SUNSTAR: AN IMPLEMENTATION OF THE GENERALIZED STAR METHOD

By

Benjamin Bettisworth

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in

Computer Science

University of Alaska Fairbanks

May 2017

Approved:

Dr. Glenn Chappell, Committe Chair
Dr. John Rhodes, Committe Member
Dr. Orion Lawlor, Committe Member
Dr. Chris Hartman, Committe Member
Dr. John Genetti, Department Chair
Department of Computer Science

Abstract

STAR (Liu et al. 2009) is a method of computing species trees from gene trees. Later, STAR was generalized and proven to be statistically consistent given a few conditions (Allman, Degnan, and Rhodes 2013). Using these conditions, it is possible to investigate robustness in the species tree inference process, the lack of which will produce instabilities in the tree resulting from STAR. We have developed a software package that estimates support for inferred trees called `SunStar`.

Contents

1 Introduction

`SunStar`[1] is a program designed to investigate the support for a species tree that has been inferred from gene trees via the STAR method.

1.1 Background

The problem of phylogenetics is an old one, going back as far as Darwin, and is a central question in the field of biology. In the following sections, we will lay out the problem at hand (Incomplete Lineage Sorting), and the attempts so far to solve the problem.

1.1.1 Problem Description

Modern phylogenetics is primarily concerned with the inference of species trees. Unfortunately, we can not directly infer species trees (defined in 1.2). Instead, phylogenetic methods are limited to traits we can observe. Historically, these traits were features like morphology of bones and limbs. With the advent of molecular sequencing, observation of traits using DNA and other molecular sequences became common, and with whose observations came the early mathematical methods of phylogenetics.

These methods were primarily using gene trees as proxies for species trees. Due to incomplete lineage sorting (ILS) (Pamilo and Nei 1988) this is not sufficient for inferring species trees. Informally, this means that genes might diverge differently than the species diverge. Therefore, just using a single gene tree as a proxy for species can be misleading. More advanced methods of inferring species trees get around the ILS difficulties by using information from multiple genes.

An example of this is the multispecies coalescent model (Maddison 1997), which is a probabilistic model of the ILS process. Using this model we can describe the way gene trees form from species trees. The methods that are discussed in this paper have been shown to be statistically consistent under this model. In this context, statistically consistent means that we can make the probability of inferring the correct species tree as close as we like to 1 if we make the sample large enough. Informally, given enough perfect data, we always get the right tree.

1.1.2 Prior Work

Previous software that infers species trees from from gene tree summaries includes ASTRAL (Mirarab and Warnow 2015), NJst (Liu and Yu 2011), STAR (Liu et al. 2009) and ASTRID (Vachaspati and Warnow 2015). All of these programs will compute species trees from existing gene trees. ASTRAL is the exception in this group, in that it does not compute a distance table from the set of gene trees passed to it. For the rest of the software, they all compute a version of a distance table.

In particular, STAR and ASTRID use a very similar method to the method used in `SunStar`, with one important difference: they do not attempt to infer the support for the tree reported. `SunStar` does, using the results from generalized STAR (Allman, Degnan, and Rhodes 2013).

1.2 Notation, Conventions and Definitions

A *tree* is a set of vertices (or nodes) and a set of edges that connect them with no cycles. A vertex with only one edge incident is called a *leaf*, plural *leaves* (we often call these *taxa* as well).

[1] The name is a slight pun on G-STAR, the original name for this project. The sun that earth orbits is a g-class star, therefore `SunStar`.

A *rooted tree* is a tree with a special node designated the *root*. The root of a tree is often labeled as ρ. A node is called an *interior* node if it is not a leaf.

Edges can have have tags which are numbers. These are typically called *weights* or *lengths*, and are often a measure of a notion of distance. The *distance* between two taxa, written $d(a, b)$, is the sum of edge weights on the path between a and b.

A tree is *trivalent* when all vertices of the tree have degree 3 or 1. A rooted trivalent tree is a rooted tree that is allowed to have a single vertex, the root, with degree 2. All trees discussed here will be trivalent unless otherwise noted.

A *gene tree* is a (rooted or unrooted) tree that and relates how a gene has diverged over time. Gene trees are usually inferred from sequences of genes, which have some mutations which differentiate them from each other. In contrast a *species tree* is a tree that relates the divergence of species. An unrooted tree is often made rooted by using an *outgroup*. This is a taxa that is intentionally distant from the other taxa, ensuring that the root is connected directly to the outgroup.

A useful bit of notation for a pair of taxa with the same parent is a *cherry*. Its called that because its a looks like a pair of cherries hanging of the stem cherry off a common stem. There is an example of a cherry in figure 1

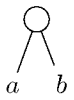

Figure 1: Example of a cherry

An *ultrametric* tree is a tree where all the distances from the root vertex to the leaves are the same.

A *distance table* for a species or gene tree is a table relating the distances between the taxa on a tree. The distance is calculated by summing the edge weights on the path between the two taxa.

We will use the notation $F(n) = \mathcal{O}(f(n))$ to denote the order of a function F. Normally, F is not reported, but instead referred to by this notation. By convention, the f reported will be close to as small as possible.

The term *stability* will be used to refer to the sensitivity of algorithms like STAR and GSTAR to inputs with equivalent topology, but different weight schedules. In general STAR should output the same tree regardless of weight schedule, (See section 1.6). Furthermore, we refer to the lack of this property as *instability*.

1.3 Newick Notation

Newick Notation is a format for specifying trees. An example of a Newick tree is as follows, with the associated tree shown in figure 2.

```
((a:.1,b:.3):1.0, (c:.43,d:.5):1.0);
```

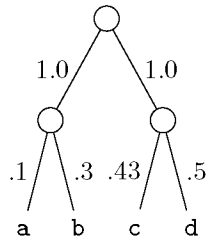

Figure 2: The associated tree

The ':' denotes a weight on an edge leading up from the node towards the root, and is optional[2].

One advantage of Newick notation is that the grammar for this language is quite simple, and requires only a few productions. This makes it easy to write a parser to recognize a Newick string. Unfortunately there are quite a few disadvantages, one of which is the non-uniqueness of the string for a tree. Specifically, there are many strings which represent the same tree, due largely to the ordering of subtrees. Note that these two trees are the same

```
(a,b); == (b,a);
```

This causes problems when attempting to compare trees, which is already a hard task. Fortunately, we can side step these issues by

- Enforcing an outgroup to insure a consistent root, and
- Enforcing an order on the taxa and subtrees.

By doing this, we can compare trees by comparing their Newick strings.

One additional problem is the lack of standardized grammar. Some implementations allow for taxa labels to start with a number. Some implementations require the semicolon at the end, others do not. It is difficult to deal with this issue, so instead I documented the grammar I used in section 3.2.

1.4 Neighbor Joining

Neighbor Joining (Saitou and Nei 1987) is a tree construction algorithm that proceeds by iteratively joining pairs together that have been selected by the four-point condition. Consider the tree in figure 3. Note that if the edges have non-negative edge weights, it must be true that

$$d(a,b) + d(c,d) \leq d(a,d) + d(b,c) = d(a,c) + d(b,d).$$

Specifically, the weight on the middle edge joining the two cherries matters. Also note that this relationship involves only distances between the leaves, and not distances between interior nodes. This is important, because we generally don't have distances from taxa to ancestor species.

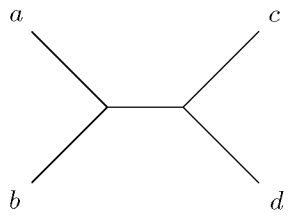

Figure 3: A 4 taxon unrooted tree

This fact informs the Neighbor Joining algorithm. By using the four point criterion, we can select pairs to join. At a high level the algorithm proceeds like so:

- Initialize the tree by creating nodes for all the N taxa, and connect them to a central node t.
- Select a pair of nodes that satisfy the above criterion the "best".
- Join the pair of nodes to a new parent node, and connect the new parent back to the central node.
- Calculate distances for the three new nodes.
- Repeat until there are 3 nodes left connected to the central node.

[2] In particular, `SunStar` only cares about the topology of the trees that it is fed, so information about edge weights is usually discarded. The exception is in the testing suite, where setting weights in this way is convenient.

Selecting the "best" pair to join is done by calculating a matrix Q, whose entries are the following

$$Q_{a,b} = (N-2)d(a,b) - R_a - R_b,$$

where a and b are indexes associated with nodes waiting to be joined, N is the number of nodes connected to the central node, $d(a,b)$ is the distance between a and b, and the quantity R_k is the sum of the distances from k to all other nodes. By calculating this matrix, and finding the smallest value in it, we can find the pair to join. Specifically, the row and column indices of the lowest entry, (a,b) correspond to the pair to join.

Using this pair (a,b), the implementation joins the pair by removing these elements from the central node c, making them the children of a new node, and then putting that new node back into the central node. We calculate the distance from the new node to the pair by the three point formula. The three point formula is for a 3 node tree, leaves x, y and z and central node r, and known distances between the leaves, we can calculate the distance between the leaves like so:

$$d(r,x) = \frac{1}{2}[d(x,y) + d(x,z) - d(y,z)],$$
$$d(r,y) = \frac{1}{2}[d(y,x) + d(y,z) - d(x,z)],$$
$$d(r,z) = \frac{1}{2}[d(z,x) + d(z,y) - d(y,x)].$$

To use the three point formula on a tree with more than three points to join, we compute the average distance between a, b and the rest of the nodes in consideration. This way, we have the 6 distances needed for the three point formula.

This process of selecting, joining, and re-inserting leaves a $N-1$ taxa connected to the central node. We repeat the process by calculating Q again, but this time only treating the nodes connected to the central node as taxa. In this way, we shrink the problem size down by 1 each iteration

When the number of nodes connected to the central node reaches 3, we skip calculating Q and just apply the three-point formulas to the remaining nodes. Importantly, this leaves us with an unrooted tree, which is the result of Neighbor Joining.

To calculate the complexity informally, we can note that each step requires $\mathcal{O}(N^2)$ work, in the form of calculating the entries of a matrix. The algorithm runs for $\mathcal{O}(N)$ steps. Therefore, the overall complexity is $\mathcal{O}(N^3)$.

1.5 STAR (estimating Species Trees using Average Ranks)

STAR infers a species tree by calculating a distance table of ranks, and then using a tree building algorithm to generate a new tree (Liu et al. 2009). Here, the term *rank* is used to mean edge length, and will be referred to as edge length in the future. We use the term here once to be consistent with Liu et al. The algorithm is as follows: Given a set of gene rooted trees, do the following for each tree.

- Set all the edge weights
  - If the edge is not connected to a leaf, set the weight to 1.
  - If the edge is connected to a leaf, set it so that the tree will be ultrametric. Practically this means that we can take the number of taxa and subtract the depth, then use that for the edge weight.
- Calculate pairwise distances, put them into a table.

Once all the tables have been calculated, do an entry-wise sum, and divide each entry by the number of gene trees. Informally, think of this as taking the average of the distance table. Once we have the "average" distance table, run a tree building algorithm on it, such as Neighbor Joining.

1.6 Generalized STAR

Generalized STAR (Allman, Degnan, and Rhodes 2013) is version of STAR where the edge weight schedule is allowed to vary. In this version, edge weights at an equal distance from the root (using graph-theoretic distance of counting edges in a path) are given the same weight. All weights must be non-negative, and all but one of the weights can be 0. Additionally, the edges connected to leaves are adjusted to make the tree ultrametric, as in STAR. Other than weight assignment, the algorithm proceeds the same.

Additionally, we can attempt to infer the stability of the species tree produced by STAR by varying the weights and reporting the different trees that come out of the tree creation step. This is the primary goal of `SunStar`.

1.7 SunStar

`SunStar` is a software package primarily intended to be used on a *nix command line, and is invoked with the command `sunstar [options]`. Currently `SunStar` has only a few options. Here they are:

- `-f --filename`: Filename for a set of Newick trees. These trees must all be rooted or all be unrooted. If they are all unrooted trees, then the `-o` is required.
- `-t --trials`: The number of trials using randomly generated schedule of weights (Section 2.4.2). If this flag is not present, then a default schedule (Section 2.4.1) will be used.
- `-r --required-ratio`: The minimum ratio required for a tree to ouput. Trees below this threshold will silenced.
- `-l --logfile`: Filename to log the sequences that are generated by the `-t` option. Defaults to `schedule.log` if not specified.
- `-o --outgroup`: The name of the outgroup taxa. This taxa must be present on all of the trees.
- `-s --silent`: Silence the progress bar, only output results.

1.7.1 Running `SunStar`

To run `SunStar` with default parameters, enter the following command

```
sunstar -f example.tree
```

Where `example.tree` is a list of Newick strings, separated by newlines. A file looks like the following for example.

```
((a,c),(d,e));
((a,c),(d,e));
((a,c),(d,e));
(a,(c,(d,e)));
(a,(c,(d,e)));
```

Weights may be included, and do not need to be stripped. Each of the trees needs to have the same set of taxa or the program will quit.

1.7.2 Output

The output of `SunStar` is a list of trees with normalized frequencies associated with them. Using the example file from above, `SunStar` produces the output

```
sunstar -f example.tree
[========================================================================] [7/7]
'(a,(c,(d,e)));' : 1
```

Note that this is a rooted tree. The root is determined by picking an outgroup arbitrarily, and is consistent across trees for a run of `SunStar`. By using the outgroup flag `-o`/`--outgroup` the specific outgroup can be specified. For example if we decide to root by `c` instead

```
sunstar -f demo.tree -o c
[==========================================================================][7/7]
'((a,(d,e)),c);' : 1
```

When the `-o` flag is specified, two things happen. First, all the trees in the input are rerooted, using the taxa as an outgroup. The second thing is that when trees are output, they are also rooted by the specified taxa.

So far only the default schedule has been used. To use the randomized schedule, the `-t` or `--trials` flag is specified. For example

```
sunstar -f demo.tree -t 1000
[================================================================][1000/1000]
'(a,(c,(d,e)));' : 1
```

The random numbers that are used for weighting, and the tree that is produced from that sequence, are logged to a file. By default this is `schedule.log`, which contains a series of JSON objects. An truncated example from the above run is

```
{"tree":"(a,(c,(d,e)));","weights": [0.713856,0.711066,0.411698]}
{"tree":"(a,(c,(d,e)));","weights": [0.0549974,0.13108,0.473839]}
{"tree":"(a,(c,(d,e)));","weights": [0.620542,0.524716,0.9447]}
{"tree":"(a,(c,(d,e)));","weights": [0.068759,0.0251013,0.694658]}
{"tree":"(a,(c,(d,e)));","weights": [0.182371,0.31707,0.83606]}
{"tree":"(a,(c,(d,e)));","weights": [0.701719,0.260946,0.639861]}
{"tree":"(a,(c,(d,e)));","weights": [0.648678,0.970904,0.509442]}
{"tree":"(a,(c,(d,e)));","weights": [0.533012,0.101569,0.370654]}
{"tree":"(a,(c,(d,e)));","weights": [0.567502,0.274677,0.371589]}
...
```

The log file name can be specified using the `-l` or `--logfile` switches.

In the case that there are many trees of low probability, the output can be trimmed using `-r` or `--require-ratio`. This will silence trees that are lower frequency than specified.

Finally, the progress bar that is output for each run is written to `stdout`. If `stdout` is redirected to a file, most of the file will be this progress bar output. To prevent this, we can silence the progress bar using `-s` or `--silent`, which will suppress the progress bar.

2 Algorithms

2.1 Basic Operations

In the following sections about complexity, we count the following as basic operations:

- Node accesses, and
- Basic Arithmetic.

Also of note is what is *not* counted. Specifically, we do not count the basic operations of most of the data structures used in computation. Counting the operations of these data structures would not change the complexity class of the problem, as the operations used on relevant data structures, mainly create and update, are constant time.

2.2 Finding the distance between two taxa

2.2.1 Specification

In order to calculate the distance table, we have to calculate the distance between any two taxa. Fortunately we are working with trees, so the path between any two nodes is unique. Therefore, if we find a path, we have found the only path. So, the strategy to find a path between two nodes is to calculate the list of ancestors for each node. Once we have this list, we walk the list, starting from the root, until we find a difference. Then, the remaining nodes in the list, plus the node right before the divergence, form the path between the two nodes.

For example, suppose we have the tree in figure 4. Then the sequences of parents associated with that tree are

$$
\begin{aligned}
s_1 &: \rho, n_1, t_1 \\
s_2 &: \rho, n_1, t_2 \\
s_3 &: \rho, n_2, t_3 \\
s_4 &: \rho, n_2, t_4.
\end{aligned}
$$

Notice that for taxa t_1 and t_2, the sequences differ only at t_1 and t_2. So backing up a step, we have the path t_1, n_1, t_2. For the taxa t_1 and t_4, the node before the difference is ρ. So, the path between t_1 and t_4 is t_1, n_1, ρ, n_2, t_4.

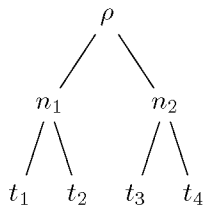

Figure 4: An example tree

2.2.2 Complexity

The complexity of finding the distance between two nodes is $\mathcal{O}(n)$, where n is the number of taxa on the tree. First, consider the steps involved with this algorithm. They are

- Make a list from taxa to root,
- Walk along that list,
- Create new path,
- Sum edge weights over path.

Note that the number of nodes on a rooted tree with n taxa is $2n - 1$. Since a tree is acyclic, each node in the path is visited at most once. Therefore the size of the lists from taxa to root is at most $\mathcal{O}(n)$. Since the list sizes are bounded by n, walking along them is also bounded by n. Creating a new path can only be as bad as concatenating two lists together. Since the two lists are of size $\mathcal{O}(n)$, concatenating them is only as bad as $2\mathcal{O}(n) = \mathcal{O}(n)$. Finally, summing over the final list of nodes requires iterating through the final path. Since the final path is $\mathcal{O}(n)$, this step is also $\mathcal{O}(n)$, and therefore this method is as well.

2.3 STAR

2.3.1 Specification

The algorithm requires a set of m rooted trees. STAR involves three major steps:

- Calculate the distance table for each tree in the set
- Average the distance tables
- Run Neighbor Joining on the new table.

Calculating the distance table requires finding the pairwise distance between all the taxa. To do this, we create a matrix with the taxa labeling the rows and columns. For each entry in the matrix, we find the distance between two taxa using the algorithm specified above. We need to do this for each tree in T.

We now have a distance matrix that records the pairwise distance for each taxa in the tree, for each tree in T. To average, we simply add each matrix, and then divide by m. Once we have this new distance matrix, we pass it to Neighbor Joining and get a new tree.

2.3.1.1 Example

Consider the trees in figure 5. If we were to run the STAR algorithm by hand on these trees, we would first produce total table in figure 6, by computing the distance between each taxa, and then adding them together.

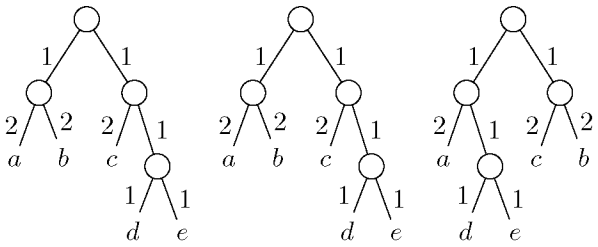

Figure 5: A set of example trees

	a	b	c	d	e
a	0	$4+4+6$	$6+6+6$	$6+6+4$	$6+6+4$
b		0	$6+6+4$	$6+6+6$	$6+6+6$
c			0	$4+4+6$	$4+4+6$
d				0	$2+2+2$
e					0

Figure 6: Total distance table associated with the trees in figure 5

The next step is to average the entries in the table, by dividing through by the number of trees. The result of this computation can be seen in figure 7.

	a	b	c	d	e
a	0	$14/3$	6	$16/3$	$16/3$
b		0	$16/3$	6	6
c			0	$14/3$	$14/3$
d				0	2
e					0

Figure 7: Average distance table associated with the trees in figure 5

After computing the average, the distance table in figure 7 is passed to Neighbor Joining, and a tree is built out of that distance table.

2.3.2 Complexity

The complexity of STAR is $\mathcal{O}(mn^3)$, where m is the number of trees, and n is the number of taxa on each tree. Calculating a single entry in the distance matrix requires a call to an $\mathcal{O}(n)$ algorithm, and there are n^2 elements in the matrix. Therefore the complexity of computing each table is $n^2\mathcal{O}(n) = \mathcal{O}(n^3)$. We need compute this distance matrix for each of the m trees in T, so $m\mathcal{O}(n^3) = \mathcal{O}(mn^3)$.

Computing the average of the distance matrices involves a series of matrix additions (mn^2) and a scalar matrix operation (n^2). These are less than the time required to simply compute a single distance matrix, and so they don't change the overall $\mathcal{O}(mn^3)$. Similarly with Neighbor Joining, which has a complexity of $\mathcal{O}(n^3)$, but only needs to be performed once.

2.4 Generalized STAR

As discussed above, Generalized STAR works by varying the schedule of weights on a set of trees. To implement this algorithm, we simply produce different schedules and then run STAR on the tree sets with the different weights. There are two strategies by which weight schedules are generated: the default schedule and the randomized schedule. These are discussed in the two following sections.

2.4.1 Default Schedule

This schedule starts by assigning a weight of 1 to every edge, and then adjusts leaf edges to be ultrametric. Then, all possible combinations of weights with at least one weight being 1 and all others being 0 are computed[3]. By the results from generalized STAR, if we have a very large sample of gene trees without inference error then the tree should come out the same, despite the 0 weights on the edges.

Intuitively, this is like canceling out parts of the tree which carry information about the differences between taxa. By setting edge weights to 0 in key places, we can possibly find that the trees output by STAR are significantly different, compared to the regular schedule. It is our belief that this strategy is effective, and early results agree.

2.4.2 Randomized Schedule

The other strategy to generate weight schedules is to generate them randomly. This strategy works by generating a list of numbers from a uniform distribution from 0 to 1. After a schedule generated and assigned, the leaf edges need to be adjusted to make the tree ultrametric. These numbers are then used as the schedule for a run of STAR.

3 Implementation

3.1 `tree.h`

3.1.1 Purpose

This file implements the class `tree_t` (read "tree type"), which is intended to store both the topology and metric information about a tree. In particular, this class needs to support the ability to store metric

[3]This should be thought of as counting in binary from 1 to the height of the tree.

information, and calculate distances between taxa.

A major complication of `tree_t` is the need to support both rooted and unrooted trees. There are two reasons for this: Neighbor Joining produces unrooted trees, and input might be in the form of unrooted trees. It might be possible to side step the Neighbor Joining issue, but the requirement to take in unrooted trees from the user is something we need to support regardless.

The support for unrooted trees also informs another requirement for `tree_t`: the ability to make minor modifications to the topology of a tree. Specifically, we need to be able to root a tree by a specified taxa called an outgroup. Therefore, there is "reroot" functionality implemented.

3.1.2 Implementation

At a high level, phylogenetic trees are implemented as doubly linked binary trees. This is to say, each node on the tree has a reference to a parent (if it exists), and references to two children (if they exist). Note that due to the nature of phylogenetic trees, nodes either have two children, or none. This also means that there is a sense of directionality towards the root. This makes finding paths between nodes easy, as we can begin at leaf nodes and proceed towards the root.

A concept that is central to the tree class the `unroot`. Traditional binary trees typically have a root node that contains pointers to the two subtrees. This works for rooted trees, but some trees we need to represent are unrooted. This leads to the concept of an `unroot`. In figure 8, there is an example of the "central" node, an unroot, connected to to three subtrees.

In this case u is the central node that we wish to represent. We can't do this with a typical node, because concepts of children an parent don't apply here. Instead, we do this by making an adjacency list for just the node u, with pointers to the nodes (representing the subtrees) a, b and c.

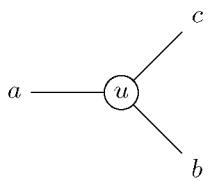

Figure 8: An example unroot

The `node` class contains the information relating to each individual node in the tree. This includes data including the label of the node, the weight of the edge towards the parent, and pointers to the children and parent.

The `tree_t` class is the main workhorse of the tree implementation. It contains the tree and the `unroot`. Since these trees do not have the normal CRUD operations, we can predetermine the size of the tree. This allows us to preallocate a contiguous section of memory to store the `node`s that make up the tree, as opposed to each node being potentially distant in memory from other nodes. By doing this, we can take advantage of cache locality, which can be a potentially large speed up on some systems. This technique I "call tree packing"; it is discussed in section 3.1.3.

The `tree_t` class is also responsible for setting the weights of the tree. This can be done in three ways. The first is to pass the `set_weight` function a function that takes a `size_t` or equivalent, and returns a `double`. Another is to pass a vector of weights which is indexed by depth. Yet another way is to pass a constant to the function, which will set the weights to be the constant, with the exception of the leaves. In all these cases, the tree remains ultrametric, i.e., the leaf edges are adjusted to ensure it.

3.1.3 Tree Packing

Tree packing is the name for producing a tree topology in a contiguous section of memory. At a high level this algorithm performs a preorder traversal of the tree, and pushes nodes onto a queue. After the traversal is complete, the queue contains the packed tree. The final step is to update the references inside the tree via a map of old references to new references.

Algorithm 1: Tree Packing

Data: List of pointers to tree roots r
Result: A packed representation of a tree in memory

```
1  stack s;
2  queue q;
3  for n in r do
4  |  push n onto s;
5  |  push n onto q;
6  while s is not empty do
7  |  n ← top of s;
8  |  pop top of s;
9  |  if n has children then
10 |  |  push children of n onto s;
11 |  |  push children of n onto q;
12 tree ← array the size of q;
13 for n in q do
14 |  push n onto tree;
15 |  update pointers of n;
```

3.1.4 Set Root

The final major role of the tree is to set and reset the root. Sometimes, unrooted trees might be passed to this function, but the algorithms only work on rooted trees. So we must be able to root a tree. To do this, the algorithm 2 is used.

3.2 `newick.h`

3.2.1 Purpose

In order to work with gene trees, I need to be able to parse Newick Notation, as specified in section 1.3. In this implementation, we expose a function `nj` which given a string returns a parsed tree. This is done via a recursive descent parser.

3.2.2 Implementation

3.2.2.1 Grammar

The grammar used has two 3 productions and 2 terminal lexemes. The terminal lexemes are labels and weights. The regular expression for labels and weight are

```
weight = [0-9]+('.'[0-9]*)?('e'[0-9]+)
label  = [A-z][A-z0-9_]
```

In other words, floating point numbers with possible exponential notation, and `c` identifiers. Every other literal is punctuation, and is there for structure only. Below is the grammar that is in the parser

Algorithm 2: Set Root

Data: List of pointers to roots of subtrees r representing the unroot of an unrooted tree, outgroup pointer o
Result: New root of a rooted tree, with o as an outgroup

```
1  if o in r then // Unroot is where it should be, just need to root the tree
2      make new node nr;
3      set nr's children to be the two other nodes in r;
4      clear r;
5      add o to r;
6      add nr to r;
7  else
8      make new node n;
9      assign n's parents and children to the three nodes of r;
10     clear r;
11     add o to r;
12     for child, parent c of n do
13         set c's parent to n
14     p ← o's parent;
15     if o is not the left child of p then
16         swap p's left child and right child;
17     set p's left child to null;
18     SwapParent(p, null) // See algorithm 3 below
```

Algorithm 3: SwapParent(node n, node p)

```
1  if p is n's left child then
2      swap n's parent and left child;
3      SwapParent (left child, n);
4  else if p is n's left child then
5      swap n's parent and right child;
6      SwapParent (right child, n);
```

```
tree    = '(' subtree ',' subtree [',' subtree] ')' [';']
subtree = label [':' weight]
        | '(' subtree ',' subtree ')' [':' weight]
```

3.2.2.2 Parser/Lexer

Since the language is so small, we have made the parser and lexer tightly integrated. Thus, there is no separate lexer module. Instead, characters are read and dealt with immediately. I read floats with the `stod` function in the standard `c++` library.

The parser itself is simple top down recursive descent parser. The parser returns a list of node pointers representing the subtrees that make up the tree. This list is intended to be the unroot.

3.3 `nj.h`

This header contains my implementation of the Neighbor Joining(Saitou and Nei 1987) algorithm. Neighbor Joining has been implemented previously. However efficient implementations are not generic, as they are specific to the tree structure being used. Therefore it was necessary to reimplement the algorithm here. A full specification of the algorithm can be found in section 1.4.

Some optimizations have been performed over the naïve implementation. The first, and largest, is that the matrix Q, which is calculated in each iteration of the algorithm, is not stored. Instead, we just store the lowest value of Q so far, and the pair associated with it. When we have finished iterating through the matrix, we simply return the pair with the lowest associated value of Q.

3.4 `star.h`

3.4.1 Purpose

The STAR algorithm is straightforward. Most of the complexity of the code comes from the tree itself, but once that is implemented, the algorithm is simple. Please refer to algorithm 4 for the details.

Algorithm 4: STAR

Data: A set of trees T, with the same set of taxa X where $|X| = n$, with weights set.
Result: A single tree s

$D \leftarrow 0^{n \times n}$;
for t *in* T **do**
    $d \leftarrow$ distance table of t;
    $D \leftarrow D + d$;
$D \leftarrow {}^1/_n \cdot D$;
$s \leftarrow$ tree from neighbor joining with D;
return s

3.5 `gstar.h`

As stated before, there are two schedules for `SunStar`. The first is the default schedule, which is run when the `--trials` option is not specified. The other is the randomized schedule, which is run when `--trials` is specified.

The default schedule is an attempt to find an error by brute force. We treat each level of edges as either having a weight of 0 or 1. We then try every possible assignment of 0s and 1s.

Algorithm 5: GSTAR-default

Data: A set of n trees, T, which all have the same set of taxa.
Result: A set of trees and associated frequencies.

$R \leftarrow$ map of trees to integers;
$h \leftarrow \max_{t \in T}$(height of t);
$S \leftarrow$ list of 1's of length h;
for i *in* $1 : 2^n$ **do**
    for t *in* T **do**
        set weights of t by schedule S;
    $t_s \leftarrow$ STAR on T;
    if $t_s \notin R$ **then**
        $R(t_s) \leftarrow 0$;
    $R(t_s) \leftarrow R(t_s) + 1$;
    `/* Here we treat S as a binary number, and decrement it */`
    decrement S;
`/* Normalize the counts into proportions */`
for r *in* R **do**
    $R(r) \leftarrow {}^1/_{2^n} \cdot R(r)$;
return R

The random schedule proceeds by randomly generating weights, and assigning them to the trees in the set.

Algorithm 6: GSTAR-random

Data: A set of n trees, T, which all have the same set of taxa. A number of trials t.
Result: A set of trees and associated frequencies.

$R \leftarrow$ map of trees to integers;
$h \leftarrow \max_{t \in T}(\text{height of } t)$;
$S \leftarrow$ a list of h random numbers, uniformly distributed on $[0, 1]$;
for i *in* t **do**
    for t *in* T **do**
        set weights of t by schedule S;
    $t_s \leftarrow$ STAR on T;
    if $t_s \notin R$ **then**
        $R(t_s) \leftarrow 0$;
    $R(t_s) \leftarrow R(t_s) + 1$;
    $S \leftarrow$ a list of h random numbers, uniformly distributed on $[0, 1]$;
`/* Normalize the counts into proportions */`
for r *in* R **do**
    $R(r) \leftarrow 1/2^n \cdot R(r)$;
return R

Early tests have shown that the default and random schedules are about equivalent in detecting error, though the representation of resulting trees differ.

4 Results

The results that are currently available are empirical investigations of the complexity of GSTAR, implemented in `SunStar`.

`SunStar` was built using `clang++` using the `-O3` and `-march=native` flags. Tests were performed on a 4.0 GHz processor. The command used to obtain these times was

```
time ./sunstar -f TESTFILE.tre -t TRIALS
```

The number reported is the user mode cpu time.

Number of Trees	Number of Taxa	Trials	CPU Time (Seconds)
10	6	1000	0.06
10	6	10000	0.54
100	6	1000	0.33
100	6	10000	3.23
10	37	1000	3.2
10	37	10000	33.5
100	37	1000	26.5
100	37	10000	266.3
424	37	1000	110.5
424	37	10000	1116.2

As it can be seen, these times fall in line with the predicted complexity of the program. Specifically, the program is linear in the number of trees and the number of trials, while being non-linear in the number of taxa. It is important to note that if we perform the calculation, the 37 taxa trials should be approximately 216 times slower than the equivalent 6 taxa trials. In these trials, they are faster than 216 times slower,

on average only 66.43 times slower. Further investigation is required to determine why we see better than expected performance in the number of taxa.

5 Conclusion

We have created a software package, `SunStar`, that uses Generalized STAR to infer the stability of a species tree produced by STAR. Furthermore, we have found early results that suggest that this technique might be useful.

6 Future Work

There are many optimizations, enhancements, and investigations we would like to do with `SunStar`. Like many software projects, `SunStar` could be developed forever, given sufficient resources. What follows is a short list of priority items that should be done before other options.

6.1 Optimizations

There are many optimizations that we would like to implement. They are in no particular order:

- Refactor distance table data structure, so that we can take advantage of symmetry.
- Refactor the node class; investigate making the class smaller in memory.
- Implement OpenMP routines on the GSTAR level of computation (Give each STAR inference a thread).
- Optimize the default schedule, find unnecessary steps.

6.2 Enhancements

There are a few enhancements that are possible for `SunStar`. Some of these are only necessary if the technique turns out to be very successful, but are included as a roadmap nonetheless.

- More distributions/schedules for weights in GSTAR.
- Add MPI support.
- Add "Full Pipeline" inference. Specifically, take in gene sequences and return a set of trees with support.

6.3 Investigations

To really investigate this method of detecting errors, we would like a study that

1. Models gene trees from a species tree using the multispecies coalescent model,
2. Generate gene sequences from those gene trees,
3. Infer new gene trees with error from the generated sequences,
4. Run `SunStar` and examine the error rate.

Most of these computations are relatively fast, except for step 3. Inferring gene trees from sequences can take quite a while. Therefore, this investigation will likely take several months of work.

7 Code

The entire code base, including this write up, is located on GitHub[4]:

[4] https://github.com/computations/SunStar.

References

Allman, Elizabeth, James Degnan, and John Rhodes. 2013. "Species Tree Inference by the STAR Method and Its Generalizations." *Journal of Computational Biology* 20 (January): 50–61. doi:10.1089/cmb.2012.0101.

Liu, Liang, and Lili Yu. 2011. "Estimating Species Trees from Unrooted Gene Trees" 60 (5): 661–67. doi:10.1093/sysbio/syr027.

Liu, Liang, Lili Yu, Dennis Pearl, and Scott Edwards. 2009. "Estimating Species Phylogenies Using Coalescence Times Among Sequences." *Systematic Biology* 58 (5): 468–77. doi:10.1093/sysbio/syp031.

Maddison, Wayne. 1997. "Gene Trees in Species Trees." *Systematic Biology* 46 (3). doi:10.1093/sysbio/46.3.523.

Mirarab, Siavash, and Tandy Warnow. 2015. "ASTRAL-II: Coalescent-Based Species Tree Estimation with Many Hundreds of Taxa and Thousands of Genes." *Bioinformatics* 31 (12): i44–i52. doi:10.1093/bioinformatics/btv234.

Pamilo, P., and Masatoshi Nei. 1988. "Relationships Between Gene Trees and Species Trees." *Molecular Biology and Evolution* 5 (5). doi:10.1093/oxfordjournals.molbev.a040517.

Saitou, Naruya, and Masatoshi Nei. 1987. "The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees." *Molecular Biology and Evolution* 4 (4): 406–25. doi:10.1093/oxfordjournals.molbev.a040454.

Vachaspati, Pranjal, and Tandy Warnow. 2015. "ASTRID: Accurate Species TRees from Internode Distances." *BMC Genomics* 16 (10). doi:10.1186/1471-2164-16-S10-S3.