
      that descend from this node.</returns>
00659         public List<string> GetNodeNames()
00660         {
00661             List<string> tbr = new List<string>();
00662 
00663             if (!string.IsNullOrEmpty(this.Name))
00664             {
00665                 tbr.Add(this.Name);
00666             }
00667 
00668             for (int i = 0; i < this.Children.Count; i++)
00669             {
00670                 tbr.AddRange(this.Children[i].GetNodeNames());
00671             }
00672             return tbr;
00673         }
00674 
00675 /// <summary>
00676 /// Get the child node with the specified name.
00677 /// </summary>
00678 /// <param name="nodeName">The name of the node to search.</param>
00679 /// <returns>The <see cref="TreeNode"/> object with the specified name, or <c>null</c> if no node with
      such name exists.</returns>
00680         public TreeNode GetNodeFromName(string nodeName)
00681         {
00682             if (this.Name == nodeName)
00683             {
00684                 return this;
00685             }
00686 
00687             for (int i = 0; i < this.Children.Count; i++)
00688             {
00689                 TreeNode item = this.Children[i].GetNodeFromName(nodeName);
00690                 if (item != null)
00691                 {
00692                     return item;
00693                 }
00694             }
00695 
00696             return null;
00697         }
00698 
00699 /// <summary>
00700 /// Get the child node with the specified Id.
00701 /// </summary>
00702 /// <param name="nodeId">The Id of the node to search.</param>
00703 /// <returns>The <see cref="TreeNode"/> object with the specified Id, or <c>null</c> if no node with
      such Id exists.</returns>
00704         public TreeNode GetNodeFromId(string nodeId)
00705         {
00706             if (this.Id == nodeId)
00707             {
00708                 return this;
00709             }
00710 
00711             for (int i = 0; i < this.Children.Count; i++)
00712             {
00713                 TreeNode item = this.Children[i].GetNodeFromId(nodeId);
00714                 if (item != null)
00715                 {
00716                     return item;
00717                 }
00718             }
00719 
00720             return null;
00721         }
00722 
00723 /// <summary>
00724 /// Get the sum of the branch lengths from this node up to the root.
00725 /// </summary>
00726 /// <returns>The sum of the branch lengths from this node up to the root.</returns>
00727         public double UpstreamLength()
00728         {
00729             double tbr = 0;
00730 
00731             TreeNode nd = this;
00732 
00733             while (nd.Parent != null)
00734             {
00735                 tbr += nd.Length;
00736                 nd = nd.Parent;
00737             }
00738 
00739             return tbr;
00740         }
```

```
00741
00742 /// <summary>
00743 /// Get the sum of the branch lengths from this node down to the leaves of the tree.  If the tree is
      not clock-like, the length of the longest path is returned.
00744 /// </summary>
00745 /// <returns>The sum of the branch lengths from this node down to the leaves of the tree.  If the tree
      is not clock-like, the length of the longest path is returned.</returns>
00746         public double LongestDownstreamLength()
00747         {
00748             if (this.Children.Count == 0)
00749             {
00750                 return 0;
00751             }
00752             else
00753             {
00754                 double maxLen = 0;
00755                 for (int i = 0; i < this.Children.Count; i++)
00756                 {
00757                     double chLen = this.Children[i].LongestDownstreamLength() +
      this.Children[i].Length;
00758
00759                     maxLen = Math.Max(maxLen, chLen);
00760                 }
00761
00762                 return maxLen;
00763             }
00764         }
00765
00766 /// <summary>
00767 /// Get the sum of the branch lengths from this node down to the leaves of the tree.  If the tree is
      not clock-like, the length of the shortest path is returned.
00768 /// </summary>
00769 /// <returns>The sum of the branch lengths from this node down to the leaves of the tree.  If the tree
      is not clock-like, the length of the shortest path is returned.</returns>
00770         public double ShortestDownstreamLength()
00771         {
00772             if (this.Children.Count == 0)
00773             {
00774                 return 0;
00775             }
00776             else
00777             {
00778                 double minLen = double.MaxValue;
00779                 for (int i = 0; i < this.Children.Count; i++)
00780                 {
00781                     double chLen = this.Children[i].ShortestDownstreamLength() +
      this.Children[i].Length;
00782
00783                     minLen = Math.Min(minLen, chLen);
00784                 }
00785
00786                 return minLen;
00787             }
00788         }
00789
00790 /// <summary>
00791 /// Get the node of the tree from which all other nodes descend.
00792 /// </summary>
00793 /// <returns>The node of the tree from which all other nodes descend</returns>
00794         public TreeNode GetRootNode()
00795         {
00796             TreeNode parent = this;
00797             while (parent.Parent != null)
00798             {
00799                 parent = parent.Parent;
00800             }
00801             return parent;
00802         }
00803
00804 /// <summary>
00805 /// Describes the relationship between two nodes.
00806 /// </summary>
00807         public enum NodeRelationship
00808         {
00809 /// <summary>
00810 /// The relationship between the nodes is unknown.
00811 /// </summary>
00812             Unknown,
00813
00814 /// <summary>
00815 /// The first node is an ancestor of the second node.
00816 /// </summary>
00817             Ancestor,
00818
00819 /// <summary>
00820 /// The first node is a descendant of the second node.
00821 /// </summary>
```

```
00822             Descendant,
00823
00824 /// <summary>
00825 /// The two nodes are relatives (i.e.  they share a common ancestor which is neither one of them).
00826 /// </summary>
00827             Relatives
00828         }
00829
00830 /// <summary>
00831 /// Get the sum of the branch lengths from this node to the specified node.
00832 /// </summary>
00833 /// <param name="otherNode">The node that should be reached</param>
00834 /// <param name="nodeRelationship">A value indicating how this node is related to <paramref
      name="otherNode"/>.</param>
00835 /// <returns>The sum of the branch lengths from this node to the specified node.</returns>
00836         public double PathLengthTo(TreeNode otherNode, NodeRelationship nodeRelationship =
      NodeRelationship.Unknown)
00837         {
00838             Contract.Requires(otherNode != null);
00839
00840             TreeNode LCA = null;
00841
00842             if (this == otherNode)
00843             {
00844                 return 0;
00845             }
00846
00847             if (nodeRelationship == NodeRelationship.Unknown)
00848             {
00849                 List<TreeNode> myChildren = this.GetChildrenRecursive();
00850                 if (myChildren.Contains(otherNode))
00851                 {
00852                     nodeRelationship = NodeRelationship.Ancestor;
00853                 }
00854                 else
00855                 {
00856                     List<TreeNode> otherChildren = otherNode.GetChildrenRecursive();
00857                     if (otherChildren.Contains(this))
00858                     {
00859                         nodeRelationship = NodeRelationship.Descendant;
00860                     }
00861                     else
00862                     {
00863                         LCA = GetLastCommonAncestor(new TreeNode[] { this, otherNode });
00864
00865                         if (LCA != null)
00866                         {
00867                             nodeRelationship = NodeRelationship.Relatives;
00868                         }
00869                         else
00870                         {
00871                             throw new InvalidOperationException("The two nodes do not belong to the
      same tree!");
00872                         }
00873                     }
00874                 }
00875             }
00876
00877             switch (nodeRelationship)
00878             {
00879                 case NodeRelationship.Relatives:
00880                     return LCA.PathLengthTo(this, NodeRelationship.Ancestor) +
      LCA.PathLengthTo(otherNode, NodeRelationship.Ancestor);
00881                 case NodeRelationship.Ancestor:
00882                     for (int i = 0; i < this.Children.Count; i++)
00883                     {
00884                         if (this.Children[i] == otherNode)
00885                         {
00886                             return this.Children[i].Length;
00887                         }
00888                         else if (this.Children[i].GetChildrenRecursive().Contains(otherNode))
00889                         {
00890                             return this.Children[i].Length + this.Children[i].PathLengthTo(otherNode,
      NodeRelationship.Ancestor);
00891                         }
00892                     }
00893                     throw new InvalidOperationException("Unexpected code path!");
00894                 case NodeRelationship.Descendant:
00895                     return otherNode.PathLengthTo(this, NodeRelationship.Ancestor);
00896                 default:
00897                     throw new InvalidOperationException("The two nodes do not belong to the same
      tree!");
00898             }
00899         }
00900
00901
00902 /// <summary>
```

```
00903 /// Get the sum of the branch lengths of this node and all its descendants.
00904 /// </summary>
00905 /// <returns>The sum of the branch lengths of this node and all its descendants.</returns>
00906         public double TotalLength()
00907         {
00908             double tbr = this.Length;
00909 
00910             for (int i = 0; i < this.Children.Count; i++)
00911             {
00912                 tbr += this.Children[i].TotalLength();
00913             }
00914             return tbr;
00915         }
00916 
00917 /// <summary>
00918 /// Sort (in place) the nodes in the tree in an aesthetically pleasing way.
00919 /// </summary>
00920 /// <param name="descending">The way the nodes should be sorted.</param>
00921         public void SortNodes(bool descending)
00922         {
00923             for (int i = 0; i < this.Children.Count; i++)
00924             {
00925                 this.Children[i].SortNodes(descending);
00926             }
00927 
00928             if (this.Children.Count > 0)
00929             {
00930                 this.Children.Sort((a, b) =>
00931                 {
00932                     int val = (a.GetLevels(true)[1] - b.GetLevels(true)[1]) * (descending ?  1 :  -1);
00933                     if (val != 0)
00934                     {
00935                         return val;
00936                     }
00937                     else
00938                     {
00939                         return string.Compare(a.GetLeafNames()[0], b.GetLeafNames()[0],
      StringComparison.InvariantCulture);
00940                     }
00941                 });
00942             }
00943         }
00944 
00945 /// <summary>
00946 /// Determine how many levels there are in the tree above and below this node.
00947 /// </summary>
00948 /// <param name="ignoreTotal">If this is <c>true</c>, the total number of levels is not computed (this
      improves performance).</param>
00949 /// <returns>
00950 /// An <see cref="int"/> array with 3 elements:  the first element is the number of levels above this
      node, the second element is the number of levels below this node, and the third element is the total
      number of levels in the tree.
00951 /// If <paramref name="ignoreTotal"/> is <c>true</c>, the third element is equal to the second.
00952 /// </returns>
00953         internal int[] GetLevels(bool ignoreTotal = false)
00954         {
00955             int upperCount = 0;
00956             TreeNode prnt = this.Parent;
00957             TreeNode lastPrnt = null;
00958             while (prnt != null)
00959             {
00960                 lastPrnt = prnt;
00961                 upperCount++;
00962                 prnt = prnt.Parent;
00963             }
00964 
00965             int lowerCount = 0;
00966             if (this.Children.Count > 0)
00967             {
00968                 for (int i = 0; i < this.Children.Count; i++)
00969                 {
00970                     TreeNode ch = this.Children[i];
00971                     lowerCount = Math.Max(lowerCount, 1 + ch.GetLevels(true)[1]);
00972                 }
00973             }
00974 
00975             if (this.Parent != null && !ignoreTotal)
00976             {
00977                 return new int[] { upperCount, lowerCount, lastPrnt.GetLevels()[2] };
00978             }
00979             else
00980             {
00981                 return new int[] { upperCount, lowerCount, lowerCount };
00982             }
00983 
00984         }
00985 
```

```
00986 /// <summary>
00987 /// Convert the tree to a Newick string.
00988 /// </summary>
00989 /// <returns></returns>
00990         public override string ToString()
00991         {
00992             return NWKA.WriteTree(this, false, true);
00993         }
00994 
00995 /// <summary>
00996 /// Determines whether the tree is clock-like (i.e.  all tips are contemporaneous) or not.
00997 /// </summary>
00998 /// <param name="tolerance">The (relative) tolerance when comparing branch lengths.</param>
00999 /// <returns>A boolean value determining whether the tree is clock-like or not</returns>
01000         public bool IsClockLike(double tolerance = 0.001)
01001         {
01002             List<TreeNode> leaves = this.GetLeaves();
01003 
01004             double len = leaves[0].UpstreamLength();
01005 
01006             foreach (TreeNode leaf in leaves)
01007             {
01008                 if (Math.Abs(leaf.UpstreamLength() / len - 1) > tolerance)
01009                 {
01010                     return false;
01011                 }
01012             }
01013 
01014             return true;
01015         }
01016 
01017 /// <summary>
01018 /// Gets the last common ancestor of all the specified nodes, or <c>null</c> if the tree doesn't
      contain all the nodes.
01019 /// </summary>
01020 /// <param name="monophyleticConstraint">The collection of nodes whose last common ancestor is to be
      determined.</param>
01021 /// <returns>The last common ancestor of all the specified nodes, or <c>null</c> if the tree doesn't
      contain all the nodes.</returns>
01022         public static TreeNode GetLastCommonAncestor(IEnumerable<TreeNode> monophyleticConstraint)
01023         {
01024             if (monophyleticConstraint.Any())
01025             {
01026                 TreeNode seed = monophyleticConstraint.ElementAt(0);
01027 
01028                 while (seed != null &&
      !seed.GetChildrenRecursive().ContainsAll(monophyleticConstraint))
01029                 {
01030                     seed = seed.Parent;
01031                 }
01032 
01033                 return seed;
01034             }
01035             else
01036             {
01037                 return null;
01038             }
01039         }
01040 
01041 /// <summary>
01042 /// Gets the last common ancestor of all the nodes with the specified names, or <c>null</c> if the
      tree doesn't contain all the named nodes.
01043 /// </summary>
01044 /// <param name="monophyleticConstraint">The collection of names representing nodes whose last common
      ancestor is to be determined.</param>
01045 /// <returns>The last common ancestor of all the nodes with the specified names, or <c>null</c> if the
      tree doesn't contain all the named nodes.</returns>
01046         public TreeNode GetLastCommonAncestor(params string[] monophyleticConstraint)
01047         {
01048             return this.GetLastCommonAncestor((IEnumerable<string>)monophyleticConstraint);
01049         }
01050 
01051 /// <summary>
01052 /// Gets the last common ancestor of all the nodes with the specified names, or <c>null</c> if the
      tree doesn't contain all the named nodes.
01053 /// </summary>
01054 /// <param name="monophyleticConstraint">The collection of names representing nodes whose last common
      ancestor is to be determined.</param>
01055 /// <returns>The last common ancestor of all the nodes with the specified names, or <c>null</c> if the
      tree doesn't contain all the named nodes.</returns>
01056         public TreeNode GetLastCommonAncestor(IEnumerable<string> monophyleticConstraint)
01057         {
01058             if (monophyleticConstraint.Any())
01059             {
01060                 TreeNode seed = this.GetNodeFromName(monophyleticConstraint.ElementAt(0));
01061 
01062                 while (seed != null && !seed.GetNodeNames().ContainsAll(monophyleticConstraint))
```

```
01063                 {
01064                     seed = seed.Parent;
01065                 }
01066 
01067                 return seed;
01068             }
01069             else
01070             {
01071                 return null;
01072             }
01073         }
01074 
01075 /// <summary>
01076 /// Checks whether this node is the last common ancestor of all the nodes with the specified names.
01077 /// </summary>
01078 /// <param name="monophyleticConstraint">The collection of names representing nodes whose last common
      ancestor is to be determined.</param>
01079 /// <returns><c>true</c> if this node is the last common ancestor of all the nodes with the specified
      names, <c>false</c> otherwise.</returns>
01080         public bool IsLastCommonAncestor(IEnumerable<string> monophyleticConstraint)
01081         {
01082             if (monophyleticConstraint.Any())
01083             {
01084                 TreeNode seed = this.GetNodeFromName(monophyleticConstraint.ElementAt(0));
01085 
01086                 while (seed != null && !seed.GetNodeNames().ContainsAll(monophyleticConstraint))
01087                 {
01088                     seed = seed.Parent;
01089                 }
01090 
01091                 return seed == this;
01092             }
01093             else
01094             {
01095                 return false;
01096             }
01097         }
01098 
01099 /// <summary>
01100 /// Transform the tree into a collection of (undirected) splits.
01101 /// </summary>
01102 /// <param name="lengthType">Determines whether the <see cref="Split.Length"/> should represent ages
      or branch lenghts.</param>
01103 /// <returns>A list of splits induced by the tree.  Each split corresponds to a branch in the
      tree.</returns>
01104         internal List<Split> GetSplits(Split.LengthTypes lengthType)
01105         {
01106             List<Split> tbr = new List<Split>();
01107 
01108             List<TreeNode> nodes = this.GetChildrenRecursive();
01109 
01110             double totalTreeLength = this.LongestDownstreamLength();
01111 
01112             if (this.Children.Count == 2)
01113             {
01114                 for (int i = 0; i < nodes.Count; i++)
01115                 {
01116                     List<string> nodeLeaves = nodes[i].GetLeafNames();
01117                     nodeLeaves.Sort();
01118 
01119                     tbr.Add(new Split(nodeLeaves.Aggregate((a, b) => a + "," + b), lengthType ==
      Split.LengthTypes.Length ?  nodes[i].Length :  (totalTreeLength - nodes[i].UpstreamLength()),
      lengthType, 1));
01120                 }
01121             }
01122             else
01123             {
01124                 List<string> allLeaves = this.GetLeafNames();
01125 
01126                 for (int i = 0; i < nodes.Count; i++)
01127                 {
01128                     List<string> nodeLeaves = nodes[i].GetLeafNames();
01129 
01130                     List<string> diffLeaves = (from el in allLeaves where !nodeLeaves.Contains(el)
      select el).ToList();
01131 
01132                     nodeLeaves.Sort();
01133                     diffLeaves.Sort();
01134 
01135                     if (diffLeaves.Count > 0)
01136                     {
01137                         List<string> splitTerminals = new List<string>() { nodeLeaves.Aggregate((a, b)
      => a + "," + b), diffLeaves.Aggregate((a, b) => a + "," + b) };
01138                         splitTerminals.Sort();
01139 
01140                         tbr.Add(new Split(splitTerminals.Aggregate((a, b) => a + "|" + b), lengthType
      == Split.LengthTypes.Length ?  nodes[i].Length :  ((totalTreeLength - nodes[i].UpstreamLength()) +
```

```
      nodes[i].Length), lengthType, 1));
01141                     }
01142                     else
01143                     {
01144                         tbr.Add(new Split(nodeLeaves.Aggregate((a, b) => a + "," + b), lengthType ==
      Split.LengthTypes.Length ?  nodes[i].Length :  (totalTreeLength - nodes[i].UpstreamLength()),
      lengthType, 1));
01145                     }
01146                 }
01147
01148             }
01149             return tbr;
01150         }
01151
01152 /// <summary>
01153 /// Gets the split corresponding to the branch underlying this node.  If this is an internal node,
      <c>side1</c> will contain all the leaves in the tree except those descending from this node, and
      <c>side2</c>
01154 /// will contain all the leaves descending from this node.  If this is the root <c>side1</c> will be
      empty and <c>side2</c> will contain all the leaves in the tree.  If the tree is rooted (the root node
      has exactly
01155 /// 2 children), <c>side1</c> will contain in all cases an additional <see langword="null"/> element.
01156 /// </summary>
01157 /// <returns>The leaves on the two sides of the split.</returns>
01158         public (List<TreeNode> side1, List<TreeNode> side2) GetSplit()
01159         {
01160             if (this.Parent == null)
01161             {
01162                 if (this.Children.Count == 2)
01163                 {
01164                     return (new List<TreeNode>() { null }, this.GetLeaves());
01165                 }
01166                 else
01167                 {
01168                     return (new List<TreeNode>(), this.GetLeaves());
01169                 }
01170             }
01171             else
01172             {
01173                 List<TreeNode> side2 = this.GetLeaves();
01174
01175                 TreeNode parent = this.Parent;
01176
01177                 while (parent.Parent != null)
01178                 {
01179                     parent = parent.Parent;
01180                 }
01181
01182                 List<TreeNode> side1 = parent.GetLeaves();
01183                 side1.RemoveAll(x => side2.Contains(x));
01184
01185                 if (parent.Children.Count == 2)
01186                 {
01187                     side1.Add(null);
01188                 }
01189
01190                 return (side1, side2);
01191             }
01192         }
01193
01194 /// <summary>
01195 /// Gets all the splits in the tree.
01196 /// </summary>
01197 /// <returns>An <see cref="IEnumerable{T}"/> that enumerates all the splits in the tree.</returns>
01198         public IEnumerable<(List<TreeNode> side1, List<TreeNode> side2, double branchLength)>
      GetSplits()
01199         {
01200             foreach (TreeNode node in this.GetChildrenRecursive())
01201             {
01202                 (List<TreeNode> side1, List<TreeNode> side2) = node.GetSplit();
01203
01204                 if (node.Parent == null)
01205                 {
01206                     yield return (side1, side2, 0);
01207                 }
01208                 else
01209                 {
01210                     yield return (side1, side2, node.Length);
01211                 }
01212             }
01213         }
01214
01215
01216
01217 /// <summary>
01218 /// Prunes the current node from the tree.
01219 /// </summary>
```

```
01220 /// <param name="leaveParent">This value determines what happens to the parent node of the current
      node if it only has two children (i.e., the current node and another node).  If this is <see
      langword="false"/>, the parent node is also pruned; if it is <see langword="true"/>, the parent node
      is left untouched.</param>
01221 /// <remarks>Note that the node is pruned in-place; however, the return value of this method should be
      used, because pruning the node may have caused the root of the tree to move.</remarks>
01222 /// <returns>The <see cref="TreeNode"/> corresponding to the root of the tree after the current node
      has been pruned.</returns>
01223         public TreeNode Prune(bool leaveParent)
01224         {
01225             if (this.Parent == null)
01226             {
01227                 return new TreeNode(null);
01228             }
01229
01230             this.Parent.Children.Remove(this);
01231
01232             if (!leaveParent)
01233             {
01234                 if (this.Parent.Children.Count == 1)
01235                 {
01236                     TreeNode parent = this.Parent;
01237                     TreeNode otherChild = this.Parent.Children[0];
01238                     if (parent.Parent != null)
01239                     {
01240                         int index = parent.Parent.Children.IndexOf(parent);
01241                         parent.Parent.Children[index] = otherChild;
01242                         otherChild.Length += parent.Length;
01243                         otherChild.Parent = parent.Parent;
01244
01245                         while (parent.Parent != null)
01246                         {
01247                             parent = parent.Parent;
01248                         }
01249
01250                         return parent;
01251                     }
01252                     else
01253                     {
01254                         if (parent.Length > 0)
01255                         {
01256                             otherChild.Length += parent.Length;
01257                         }
01258
01259                         otherChild.Parent = null;
01260                         return otherChild;
01261                     }
01262                 }
01263                 else
01264                 {
01265                     TreeNode parent = this.Parent;
01266
01267                     while (parent.Parent != null)
01268                     {
01269                         parent = parent.Parent;
01270                     }
01271
01272                     return parent;
01273                 }
01274             }
01275             else
01276             {
01277                 TreeNode parent = this.Parent;
01278
01279                 while (parent.Parent != null)
01280                 {
01281                     parent = parent.Parent;
01282                 }
01283
01284                 return parent;
01285             }
01286         }
01287
01288 /// <summary>
01289 /// Prunes a node from the tree.
01290 /// </summary>
01291 /// <param name="nodeToPrune">The node that should be pruned.</param>
01292 /// <param name="leaveParent">This value determines what happens to the parent node of the pruned node
      if it only has two children (i.e., the pruned node and another node).  If this is <see
      langword="false"/>, the parent node is also pruned; if it is <see langword="true"/>, the parent node
      is left untouched.</param>
01293 /// <remarks>Note that the node is pruned in-place; however, the return value of this method should be
      used, because pruning the node may have caused the root of the tree to move.</remarks>
01294 /// <returns>The <see cref="TreeNode"/> corresponding to the root of the tree after the <paramref
      name="nodeToPrune"/> has been pruned.</returns>
01295         public TreeNode Prune(TreeNode nodeToPrune, bool leaveParent)
01296         {
```

```
01297             if (nodeToPrune == null)
01298             {
01299                 if (this.Parent != null)
01300                 {
01301                     TreeNode parent = this.Parent;
01302 
01303                     while (parent.Parent != null)
01304                     {
01305                         parent = parent.Parent;
01306                     }
01307 
01308                     return parent;
01309                 }
01310                 else
01311                 {
01312                     return this;
01313                 }
01314             }
01315 
01316             return nodeToPrune.Prune(leaveParent);
01317         }
01318 
01319 /// <summary>
01320 /// Creates a lower triangular distance matrix, where each entry is the path length distance between
      two leaves in the tree.  Entries are in the same order as returned by the <see cref="GetLeaves"/>
      method.
01321 /// </summary>
01322 /// <param name="maxDegreeOfParallelism">Maximum number of threads to use, or -1 to let the runtime
      decide.  If this argument is set to 0 (the default), the value used is 1 for trees with 1500 or fewer
      leaves, or -1 for larger trees.</param>
01323 /// <param name="progressCallback">A method used to report progress.</param>
01324 /// <returns>A <see cref="T:double[][]"/> jagged array containing the distance matrix.</returns>
01325         public double[][] CreateDistanceMatrixDouble(int maxDegreeOfParallelism = 0, Action<double>
      progressCallback = null)
01326         {
01327             List<TreeNode> leaves = this.GetLeaves();
01328             List<TreeNode> nodes = this.GetChildrenRecursive();
01329 
01330             if (maxDegreeOfParallelism == 0)
01331             {
01332                 if (leaves.Count <= 1500)
01333                 {
01334                     maxDegreeOfParallelism = 1;
01335                 }
01336                 else
01337                 {
01338                     maxDegreeOfParallelism = -1;
01339                 }
01340             }
01341 
01342             HashSet<int>[] ancestors = new HashSet<int>[nodes.Count];
01343             HashSet<int>[] descendants = new HashSet<int>[nodes.Count];
01344             int[] leafIndices = new int[leaves.Count];
01345 
01346             for (int i = 0; i < nodes.Count; i++)
01347             {
01348                 if (nodes[i].Parent != null)
01349                 {
01350                     int parentIndex = nodes.IndexOf(nodes[i].Parent);
01351                     ancestors[i] = new HashSet<int>(ancestors[parentIndex]);
01352                     ancestors[i].Add(i);
01353                 }
01354                 else
01355                 {
01356                     ancestors[i] = new HashSet<int>() { i };
01357                 }
01358 
01359                 if (nodes[i].Children.Count == 0)
01360                 {
01361                     leafIndices[leaves.IndexOf(nodes[i])] = i;
01362                 }
01363             }
01364 
01365             for (int i = 0; i < nodes.Count; i++)
01366             {
01367                 descendants[i] = new HashSet<int>();
01368             }
01369 
01370             for (int i = 0; i < leaves.Count; i++)
01371             {
01372                 foreach (int j in ancestors[leafIndices[i]])
01373                 {
01374                     descendants[j].Add(i);
01375                 }
01376             }
01377 
01378             double[][] tbr = new double[leaves.Count][];
```

```
01379
01380             for (int i = 0; i < tbr.Length; i++)
01381             {
01382                 tbr[i] = new double[i];
01383             }
01384
01385             if (maxDegreeOfParallelism == 1)
01386             {
01387                 for (int i = 1; i < nodes.Count; i++)
01388                 {
01389                     double length = nodes[i].Length;
01390
01391                     for (int k = 0; k < leaves.Count; k++)
01392                     {
01393                         if (!descendants[i].Contains(k))
01394                         {
01395                             foreach (int j in descendants[i])
01396                             {
01397                                 tbr[Math.Max(j, k)][Math.Min(j, k)] += length;
01398                             }
01399                         }
01400                     }
01401
01402                     progressCallback?.Invoke((double)i / (nodes.Count - 1));
01403                 }
01404             }
01405             else
01406             {
01407                 int progress = 0;
01408                 object progressLock = new object();
01409
01410                 System.Threading.Tasks.Parallel.For(1, nodes.Count, new
      System.Threading.Tasks.ParallelOptions() { MaxDegreeOfParallelism = maxDegreeOfParallelism }, i =>
01411                 {
01412                     double length = nodes[i].Length;
01413
01414                     for (int k = 0; k < leaves.Count; k++)
01415                     {
01416                         if (!descendants[i].Contains(k))
01417                         {
01418                             foreach (int j in descendants[i])
01419                             {
01420                                 Add(ref tbr[Math.Max(j, k)][Math.Min(j, k)], length);
01421                             }
01422                         }
01423                     }
01424
01425                     if (progressCallback != null)
01426                     {
01427                         lock (progressLock)
01428                         {
01429                             progress++;
01430                             progressCallback.Invoke((double)progress / (nodes.Count - 1));
01431                         }
01432                     }
01433                 });
01434             }
01435
01436             return tbr;
01437         }
01438
01439 /// <summary>
01440 /// Creates a lower triangular distance matrix, where each entry is the path length distance between
      two leaves in the tree.  Entries are in the same order as returned by the <see cref="GetLeaves"/>
      method.
01441 /// </summary>
01442 /// <param name="maxDegreeOfParallelism">Maximum number of threads to use, or -1 to let the runtime
      decide.  If this argument is set to 0 (the default), the value used is 1 for trees with 1500 or fewer
      leaves, or -1 for larger trees.</param>
01443 /// <param name="progressCallback">A method used to report progress.</param>
01444 /// <returns>A <see cref="T:float[][]"/> jagged array containing the distance matrix.</returns>
01445         public float[][] CreateDistanceMatrixFloat(int maxDegreeOfParallelism = 0, Action<double>
      progressCallback = null)
01446         {
01447             List<TreeNode> leaves = this.GetLeaves();
01448             List<TreeNode> nodes = this.GetChildrenRecursive();
01449
01450             if (maxDegreeOfParallelism == 0)
01451             {
01452                 if (leaves.Count <= 1500)
01453                 {
01454                     maxDegreeOfParallelism = 1;
01455                 }
01456                 else
01457                 {
01458                     maxDegreeOfParallelism = -1;
01459                 }
```

```
01460             }
01461
01462             HashSet<int>[] ancestors = new HashSet<int>[nodes.Count];
01463             HashSet<int>[] descendants = new HashSet<int>[nodes.Count];
01464             int[] leafIndices = new int[leaves.Count];
01465
01466             for (int i = 0; i < nodes.Count; i++)
01467             {
01468                 if (nodes[i].Parent != null)
01469                 {
01470                     int parentIndex = nodes.IndexOf(nodes[i].Parent);
01471                     ancestors[i] = new HashSet<int>(ancestors[parentIndex]);
01472                     ancestors[i].Add(i);
01473                 }
01474                 else
01475                 {
01476                     ancestors[i] = new HashSet<int>() { i };
01477                 }
01478
01479                 if (nodes[i].Children.Count == 0)
01480                 {
01481                     leafIndices[leaves.IndexOf(nodes[i])] = i;
01482                 }
01483             }
01484
01485             for (int i = 0; i < nodes.Count; i++)
01486             {
01487                 descendants[i] = new HashSet<int>();
01488             }
01489
01490             for (int i = 0; i < leaves.Count; i++)
01491             {
01492                 foreach (int j in ancestors[leafIndices[i]])
01493                 {
01494                     descendants[j].Add(i);
01495                 }
01496             }
01497
01498             float[][] tbr = new float[leaves.Count][];
01499
01500             for (int i = 0; i < tbr.Length; i++)
01501             {
01502                 tbr[i] = new float[i];
01503             }
01504
01505             if (maxDegreeOfParallelism == 1)
01506             {
01507                 for (int i = 1; i < nodes.Count; i++)
01508                 {
01509                     float length = (float)nodes[i].Length;
01510
01511                     for (int k = 0; k < leaves.Count; k++)
01512                     {
01513                         if (!descendants[i].Contains(k))
01514                         {
01515                             foreach (int j in descendants[i])
01516                             {
01517                                 tbr[Math.Max(j, k)][Math.Min(j, k)] += length;
01518                             }
01519                         }
01520                     }
01521
01522                     progressCallback?.Invoke((double)i / (nodes.Count - 1));
01523                 }
01524             }
01525             else
01526             {
01527                 int progress = 0;
01528                 object progressLock = new object();
01529
01530                 System.Threading.Tasks.Parallel.For(1, nodes.Count, new
      System.Threading.Tasks.ParallelOptions() { MaxDegreeOfParallelism = maxDegreeOfParallelism }, i =>
01531                 {
01532                     float length = (float)nodes[i].Length;
01533
01534                     for (int k = 0; k < leaves.Count; k++)
01535                     {
01536                         if (!descendants[i].Contains(k))
01537                         {
01538                             foreach (int j in descendants[i])
01539                             {
01540                                 Add(ref tbr[Math.Max(j, k)][Math.Min(j, k)], length);
01541                             }
01542                         }
01543                     }
01544
01545                     if (progressCallback != null)
```

```
01546                     {
01547                         lock (progressLock)
01548                         {
01549                             progress++;
01550                             progressCallback.Invoke((double)progress / (nodes.Count - 1));
01551                         }
01552                     }
01553                 });
01554             }
01555 
01556             return tbr;
01557         }
01558 
01559 /// <summary>
01560 /// Interlocked add for double, from https://stackoverflow.com/a/16893641.
01561 /// </summary>
01562         private static double Add(ref double location1, double value)
01563         {
01564             double newCurrentValue = location1; // non-volatile read, so may be stale
01565             while (true)
01566             {
01567                 double currentValue = newCurrentValue;
01568                 double newValue = currentValue + value;
01569                 newCurrentValue = Interlocked.CompareExchange(ref location1, newValue, currentValue);
01570                 if (newCurrentValue.Equals(currentValue))
01571                 {
01572                     return newValue;
01573                 }
01574             }
01575         }
01576 
01577 /// <summary>
01578 /// Interlocked add for float, adapted from https://stackoverflow.com/a/16893641.
01579 /// </summary>
01580         private static float Add(ref float location1, float value)
01581         {
01582             float newCurrentValue = location1; // non-volatile read, so may be stale
01583             while (true)
01584             {
01585                 float currentValue = newCurrentValue;
01586                 float newValue = currentValue + value;
01587                 newCurrentValue = Interlocked.CompareExchange(ref location1, newValue, currentValue);
01588                 if (newCurrentValue.Equals(currentValue))
01589                 {
01590                     return newValue;
01591                 }
01592             }
01593         }
01594 
01595     }
01596 }
```

8.29 TreeNode.ShapeIndices.cs

```
00001 using System;
00002 using System.Collections.Generic;
00003 using System.Linq;
00004 using System.Text;
00005 
00006 namespace PhyloTree
00007 {
00008     public partial class TreeNode
00009     {
00010 /// <summary>
00011 /// Null hypothesis for normalising tree shape indices.
00012 /// </summary>
00013         public enum NullHypothesis
00014         {
00015 /// <summary>
00016 /// Yule-Harding-Kingman model (also known as Yule model or Equal-rates Markov model).  At each step
      in growing the tree, a new leaf is added as a sibling to an existing leaf.
00017 /// </summary>
00018             YHK,
00019 
00020 /// <summary>
00021 /// Proportional to distinguished arrangements model (also known as uniform model).  At each step in
      growing the tree, a new leaf is added as a sibling to an existing (possibly internal) node.
00022 /// </summary>
00023             PDA,
00024 
00025 /// <summary>
00026 /// Do not perform any normalisation.
00027 /// </summary>
```

```
00028             None
00029         }
00030
00031 /// <summary>
00032 /// Compute the depth of the node (number of branches from this node until the root node).
00033 /// </summary>
00034 /// <returns>The depth of the node.</returns>
00035         public int GetDepth()
00036         {
00037             return this.GetDepth(0);
00038         }
00039
00040         private int GetDepth(int currentDepth = 0)
00041         {
00042             if (this.Parent == null)
00043             {
00044                 return currentDepth;
00045             }
00046
00047             return this.Parent.GetDepth(currentDepth + 1);
00048         }
00049
00050 /// <summary>
00051 /// Computes the Sackin index of the tree (sum of the leaf depths).
00052 /// </summary>
00053 /// <param name="model">If this is <see cref="NullHypothesis.None"/>, the raw Sackin index is
      returned.  If this is <see cref="NullHypothesis.YHK"/> or <see cref="NullHypothesis.PDA"/>, the Sackin
00054 /// index is normalised with respect to the corresponding null tree model (which makes scores
      comparable across trees of different sizes).</param>
00055 /// <returns>The Sackin index of the tree, either as a raw value, or normalised according to the
      selected null tree model.</returns>
00056         public double SackinIndex(NullHypothesis model = NullHypothesis.None)
00057         {
00058             List<double> leafDepths = new List<double>();
00059
00060             List<TreeNode> leaves = this.GetLeaves();
00061
00062             foreach (TreeNode leaf in leaves)
00063             {
00064                 leafDepths.Add(leaf.GetDepth());
00065             }
00066
00067             double averageLeafDepth = leafDepths.Average();
00068
00069             int sackinIndex = (int)leafDepths.Sum();
00070
00071             switch (model)
00072             {
00073                 case NullHypothesis.None:
00074                     return sackinIndex;
00075                 case NullHypothesis.YHK:
00076                     return (sackinIndex - 2 * leaves.Count * (from el in Enumerable.Range(2,
      leaves.Count - 1) select 1.0 / el).Sum()) / leaves.Count;
00077                 case NullHypothesis.PDA:
00078                     return sackinIndex / Math.Pow(leaves.Count, 1.5);
00079             }
00080
00081             return double.NaN;
00082         }
00083
00084
00085         private (int score, int leaves) ComputeCollessInner()
00086         {
00087             if (this.Children.Count > 0)
00088             {
00089                 (int score1, int leaves1) = this.Children[0].ComputeCollessInner();
00090                 (int score2, int leaves2) = this.Children[1].ComputeCollessInner();
00091
00092                 return (score1 + score2 + Math.Abs(leaves1 - leaves2), leaves1 + leaves2);
00093             }
00094             else
00095             {
00096                 return (0, 1);
00097             }
00098         }
00099
00100 /// <summary>
00101 /// Computes the expected value of the Colless index under the YHK model.
00102 /// </summary>
00103 /// <param name="numberOfLeaves">The number of leaves in the tree.</param>
00104 /// <returns>The expected value of the Colless index for a tree with the specified <paramref
      name="numberOfLeaves"/>.</returns>
00105 /// <remarks>Proof in DOI: 10.1214/105051606000000547</remarks>
00106         public static double GetCollessExpectationYHK(int numberOfLeaves)
00107         {
00108             static double tN(int n)
00109             {
```

```
00110                 if (n % 2 == 0)
00111                 {
00112                     return (n - 2) / 4.0;
00113                 }
00114                 else
00115                 {
00116                     return (n - 1) * (n - 1) / (4.0 * n);
00117                 }
00118             }
00119 
00120             double sum = 0;
00121 
00122             for (int k = 1; k < numberOfLeaves; k++)
00123             {
00124                 sum += (k - 1 - 2 * tN(k)) / ((k + 1) * (k + 2));
00125             }
00126 
00127             return numberOfLeaves - 1 - 2 * tN(numberOfLeaves) + 2 * (numberOfLeaves + 1) * sum;
00128         }
00129 
00130 /// <summary>
00131 /// Compute the Colless index of the tree.
00132 /// </summary>
00133 /// <param name="model">If this is <see cref="NullHypothesis.None"/>, the raw Colless index is
      returned.  If this is <see cref="NullHypothesis.YHK"/> or <see cref="NullHypothesis.PDA"/>, the
      Colless
00134 /// index is normalised with respect to the corresponding null tree model (which makes scores
      comparable across trees of different sizes).</param>
00135 /// <param name="yhkExpectation">If <paramref name="model"/> is <see cref="NullHypothesis.YHK"/>, you
      can optionally use this parameter to provide a pre-computed value for the expected value of the
00136 /// Colless index under the YHK model.  This is useful to save time if you need to compute the Colless
      index of many trees with the same number of leaves.  If this is <see cref="double.NaN"/>, the
00137 /// expected value under the YHK model is computed by this method.</param>
00138 /// <returns>The Colless index of the tree.</returns>
00139         public double CollessIndex(NullHypothesis model = NullHypothesis.None, double yhkExpectation =
      double.NaN)
00140         {
00141             (int score, int leaves) = this.ComputeCollessInner();
00142 
00143             switch (model)
00144             {
00145                 case NullHypothesis.None:
00146                     return score;
00147                 case NullHypothesis.YHK:
00148                     if (double.IsNaN(yhkExpectation))
00149                     {
00150                         yhkExpectation = GetCollessExpectationYHK(leaves);
00151                     }
00152                     return (score - yhkExpectation) / leaves;
00153                 case NullHypothesis.PDA:
00154                     return score / Math.Pow(leaves, 1.5);
00155             }
00156 
00157             return double.NaN;
00158         }
00159 
00160 /// <summary>
00161 /// Computes the number of cherries in the tree.
00162 /// </summary>
00163 /// <param name="model">If this is <see cref="NullHypothesis.None"/>, the raw number of cherries is
      returned.  If this is <see cref="NullHypothesis.YHK"/> or <see cref="NullHypothesis.PDA"/>, the number
00164 /// of cherries is normalised with respect to the corresponding null tree model (which makes scores
      comparable across trees of different sizes).</param>
00165 /// <returns>The number of cherries in the tree.</returns>
00166 /// <remarks>Proofs in DOI: 10.1016/S0025-5564(99)00060-7</remarks>
00167         public double NumberOfCherries(NullHypothesis model = NullHypothesis.None)
00168         {
00169             List<TreeNode> leaves = this.GetLeaves();
00170 
00171             int numberOfCherries = 0;
00172 
00173             for (int i = 0; i < leaves.Count; i++)
00174             {
00175                 if (leaves[i].Parent.Children.Count == 2 &&
      leaves[i].Parent.Children[0].Children.Count == 0 && leaves[i].Parent.Children[1].Children.Count == 0)
00176                 {
00177                     numberOfCherries++;
00178                 }
00179             }
00180 
00181             numberOfCherries /= 2;
00182 
00183             switch (model)
00184             {
00185                 case NullHypothesis.None:
00186                     return numberOfCherries;
00187 
```

```
00188                 case NullHypothesis.YHK:
00189                     return (numberOfCherries - leaves.Count / 3.0) / Math.Sqrt(2.0 * leaves.Count /
      45.0);
00190 
00191                 case NullHypothesis.PDA:
00192                     double mu = (double)leaves.Count * (leaves.Count - 1) / (2.0 * (2 * leaves.Count -
      5));
00193                     double sigmaSq = (double)leaves.Count * (leaves.Count - 1) * (leaves.Count - 4) *
      (leaves.Count - 5) / (2.0 * (2 * leaves.Count - 5) * (2 * leaves.Count - 5) * (2 * leaves.Count - 7));
00194                     return (numberOfCherries - mu) / Math.Sqrt(sigmaSq);
00195             }
00196 
00197             return double.NaN;
00198         }
00199     }
00200 }
```

Index