Three-State Perfect Phylogeny

Clodagh Kenny, Srikar Annapragada, and Nakul Iyer

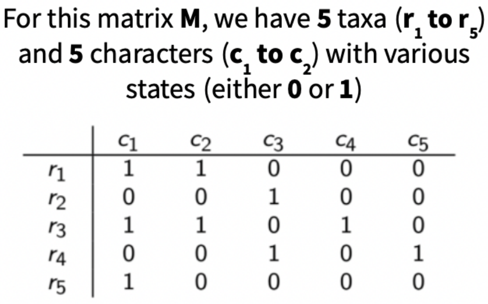
For CS 466 with Professor Mohammed El-Kebir at UIUC

December 18, 2022

Introduction

Note: all code referenced in this document is open-source and publicly available at https://github.com/nakuliyer/Multistate-Phylogeny/.

The goal of this implementation was to find an algorithm to solve the 3-state perfect phylogeny problem. In this class we have gone over the perfect phylogeny problem with both one and two states, and have discussed the algorithms that are used to solve these problems. In the perfect phylogeny problem, an $n \times m$ matrix, called M, with n taxa and m characters is inputted. The rules in order to verify that the tree exists and the tree is a perfect phylogeny are if M has state s at taxon f and character c, then f has the state s for character c. Additionally, every taxon labels one leaf and, starting from the root, every state shift on the edges leads to the taxon with those exact state shifts. The goal is to find a tree such that every edge represents a single character state change and every state change represents a single edge. Specifically, in the two state phylogeny, the input is a matrix M, with various state changes of either 0 or 1. As shown in the figure below this matrix M has 5 taxa (r1 to r5) and 5 characters (c1 to c5).



For this matrix **M**, we have **5** taxa (**r_1 to r_5**) and **5** characters (**c_1 to c_2**) with various states (either **0** or **1**)

	c_1	c_2	c_3	c_4	c_5
r_1	1	1	0	0	0
r_2	0	0	1	0	0
r_3	1	1	0	1	0
r_4	0	0	1	0	1
r_5	1	0	0	0	0

Figure 1: Matrix M for Two State Perfect Phylogeny

In order to create this tree, each row of matrix M is iteratively considered choosing a root node, and a starting node, where each of the subtrees is built out from the root to the last matching characters. In the figure below, the final tree for this matrix is shown where each edge c_i represents c_i changing from 0 to 1, which for each edge occurs only once in the whole tree. This tree will be considered a perfect phylogeny if all the rules, as discussed above, have been met. This algorithm can be solved in polynomial time.

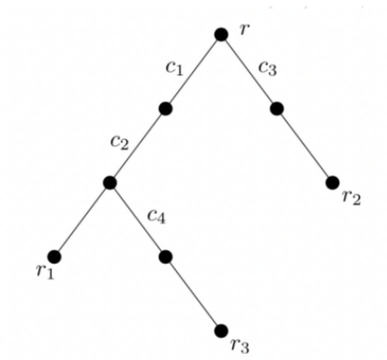


Figure 2: Final tree for Two State Perfect Phylogeny

Our algorithm focused on the 3-state problem which is much more challenging than the two state perfect phylogeny. In a two state the state tree for a cladistic character could be only 0 or 1, however now in a 3-state it can be 0,1 or 2. This gets much more complicated as a character can shift from 0 to 1 on one branch and 1 to 2 on that same branch or from 0 to 2 on one branch and 2 to 1 on that same branch as well as from 0 to 1 on one branch and 0 to 2 on another branch. The figure below shows this visually.

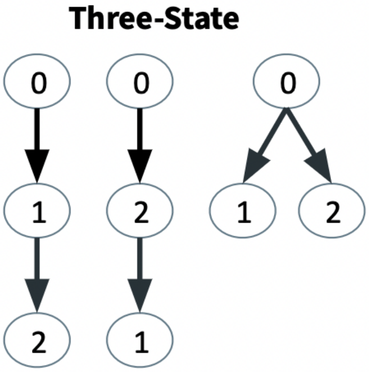


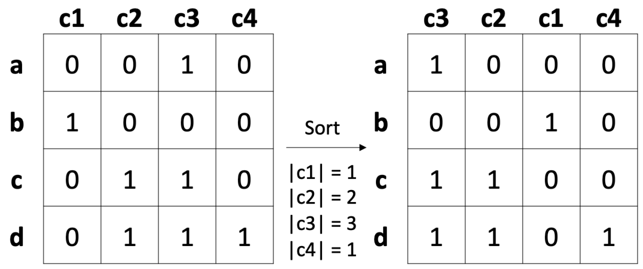
Figure 3: Multi-state Clastic Trees

The general 3-state phylogeny problem is NP-Hard, however in limiting the number of states to some k, then we can try all possible state trees for every character. For the three state problem there can be 3^n possible two state matrices. Since the two state phylogeny problem can be solved in linear time, the 3-state phylogeny problem can be broken into the two state problem, and if any of these are solvable, then for this set of state trees for all characters, the 3-state perfect phylogeny problem has a solution. This was the basis for which our algorithm was implemented, and our algorithm runs in $O(3^c)$ time for c characters. Note that our algorithm does not attempt to reduce or optimize the runtime beyond this; our goal was only to solve the problem.

Methods

Our algorithm is implemented in Python. We utilized the libraries NumPy and NetworkX, a useful tool for storing tree and graph data structures. The implementation code (in `implementation.py`) is split into four key methods:

`sort_characters(M: np.ndarray) -> Tuple[np.ndarray, np.array]` simply sorts (out-of-place) the columns of `M` based on the number of `1`s in the column in descending order. It returns the new matrix and an array representing the column indices of the original matrix according to the way they were sorted.



	c1	c2	c3	c4
a	0	0	1	0
b	1	0	0	0
c	0	1	1	0
d	0	1	1	1

	c3	c2	c1	c4
a	1	0	0	0
b	0	0	1	0
c	1	1	0	0
d	1	1	0	1

Figure 4: Sorting process for an example matrix (borrowed from CS 466 lecture slides). In this case, the second array returned would correspond to [c3, c2, c1, c4] as the new first column corresponds to the old third column, etc.

`extend_tree_for_taxum(T: nx.DiGraph, M: np.ndarray, taxum_idx: int) -> bool` is the key function which enables the construction of a two-state phylogeny tree. In constructing the two-state tree, this method is called on every row/taxum and extends the tree `T` to include the taxum and any new characters/edits it may have.

Firstly, a set `edits` is initialized to hold all the indices of all characters that are mutated for the taxum (i.e. $M[taxum\_idx][c] = 1 \Rightarrow c \in edits$). Then, the tree `T` is traversed in the following manner: starting from the root, for each node, if any outgoing edge is marked by a character in `edits` then remove it from `edits` and visit the node at the endpoint of the edge. This process continues until some node that has no outgoing edges labeled by a character in `edits` is visited.

Now, the remaining characters in `edits` are the characters we must extend the tree for. For each character c in `edits` (note that order matters, hence the reason for `sort_characters`), if c is already an edge in `T` then we return `false` indicating a conflict; otherwise, we add an edge labeled by c from the current node to a new node and traverse the edge. Finally, we add an edge from the current node to a new node labeled by the taxum.

For the following matrix, we demonstrate the process of calling `extend_tree_for_taxum` many times:

	c1	c2	c3	c4	c5
r1	1	1	0	0	0
r2	0	0	1	0	0
r3	1	1	0	0	1
r4	0	0	1	1	0
r5	0	1	0	0	0

Matrix M

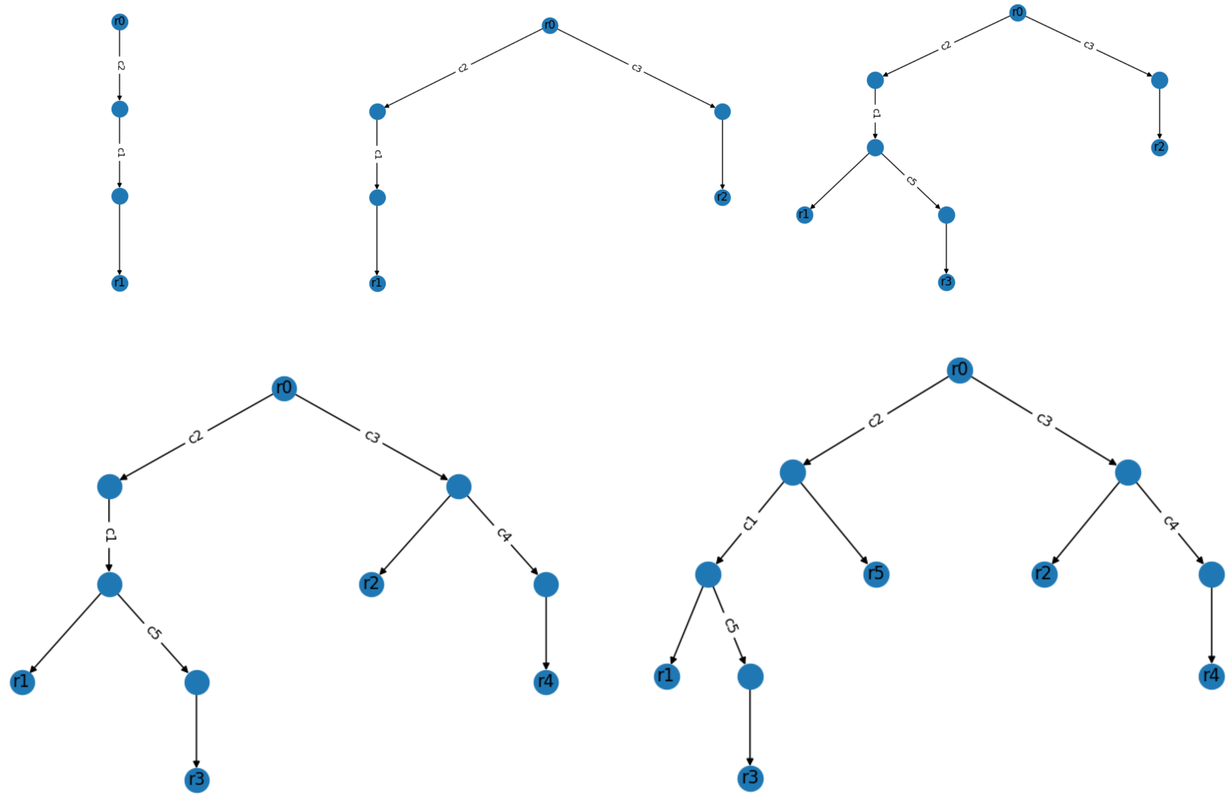


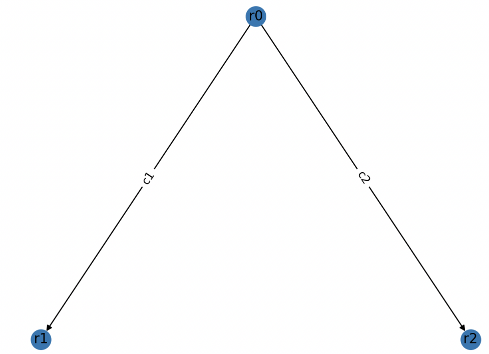
Figure 5: 5 iterations of `extend_tree_for_taxum` *for matrix M. Observe that in the first iteration, the edge c_2 is added* before *c_1 because the c_2 column for M has more 1s.*

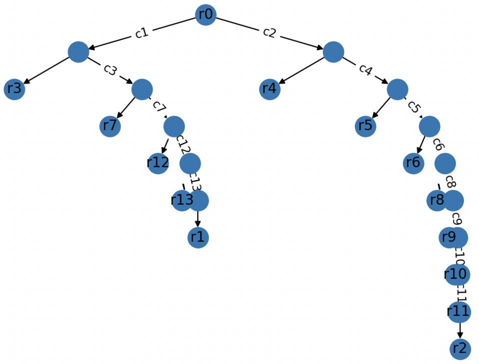
`two_state_phylo(M: np.ndarray, draw=False) -> nx.DiGraph` attempts to construct a conflict-free tree for an $n \times m$ input matrix `M` with n taxa and m characters. This method is undefined when `M` contains cells that are not `0` or `1`. If such a construction is impossible, `None` is returned; otherwise, the tree will be returned or optionally drawn if `draw=True`.

`three_state_phylo(M: np.ndarray, draw=False) -> nx.DiGraph` attempts to construct a conflict-free tree for an $n \times m$ input matrix `M` with n taxa and m characters. This method is undefined when `M` contains cells that are not `0`, `1`, or `2`. It does this by creating a two-state matrix for all permutations of all three state trees for every character (there are 3^c such matrices). If `two_state_phylo` succeeds in constructing a tree for any one of these matrices, the 3-state tree corresponding to it is returned optionally drawn if `draw=True`; otherwise, `None` is returned.

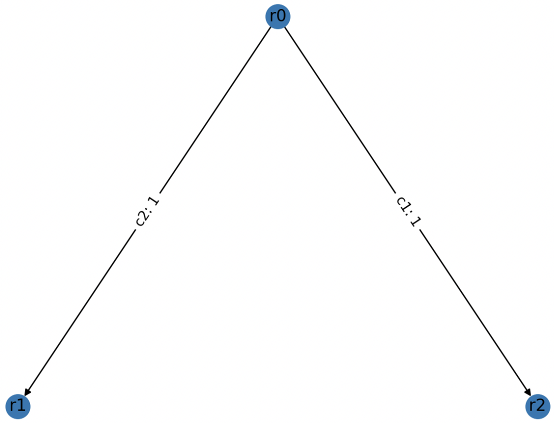
Validation Results

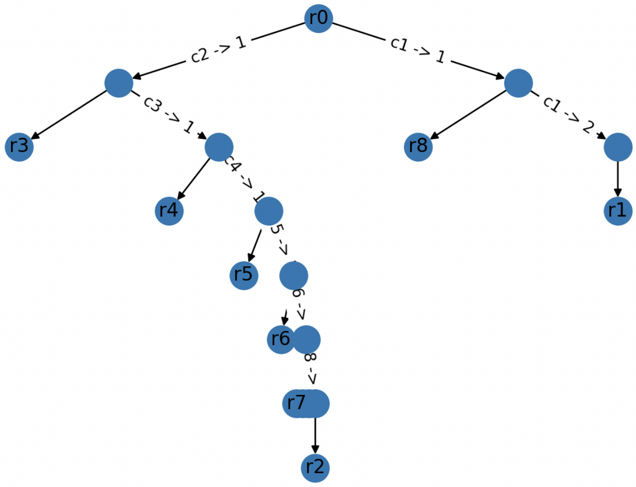
In order to verify our results, we had to think of datasets of phylogenetic matrices that have 2 states and 3 states so that we can test our algorithms. However, after searching for them, we have only come across limited ones. So we decided another approach which was to build a rooted tree starting from the top and edge by edge and node by node, making sure that each extension would satisfy the 2 state or 3 state perfect phylogeny conditions. This way, after extending the tree for some number of times, we can convert this tree into a Matrix and feed that into our algorithm to test it. These extensions of the tree occur randomly, so we can generate many different trees this way and have many different phylogenetic matrices to test.
For example, a generated 2-state test:





The above tree has been generated and transformed into a matrix, and has been passed into the `two_state_phylo` method to verify its perfect phylogeny. And similarly for a 3-state generation:





Now in order to test that the algorithm will fail on non-perfect state trees, we need to generate those two. Our tree generator can do this through the method `add_char_2state` or `add_char_3state.` If we pass in a False boolean in these methods, it will extend our tree by adding in an edge/ state change that will break our perfect phylogeny conditions. Overall to test all of these cases, we have run 100 tests for success giving a good matrix and 100 tests for a failure giving a bad matrix. Running `python testing.py` will run these tests and verify the correctness of our algorithm.

Our test code was able to generate 400 random perfect phylogeny trees and solve them all in 1.2176 seconds. Out of 50 randomly generated trees, our code was able to find a perfect phylogeny for 30 of them and reported conflicts for 20 of them in 21.1981 seconds. The extra time in randomly generated trees is owed to the fact that the algorithm has to iterate over all 3^c two-state matrices and find conflicts in all of them.

Conclusion

In conclusion, our algorithm solved the 3-state perfect phylogeny problem. This project allowed for the implementation of an algorithm that was covered in class but was not implemented as a part of the coursework. We were able to develop a greater understanding on the criteria for a perfect phylogenetic tree as well as developing greater knowledge in clastic state trees and the differentiation between two-state and multi state phylogeny.

Future Work

In the future, the 3-state phylogeny can be expanded in many ways, such as inputting a larger number of state changes beyond 3, or 4 etc. In this way you could have a substantially larger number of taxa, characters and species that could be represented. Additionally, we could also work to have matrices inputted with missing data to fill out the matrix and create a tree with a perfect phylogeny.

This project also largely ignored optimizations that can be done to reduce the time complexity of the three-state solution. Future iterations of this implementation might find ways to immediately detect a two-state matrix as conflicted based on the existence of particular sub-matrices. Doing so would drastically reduce the time complexity of the three-state algorithm since it requires 3^c calls to the two-state algorithm.

References

"2.2 Building Trees." Digital Atlas of Ancient Life, 15 Oct. 2019, https://www.digitalatlasofancientlife.org/learn/systematics/phylogenetics/building-trees/.

Korhonen, Tuuka, and Matti Jarvisalo. "Finding Most Compatible Phylogenetic Trees over Multi-State Characters." AAAI Publications, 2020, https://ojs.aaai.org/index.php/AAAI/article/view/5514/5370.

Rubin, Simon. "Multi-State Perfect Phylogeny." Brown CS: Student Theses, 2016, https://cs.brown.edu/research/pubs/theses/capstones/2016/rubin.simon.pdf.