**Homework 6 Answers**

1. When we use the molecular data to build the phylogeny tree, we often assume that nucleotide changes should accumulate in some constant proportion to time, where the constant of proportionality may differ in different lineages. In reality, what factors could make this hypothesis not true?

The assumption that nucleotide changes should accumulate in some constant proportion to time within each lineage is a generalization of the “molecular clock hypothesis”. In the original molecular clock hypothesis, the assumption is that there is some constant rate of mutation which are fixed in the population (meaning a new mutation in one individual eventually spreads out the entire population) at some uniform rate. The original molecular clock hypothesis posits a single rate over the entire tree. The generalization here is that this uniform rate applies within each lineage in the tree, but across different lineages (edges of the tree), there might be different uniform rate (e.g., the human lineage might be evolving at some constant but faster rate than gorilla).

The fundamental basis of “molecular clock hypothesis” is that the mutation rate is steady for the same lineages during evolution. However, fixation rate of new mutations for different DNA regions of the same lineage could be different. In some regions of genome, for example intron or noncoding sequence, they are “neutral” and nonfunctional, which means that they are not under selective pressure and their substitutions could occur in proportion to time since it is only determined by mutation rate. However, for most coding sequence in the genome, the fixation rates might be determined by many evolutionary parameters. For example, directional selections are always for some certain variants carrying beneficial mutations. Furthermore, the mutation rate is always not steady during evolutionary time and in different parts of the genome. The rate may differ between different time periods due to changes in mutagens in the environment, changes in key genes involved in the genome stability, damage response, recombination repair, animal age or sex etc. Different parts of a genome also have different mutation rates due to factors like differences in chromatins, replication rate, etc. Finally, effective population size and reproductive strategies are also very important factors for fixation rate.

1. We have talked about methods for phylogeny estimation. Discuss how we could measure the accuracy of the estimated phylogeny. What approaches would you take to assess the utility of different methods or algorithms?

We could compare the accuracy of different methods based on some standard data with validated known phylogeny or simulated data. One possible validated known phylogeny is from experimental evolution of fast evolving organisms such as bacteria or viruses. In some limited cases, we might have detailed fossil records that help establish nearly certain phylogenies. Simulation is a common method where some stochastic process for character evolution is created over a known phylogeny tree graph. We might run multiple simulations over a variety of trees and measure the similarity of the estimated tree to the known model tree.

Lastly, an alternative approach is to attempt to establish some kind of statistical confidence bounds on the estimated trees. This is similar to estimating the mean of a normal distribution along with a confidence limits on the estimated mean. For the phylogeny estimates, the bootstrap method is a resampling method to create an ensemble of estimated trees from variants (created by bootstrap resampling) of the input data. This link discusses this idea:

<https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1558-5646.1985.tb00420.x>

1. In class, we discussed two different greedy sequential tree construction method, one with O(n^3) and another with O(n^2) complexity where n is the number of leaves. Suggest another possible greedy tree construction method with polynomial time complexity. Give the time complexity for your approach. Analyze pros and cons with respect to the two O(n^3) and O(n^2) algorithms discussed in class.

**Some students’ answers:**

1. Another possible greedy tree construction method could start by randomly picking three leaves of the t total leaves without replacement and construct a T3 tree. Do this m times where m < C(t,3) such that none of the T3 trees are identical. (Note this is a subset of all the possible T3 trees that could be constructed). Of the m different T3 trees, pick the best T3 tree. We will continue with this tree. Randomly choose the next leaf (from the t-3 leaves remaining); we will use it to create T4 from T3. Note that there are 2k-3 edges to add this leaf. Add it to min (m, 2k-3) different positions such that none of the min (m, 2k-3) trees are identical. Choose the best T4 tree of the min (m, 2k-3) trees. Continue until all the leaves have been added. At each iteration, we are choosing an optimal tree from a subset of possible trees. If we assume that, on average, the min (m, 2k-3) = m, the complexity of this algorithm is approximately O(tm).

Note that the solution to these three algorithms is not guaranteed to be globally optimal. In terms of complexity, my algorithm is intermediate O(t3) and O(t2): O(t2) < O(tm) < O(t3). If the number of leaves gets too large, O(t3) and O(tm) algorithms may become too complex and O(t2) would be preferred. However, on smaller datasets, O(tm) may provide a faster and similar solution to O(t3).

1. Another possible greedy tree construction:
2. Create all subsets of 3 leaves and select the optimal based on some given objective function to be used as the base
3. Next add the edge to the most optimal next edge after considering all 2n-3 edges
4. Create a constant to remove worst leaves
5. Repeat step “b”

Analysis of Complexity: This algorithm has complexity O(n3). The first step involved nC3 possibilities which is of the magnitude n3. Steps 2 and 3 are each O(n) time since in one case we are looking at 2n-3 edges and in the other we have set a constant for the time to remove these leaves. Overall our run-time therefore is O(n3)

Pros:

* Should construct a better tree than the O(n3) algorithm in the text since we are removing the worst leaves
* Uses the optimal 3 leaves a base rather than randomly choosing 3 like in the O(n2)

Cons:

* Slower runtime than O(n2)
* Greedier approach than O(n3) since we are removing some parts to increase optimality

1. Build a completely resolved tree, starting with all sequences connected to a single ‘hub’ node. At each step, two lineages attached to the hub node are joined, becoming neighbors. Neighbors are chosen so that tree is optimal at each step. Time complexity: O(n^3). n-leaf tree requires n-3 iterations. At each step, one has to build a n-by-n search matrix, then the next is (n-1) x (n-1) Pros: Fast and additional sequence is irrelevant. Alternative additional sequences do not yield different trees as in the case of sequential construction. Cons: Similar to sequential construction: Yields one tree, often not global. Unique: neighbors cannot be dismantled at later steps; ties broken arbitrarily.
2. In the sequential construction algorithm discussed in class, at each iteration one additional node is added to the nascent tree at each iteration. However, this potentially led to local optima, as we can imagine a situation where the global loss function is minimized when a pair of nodes are added in a way that is different than them being added sequentially. An algorithm to address this may consider two nodes at a time at each iteration. For each of the t iterations, the number of possible ways to add two additional nodes to a nascent tree of size k is something like 2 \* choose(k,2), so the overall runtime is O(t3). This is faster than the O(N^2) algorithm in class and equal asymptotically to O(n^3) and has the benefits of being easy to implement and somewhat avoids local optima over the O(N^2) algorithm. However, it may be prohibitively slow on large datasets and is susceptible to local optima for pairs of nodes.
3. We have a divide-and-conquer algorithm that we are going to use for sequential tree construction. We recursively call the method on the array of organism sequences, and at each step it divides the array in half. Eventually, the base case is reached where only one or two elements are in a tree. Then, the method returns one tree and goes up a step. The method call a step up then receives two trees, and it then merges these two trees by taking one of the trees and putting it into the optimal location on another tree. This keeps happening until the method returns to the top and there is one tree.
4. Proof of Runtime: There are at most O(n) merges done at each "level", and each merge takes O(n) time because there are are O(n) insert locations on one of the trees. Finally, there are O(n) "levels", which are recursive method calls, so this means there are O(n)\*O(n)\*O(n) = O(n^3) operations. I know that the method is actually in terms of log\_2(n) for runtime bound, but I wanted to prove polynomial time. Pros of this method: Each tree that is built up is close to a locally optimal tree. It is a good method to approximate the globally optimal tree, as the globally optimal tree is built from locally optimal trees. It is space efficient and does not create trees which aren't used. Cons of this method: It is not faster than the other two methods, as the other two are O(n^2) and O(n^3), but since we did not prove the tightest bound this may not be totally true, as it may be a little faster. It is a greedy method, similar to the other methods, so it does not produce an optimal solution. The recursive calls may have local variables that can take up space, and there are a lot of recursive calls.
5. Build a starting tree by any method (i.e. randomly join the leaves into a tree) • For each leaf in the tree:
   1. –  Consider moving the leaf to each possible edge in the current tree
   2. –  Move the leaf to the edge in the current tree which causes the largest increase in the objective function (if no movement causes an increase, leave the leaf where it is)

This procedure considers (2t − 3) possible leaves for each of t steps, giving a total of ΣT (2t−3) = t(2t−3) steps, for an overall time complexity of O(t2). Building the starting tree would take O(t) time if done at random, although one could use more complex methods if needed.

Note that this is still a greedy algorithm (the comparison for each edge still chooses the locally optimal solution), and similar to the streamwise O(t2) algorithm discussed in class, it will be sensitive to the order in which leaves are moved or considered to move.

1. Idea – I would implement what is essentially the O(n^2) algorithm except it iterates through a mix of choosing the neutral and the optimal solution for a proportion of steps. After this iteration it chooses the optimal solution from this small set. The algorithm goes as follows:

For the leaf set L, randomize the order.

Choose the first three leaves in the input order and construct a 3-leaf tree.

Let k = the string of values {0, 1, 2, 3, 4, 5, 6, 7, 8}

For the first n/(k+2) leaves

Select the next leaf in the input order and examine each of the possible edges and select the neutral (middle of worst and best) edge for insertion by evaluating the objective function.

Continue until all n/2 leaves have been added.

For the last n/k+2 leaves

Select the next leaf in the input order and examine each of the possible edges and select the optimal edge for insertion by evaluating the objective function.

Continue until all n leaves have been added.

Compare the resulting 9 trees and choose the optimal solution

Analysis

* + 1. Time complexity = 9\*O(n^2) = O(n^2)
    2. Pros: attempts to remediate the bias early on towards the funneling to a local maxima solution
    3. Cons: does not guarantee you are selecting for a global maximum solution, worst case it is as bad at finding a global maximum as the simple O(n^2) algorithm.

1. Let the objective function be denoted 𝜑 (D, θ, T), where D is the dataset, θ are our parameters, and T is the tree topology parameters as discussed in class.

As with the O(n^3) time algorithm, consider all possible C(t,3) 3-leaf subsets of L, and compute a score for them using function 𝜑.

Sort the trees from most to least optimal. Beginning with the tree that gives the optimal value, greedily merge the trees by combining similar nodes. If the next encountered tree does not have any similar values, push the tree to later in the list\*\*. Continue until all leaves are in the tree.

The runtime of this would be O(n^3), as we are still computing scores for all triplets (see note below for more regarding runtime). While this doesn't improve the time-complexity, it would likely result in a tree with fairly accurate placement of members that are closely related, perhaps partially eliminating the issue of being "too greedy." As with the given O(n^3) algorithm, this would not be a good algorithm to use with very large datasets. That said, it it may alleviate some of the issues caused by choosing individual nodes greedily as seen in the O(n^3) algorithm.

\*\* It's hard to say where to put this that that will have a good balance of runtime and optimality, as pushing it to the end of the queue will \_definitely\_ be suboptimal, but just pushing it back one spot could have implications on the runtime — a case where a the nearest sub-tree with merge-able nodes was at the end of the list could lead to an exponential time algorithm. Maybe putting it at the middle of the queue is a good compromise...

1. Consider the following algorithm:
   1. First look at all of the n leaves in 3 leaf subsets and choose the optimal 3 leave tree given the objective function.
   2. Then, choose an edge at random and look at the remaining edges not in the tree and find the node that maximizes the objective function.
   3. Repeat this process until all of the nodes have been added.
   4. Runtime: Similar to the algorithm where the node order is randomized, this takes O(n^2) time.
   5. Pros:
      1. Faster than the O(n^3) runtime algorithm
      2. Considers all of the nodes at each addition unlike the node randomization O(n^2) algorithm,
   6. Cons:
      1. No control over the tree topology which is a major part of optimizing a phylogenetic tree.
2. Algorithm:

1. Use all the t leaves to build a random polygenetic tree that contains (2t-3) edges.

2. Choose an edge and calculate the objective function to estimate the phylogenetic distance. With the rest of the tree, then rearrange the edge based on the estimation.

3. Repeat step 2 until are the edges are iterated

Time complexity:

Since there are (2t-3) edges in the tree, the greedy algorithm presented will cost O(t2) time. Pros: The algorithm is equally or more efficient that the O(t2) and O(t3) algorithm discussed in class Cons: Greedy approach is more likely output a locally optimal solution.

1. Instead of randomizing the leaf insertion order, you can randomize the leaf placement. The resulting algorithm would start with a random three leaf tree, then randomly pick a position to insert the next leaf at (O(1)). It would then iterate over all the possible leaves to find the one that would yield the highest value of the objective function and insert that leaf (O(n)). It would then repeat this n times until the tree was complete.

This algorithm runs in O(n^2) time. In terms of performance, it should be roughly equivalent to randomizing the leaf insertion order. When compared to the O(n^3) algorithm, it should perform worse in optimizing the objective function, but will perform much faster. If you were to find a good way to choose which location to search nodes via, you could improve its performance.

For example, if you maintain a biograph where one side consists of leaves in the tree and the edges between them consist of pairwise alignment distances, you could insert the next leaf adjacent to the leaf in the tree with the smallest distance to a leaf not yet inserted. This would add substantial overhead though.

1. In the O(n^3) strategy, we first compare all possible 3-leaf subsets of L and chose the optimal one. We then consider each possible additional leaf and tree shape for a 4-leaf tree and select the optimal one. We then continue examining each possible leaf and position for all remaining leaves. In the O(n^2) strategy, we first give an arbitrary number to each leaf. We then construct the optimal tree from the first three leaves and continue adding each leaf onto the tree in order in the optimal position. The O(n^3) strategy probably results in a better tree but uses more compute. The O(n^2) strategy may result in a less precise tree but uses less compute. An additional strategy with polynomial time would be to first examine all possible 3-leaf subsets of L and start with the optimal one, and then arbitrarily number the remaining leaves and add in order in the optimal position.
2. A greedy tree constructive method I propose is a bootstrap version of quadratic construction. Given a set of N leaves, create N randomly ordered lists of leaves and produce N 3-leaf trees. Then remove any redundant 3 leaf trees. Then proceed to add the next leaf for each tree and select the optimal edge for insertion tree according to the objective function and remove the least optimal tree each round. Continue this until all leaves for the set are added, even if there is only one tree remaining at some point. This algorithm is O(N^2) complexity because it essentially doing the quadratic search albeit slower since it has to consider multiple starting trees but as the number of possible trees grows larger it reduces this number. The pros of this algorithm is it explores the search space broadly by starting in a lot of different areas in that space, which is better than just the quadratic method which can get stuck in a local optimum quite easily. However it doesn’t effectively search the local space as well as tree bisection and regrafting since the poorest tree is removed each round. It should be faster than tree bisection, since it reduces the number of trees constructed by removing the least optimal one each round. The proposed algorithm should be able to identify more optimal configurations than the quadratic construction algorithm.
3. Method:
   1. Split data up into random sets of leaves of similar sizes.
   2. Optimize each set into a tree.
   3. Combine pairs of sets to create larger optimized trees. (Do so by inserting one tree into the other tree in the optimal spot)
   4. Continue until only one tree remains

This method should also take O(n^2) time. It is good because it should complete much faster than the other two methods, but it might not create as good of a tree. This is because it does less comparisons overall in order to try and be more efficient

1. a. We give the following algorithm:
   * 1. Construct all possible subsets of 3 leaves. Select the one whose value is optimal based off a given objective function. We will use this tree as our base.
     2. Next, consider the 2n-3 edges that the current tree contains (the base has n = 3), then add the edge that gives the most optimal value.
     3. Set a constant for the occasional removal of the ‘worst’ leaves, resuming to step (iv)
     4. Continue to add the edges in the most optimal way based off (ii)
   1. Runtime analysis: This is still an O(t^3) algorithm since the first step requires setting up all possible combinations of 3 leaves given t nodes. To find the possible subsets we have (t choose 3) = (t)(t – 1)(t – 2)/3!, which is O(t^3) for the first step. For step 2), we look at the most optimal tree generated from step 1) and consider the 2t-3 edges which is O(t) time and add the optimal edge. For step 3), we set a constant to occasionally remove one of the worst leaves, which means that we must repeat an additional kt looks of edges where k represents the constant, however this is still O(t) time for each of these operations, since k is a constant. Therefore, our runtime is still bounded by O(t^3) time.
   2. Advantages:
      1. Compared to the O(t^3) algorithm, this should increase the constructed tree optimality since we are also removing the proposed non-optimal edges in aim to construct an even better edge
      2. Compared to the O(t^2) algorithm, this considers the optimal 3-leaf tree at the beginning for the base, instead of randomly selecting 3 leaves to use
      3. We have better control of the topology construction of the tree
   3. Disadvantages:
      1. Compared to the O(t^2) algorithm, this has a slower runtime.
      2. Compared to the O(t^3) algorithm, this is an even greedier approach, since it is deselecting some edges to improve optimality
      3. There is still no way for us to determine what the best way to set the constant, k, is
2. First, we construct the 3-leaf tree with S1, S2 and S3.

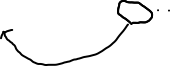
Then, we keep adding vertices onto the same first three edges until we reach the end.

The next step is to remove one edge from the end and add it to another “first 3 edges”.

Do the same thing recursively until we get all the trees.

For the time complexity, I am really stuck here so I would like to make a guess: it will be O(n^2).Pros: Save time comparing with O(n^3) construction, not so biased as total random.Cons: If the original model(the 3-leaf tree) is wrong, then the whole set of trees would be a bad set.

Like this:



1. I can run O(n^2) algorithm twice and build the consensus tree, and then, from the consensus tree, run O(n^3) algorithm to add the rest vertices. This algorithm is O(n^3) time complexity.

|  |  |  |
| --- | --- | --- |
| Method | Pros | Cons |
| O(n^2) | Can be faster | Is very greedy and the result could be influenced by random occurrence |
| O(n^3) | 1. Is more likely to find a better tree (but cannot guarantee it)  2. The tree is consistent every time we run this algorithm | 1. Is not suitable for larger trees  2. May not perform better than O(n^2) algorithm |

1. Proceed similarly to the O(n^3) algorithm, except after every two iterations, instead of adding on the next leaf that produces the best tree, remove the leaf that will result in the best tree. This “two steps forward, one step back” algorithm should maybe reduce the probability of getting trapped in some incorrect local maxima, improving slightly on the original O(n^3) algorithm and of course on the even rougher O(n^2) algorithm. We only have to traverse the tree once for each removing iteration, which takes O(n), and we do this O(n) times. That is an extra O(n^2) step overall and so the algorithm is still O(n^3). The drawback is that this method is even more costly than the original algorithm, and that the removal step may not even do that much to avoid incorrect greedy climbs.
2. In class, we discussed sequential tree construction with and without a random order for the leaf set. Another method is the divide and conquer method that uses subsets of trees with O(n^3) complexity. The method breaks down the problem to smaller trees, and later aggregates the small trees to create a large tree. One problem with this tree is based on unresolved data, there may be polytomus vertices. From a group of subtrees, each edge could be pasted together resulting in O(n^2) potential edit operations. Thus, each tree has O(n^3) potential neighbors for the edit operation. There is a tradeoff between neighborhood size and efficiency with the global optimum. This is less efficient than the O(n^2) algorithm from class which uses a random set of leaves. It is as efficient as the other O(n^3) algorithm, but the other O(n^3) algorithm is sometimes “too greedy” and is more likely to reach a local than a global optimal solution. A con to this algorithm is that if all small trees are equal neighbors, this is as efficient as an exhaustive search.
3. Similar to the greedy sequential tree construction method with O(n^2) complexity, you could start with a random tree but instead with 4 leaves. Then you would continue the same by iteratively finding the optimal place to add each leaf. This algorithm would have O(n^2) time as well, which is good, but it will not be locally accurate for the fourth tree, which is bad. However, there is a chance that the random fourth tree contributes to a better global accuracy.
4. Another greedy algorithm for tree construction is as follows. Arbitrarily select three leaves to constitute a T3 tree. Next arbitrarily add an edge anywhere on the tree. Then for each remaining leaf calculate \phi of T4 with the new leaf added to the new edge. Select the tree which is most optimal. Continue the process until all of the leaves have been added.

Analysis: Creating T3 can be done in O (1). Adding a new edge can be done in O (1). Calculating \phi of the new tree can be done in O (1) and this operation will be done O(t - k) = O(t) times for each new edge added. Thus, for t insertions the algorithm takes O(t^2) time.

The advantage of the O(t^3) algorithm is it does not assume an order of the leaves. Furthermore, at each addition it evaluates all possible insertions of each leaf. The disadvantage is it has a longer runtime and larger space complexity. The advantage of the O(t^2) algorithm is it runs faster and takes up less space. The downside is it assumes an order of insertions for the leaves which is not necessarily correct.

1. One approach we could take is ordering the leaves nodes by some sort of similarity score before inputting them to create trees. One way we could compute a similarity score is by first creating a ‘mode string’ that reflects the most frequent character at each position of the string. Next, for every string, sum up the number of position matches and return the sum as the similarity score. Computing similarity scores would take O(n), by using sorting algorithms such as quicksort we could sort leaves by similarity scores in O(nlogn), finally we can continue with the quadratic algorithm by taking in the sorted leaves and selecting the optimal edge for insertion by evaluating the objective function in O(n^2). All in all, this algorithm would take O (n + nlogn + n^2), or O(n^2) time.

With respect to the O(n^3) algorithm, this algorithm may not be guaranteed to find the best possible solution because it is not completely exhaustive (since we are fixing the ordering in which leaves may be inserted into the tree). However, it saves on time and is more feasible for computationally intense, large data sets.

With respect to the O(n^2) algorithm, this algorithm takes slightly more time at the beginning to create a ordering based on similarity scores (O(nlogn)) rather than generating a random ordering (O(n)). However, this ordering based on similarity score may lead to more optimal constructed trees since the inputted leaves are relatively alike to their neighbors.

1. Given a leaf set L of N leaves, partition it into groups of 3 randomly such that there are N/3 groups. Find the group whose 3-leaf tree gives the optimal value. This takes O(N/3) = O(N) time. Then starting from that group, select the next leaf in the input order and select the optimal edge to insert it into the starting group. This takes O(N) time when inserting every leaf. Continue this process until all leaves have been added. Since there are O(N) leaves to add, the process of inserting all leaves is O(N^2) time. So, the total time is O(N) + O(N^2) = O(N^2) time for this algorithm.