TreeNode

1.2.0

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

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

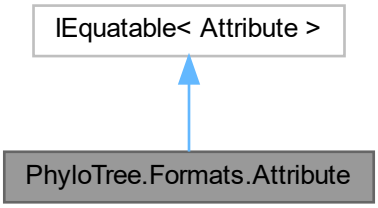
Chapter 7

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.

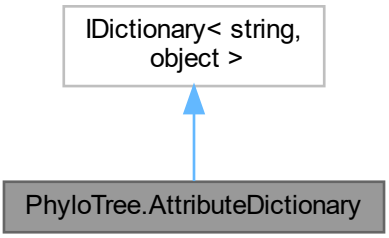
The documentation for this struct was generated from the following file:

- 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.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.

7.5.1 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.

7.5.2 Member Function Documentation

7.5.2.1 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.

7.5.2.2 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.

7.5.2.3 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.

7.5.2.4 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.

7.5.2.5 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.

7.5.2.6 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.

7.5.2.7 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.

7.5.2.8 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.

7.5.2.9 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.

7.5.2.10 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.

7.5.2.11 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.

7.5.2.12 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

7.6 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 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 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.

7.6.1 Detailed Description

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

Definition at line 12 of file NcbiAsnText.cs.

7.6.2 Member Function Documentation

7.6.2.1 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.

7.6.2.2 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.

7.6.2.3 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.

7.6.2.4 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.

7.6.2.5 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.

7.6.2.6 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.

7.6.2.7 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.

7.6.2.8 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.

7.6.2.9 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.

7.6.2.10 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.

7.6.2.11 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.

7.6.2.12 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.

7.6.2.13 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

7.7 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.

7.7.1 Detailed Description

Contains methods to read and write trees in NEXUS format.

Definition at line 14 of file NEXUS.cs.

7.7.2 Member Function Documentation

7.7.2.1 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.

7.7.2.2 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.

7.7.2.3 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.

7.7.2.4 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.

7.7.2.5 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.

7.7.2.6 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.

7.7.2.7 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.

7.7.2.8 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.

7.7.2.9 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.

7.7.2.10 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.

7.7.2.11 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.

7.7.2.12 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

7.8 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.

7.8.1 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.

7.8.2 Member Function Documentation

7.8.2.1 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.

7.8.2.2 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.

7.8.2.3 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.

7.8.2.4 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.

7.8.2.5 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.

7.8.2.6 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.

7.8.2.7 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.

7.8.2.8 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.

7.8.2.9 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.

7.8.2.10 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.

7.8.2.11 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.

7.8.2.12 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.

7.8.2.13 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.

7.8.2.14 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.

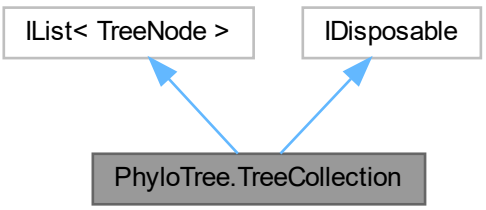
The documentation for this class was generated from the following file:

- NWKA.cs

7.9 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.

7.9.1 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.

7.9.2 Constructor & Destructor Documentation

7.9.2.1 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.

7.9.2.2 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.

7.9.3 Member Function Documentation

7.9.3.1 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.

7.9.3.2 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.

7.9.3.3 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.

7.9.3.4 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.

7.9.3.5 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.

7.9.3.6 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.

7.9.3.7 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.

7.9.3.8 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.

7.9.3.9 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.

7.9.3.10 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.

7.9.3.11 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.

7.9.4 Property Documentation

7.9.4.1 Count

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

The number of trees in the collection.

Definition at line 87 of file TreeCollection.cs.

7.9.4.2 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.

7.9.4.3 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.

7.9.4.4 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.

7.9.4.5 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

7.10 PhyloTree.TreeNode Class Reference

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

Public Types

- 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.
- 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 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.

7.10.1 Detailed Description

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

Definition at line 9 of file TreeNode.Comparisons.cs.

7.10.2 Member Enumeration Documentation

7.10.2.1 NodeRelationship

```
enum PhyloTree.TreeNode.NodeRelationship
```

Describes the relationship between two nodes.

Definition at line 807 of file TreeNode.cs.

7.10.2.2 NullHypothesis

```
enum PhyloTree.TreeNode.NullHypothesis
```

Null hypothesis for normalising tree shape indices.

Definition at line 13 of file TreeNode.ShapeIndices.cs.

7.10.3 Constructor & Destructor Documentation

7.10.3.1 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.

7.10.4 Member Function Documentation

7.10.4.1 Clone()

```
TreeNode PhyloTree.TreeNode.Clone ( )
```

Recursively clone a TreeNode object.

Returns

The cloned TreeNode

Definition at line 554 of file TreeNode.cs.

7.10.4.2 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.

7.10.4.3 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.

7.10.4.4 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.

7.10.4.5 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.

7.10.4.6 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.

7.10.4.7 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.

7.10.4.8 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.

7.10.4.9 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.

7.10.4.10 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.

7.10.4.11 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.

7.10.4.12 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.

7.10.4.13 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.

7.10.4.14 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.

7.10.4.15 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.

7.10.4.16 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.

7.10.4.17 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.

7.10.4.18 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.

7.10.4.19 GetSplit()

```
List< TreeNode > List< TreeNode > side2 PhyloTree.TreeNode.GetSplit ( )
```

Definition at line 1158 of file TreeNode.cs.

7.10.4.20 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.

7.10.4.21 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.

7.10.4.22 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.

7.10.4.23 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.

7.10.4.24 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.

7.10.4.25 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.

7.10.4.26 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.

7.10.4.27 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.

7.10.4.28 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.

7.10.4.29 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.

7.10.4.30 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 17 of file TreeNode.Comparisons.cs.

7.10.4.31 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 29 of file TreeNode.Comparisons.cs.

7.10.4.32 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.

7.10.4.33 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.

7.10.4.34 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.

7.10.4.35 ToString()

```
override string PhyloTree.TreeNode.ToString ( )
```

Convert the tree to a Newick string.

Returns

Definition at line 990 of file TreeNode.cs.

7.10.4.36 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.

7.10.4.37 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.

7.10.5 Member Data Documentation

7.10.5.1 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.

7.10.6 Property Documentation

7.10.6.1 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.

7.10.6.2 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.

7.10.6.3 Id

```
string PhyloTree.TreeNode.Id [get]
```

A univocal identifier for the node.

Definition at line 372 of file TreeNode.cs.

7.10.6.4 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.

7.10.6.5 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.

7.10.6.6 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.

7.10.6.7 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

7.11 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)

  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.

7.11.1 Detailed Description

Useful extension methods

Definition at line 16 of file Extensions.cs.

7.11.2 Member Function Documentation

7.11.2.1 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 25 of file Extensions.cs.

7.11.2.2 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 66 of file Extensions.cs.

7.11.2.3 GetConsensus()

```
static TreeNode PhyloTree.Extensions.TypeExtensions.GetConsensus (
            this IEnumerable< TreeNode > trees,
            bool rooted,
```

```
            bool clockLike,
            double threshold,
            bool useMedian,
            Action< double > progressAction = null )  [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.

Returns

A rooted consensus tree.

Definition at line 111 of file Extensions.cs.

7.11.2.4 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 87 of file Extensions.cs.

7.11.2.5 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 44 of file Extensions.cs.

7.11.2.6 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 181 of file Extensions.cs.

7.11.2.7 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 269 of file Extensions.cs.

7.11.2.8 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 325 of file Extensions.cs.

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

- Extensions.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.Generic;
00003 using System.Diagnostics.Contracts;
00004 using System.IO;
00005 using System.Linq;
00006 using System.Text;
00007
00008 /// <summary>
00009 /// Contains useful extension methods.
00010 /// </summary>
00011 namespace PhyloTree.Extensions
00012 {
00013 /// <summary>
00014 /// Useful extension methods
00015 /// </summary>
00016     public static class TypeExtensions
00017     {
00018 /// <summary>
00019 /// Determines whether <paramref name="haystack"/> contains all of the elements in <paramref
      name="needle"/>.
00020 /// </summary>
00021 /// <typeparam name="T">The type of the elements in the collections.</typeparam>
00022 /// <param name="haystack">The collection in which to search.</param>
00023 /// <param name="needle">The items to be searched.</param>
00024 /// <returns><c>true</c> if haystack contains all of the elements that are in needle or needle is
      empty.</returns>
00025         public static bool ContainsAll<T>(this IEnumerable<T> haystack, IEnumerable<T> needle)
00026         {
00027             Contract.Requires(needle != null);
00028
00029             foreach (T t in needle)
00030             {
00031                 if (!haystack.Contains(t))
00032                 {
00033                     return false;
00034                 }
00035             }
00036             return true;
00037         }
00038
00039 /// <summary>
00040 /// Compute the median of a list of values.
00041 /// </summary>
00042 /// <param name="array">The list of values whose median is to be computed.</param>
00043 /// <returns>The median of the list of values.</returns>
00044         public static double Median(this IEnumerable<double> array)
00045         {
00046             List<double> ordered = new List<double>(array);
00047             ordered.Sort();
00048
00049             if (ordered.Count % 2 == 0)
00050             {
00051                 return 0.5 * (ordered[ordered.Count / 2] + ordered[ordered.Count / 2 - 1]);
00052             }
00053             else
00054             {
00055                 return ordered[ordered.Count / 2];
00056             }
00057         }
00058
00059 /// <summary>
00060 /// Determines whether <paramref name="haystack"/> contains at least one of the elements in <paramref
      name="needle"/>.
00061 /// </summary>
00062 /// <typeparam name="T">The type of the elements in the collections.</typeparam>
00063 /// <param name="haystack">The collection in which to search.</param>
00064 /// <param name="needle">The items to be searched.</param>
00065 /// <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>
00066         public static bool ContainsAny<T>(this IEnumerable<T> haystack, IEnumerable<T> needle)
00067         {
00068             Contract.Requires(needle != null);
00069
00070             foreach (T t in needle)
00071             {
00072                 if (haystack.Contains(t))
00073                 {
00074                     return true;
00075                 }
00076             }
00077             return false;
00078         }
00079
00080 /// <summary>
00081 /// Computes the intersection between two sets.
```

```
00082 /// </summary>
00083 /// <typeparam name="T">The type of the elements in the collections.</typeparam>
00084 /// <param name="set1">The first set.</param>
00085 /// <param name="set2">The second set.</param>
00086 /// <returns>The intersection between the two sets.</returns>
00087         public static IEnumerable<T> Intersection<T>(this IEnumerable<T> set1, IEnumerable<T> set2)
00088         {
00089             Contract.Requires(set1 != null);
00090             Contract.Requires(set2 != null);
00091 
00092             foreach (T element in set1)
00093             {
00094                 if (set2.Contains(element))
00095                 {
00096                     yield return element;
00097                 }
00098             }
00099         }
00100 
00101 /// <summary>
00102 /// Constructs a consensus tree.
00103 /// </summary>
00104 /// <param name="trees">The collection of trees whose consensus is to be computed.</param>
00105 /// <param name="rooted">Whether the consensus tree should be rooted or not.</param>
00106 /// <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>
00107 /// <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>
00108 /// <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>
00109 /// <param name="progressAction">An <see cref="Action"/> that will be invoked as the trees are
      processed.</param>
00110 /// <returns>A rooted consensus tree.</returns>
00111         public static TreeNode GetConsensus(this IEnumerable<TreeNode> trees, bool rooted, bool
      clockLike, double threshold, bool useMedian, Action<double> progressAction = null)
00112         {
00113             Contract.Requires(trees != null);
00114 
00115             Dictionary<string, List<double» splits = new Dictionary<string, List<double»();
00116 
00117             int totalTrees = 0;
00118 
00119             Split.LengthTypes lengthType = clockLike ?  Split.LengthTypes.Age :
      Split.LengthTypes.Length;
00120 
00121             int count = -1;
00122 
00123             if (trees is IReadOnlyList<TreeNode> list)
00124             {
00125                 count = list.Count;
00126             }
00127 
00128             foreach (TreeNode tree in trees)
00129             {
00130                 List<Split> treeSplits = tree.GetSplits(lengthType);
00131 
00132                 for (int i = 0; i < treeSplits.Count; i++)
00133                 {
00134                     if (splits.TryGetValue(treeSplits[i].Name, out List<double> splitLengths))
00135                     {
00136                         splits[treeSplits[i].Name].Add(treeSplits[i].Length);
00137                     }
00138                     else
00139                     {
00140                         splits.Add(treeSplits[i].Name, new List<double>() { treeSplits[i].Length });
00141                     }
00142                 }
00143 
00144                 totalTrees++;
00145 
00146                 if (count > 0)
00147                 {
00148                     progressAction?.Invoke((double)totalTrees / count);
00149                 }
00150                 else
00151                 {
00152                     progressAction?.Invoke(totalTrees);
00153                 }
00154             }
00155 
00156             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)));
00157 
```

```
00158             List<Split> finalSplits = new List<Split>();
00159 
00160             for (int i = 0; i < orderedSplits.Count; i++)
00161             {
00162                 if (orderedSplits[i].IsCompatible(finalSplits))
00163                 {
00164                     finalSplits.Add(orderedSplits[i]);
00165                 }
00166             }
00167 
00168             return Split.BuildTree(finalSplits, rooted);
00169         }
00170 
00171 /// <summary>
00172 /// Reads the next non-whitespace character, taking into account quoting and escaping.
00173 /// </summary>
00174 /// <param name="reader">The <see cref="TextReader"/> to read from.</param>
00175 /// <param name="escaping">A <see cref="bool"/> indicating whether the next character will be
      escaped.</param>
00176 /// <param name="escaped">A <see cref="bool"/> indicating whether the current character will be
      escaped.</param>
00177 /// <param name="openQuotes">A <see cref="bool"/> indicating whether double quotes have been
      opened.</param>
00178 /// <param name="openApostrophe">A <see cref="bool"/> indicating whether single quotes have been
      opened.</param>
00179 /// <param name="eof">A <see cref="bool"/> indicating whether we have arrived at the end of the
      file.</param>
00180 /// <returns>The next non-whitespace character.</returns>
00181         public static char NextToken(this TextReader reader, ref bool escaping, out bool escaped, ref
      bool openQuotes, ref bool openApostrophe, out bool eof)
00182         {
00183             Contract.Requires(reader != null);
00184 
00185             int i = reader.Read();
00186 
00187             if (i < 0)
00188             {
00189                 eof = true;
00190                 escaped = false;
00191                 return (char)i;
00192             }
00193 
00194             eof = false;
00195             char c = (char)i;
00196 
00197             if (!escaping)
00198             {
00199                 escaped = false;
00200                 if (!openQuotes && !openApostrophe)
00201                 {
00202                     while (Char.IsWhiteSpace(c))
00203                     {
00204                         i = reader.Read();
00205 
00206                         if (i < 0)
00207                         {
00208                             eof = true;
00209                             escaped = false;
00210                             return (char)i;
00211                         }
00212 
00213                         c = (char)i;
00214                     }
00215 
00216                     switch (c)
00217                     {
00218                         case '\\':
00219                             escaping = true;
00220                             break;
00221                         case '"':
00222                             openQuotes = true;
00223                             break;
00224                         case '\'':
00225                             openApostrophe = true;
00226                             break;
00227                     }
00228                 }
00229                 else if (openQuotes)
00230                 {
00231                     switch (c)
00232                     {
00233                         case '"':
00234                             openQuotes = false;
00235                             break;
00236                         case '\\':
00237                             escaping = true;
00238                             break;
```

```
00239                     }
00240                 }
00241                 else if (openApostrophe)
00242                 {
00243                     switch (c)
00244                     {
00245                         case '\"':
00246                             openApostrophe = false;
00247                             break;
00248                         case '\\':
00249                             escaping = true;
00250                             break;
00251                     }
00252                 }
00253             }
00254             else
00255             {
00256                 escaping = false;
00257                 escaped = true;
00258             }
00259 
00260             return c;
00261         }
00262 
00263 /// <summary>
00264 /// Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.
00265 /// </summary>
00266 /// <param name="reader">The <see cref="TextReader"/> to read from.</param>
00267 /// <param name="eof">A <see cref="bool"/> indicating whether we have arrived at the end of the
    file.</param>
00268 /// <returns>The next word.</returns>
00269         public static string NextWord(this TextReader reader, out bool eof)
00270         {
00271             Contract.Requires(reader != null);
00272 
00273             StringBuilder sb = new StringBuilder();
00274 
00275             int c = reader.Read();
00276 
00277             while (c >= 0 && Char.IsWhiteSpace((char)c))
00278             {
00279                 c = reader.Read();
00280             }
00281 
00282             if (c >= 0)
00283             {
00284                 sb.Append((char)c);
00285             }
00286 
00287             if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00288             {
00289                 eof = false;
00290                 return sb.ToString();
00291             }
00292 
00293             c = reader.Peek();
00294 
00295             while (c >= 0 && !Char.IsWhiteSpace((char)c))
00296             {
00297                 if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00298                 {
00299                     break;
00300                 }
00301                 c = reader.Read();
00302                 sb.Append((char)c);
00303                 c = reader.Peek();
00304             }
00305 
00306             if (c < 0)
00307             {
00308                 eof = true;
00309             }
00310             else
00311             {
00312                 eof = false;
00313             }
00314 
00315             return sb.ToString();
00316         }
00317 
00318 /// <summary>
00319 /// Reads the next word, taking into account whitespaces, square brackets, commas and semicolons.
00320 /// </summary>
00321 /// <param name="reader">The <see cref="TextReader"/> to read from.</param>
00322 /// <param name="eof">A <see cref="bool"/> indicating whether we have arrived at the end of the
    file.</param>
00323 /// <param name="headingTrivia">A string containing any whitespace that was discarding before the
```

```
      start of the word.</param>
00324 /// <returns>The next word.</returns>
00325         public static string NextWord(this TextReader reader, out bool eof, out string headingTrivia)
00326         {
00327             Contract.Requires(reader != null);
00328 
00329             StringBuilder sb = new StringBuilder();
00330 
00331             StringBuilder headingTriviaBuilder = new StringBuilder();
00332             StringBuilder trailingTriviaBuilder = new StringBuilder();
00333 
00334             int c = reader.Read();
00335 
00336             while (c >= 0 && Char.IsWhiteSpace((char)c))
00337             {
00338                 headingTriviaBuilder.Append((char)c);
00339                 c = reader.Read();
00340             }
00341 
00342             headingTrivia = headingTriviaBuilder.ToString();
00343 
00344             if (c >= 0)
00345             {
00346                 sb.Append((char)c);
00347             }
00348 
00349             if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00350             {
00351                 eof = false;
00352                 return sb.ToString();
00353             }
00354 
00355             c = reader.Peek();
00356 
00357             while (c >= 0 && !Char.IsWhiteSpace((char)c))
00358             {
00359                 if ((char)c == '[' || (char)c == ']' || (char)c == ',' || (char)c == ';')
00360                 {
00361                     break;
00362                 }
00363                 c = reader.Read();
00364                 sb.Append((char)c);
00365                 c = reader.Peek();
00366             }
00367 
00368             if (c < 0)
00369             {
00370                 eof = true;
00371             }
00372             else
00373             {
00374                 eof = false;
00375             }
00376 
00377             return sb.ToString();
00378         }
00379     }
00380 }
```

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))
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 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>, 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 }
```

8.9 TreeNode.Comparisons.cs

```
00001 using PhyloTree.Extensions;
00002 using System;
00003 using System.Collections.Generic;
00004 using System.Linq;
00005 using System.Text;
00006
00007 namespace PhyloTree
00008 {
00009     public partial class TreeNode
00010     {
00011 /// <summary>
00012 /// Computes the Robinson-Foulds distance between the current tree and another tree.
00013 /// </summary>
00014 /// <param name="otherTree">The other tree whose distance to the current tree is computed.</param>
00015 /// <param name="weighted">If this is <see langword="true" />, the distance is weighted based on the
      branch lengths; otherwise, it is unweighted.</param>
00016 /// <returns>The Robinson-Foulds distance between this tree and the <paramref
      name="otherTree"/>.</returns>
00017         public double RobinsonFouldsDistance(TreeNode otherTree, bool weighted)
00018         {
00019             return RobinsonFouldsDistance(this, otherTree, weighted);
00020         }
00021
00022 /// <summary>
00023 /// Computes the Robinson-Foulds distance between two trees.
00024 /// </summary>
00025 /// <param name="tree1">The first tree.</param>
00026 /// <param name="tree2">The second tree.</param>
00027 /// <param name="weighted">If this is <see langword="true" />, the distance is weighted based on the
      branch lengths; otherwise, it is unweighted.</param>
00028 /// <returns>The Robinson-Foulds distance between <paramref name="tree1"/> and <paramref
      name="tree2"/>.</returns>
00029         public static double RobinsonFouldsDistance(TreeNode tree1, TreeNode tree2, bool weighted)
00030         {
00031             if (tree1 == null)
00032             {
00033                 throw new ArgumentNullException(nameof(tree1), "The first tree cannot be null!");
00034             }
00035
00036             if (tree2 == null)
00037             {
00038                 throw new ArgumentNullException(nameof(tree2), "The second tree cannot be null!");
00039             }
00040
00041             List<(string[], string[], double)> splits1 = (from el in tree1.GetSplits()
00042                                                            select (
00043                                                            (from el1 in el.side1 where el1 == null ||
      !string.IsNullOrEmpty(el1.Name) select el1 == null ?  "@Root" :  el1.Name).ToArray(),
00044                                                            (from el2 in el.side2 where el2 == null ||
      !string.IsNullOrEmpty(el2.Name) select el2 == null ?  "@Root" :  el2.Name).ToArray(),
00045                                                            el.branchLength
00046                                                            )).ToList();
00047
```

```
00048             List<(string[], string[], double)> splits2 = (from el in tree2.GetSplits()
00049                                                            select (
00050                                                            (from el1 in el.side1 where el1 == null ||
      !string.IsNullOrEmpty(el1.Name) select el1 == null ?  "@Root" :  el1.Name).ToArray(),
00051                                                            (from el2 in el.side2 where el2 == null ||
      !string.IsNullOrEmpty(el2.Name) select el2 == null ?  "@Root" :  el2.Name).ToArray(),
00052                                                            el.branchLength
00053                                                            )).ToList();
00054
00055             static bool AreSameSplit((string[], string[], double) split1, (string[], string[], double)
      split2)
00056             {
00057                 if (split1.Item1.Length == split1.Item2.Length || split2.Item1.Length ==
      split2.Item2.Length)
00058                 {
00059                     if (split1.Item1.Length == split1.Item2.Length && split2.Item1.Length ==
      split2.Item2.Length)
00060                     {
00061                         return AreSameSplit2(split1.Item1, split1.Item2, split2.Item1, split2.Item2)
      || AreSameSplit2(split1.Item1, split1.Item2, split2.Item2, split2.Item1);
00062                     }
00063                     else
00064                     {
00065                         return false;
00066                     }
00067                 }
00068
00069                 string[] split11, split12;
00070
00071                 if (split1.Item1.Length > split1.Item2.Length)
00072                 {
00073                     split11 = split1.Item1;
00074                     split12 = split1.Item2;
00075                 }
00076                 else
00077                 {
00078                     split11 = split1.Item2;
00079                     split12 = split1.Item1;
00080                 }
00081
00082                 string[] split21, split22;
00083
00084                 if (split2.Item1.Length > split2.Item2.Length)
00085                 {
00086                     split21 = split2.Item1;
00087                     split22 = split2.Item2;
00088                 }
00089                 else
00090                 {
00091                     split21 = split2.Item2;
00092                     split22 = split2.Item1;
00093                 }
00094
00095                 return AreSameSplit2(split11, split12, split21, split22);
00096             }
00097
00098
00099             static bool AreSameSplit2(string[] split11, string[] split12, string[] split21, string[]
      split22)
00100             {
00101                 if (split11.Length != split21.Length || split12.Length != split22.Length)
00102                 {
00103                     return false;
00104                 }
00105
00106                 HashSet<string> union2 = new HashSet<string>(split12.Length);
00107
00108                 for (int i = 0; i < split12.Length; i++)
00109                 {
00110                     union2.Add(split12[i]);
00111                     union2.Add(split22[i]);
00112                 }
00113
00114                 if (union2.Count != split12.Length)
00115                 {
00116                     return false;
00117                 }
00118
00119                 HashSet<string> union1 = new HashSet<string>();
00120
00121                 for (int i = 0; i < split11.Length; i++)
00122                 {
00123                     union1.Add(split11[i]);
00124                     union1.Add(split21[i]);
00125                 }
00126
00127                 return union1.Count == split11.Length;
```

```
00128             }
00129
00130
00131             bool?[] matched1 = new bool?[splits1.Count];
00132             bool?[] matched2 = new bool?[splits2.Count];
00133
00134             for (int i = 0; i < splits1.Count; i++)
00135             {
00136                 matched1[i] = false;
00137
00138                 for (int j = 0; j < splits2.Count; j++)
00139                 {
00140                     if (AreSameSplit(splits1[i], splits2[j]))
00141                     {
00142                         matched1[i] = true;
00143                         matched2[j] = true;
00144                         break;
00145                     }
00146                 }
00147             }
00148
00149             for (int j = 0; j < splits2.Count; j++)
00150             {
00151                 if (matched2[j] == null)
00152                 {
00153                     matched2[j] = false;
00154
00155                     for (int i = 0; i < splits1.Count; i++)
00156                     {
00157                         if (AreSameSplit(splits1[i], splits2[j]))
00158                         {
00159                             matched2[j] = true;
00160                             break;
00161                         }
00162                     }
00163                 }
00164             }
00165
00166             if (!weighted)
00167             {
00168                 return matched1.Count(x => x == false) + matched2.Count(x => x == false);
00169             }
00170             else
00171             {
00172                 double tbr = 0;
00173
00174                 for (int i = 0; i < matched1.Length; i++)
00175                 {
00176                     if (matched1[i] == false)
00177                     {
00178                         tbr += splits1[i].Item3;
00179                     }
00180                 }
00181
00182                 for (int i = 0; i < matched2.Length; i++)
00183                 {
00184                     if (matched2[i] == false)
00185                     {
00186                         tbr += splits2[i].Item3;
00187                     }
00188                 }
00189
00190                 return tbr;
00191             }
00192         }
00193     }
00194 }
```

8.10 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 the 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
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 }
```

8.11 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 - 7));
00194                     return (numberOfCherries - mu) / Math.Sqrt(sigmaSq);
00195             }
00196
00197             return double.NaN;
00198         }
00199     }
```

```
00200 }
```

Index