Evolving Evolutionary Tree Computation

Ross Penna
Bemidji State University
ross_penna@yahoo.com

Abstract

The need for computational analysis on evolutionary hypotheses (trees) has created the need for a set of classes for representing evolutionary trees. This paper describes a new collection of classes based upon a previously developed set of classes [1]. The new classes were developed using the object-oriented paradigm to provide a flexible basis for manipulating evolutionary trees, especially in large numbers. To demonstrate the classes, two applications, treefilter and NNI, were also developed. These tree-based applications are used to ascertain the interrelationships among groups of trees.

Introduction

With the progress in the fields of molecular biology and genomics, nucleotide sequence data is becoming a common data set for computational analysis. Programs, like TrExML [2] and others mentioned in that paper, have been developed that are capable of creating large numbers of evolutionary hypotheses from this sequence data. These hypotheses are represented as evolutionary trees. Due to their hypothetical nature, computational methods for the analysis of trees are needed. This project intends to support such computations. A set of base classes was developed to allow an object to acquire the traits of a hypothesis, or tree. The development is intended to be flexible as the classes allow for several different representations of the tree, supporting computation in different ways. However, the standard and most common representation, or format, for the trees is a comma-delimited, parenthesis-grouped string of species names, often called a topology. The TrExML program, and others like it, produces trees in this form. This representation, the bifurcating structure, and the ID representation form the essence of the base classes, which are described next. Descriptions of treefilter, a program for finding and removing duplicates from a file of trees, and NNI, a program for computing all of the nearest neighbor interchanges of a tree, follow the base classes' discussion.

Base classes

The purpose of the base classes is to provide fundamental representations for an evolutionary tree. In doing so, the standard representation is taken and manipulated to create the other forms that are more conducive to computation. The classes were developed with an object-oriented methodology in C++ and drew from a set of previously developed tree classes [1]. Throughout the discussion on the base classes, the topology *(A,B,(C,D))* will be used as an example.

intMap object

The first base class, which is a significant departure from the original implementation of trees, is the implementation of the intMap object. The purpose of an intMap object is to introduce a compact, uniform naming convention. It replaces the species names with integers. The object creates a mapping between the species names and a set of integers. The mappings were originally stored internally as a static private data member in each tree object, so that any workings of the intMap object were dealt with in the tree object. The decoupling of the intMap object from the tree object allows for the processing of trees on different species with in the same program.

intMap class

The two main functions in the intMap class support fast lookup of a species name given an integer, and vice versa. There also exists a function to print out the list of integers and the species that they index in the format

Index: species name

The printed index is primarily used for debugging purposes. For the example topology, the mapping printout would look like:

```
0 : A
1 : B
2 : C
3 : D
```

ID object

Another base class that diverges considerably from the original implementation is the ID class. The ID, like the intMap object, was stored internally in the tree object, and was therefore tightly coupled with the tree's implementation. The ID object generates a representation based on a canonical ordering of evolutionary trees. It is the ID of a tree that is used in determining equality and other relational operations. The ID object is constructed from the standard representation of a tree where the species names have been replaced with integers (obtained from the intMap mapping).

ID creation algorithm

An ID is an ordering of the integers, unique to each unique tree, constructed by an ordered replacement of leaf pairs in the bifurcating structure until only three leaves remain. Because of the destructive nature of the algorithm, it is performed on its own copy of the tree's bifurcating structure. The Standard Template Library pair structure was used to support the ID creation. The algorithm takes the bifurcating structure and identifies each of the existing leaf pairs, and then stores each in a pair structure. The pair structures are ordered in ascending order based on the larger of the two leaf values in each pair. The first, or smallest pair is replaced in the structure by a new leaf. The pair is then added to the ID and removed from the list of pairs. In replacing the pair, the node holding it is deleted from the bifurcating structure. If adding the new leaf creates a new pair, this pair is added to the list of pairs. This process continues until the only remaining node in the structure is the rootTopoNode.

ID class

The ID class includes functions for copying and printing an ID, relational operators for comparing IDs, and a method to return the size of an ID. IDs are printed as a simple, parentheses-enclosed, comma-separated string of integers. For the example topology, the ID would be displayed as

(0,1,2,3).

bifurcating structure object

Evolutionary trees are typically treated as unrooted, as in figure 1.

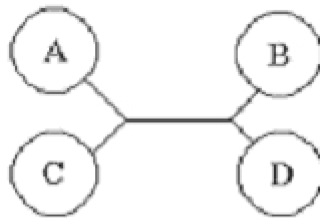

Figure 1: evolutionary tree

A, B, C, and D are subtrees with a minimum of one leaf. The notion of a root node is introduced as a convenience, giving the computational algorithms a starting place in the tree. The topoNode object, along with the rootTopoNode object, implements the bifurcating structure representation of an evolutionary tree. The new implementation differs little from the original. As with the original, each node in the structure has three neighbors. Maintaining this structure invariant in the root requires the rootTopoNode object. The invariant also makes the choice for the root node of the tree arbitrary. The bifurcating structure is useful for several immediate concerns, including the construction of the ID of a tree and the NNI operation. For the example topology, the structure would be:

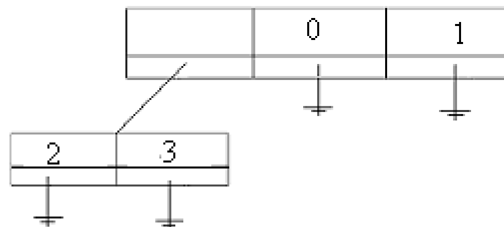

Figure 2: bifurcating structure

topoNode and rootTopoNode classes

Each topoNode in the bifurcating structure represents an internal node in the evolutionary tree. A topoNode has four data members: lName, rName, lChild, and rChild (figure 3).

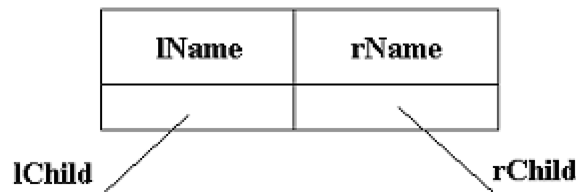

Figure 3: topoNode structure

Both lName and rName are integers that correspond to species names via the aforementioned intMap object. Each node is maintained such that lName < rName, thus creating an invariant for the topoNode class. The lChild and rChild are pointers to other topoNodes. From this perspective, the structure imitates a standard binary tree, however, it is in the rootTopoNode that the structures are set apart. The rootTopoNode (figure 4) is actually a specialized topoNode in that it has the same data members, but adds mName and mChild.

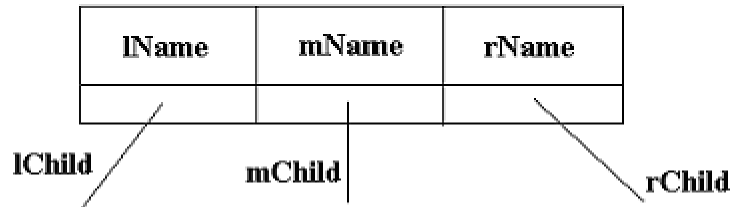

Figure 4: rootTopoNode structure

The relationship between mName and mChild is identical to those in the topoNode class. Also similar to the topoNode class is the invariant that lName < mName < rName. The three pairs of data members allow the representation of the chosen root node to maintain the structure invariant (each node has three neighbors). This set of classes is considerably larger than the other base classes, providing functions for manipulating each of the data members, in addition to finding the parent of a node that contains a given pair of integers. The latter also allows for deleting that Child that contains the pair and replacing the corresponding Name with a new name (this function is for use in the ID creation algorithm).

tree object

The tree object is used to represent evolutionary trees for computational purposes and is implemented in the tree class. It brings the different representations of a tree into a single object. Along with the ID and the bifurcating structure, the object contains the topology, and a similar representation with integers in-place of the names. All representations, aside from the common string representation, are constructed using the intMap-referenced integers for the species names. By maintaining each of the representations, the tree object allows for computation on

trees in several different ways. Some computations are easier or more logical to implement using representations other than the topology.

tree class

The tree class contains five data members: a pointer to the bifurcating structure representation, a pointer to the ID representation, the topology, the string representation with integers, and the number of leaves in the tree. The constructors for the tree class, aside from the default constructor, all require an intMap object pointer as a parameter. The default constructor has an empty implementation due to the need for a common intMap object. By maintaining a common intMap object, external to the tree object, many trees may share a similar mapping. This is essential when comparing trees. Trees using different mappings can give incorrect results when dealing with equality and other relational operations because the relationships are based on the IDs, which are computed from the mapped integers from the intMap object. The tree class supports relational operators as well as methods for accessing each representation and printing the topology.

ftree object

The tree object represents the essence of an evolutionary tree and is useful for performing computations on a single tree. However, the ftree object is needed for processing an entire file of trees. An ftree object is a tree object that contains additional information necessary for performing computations on the relationships among trees in a file. This information includes its relative position in the file, as well as the positions of other topologies in the file that represent the same tree.

ftree class

The ftree class is implemented as a descendent of the tree class. However, the ftree class includes data members to hold the position of the tree in a file and a list of the positions of all the trees in the file that are identical to it. The additional functions in the ftree class support accessing and manipulation of the new data members. In the original implementation, the tree class held the data members and functions of both tree and ftree classes. However, neither the file position nor the list of identical trees is essential to the fundamental nature of an evolutionary tree. Rather, they are merely pieces of data utilized in the computation of trees in a set of specific family of applications, primarily those that deal with the interrelationship of multiple trees within a file.

Tree-based applications

With the development of the base classes, applications can be constructed that allow computations to be performed on trees. One area of tree computation focuses on the interrelationship of multiple trees.

treefilter

One such tree-based application that deals with interrelationships is the treefilter program. The treefilter program is used to process a file of trees and find each unique tree. The program uses a common intMap object for all trees in the file and tracks each tree's relative position.

input and computation

The input file consists of a series of topologies, terminated by a semicolon. The program reads in each topology and uses it along with the common intMap object and the position of the topology in the file to construct an ftree object. The new ftree object is then compared against the set of ftree objects already extracted from the file to find a match. If a match is found, the position of the new object is added to the matching objects list of identical trees. If a match is not found, the new ftree object is inserted into the set of extracted trees.

output

Once the file has been completely processed, the program writes the appropriate output files. The options for output files include files with extensions of .unique, .references, and .frequencies. The file with the .unique extension is a default output and always written. Each unique tree's topology is printed in this file. The .frequencies file is also default, but can be prevented from being written by using the –f option in command line. This file gives the position in the file of each of the unique trees, along with the number of occurrences of each in the file. Unlike the previous two, the .references file is not written by default. However, the –r option in the command line enables the writing of the file. The .references lists each unique topology along with its position in the file. This file is necessary for meaningful analysis of the .frequencies file, in that the file positions mean nothing without knowing which tree each position refers to.

NNI (Nearest Neighbor Interchange)

The NNI program is another in the family of applications that deal with the interrelationships of files of trees, but it is much more computationally intense

than treefilter. Typically, the nearest neighbor interchange method is used as a metric for measuring how similar or dissimilar two trees are. The degree of similarity is based on the distance between two trees, or how many nearest neighbor interchanges are required to transform one tree into the other. However, this program looks specifically for those pairs of trees that are precisely one nearest neighbor interchange apart.

nearest neighbor interchange algorithm

The first step in the nearest neighbor interchange is locating an internal edge. An internal edge is an edge on which there are four adjacent subtrees (figure 5).

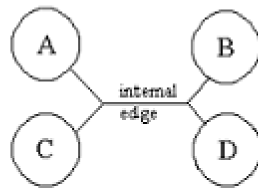

Figure 5: internal edge of an evolutionary tree

Then, a subtree (A) is selected from one end and a subtree (B) is selected from the opposite end, and they are swapped. For each internal edge, there are two possible nearest neighbor interchanges, A and B, and A and D. C and A are interchangeable, as are B and D, and result in an identical tree when switched.

input and computation

The input file for NNI is of the same format as treefilter. In fact, NNI utilizes a function similar to treefilter to extract the unique trees from the file – instead of writing the trees to files as treefilter does; the function returns a set of pointers to ftree objects. Once all the trees have been extracted from the file, the program computes all possible NNIs for each tree and stores them. Then, for each computed NNI, the original trees are searched for a match. If a match is found, then the matching tree is said to be one nearest neighbor interchange away from the original tree. The object of the NNI program is to discover all such relationships among the trees.

output

The NNI program writes output to two different files. One file is the .nni file, and this file contains a graph that displays the trees that are a single nearest neighbor interchange apart (figure 6).

```
1:     2  3          ;
2: 1      3          ;
3: 1  2              ;
4: 1           5  6  ;
5: 1        4        ;
6:             5     ;
```

Figure 6: graph of trees one NNI apart

For example, in figure 6, the first tree in the file is one NNI away from the second and third trees in the file. The second file has an extension of .nnireference. Like the treefilter .references file, this file contains the topology of each tree, preceded by its position in the file. This file serves the same purpose for the .nni file as the .references file served for the .frequencies file in treefilter.

Implementation improvements

The goals for rewriting the base classes and the applications were two-fold. The first objective was to increase flexibility by utilizing object-oriented design principles in the redesigning of the classes and applications. The second was to reduce runtimes and improve the reliability of the software by utilizing both intuitive improvements and efficient data structures.

object-oriented programming

By utilizing object-oriented design and programming techniques, such as inheritance and delegation, the classes were made more reliable, maintainable, robust, and extensible. With the isolation of objects into separate classes, the coding is more extensible and robust in that each class is only responsible for a certain object; each class is more intuitive and easier to reuse. The code has proven to be more reliable by successfully processing input that caused errors in the original implementation. The maintainability has improved by virtue of making error tracking and correction easier via the functionally separate modules of code. The object-oriented approach is most evident in the base classes. Originally, what are now the intMap and ID objects were imbedded in the tree object. However, by creating them as separate objects, their implementation is decoupled from the implementation of trees. The decoupling allows trees to delegate the responsibilities for creating and using a valid intMap or ID object to the intMap or ID class respectively. This delegation improves the reliability of the objects created due to the creation process being autonomous. It is not affected by operations irrelevant to the object that are executed in the tree class because the object's value can only be affected by the provided class functions. The class functions do not allow for operations that compromise the integrity of the object once it is created, and the creation operations have built in error checking. Another object-oriented induced change was the development of the

ftree class. By using inheritance to develop a subclass that was useful for computations involving a file of trees, the ftree class, the tree class is able to maintain the qualities that define an evolutionary tree and only those qualities.

Another feature of the object-oriented methodology exploited was the reusing of code. Primarily, this involved using the Standard Template Library for data structures and algorithms instead of creating new ones. The three most significant examples were using the set and algorithm classes and the pair structure. In using these, the implementation is more reliable in that these pieces of code are thoroughly tried and tested.

data structure efficiency

The second purpose was to utilize data structures and algorithms to reduce computation and overall execution times. This was accomplished in several classes and both treefilter and NNI in a couple of different ways. In the treefilter and NNI applications, vector usage was replaced with sets. In the original implementations, the vectors were filled, then sorted, and finally searched. The sorting operation, which runs in O(N Log N) time, dominates the collection of operations. By utilizing sets, the filling and sorting operations were combined into a single operation and the use of the binary search algorithm is forced, resulting in a collection of operations also running in O(N Log N) time. While this appears to be the same as the previous analysis, the new operations execute in less time due to the combining of the two operations. Another example of utilizing data structures to decrease runtime was the use of a (min) heap (a priority queue from the STL) in the ID class implementation to hold the pairs. Because the ID creation algorithm was based on always consuming the smallest leaf pair available, the ability to retrieve the smallest item in constant time is advantageous. Additionally, allowing for removing and inserting values in O(Log N) time, giving an overall runtime in O(N Log N), made the heap ideal. Originally implemented as a list, the pair operations were dominated by the sorting algorithm, which ran in O(N Log N) time. Again, the combining of the insertion and sorting operations results in faster runtimes.

Closing

Biological data in the form of DNA or RNA nucleotide sequences is often used to organize species in evolutionary trees that provide hypotheses for the possible evolutionary relationships among the species. To support computational methods on these hypotheses (or trees), a set of base classes was developed. By applying object-oriented software development techniques and improving data structure efficiency in the redevelopment, the software is more robust, maintainable, secure, reliable, and extensible. Nevertheless, object-oriented coding does lead to some quirky situations, such as the lack of a (min) heap implementation in the

STL. Although there is a priority queue adapter class, utilizing it forces the heap to be a max heap. Overcoming issues such as this sometimes leads to peculiar coding, such as the reversing of the relational operators (except for equality and inequality) for the pairs in the ID class. Another difficulty faced in the development of the base classes is the foreknowledge of a particular application (or applications) that would utilize the classes. Often, coding would start to become tailored to the specific uses of those programs. Tailored coding severely degrades the reusability, robustness, and extensibility of the code, all of which are paradigms of object-oriented programming and aspirations of this project. Future goals include adding additional error checking and handling to the base classes, as well as further application development. To obtain the software, visit http://whitetail.bemidjistate.edu/software.

References

1. Wolf, Marty J. (1999). tree.h, tree.cc, topoNode.h, topoNode.cc, options.h, options.cc, treefilter.h, treefilter.cc [Computer Software]. Bemidji, MN.

2. Wolf, Marty J., Easteal, Simon, Kahn, Margaret, McKay, Brendan D., Jermiin, Lars S. (2000) TrExML-a maximum-likelihood program for extensive tree-space exploration. *Bioinformatics* 16: 383-394

Acknowledgements

For his assistance in the code-development stage as well as the preparations of these materials, I would like to thank Marty Wolf.