30/30

William Argus
A12802324
ECE 208

Homework 5 Writeup

Valid normalization of GTR transition rates:

Since the units of GTR rate matrix and units of branch length that are being scored need to be the same, the GTR rates must be normalized to be in substitution units. In order to apply the formula in the generative process, $P = e^{tR}$, time has to be scaled by dividing by *d*, given in the following formula from the lecture slides: $d = \sum -\pi_i R_{i,i}$. This was found in the code by multiplying the diagonals of the R matrix by the corresponding π_i value for that diagonal and then summing it. Then, as the same lecture suggested, when applying the formula $P = e^{tR}$, *t* was divided by *d* to make the formula for the R matrix for each node written as $P = e^{R\frac{t}{d}}$. It should be noted that since the value of *t* is different for node, this matrix *P*, which gives the probability of transition for each character when going from the parent to the given node, was recalculated and different for each node.

Description of Sequence Simulation:

Here the code will be described so that the time complexity analysis can be completed below. The first function, random_seq, was used to calculate a random sequence given the length of the sequence (k) and the probability of each letter occurring (gtr_probs). For this, numpy's random choice function was used to return a list of letters of length k, which letter selected by using the gtr_probs as the required probability distribution. Then using a for loop to loop through the sequence, each item in the list is appended to a string with the string being returned as the random sequence.

The second function, evolve simulates the evolution down the tree to generate a sequence for each leaf. This function is passed the tree, the root sequence (either the one given as an argument or randomly generated if only an integer was given), the gtr_probs (the stationary probabilities), and the gtr_rates (the transition rates). Two helper functions are defined within the evolve function. First is the normalize function, which takes in a 4x4 numpy array and divides every element in each of its rows by the sum of the elements in that row, in order to normalize each row, thus making it a valid probability distribution.

The second helper function is made to generate the new sequence for a node that is passed it, and runs on the assumption that this node's parent has already had its sequence generated previously (this is a fine assumption since the main loop uses preorder traversal, which, by definition, visits each nodes parent node before visiting that given node.) This function gets the node label, then sets the sequence as the root_seq. This is because if the root is passed in, it won't have a parent, so its sequence should just be the root sequence, and in an unrooted tree, the root node would not get past the if statement on the next line which checks if the node has a label. By not getting past that if statement, the root node's sequence is returned as the root_seq, which is the goal. The other nodes that get past the if statement then have their parent nodes and branch lengths collected and seq is cleared. Then using the R matrix for the correct probability distributions, the parent sequence is looped through and each character of the node's sequence is selected using the aforementioned probability distributions. This sequence is then returned.

In the main part of the evolve function, the R matrix is constructed according to the following visual, found in the lecture slides.

$$\begin{array}{c|cccc|}
 & A & G & C & T \\
A & -(\alpha\pi_G+\beta\pi_C+\gamma\pi_T) & \alpha\pi_G & \beta\pi_C & \gamma\pi_T \\
G & \alpha\pi_A & -(\alpha\pi_A+\delta\pi_C+\varepsilon\pi_T) & \delta\pi_C & \varepsilon\pi_T \\
C & \beta\pi_A & \delta\pi_G & -(\beta\pi_A+\delta\pi_G+\eta\pi_T) & \eta\pi_T \\
T & \gamma\pi_A & \varepsilon\pi_G & \eta\pi_C & -(\gamma\pi_A+\varepsilon\pi_G+\eta\pi_C)
\end{array}$$

The value for *d* is then found using the method mentioned in the previous section. In the main for loop, the nodes are looped through in preorder. First each node is checked if it is the root node, and if so, its label is assigned as the root sequence. This renders the if statement in the new_Seq function redundant but both were left in to ensure there were no errors. If the node is node is not the root, then the sequence for it is found using new_Seq. This sequence is then added to the seqs dictionary is the node is a leaf, otherwise it is set as the label for that node to be used when finding the sequences of that nodes children. Then the dictionary seqs is returned.

<u>Time Complexity Analysis:</u>

Note that *k* is the length of the sequence and *n* is the number of nodes in the tree passed in.

The first time intensive part of the code that depends on either sequence length or size of tree is the random_seq function. As seen earlier this function loops through a list of the length of the desired random sequence so it has runtime of *O(k)*, where k is the length of the sequence.

The second part is the preorder tree traversal in the evolve function. This loops through all nodes in the tree except the root and is therefore *O(n-1)*. Within this loop, the function new_Seq is called for each node, which loops through the sequence for each node. Accordingly this function has a runtime of *O(k)* for each time it is called. Since it is called within the tree traversal loop, this evolve function is *O(nk - k)*.

The total runtime is therefore *O(k)* (from the random_seq function) + *O(nk - k)* (from the evolve function).

Combined, this gives a worst case runtime of the entire program to be *O(nk)*.

Where *k* is the length of the sequence and *n* is the number of nodes in the tree passed in.