Computer Science 241

(Pair) Program 3 (15 points)
Due Sunday, June 5th, 2016 at 11:59 PM

Overview

For the third programming assignment you will work with an assigned partner to create a program that builds and manipulates phylogenetic trees. Specifically, it will build a tree from a set of aligned amino acid sequences - one for each species in a set of species. By comparing the differences between these sequences, your program will be able to infer a hierarchical structuring of the species reflective of the evolutionary relationships between them. Species live on the leaves of the phylogenetic tree, while non-terminals represent inferred evolutionary predecessors. Each edge from parent to child has an associated weight that indicates to what extend the child deviates from the (inferred) parent. You will construct the phylogenetic tree using the pairwise distances between the sequences for each species and a simple "agglomerative" clustering algorithm. Once generated, you will be able to query the trees to find common ancestors, measure evolutionary distance between species and compute basic properties of the tree and its nodes.

You and your partner will implement this functionality in a new class: `PhyloTree`. A skeleton `.java` file with several public and private dummy methods have been provided. The pre- and post-conditions for each method are specified; it will be your job to complete all of the provided methods according to those conditions. Additionally, these will rely upon two classes that have already been implemented for you: `PhyloTreeNode`, which defines a node type which will be the building block of your `PhyloTree` objects, `Species`, which defines a type to represent a species. A third class, `MultiKeyMap` has also been provided for your use, although it is not required that you use it.

Pair Programming

As in Program 1, this is a pair programming assignment. You must work together with your assignment partner to design, develop and debug your work for Program 3. Recall that at any given moment when coding, one person (the "driver") will be writing the code, while the other will be reviewing and offering suggestions (the "navigator"). You and your partner should switch roles frequently. To succeed, you must both be on the same page and both be engaged in the development of the program. No designing, development or debugging can be done without both partners present. The goal of this approach is to increase your productivity and expose you to new approaches to developing code. Often the same bug that would take you three bleary-eyed hours to hunt down alone can be found within minutes by a second pair of eyes. Be patient: there is plenty of time allotted to complete this assignment as long as you proceed at a steady pace. Ask for help from me or the department tutors if you need it.

I will assign your partner via Canvas. You will need to coordinate with your partner to find times when you can both be present, and contact me ASAP if you cannot find sufficient overlap in your schedules to collaborate.

Development and Testing

As in past programs, a driver program (this time named `Program3`) has been provided for you to use during development and debugging. Do not modify Program3's code. This program drives your `PhyloTree` class, reading test input alignment files (in "FASTA" format), generating the inferred trees, querying for all pairwise evolutionary distances, and reporting various statistics of the trees. It accomplishes this by generating new instances of type `PhyloTree`. Because `Program3` interacts with your classes, you must not change the method header for any of the public methods in either class (including adding new `throws` elements). You must supply this program with two arguments: the name of an "alignment list" (just a list of alignment files) and the name of an (already created) output directory, where the constructed tree and pairwise evolutionary distances will be written. An example usage is:

```
C:> java.exe Program3 someListFile.txt someOutputDirectory
```

or (in Linux):

```
$ java Program3 someListFile.txt someOutputDirectory
```

Grading

Submitting your work

Submit over Canvas a zip archive that contains the following material:

- `PhyloTree.java`
- `MultiKeyMap.java` (if modified)
- Your write-up
- At least two new test input files you have created
  - Name your new test files `test1.fasta`, `test2.fasta`, etc.
  - Create a new test list named `test.list` that contains your test fasta files
- Any other source code needed to compile your program / classes

Upon checking out your files, I will replace your versions of `Program3.java`, `PhyloTreeNode.java` and `Species.java` with my originals, compile all `.java` files, run `Program3` against a series of test alignment files, analyze your code, and read your documentation.

Points

This assignment will be scored by taking the points earned and subtracting any deductions. You can earn up to 15 points:

Component	Points
Write Up & Test Cases	1.5
`constructor`	0.6
`getOverallRoot`	0.4
`toString` (both)	0.9
`toTreeString` (both)	0.9
`getHeight`	0.4
`getWeightedHeight`	0.4
`countAllSpecies`	0.4
`getAllSpecies`	0.4
`findTreeNodeByLabel` (both)	0.6
`findLeastCommonAncestor` (both)	1.2
`findEvolutionaryDistance`	1.2
`buildTree`	3
`nodeDepth`	0.6
`nodeHeight`	0.6
`weightedNodeHeight`	0.6
`loadSpeciesFile`	0.6
`getAllDescendantSpecies`	0.7
Total	**15**

Several deductions may decrease your score, with penalties up to those listed below:

You may also have deductions from your score for

- Poor code style (e.g. bad indentation, non-standard naming conventions)
- Inadequate versioning
- Errors compiling or running

Write-Up & Test Cases

In one or two pages provide a write-up of your implementation. Submit your writeup as a **plaintext** file[1] named **writeup.txt**. Please include all of the following:

1. The name of both group members.
2. An acknowledgement and discussion of any parts of the program that are not working. Failure to disclose obvious problems will result in additional penalties.
3. **An acknowledgment and discussion of any parts of the program that appear to be inefficient (in either time or space complexity).**
4. A discussion of the portions of the assignment that were most challenging. What about those portions was challenging?
5. A discussion on how you approached testing that your program was correct and asymptotically efficient. What did `test1.fasta` test? What did `test2.fasta` test?
6. Any other thoughts or comments you have on the assignment and its implementation.

[1]E.g. created by Notepad, or vim, or emacs.

Algorithms

Building the Tree

You will construct the tree using an agglomerative clustering method. With agglomerative clustering, you start with a set of items, and iteratively combine them into larger groups until there are a specified number of groups. In the case of phylogenetic tree construction, you begin with a forest of trees, where each tree is just a single node, one for each species, and then you iteratively build trees into larger trees until you have only one tree left. In each step of the clustering process, you will combine the two trees that have the smallest *distance* to each other, since these are the most likely to be closely genetically related.[2] Conceptually, the algorithm is simple. It follows these steps:

- Create a forest containing n trees, one for each of species, where each tree is just a single node.
- Compute the pairwise distances between each of the trees using `Species.distance`. **Once you have computed these distances, you should no longer view your nodes as species; instead, view them just as trees in a forest with distances between them.**
- While there is more than one tree in the forest do
  1. Find the two trees that have the minimum distance to each other. Of these two trees, let's call the tree that is alphabetically earlier to be T_1 and the other T_2.
  2. Remove T_1 and T_2 from the forest, and then create and add a new tree T_{new} whose left child is T_1 and right child is T_2.
  3. Now we need to compute the distances between the new tree T_{new} and every other tree still in the forest. One by one, for every other tree T_{other} in the forest, calculate the distance from the new tree T_{new} to the other tree T_{other} as follows:

$$D(T_{new}, T_{other}) = \frac{|T_1|}{|T_1| + |T_2|} D(T_{other}, T_1) + \frac{|T_2|}{|T_1| + |T_2|} D(T_{other}, T_2)$$

Here the "absolute value" notation of a tree means the number of species in that tree (i.e. $|T_i|$ denotes the number of species in the tree T_i). Note that only leaves count as species! All this is doing is coming up with a distance between new tree T_{new} and each other tree in the forest T_{other} as a weighted combination of the distances of T_{new}'s children (T_1 and T_2) to T_{other}. For example, if T_1 denotes a tree of mice species, T_2 denotes a family of rat species, T_{new} denotes an inferred predecessor of both, and T_{other} denotes a family of lion species, then $D(T_{other}, T_1)$ will be large and $D(T_{other}, T_2)$ will be large, which means that their weighted combination, $D(T_{new}, T_{other})$, will be large. Although the algorithm does not require it, we will follow convention that the label for the root of each new node T_{new} should be its left child's label followed by the plus sign followed by its right child's label (no spaces). This convention is only for helping debugging; the names *should not* be used for anything other than printing (e.g. you cannot use them to help locate a common ancestor). The implementation of this algorithm can be a bit tricky. You may use objects of the type `MultiKeyMap`, if you find it useful.

[2]Important note: if multiple pairs of trees tie for smallest distance, then take the one that is alphabetically least, using the naming conventions described in this section.

File Formats

Input Fasta Format

The provided alignments are in a simple but standard format, known as "FASTA". Each species has a header line, which begins with the `>` character. After `>`, there are up to seven fields, each separated by the pipe/bar character: |. These fields correspond to

1. NCBI taxon ID
2. database of origin
3. gi
4. NCBI gi number
5. ref
6. NCBI accession number
7. species name

We will only care about the last field: the species name, which will be the name used in our `Species` object. All other fields can be ignored. Some species in the input data will not have all seven fields; we will ignore these species.

After each header are several lines for the aligned sequence. All such lines should be read and "glued" back together into one long sequence with no spaces or newlines. We will split these sequences into arrays of Strings, with each String storing a single character in the sequence; thus, our array of strings will contain as many entries as there are characters in the sequence string.

As an example, consider the following toy FASTA file:

```
>DONT|CARE|WHAT|THESE|FIELDS|ARE|Faecaibacterium_prausnitzii_A2-165
-----QKNSYQWFL--------DEGLKEVF
--------------GTIEDYTGNLA-----
>DONT|CARE|WHAT|THESE|FIELDS|ARE|Campyobacter_coi_RM2228
----DFSNISKQ------SGIE------KV
--------------IRECMER---------
```

This file defines two species: `Faecaibacterium_prausnitzii_A2-165`, which has the following 60 character sequence

```
-----QKNSYQWFL--------DEGLKEVF--------------GTIEDYTGNLA-----
```

and `Campyobacter_coi_RM2228`, which has the following 60 character sequence

```
----DFSNISKQ------SGIE------KV--------------IRECMER---------
```

The distance between these two sequences is simply the number of characters where they differ, divided by their length (in this case 60). General FASTA files will have many more species defined, and longer sequences per species.

toString Tree Format

`PhyloTree`'s `toString` method will produce a "graphical" representation of your tree. It is similar to the `printSideways` method defined on pages 1040 and 1041 of the Java textbook. Each node will be printed, with an indentation proportional to its weighted depth. To make the tree more readable, nodes will actually be printed during reverse inorder traversal. The pseudo code is simple:

```
PrettyPrint(node)
    PrettyPrint the node's right child
    Print k periods, and then print node with PhyloTreeNode's toString
    PrettyPrint the node's left child
End PrettyPrint
```

To compute k, use the following equation:

`k = printingDepth * (weightedDepth(node) / weightedHeightOfTree)`

Where `printingDepth` is a field in the `PhyloTree` class that specifies how many periods to print for the *deepest* node in the tree. `weightedDepth(node)` is the sum of the weights on the edges from the root to `node`. Finally, `weightedHeightOfTree` is the maximum `weightedDepth` of any node in the tree. For example, consider the following toy problem with six species, each characterized by a binary string of length six:

```
>0|1|2|3|4|5|A
000000
>0|1|2|3|4|5|B
000001
>0|1|2|3|4|5|C
100111
>0|1|2|3|4|5|D
100110
>0|1|2|3|4|5|E
001011
>0|1|2|3|4|5|F
111111
```

The corresponding output tree (with `printingDepth` at 40) would be:

```
..................................F
.....................[NONTERM 0.21]
........................................D
..................................[NONTERM 0.08]
........................................C
[NONTERM 0.32]
..................................E
.....................[NONTERM 0.21]
........................................B
..................................[NONTERM 0.08]
........................................A
```

toTreeString Output Format

You will be writing your data in a format that can be displayed nicely with an external tool. In this case the external tool will be `FigTree`[3], which you can download and "install" (just unpack) under your home directory. It reads `.tree` files. In this format, a leaf node is written as `label:weight`, where label is the label of the node and weight is the cost on the edge between the node and its parent, printed to five decimal places. Non-leaf nodes are written as `(subtree1,subtree2):weight`, where subtrees 1 and 2 are either leaf or non-leaf nodes, and weight is the weight from this node to its parent (also printed to five decimal places). For the root node, the format is just `(subtree1,subtree2)` without any weight specified. There are no spaces or newlines. The basic pseudo-code is:

```
TreePrint(node)
    if leaf then
        Print label:weight
    else
        Print "("
        TreePrint the node's right child
        Print ","
        TreePrint the node's left child
        Print "):weight" // weight part is omitted for root
fi
End TreePrint
```

In tree format, the toy example discussed in the previous section is:
((F:0.20833,(D:0.08333,C:0.08333):0.20833):0.32407,(E:0.20833,(B:0.08333,A:0.08333):0.20833):0.32407)

To display a tree file using FigTree, cd to the FigTree directory (e.g. FigTree_v1.4.0). From that directory, run

```
bash bin/figtree ~/path/to/your/tree/file.tree
```

where you replace `~/path/to/your/tree/file.tree` with the full path to your tree file. This should open the FigTree program and display your tree.

Academic Honesty

To remind you: aside from your designated partner, you must not share code with your classmates: you must not look at others' code or show your classmates your code. You cannot take, in part or in whole, any code from any outside source, including the internet, nor can you post your code to it. If you and your partner need help from another pair, all involved should step away from the computer and *discuss* strategies and approaches, not code specifics. I am available for help during office hours, as are department tutors, but you should attend these hours with your partner. I am also available via email (make sure you and your partner is included in the email and do not wait until the last minute to email). If you participate in academic dishonesty, you will fail the course.

[3]http://tree.bio.ed.ac.uk/software/figtree/