Iowa State University
Digital Repository

Creative Components

Iowa State University Capstones, Theses and Dissertations

Spring 2019

Finding Median Reticulation Network

Venkata Sai Krishna Teja Vadali

Follow this and additional works at: https://lib.dr.iastate.edu/creativecomponents

Part of the Theory and Algorithms Commons

Recommended Citation

Vadali, Venkata Sai Krishna Teja, "Finding Median Reticulation Network" (2019). *Creative Components*. 269.
https://lib.dr.iastate.edu/creativecomponents/269

This Creative Component is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Creative Components by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

IOWA STATE UNIVERSITY, COMPUTER SCIENCE DEPARTMENT

Finding Median Reticulation Network

Krishna Vadali (vvadali@iastate.edu), Oliver Eulenstein,

Tavis Anderson, Alexy markin

January 8, 2019

1 INTRODUCTION

In recent years, the study of evolutionary virus networks has become particularly important. These studies give us insight into how and when various viruses are born and evolved. However, inferring evolutionary networks is an extreme computational challenge [5]. Evolutionary tree inference has a very long history and is a well-developed area. One of the most common and powerful approaches for large-scale evolutionary tree inference is to solve a so-called Median Tree problem. In the Median Tree problem, we are given a set of trees G, we find a tree T, called Median Tree, which closely represents all the trees in G. While generally, the median tree problems that are used in practice are NP-hard, they have been successfully tackled using the local search heuristics operating on standardized spaces of evolutionary trees.

However, such methods are not explored thoroughly for *Reticulation Networks*. A *Reticulation Network* is a phylogenetic tree with *reticulation vertices*. A *reticulation vertex* is a vertex in the network with more than one parent vertices. In this work, we provide a *Local Search Heuristic approach* to solve *Median Reticulation Networks Problem*: Given a set of input trees, find a Reticulation Network that displays all input trees with minimum reticulation vertices. In *Local Search Heuristics*, we define a local search space to optimize a reticulation network. We use standard *1-SNPR* operation defined by [3] to build local search space. We also introduce a new class of measurement to measure the distance between a Reticulation Network to a set of trees under *Robinson Foulds distance(RF)*. *RF* distance tells us how similar or different, two trees are. Here, we identify that computing *RF* between a Reticulation Network and a set of input trees is NP-Hard, and so we provide a faster algorithm which uses memorization for computing this measurement. Also, we provide a faster algorithm that explores the local search space in an intelligent manner to give us run-time benefits in practice.

2 RELATED WORK

Median tree problems which are typically used in practice, are NP-hard [2] in general, and therefore are approached by using local search heuristics [1, 4, 6, 7, 9] that make truly large-scale phylogenetic analyses feasible [7, 9]. Effective local search heuristics have been pro-

posed and analyzed [1, 4, 6, 7, 9], and provided various credible species trees [7, 9]. In this work, we apply similar techniques on phylogenetic networks solve Median Reticulation Network problem. The *Median Reticulation Network Problem* is to find a network that displays all the input trees with a minimum degree of reticulations. The problem of finding such Reticulation Network has already been explored [10] under deep coalescence evolutionary model using local search approach. But, it suffers from two issues: 1. Deep coalescence events are not very common in viruses. 2. The solution does not take errors in the input trees into consideration. Recently in 2017, [3] published a rigorous way explore local search space on a network using *SNPR* operations.

3 CONTRIBUTION

In this work, we provide a *Local Search heuristic* approach to solve *Median Reticulation Network Problem*. Although a Median Reticulation Network requires every input tree to be displayed by it exactly, in practice, the input trees are not always correct. We introduce *RF* based measurement to measure the distance between the set of trees and a network that takes incorrect trees into consideration. Here, we identify that computing this measurement is NP-Hard, and so we provide a faster algorithm which uses memorization to improve runtime. Finally, we provide a faster algorithm that explores the local search space in an intelligent manner that provides run-time benefits in practice.

4 PRELIMINARIES

A *(phylogenetic) network* is a directed acyclic graph (*DAG*) with a designated root and all other vertices are either of in-degree one and out-degree two (*tree vertices*), in-degree two and out-degree one (*reticulation vertices*), or in-degree one and out-degree zero (*leaves*). All leaves are bijectively labeled by a label-set X and are identified with the elements of X. For convenience, networks are *planted*, i.e., the root has in-degree zero and out-degree one. For a network N the root and leaves are denoted by $\rho(N)$ and $\mathsf{L}(N)$ respectively. For every node v we denote the set of children, parent(s), and sibling(s) by $\mathsf{Ch}(v)$, $\mathsf{Pa}(v)$, and $\mathsf{Sb}(v)$ respectively.

We distinguish *reticulation edges* – edges entering a reticulation vertex – and *tree edges* – edges entering a tree vertex. A *tree-path* is a directed path that consists of tree edges.

A vertex v is a *descendant* of w is there is a directed path from w to v (we consider each vertex to be a descendant of itself); w is also called an *ancestor* of v. A *cluster* of vertex v, C_v, is the set of leaves that are descendants of v.

A *(phylogenetic) tree* is a network with no reticulation vertices. A *least common ancestor (LCA)* of two nodes v, w in a tree, $\mathsf{lca}(v, w)$, is the farthest from the root vertex x such that v and w are descendants of x. For a node v in a tree T by T_v we denote the subtree of T rooted at v. The size of a tree T is defined as $|T| := |\,\mathsf{L}(T)|$.

A tree T is *displayed* in a network N (with the same leaf set), if one can remove exactly one reticulation edge from each reticulation node, then remove all potentially appearing non-labeled vertices with out-degree zero, and obtain a subdivision of T.

Tree-child networks, A network is called *tree-child* if each node has at least one outgoing tree edge. It is easy to see that each node in a tree-child network must have a tree-path going to some leaf.

Embedding cost, We define the cost of embedding a tree T in a network N on the same leaf set using the standard Robinson-Foulds (RF) distance. The cost should be zero, when the tree is displayed in the network and positive otherwise. Hence, we define the cost as follows: let $\mathcal{P}_N$ be a set of all trees displayed in N, then

$$d(T, N) := \min_{G \in \mathcal{P}_N} RF(T, G),$$

where $RF(T, G)$ is the Robinson-Foulds (cluster) distance defined as the size of the symmetric difference between the cluster representations of two trees [8].

5 PROBLEM DEFINITION

Given a set of m trees $G = \{t_1, t_2, ...t_m\}$ with $Taxon \subseteq T = \{1, 2, ...n\}$, and $K \geq 0$, maximum number of reticulation vertices allowed, the Median Reticulation Network problem to find a Reticulation Network N, containing n Taxon, such that N displays all the trees in

G as well as possible with at most K reticulation vertices. Here, the trees in G may not be complete, meaning a Tree can contain Taxon $\subseteq$ T.

The above problem definition is not complete until we define how do we quantitatively measure the distance between Trees in G and a given network N. We define the distance $D(G, N)$ to be the summation of embedded costs of each of the trees in G.

$$\text{Distance between } G \text{ and } N,\ D(G, N) = \sum_{tinG} d(t, N)$$

6 METHODOLOGY

To make the further discussion easier and to get an intuition about what we are trying to solve, we begin this section by detailing a little bit on a simple example.

6.1 Example

Let us consider two gene trees G_1, G_2 displayed below and we seek best Median Reticulation network with K = 1, any correct algorithm could potentially produce N displayed below since it is one of the optimal networks. Next, we shall go understand why N is considered as one of the optimal networks.

The optimal network N has one reticulation vertex which has two parent edges (Red and Purple as indicated in the figure). This means that the network displays two trees T_1, and T_2 each one choosing one of the edges respectively. Let us say, with out loss of generality, T_1 chooses purple edge and so T_2 chooses Red edge. Then, $T_1's$ non-trivial clusters are { {b, c }, {e, b, c }, {a, d }, {a, b, c, d, e } } and similarly in T2 we have { {b, c }, {a, b, c }, {a, b, c, d }, {a, b, c, d, e } }.

Also, G_1 and G_2 have { {b, c }, {a, b, c }, {d, e }, {a, b, c, d, e } } and { {b, c }, {e, b, c }, {a, d }, {a, b, c, d, e } } non-trivial clusters respectively. When we try to superimpose G_1 on both T_1 and T_2, G_1 fits into T_2 better because it only has one cluster different from T_2, whereas with T_1 it has two different clusters and therefore we say T_2 closely displays G_1. Similarly G_2 is completely displayed by T_1 with zero mismatches. Also there is no N with one reticulation that has less than 1 mismatch and therefore N is one of the optimal network.

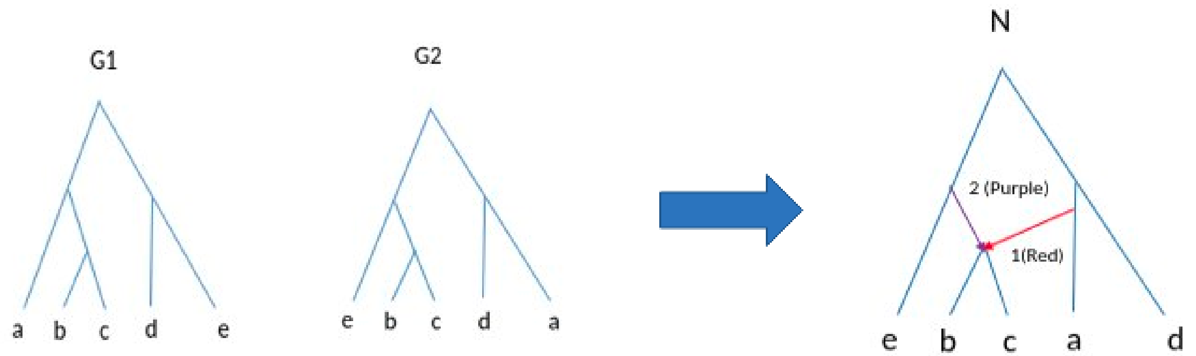

Figure 1 : G1, G2 are Gene Trees , N is one of the optimal Reticulation Network with **one** Reticulation Vertex.

Figure 1: Optimal Network with One Reticulation.

6.2 Naive Method

In this section, we describe a naive method to compute *Median Reticulation Network* that runs in exponential time in n and K.

The input to above method are set of gene trees G, the number of Taxon n and a maximum number of reticulation vertices possible K. The process begins by enumerating all possible trees on given n Taxon in line 1. The main *While* loop from line numbers 4-16 computes set of all possible networks that are optimal with a reticulation vertices. The *For* loop in Lines 6-15 picks each optimal network t from the previous iteration and finds possible optimal networks that can root out of t. In line 7, it enumerates all possible networks that are valid after adding one additional reticulation vertex. The *For* loop from 8-15 looks at each of these networks, and for each of the network r, it computes *RF* distance from G to network. If the distance is smaller than the $bestScore$, it treats r as a potential candidate network and puts it in the candidate set $best$ in lines 13 and 15. When the main *While* loop breaks, the method returns any random network form the available optimal networks.

The method $MedianReticulationNetworkNaive$ correctly computes an optimal network under *RF* distance measurement. The main argument for proof of correctness is that the

Algorithm 1: MedianReticulationNetworkNaive (set of input trees G, number of taxon n, maximum reticulation vertices possible K)

```
1  best ← {Enumeration of all possible Trees on n Taxon}
2  bestScore ← ∞
3  a ← 0
4  while a < K do
5  |  nextBest ← best
6  |  foreach t in nextBest do
7  |  |  R ← {Enumeration of all possible one reticulations on t}
8  |  |  foreach r in R do
9  |  |  |  score ← ComputeRFBasic(G, r)
10 |  |  |  if score < bestScore then
11 |  |  |  |  bestScore ← score
12 |  |  |  |  best ← { }
13 |  |  |  |  best ← nextBest ∪ r
14 |  |  |  if score = bestScore then
15 |  |  |  |  best ← nextBest ∪ r
16 |  a ← a + 1
17 return any network in best
```

method explores every possible network and finds the set of networks that are optimal. Also, since the two *For* loops and the *While* loop only run a finite number of iterations and also as we will see $ComputeRFBasic$ force albeit runs in exponential time, returns eventually, the method will terminate.

Although method $MedianReticulationNetworkNaive$ terminates eventually, it runs in exponential time and hence not practical for any larger inputs. This can be seen easily, initially, the number of trees is exponential in n, also $ComputeRFBasic$ runs in exponential in a (Number of reticulation vertices it contains).

6.3 Local Search Heuristics

As we observed in the previous section, the naive method runs in exponential time to compute an optimal Reticulation Network. In fact, the problem of computing *RF* distance is $NP-Hard$ (Proven by our group, will be published), this implies finding an optimal reticulation network under *RF* measure is NP-Hard.

In this section, we explore another exponential heuristic method that does not compute the exact optimal network, but a close enough one for most practical purposes. Indeed, this is still exponential but in k not in n and k. This is useful since it is known that in evolutionary history there are not many reassortment events, it is known that these events are typically less than 20.

We begin by exploring a general method to solve this problem and provide naive implementations of sub-operations. Then, we optimize some of these sub-operations, especially RF-Computation and local search space exploration to get a better runtime practice.

Instead of working on all possible trees, the $HeuristicMedianReticulationNetwork$ takes a different direction in that, it computes Median Tree of input Gene Trees G. There are several methods that compute an exact Median tree but take exponential time, also there are good heuristic methods that compute good Median Tree in polynomial time. Depending on the size of the input we can either compute the exact Median Tree or a good enough one in line 1. We then run the main *While* loop like in the naive approach from lines 4 to 8. Again, like in the brute naive approach, we find the best network after introducing one reticulation in N in line 6. In line 7, we explore the local SNPR space until it converges at some local minimum.

Algorithm 2: HeuristicMedianReticulationNetwork (G, n, K)

```
1 best ← ComputeMedianTree(G)
2 bestScore ← ∞
3 a ← 0
4 while a < K do
5     nextBest ← best
6     R, bestScore ← FindBestNetworkAfterOneAdditionalReticulation(G, bestScore,
        nextBest)
7     best, bestScore ← FindOptmialLocalNetworkBySNPR(G, bestScore, R)
8     a ← a + 1
9 return best
```

Before looking into the sub-operations, we will argue that the method $HeuristicMedianReticulationNetwork$ terminates. It is known that $ComputeMedianTree$ terminates. We need to argue that sub operations $FindBestNetworkAfterOneAdditionalReticulation$ and $FindOptmialLocalNetworkBySNPR$ terminate further.

$FindBestNetworkAfterOneAdditionalReticulation$ considers every pair of edges and finds a pair of edges that produce the best score and uses those edges to introduce additional reticulation vertex. Hence it only runs in the polynomial in n and therefore it terminates. On the other-hand $FindOptmialLocalNetworkBySNPR$ runs until it converges in some local optimum, we cannot really say that this runs in some polynomial time, but we know that the lower bound on score is 0 therefore, it cannot run infinitely since once the score reaches 0, It knows that the current network is optimal network and there is no better network possible.

Algorithm 3: FindBestNetworkAfterOneAdditionalReticulation(Set of Gene-Trees G, Optimal Score S, Network N)

1 $E \leftarrow N.edges$
2 $best \leftarrow N$
3 **foreach** $e1$ *in* E **do**
4   **foreach** $e2$ *in* E **do**
5   **if** *ValidReticulation(*N*,* $e1$*,* $e2$*)* **then**
6             $R \leftarrow$ Reticulate(N, $e1$, $e2$)
7             $D \leftarrow$ ComputeRFBruteForce(G, R)
8   **if** $D < S$ **then**
9                 $best \leftarrow R$
10                $S \leftarrow D$
11 return $best$, S

To add a new reticulation, we need to choose a valid vertex to recirculate. Two critical properties that must be preserved.

1. N must not contain a directed cycle.
2. N must not violate Tree-Child Property.

Function $ValidReticulation$ exactly checks these properties after performing reticulate operation given Network N, $parent$, $child$ edges

Function ValidReticulation(Netwrok N, parent edge $parent$, child edge $child$)

1 $R \leftarrow$ Reticulate(N, $e1$, $e2$)
2 **if** R *contains directed cycle OR treechildPropertyViolated(*R*)* **then**
3   return NO
4 return YES

The main *while* loop in $FindOptmialLocalNetworkBySNPR$ runs until it reaches N that is optimum in $1-SNPR$ pace. The *For* loops inside compute the best network reachable

by doing $1 - SNPR$ operation, call it $best$. Next, we move to the $best$ network and try to optimize further by finding $1 - SNPR$ network in its space. Note that we can never end up in a loop because a network cannot be reached twice. A single iteration of the outer *While* loop runs in time O($|N.V|^2$ * (Time to compute RF)).

Algorithm 4: FindOptmialLocalNetworkBySNPR(Set of Gene-Trees G, Optimal Score S, Network N)

```
1  optimumNotReached ← True
2  while optimumNotReached do
3      optimumNotReached ← False
4      E ← N.edges
5      foreach e1 in E do
6          foreach e2 in E do
7              if ValidSNPRMove(e1, e2) then
8                  R ← PerformSNPRMove(N, e1, e2)
9                  D ← ComputeRFBruteForce(G, R)
10                 if D < S then
11                     best ← R
12                     S ← D
13                     optimumNotReached ← True
14     N ← best
```

$ValidSNPRMove$ works exactly like $ValidReticulation$, it checks after doing SNPR move if the network is an invalid condition, which means preserving the below properties.

1. N must not contain a directed cycle.
2. N must not violate Tree-Child Property.
3. N must not be broken, meaning the network should be connected.

Function $ValidSNPRMove$ exactly checks these properties after performing SNPR operation given Network N, $parent$, $child$ edges

Function validSNPRMove(Network N, Parent edge $parent$, Child edge $child$)

```
1  R ← performSNPRMove(N, e1, e2)
2  if R contains directed cycle OR treechildPropertyViolated(R) OR R is not connected
     then
3      return NO
4  return YES
```

The function $treechildPropertyViolated$ helps us find it, the network still preserves Tree-Child property.

1. Every vertex must have at least one tree edge going out of it.

Function treechildPropertyViolated(Network N)

```
1  V ← N.V for v in V do
2      if v is a reticulation vertex then
3          for (v, w) in N.E do
4              if w is a reticulation vertex then
5                  return YES
6      else
7          for (v, w) in N.E do
8              if w is not a reticulation vertex then
9                  return NO
10         return YES
11 return NO
```

The $ComputeRFBruteForce$ method as the name suggests computes the *RF* distance between the network N and set of Gene Trees G in a naive manner. It starts by enumerating all the edges into sets of edges such that in each of the set, exactly one parent edge is included for every vertex v in $N.V$ except the root.

Observe that each of these edge sets, along with the vertices in $N.V$, form a tree. for each of these trees, we run the *For* loop to compute the *RF* from each of the input trees in G and record the minimum score. We then return the sum of the scores as a distance from N to G. it is easy to see that this method runs in exponential on the number of reticulation vertices since every reticulation vertex has two choices for choosing the parent.

Algorithm 5: ComputeRFBruteForce(set of Gene Trees G, Network N)

```
1  score ← {∞, ∞, ....., |G| times }
2  S ← { Enumeration of all combination of edges such that for each v in N.V, we
       choose exactly one parent }
3  foreach edgeSet in S do
4  |   C1 ← nontrivialClusters(N, edgeSet)
5  |   foreach g in G do
6  |   |   C2 ← nontrivialClusters(g)
7  |   |   common ← commonclusters(C1, C2)
8  |   |   score[g] ← min(score[g], |C1| + |C2| − 2 ∗ |common|)
9  totalScore ← 0
10 foreach g in G do
11 |   totalScore ← totalScore + score[g]
12 return totalScore
```

7 ADVANCED OPERATIONS

Here, we re-visit the above heuristic method and propose some improvements for a better run-time of the algorithm.

7.1 Fast RF computation

In the basic method, as we have seen, for every input Gene Tree g we generate all possible embedded trees in the network and try to find the best one, which is having the least *RF* distance

to g. We identify two significant improvements to this method. First, the cluster comparison is O(n), we would use LCA comparison to which is O(1). Second, we identify clusters which change and clusters which remain the same among embedded trees and memorize meta-data to avoid recomputing these values. Both of these improvements give us an improvement in run-time.

The idea of computing LCA's to compare clusters has been explored before. The key idea is the following: For any arbitrary vertex v having children C in a network N, if $LCA(C) == l$, where l is any vertex in the Gene Tree g, and if $|leafs(l)| = |leafs(v)|$, i.e the cluster sizes are equal for v in N and l in g, then their clusters should match.

We use the above idea and compare cluster sizes instead of comparing each element in the clusters. Clearly, this needs us to find the $LCA(C)$ as efficiently as possible. But, since input trees are static and do not change at all in the process of the algorithm, for each Gene Tree g in G, we precompute $LCA(u, v)$ where $u, v \in g.v$. Hence $Lca(C)$ should take only constant time.

The second improvement comes from the observation that only a few nodes have their clusters changed from one embedded tree to the other. If we order these embedded trees in N in such a way that there is only a few nodes have their clusters changed from the last processed embedded tree, then we can get away by only recomputing those changed nodes. The hard part is finding such an order. Since N does not contain a cycle (maintained by Tree-Child property), N is a Directed Acyclic Graph (DAG). We know that every DAG has a topological order, and so we use this fact to find the topological order for N.

Now, let $Top(N) = \{v_1, v_2, \ldots, v_n\}$ be the Topological order of vertices in N. Without loss of generality, let us assume v_k is the last reticulation vertex. Then we know that all the nodes from $\{ v_{k+1}, v_{k+2}, \ldots, v_n \}$ have their clusters not changed through out all the embedded trees. Therefore we only need to compute and match these clusters only once.

We can extend the above idea further on. Let $\{ r_1, r_2 \ldots r_k \}$ are the reticulation vertices ordered in topological order. Then, $\forall p \in \{1, 2, \ldots, k\}$, we can have same edge choices for the nodes in $\{r_p, r_{p+1} \ldots, r_k\}$ for all combinations of edge choices in $\{r_1, r_2, \ldots, r_{p-1}\}$. This means, we are only recomputing clusters for some subset of vertices in $Top(N)$. The below

$FastRFCompute$ method formalizes these two ideas to attain a better run-time.

The method $FastRFCompute$ starts by computing the LCA's for each pair of nodes in for each of the input Gene Trees in line number 1. In line 2, it computes the topological order of vertices in N. In line number 3, we extract the set D of vertices from N, such that D contains only the vertices that have a directed path to a reticulation vertex. in line 4, we order these vertices according to order in $Top(N)$. The main ***For*** loop from lines 6 to 9 executes the main operation on all the bit vectors of length equal to the number of reticulation vertices considered in lexicographic order. In lines 7 and 8, we call a sub-routine $ComputeSimilarityDynamic$ which computes the similarity score between the input trees T and the embedded tree displayed by $N.V$ along with the bit vector A. If $A[i] = 0$, we pick the first parent and ignore the second parent, similarly, if $A[i] = 1$, we pick the second parent for the ith reticulation vertex in the order. In line 9, we update the best score for each of the input trees. In Line 12, we return the distance between input trees T and the network N.

Now, we argue that $FastRFCompute$ terminates. We know that Computing LCA's on T and Topological ordering on N terminate. Also, the ***For*** loop executes exactly $2^{|reticulations|}$ times, hence it terminates as long as $ComputeSimilarityDynamic$ terminates.

Now, we argue that $FastRFCompute$ is correct by arguing that it considers all possible embedded trees and not more or less. It is easy to see this is true, we know that any vertex v in $N.V$ either has exactly one or two parents (except the root vertex, which has 0). Also, we know that all the normal (not reticulation) vertices have exactly one parent. Therefore, there is no choice but to pick it. The reticulation vertices, however, have 2 parents, so for every combination of these edges, we get a different embedded tree. The ***For*** loop considers all of these possible choices to build different embedded trees which are $2^{|reticulations|}$ which is exactly number of times the ***For*** loop executes.

The function $ComputeSimilarityDynamic$ computes similarity score, which is the number of matching clusters between input tree T and the embedded tree displayed by r-bit vector A. The main ***For*** loop runs from start index j to end index k indices, in each of the iterations it computes the similarity score for one vertex v that changes its cluster from the previous embedded tree. if v is leaf we mark the size of the cluster is 1 and similarity as 1 and mark

Algorithm 6: FastRFCompute(Network N, Gene Trees T)

1 Precompute T to find LCAs for all pairs of nodes in T in constant time

2 Let Top be a topological ordering of vertices in N; and $V := Top$-reversed

3 Let D be the set of vertices in N that have a directed path to a reticulation vertex (i.e., all ancestors of reticulation nodes)

4 $P := V[D]$ (ordering V restricted to D)

5 $s := 0$

6 **for** *each r-bit binary vector A in the lexicographic order* **do**

7 $A = 00\ldots0$ then $ComputeSimilarityDynamic(V, 1, |V|, A)$

8 Otherwise let $1 \leq i \leq |V|$ be the left-most position in A by which A differs from the previous vector in the lexicographic order. Then $ComputeSimilarityDynamic(P, i, |P|, A)$

9 If after the computations $\sigma(\rho(N)) > s$, then $s := \sigma(\rho(N))$

10 **return** $2 \cdot (2|T| - 1) - 2s$.

$lca(v)$ = leaf from T with same label as v. If v is not a leaf and is a reticulation vertex, then we copy similarity, size and $lca(v)$ from its only child. If v is not a reticulation vertex and does not point to a reticulation vertex, then we compute its similarity by summing the similarity of its children, but adding 1 to the similarity score if $|lca(v.childern)|$ = $|v|$, also, we compute size as the sum of the sizes of its children. If v is pointing to a reticulation node and if the edge is part of the current embedded tree, then we consider it as a normal vertex and do the same computation as above. If v is not reticulation and is pointing to a reticulation vertex, but the edge is not part of the current embedded tree, then we copy the similarity, size, and lca from its valid child.

Here, we argue that $ComputeSimilarityDynamic$ terminates. It executes at most O(n) iterations once for each node, where n is the size of the N.

Now, we argue that $ComputeSimilarityDynamic$ computes similarity score correctly. When v is a leaf, this is trivial. If v is a reticulation vertex, it has the same cluster as its only child, hence this is true as well. If v is not a reticulation vertex and both edges are

Function ComputeSimilarityDynamic(Leaf ordering O, start index j, end index k, r-bit vector A)

```
1  for i ∈ j, j+1, ..., k do
2      Node v := O[i];
3      if v is a leaf then
4          μ(v) := leaf from T with same label as v;
5          λ(v) := 1; σ(v) := 1
6      else
7          if v is a reticulation vertex then
8              Let c be the child of v; μ(v) := μ(c); λ(v) := λ(c);
9              σ(v) := σ(c)
10         else
11             Let c_1 and c_2 be children of v
12             if neither c_1 nor c_2 are reticulations then
13                 μ(v) := lca_T(μ(c_1), μ(c_2)); λ(v) := λ(c_1) + λ(c_2)
14                 σ(v) := σ(c_1) + σ(c_2) + I[|T_μ(v)| = λ(v)]
15             else
16                 WLOG assume that c_1 is a reticulation and v is the first parent of c_1
17                 If A[c_1] is 0 (i.e., first parent is chosen) then proceed as in lines 13, 14
18                 Otherwise proceed as in line 9 with c := c_2
```

part of the embedded tree, the size must of the sizes of the children, and LCA will be the $LCA(v.children)$, and the similarity score will be the sum of the similarity of children. Only if the $cluster(v)$ is same as the $cluster(LCA(v))$ we add one to similarity, this is precisely what we are doing with the LCA size comparison. If one of the outgoing edges of v is not in the embedded tree is not part of the tree, we can compress another edge, we have the same effect by copying values from the valid child.

7.2 Fast Local SNPR Search

The key idea behind the $FastLocalSNPRSearch$ roots from the observation that, a NNI move in a network can change the *RF* distance by atmost 1. So, for each edge (w, x) in the network, we prune the edge (w, x) and perform *NNI* moves in the topological order of the network, computing exact *RF* distance only whenever it is required. For any edge (u, v), we compute the *RF* distance by re-grafting the edge (u, v) if the minimum distance obtained re-grafting every valid edge (y, u), d - 1 is less than optimum distance obtained so far. If not, we mark the minimum distance at v as d - 1.

Here, we will argue that the algorithm $FastLocalSNPRSearch$ terminates. We already argued that the outer *While* loop terminates. Also, we only have $O(n)$ edges and for each edge, we perform $O(n)$ *NNI* moves. We also know that at any point of time the network does not contain a cycle (Its a *DAG*), hence no edge is re-grafted twice. Together, we have $O(n^2)$ operations for one iteration of the outer *While* loop and since the *While* loop terminates, the algorithm $FastLocalSNPRSearch$ terminates.

Now, we argue that $FastLocalSNPRSearch$ is correct in that it does not miss any network in the search space. We know we can generate all *SNPR* moves by series of *NNI* operations, also we perform all possible *NNI* moves, this ensures that all *SNPR* networks are generated.

Below, we provide pseudo-code for $FastLocalSNPRSearch$ algorithm. The algorithm takes a network N, set of gene trees G, the optimum distance has seen so far with the trees $OptDistance$, and computes optimum network in *1-SNPR* space.

Algorithm 7: FastLocalSNPRSearch(Network N, Set of Gene Trees G, Optimum distance seen so far $OptDist$)

```
1  while N is not local optimum do
2      optNetwork ← N
3      top ← TopologicalOrder(N)
4      foreach Edge e(u, v) in N.E do
5          optDistance ← {∞, ∞, ..., N.Vtimes}
6          optDistance[N.root] ← optDist
7          foreach w in top do
8              if (Can Perform SNPR on ((w, w.left), e)) then
9                  if optDistance[w] − 1 < OptDist then
10                     M ← perfomSNPR(N, ((w, w.left), e))
11                     dist ← computeDistance(M, G)
12                     if dist < OptDist then
13                         optNetwork ← M
14                         optDist ← dist
15                     optDistance[w.left] = min(optDistance[w.left], dist)
16                 else
17                     optDistance[w.left] =
                        min(optDistance[w.left], optDistance[w] − 1)
18             if (Can Perform SNPR on ((w, w.right), e)) then
19                 if optDistance[w] − 1 < OptDist then
20                     M ← perfomSNPR(N, ((w, w.right), e))
21                     dist ← computeDistance(M, G)
22                     if dist < OptDist then
23                         optNetwork ← M
24                         optDist ← dist
25                     optDistance[w.right] = min(optDistance[w.right], dist)
26                 else
27                     optDistance[w.right] =
                        min(optDistance[w.right], optDistance[w] − 1)
28     N ← optNetwork
29 return (optNetwork, OptDist)
```

8 EXPERIMENTS

We conducted experiments to verify our solution and compared with the results from the naive method. Below are some of the observations made.

Parameters To compare	Brute Force	Our Approach
26 Taxon, with 2 input trees	Terminates with a network with 9 reticulations	Terminates with a network with 11 reticulations
26 Taxon, with 8 input trees	does not terminate	Runs upto 18 reticulations and then slows
120 Taxon, with 8 input trees	does not terminate	We have ability to run upto certain reticulations and then terminate.

Also, we ran our $HeuristicMedianReticulationNetwork$ on 26 Taxon with 2 input trees, using $ComputeRFBruteForce$, $FastRFCompute$ and $FastLocalSNPRSearch$. They took 304 and 108, 88 secs time respectively, thus $FastRFCompute$ gave us 65% gain and $FastRFCompute$ along with $FastLocalSNPRSearch$ gave 71% gain in run time.

Below are the networks computed on 26 Taxon data and H3 data set with 164 Taxon.

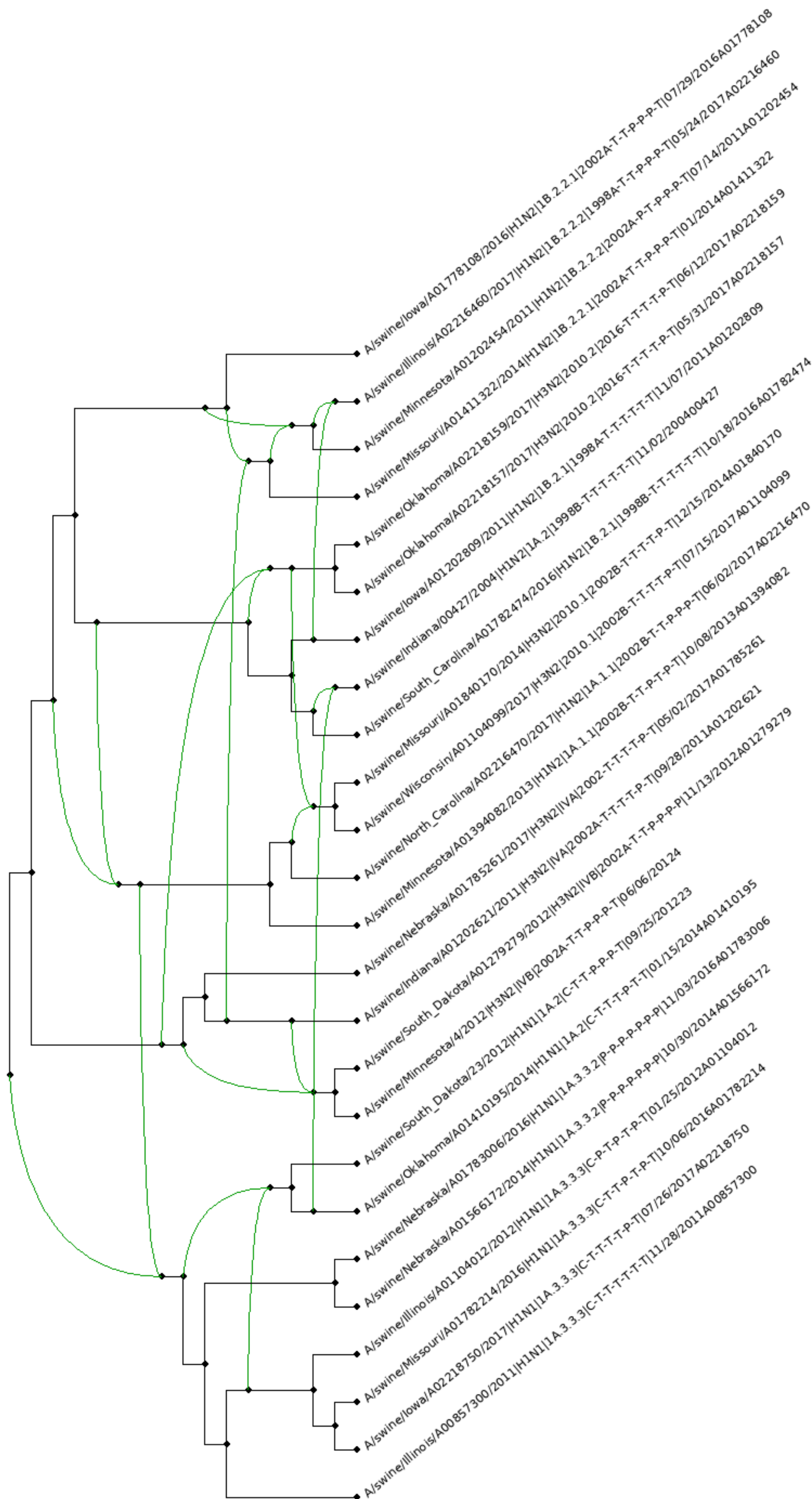

Figure 2: Optimal Reticulation Network on 26 Taxon data set computed with HA and NA trees and 10 Reticulations.

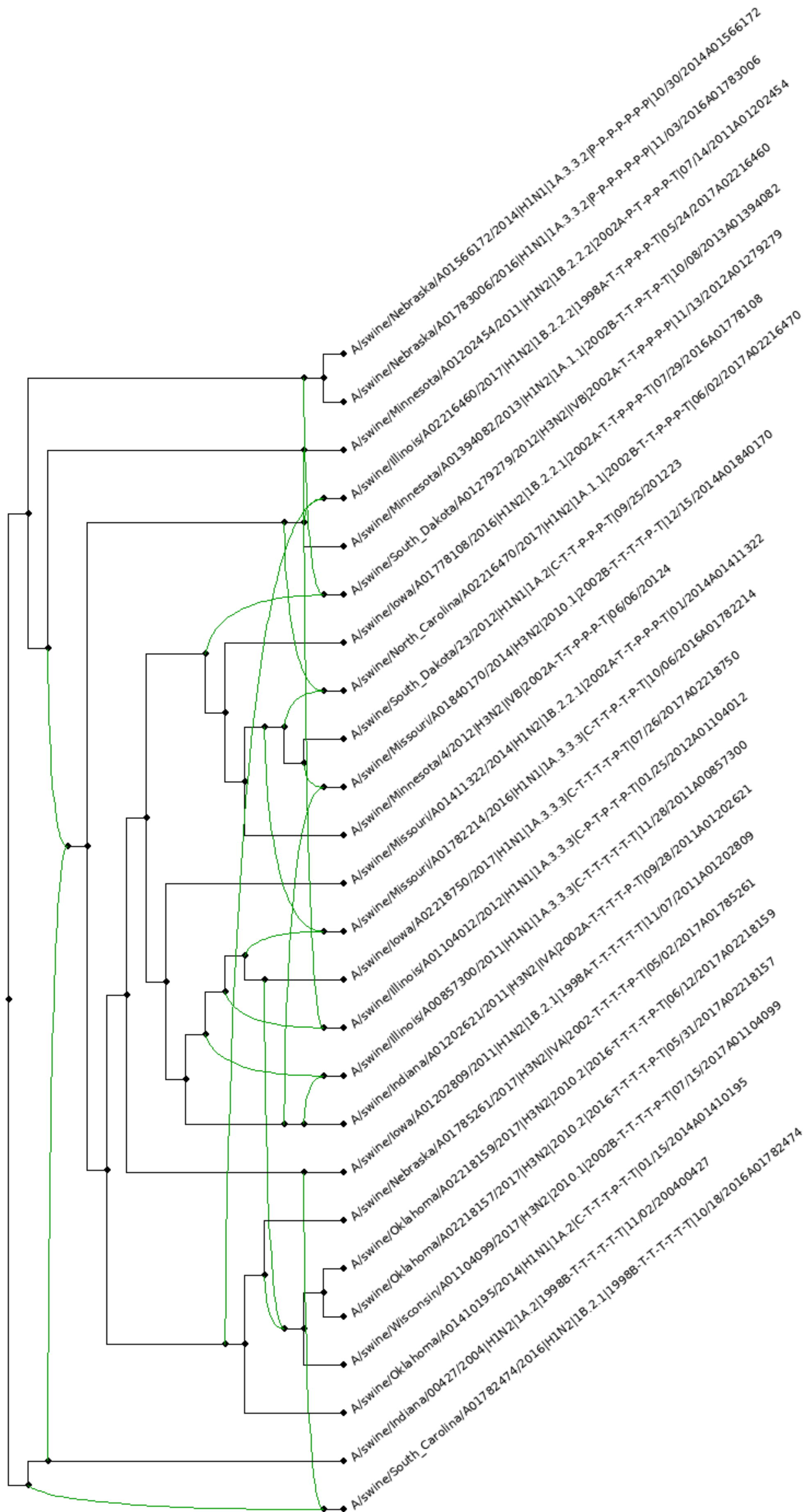

Figure 3: Optimal Reticulation Network on 26 Taxon data set, computed using all 8 input trees and 10 Reticulations.

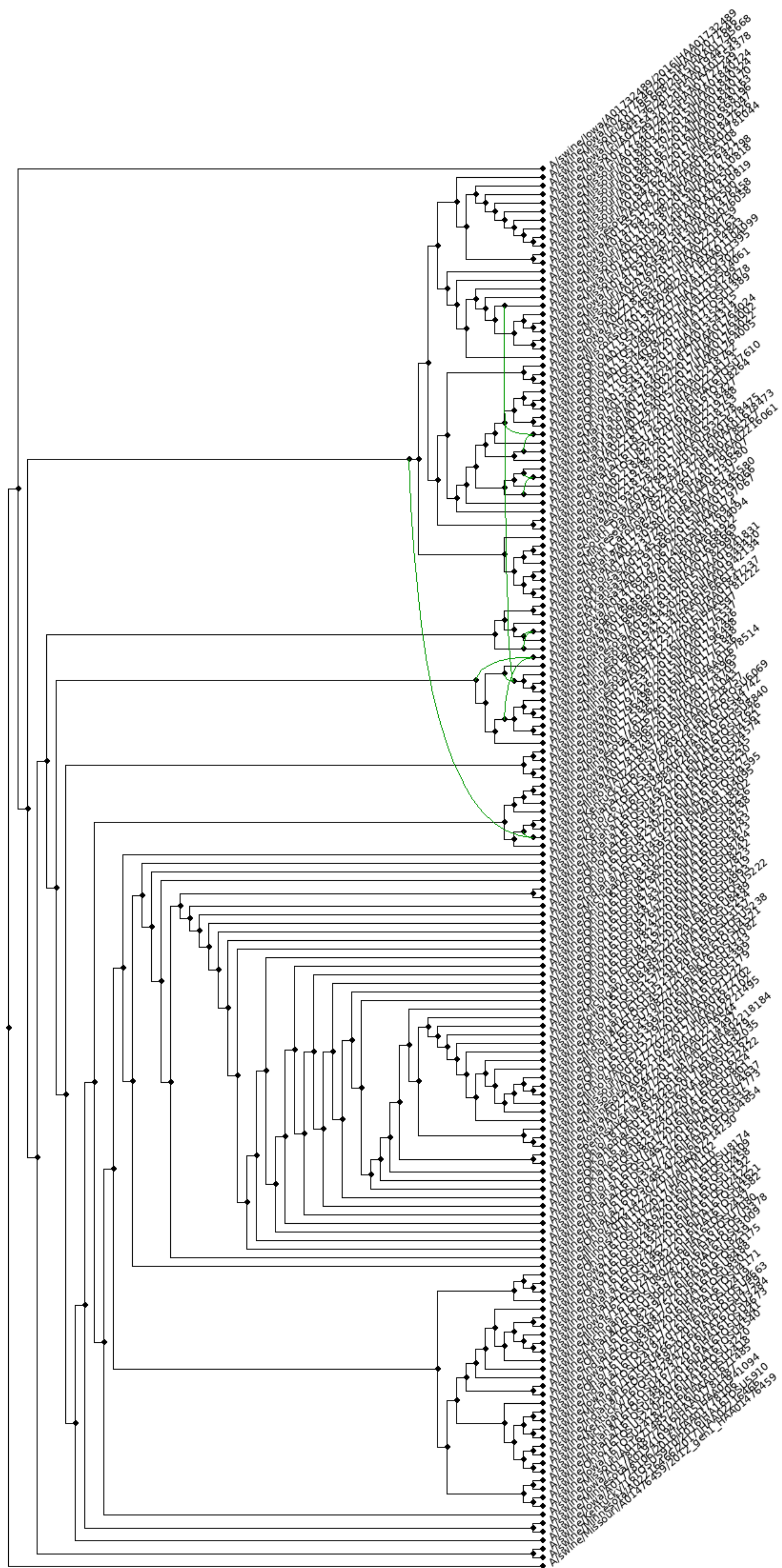

Figure 4: Optimal Reticulation Network on H3 data set (164 Taxon), computed using HA, NA

REFERENCES

[1] M. S. Bansal, J. G. Burleigh, and O. Eulenstein. Efficient genome-scale phylogenetic analysis under the duplication-loss and deep coalescence cost models. *BMC Bioinformatics*, 11 Suppl 1:S42, 2010.

[2] O. R. Bininda-Emonds, editor. *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life*, volume 4 of *Computational Biology*. Springer Verlag, 2004.

[3] M. Bordewich, S. Linz, and C. Semple. Lost in space? generalising subtree prune and regraft to spaces of phylogenetic networks. *Journal of theoretical biology*, 423:1–12, 2017.

[4] R. Chaudhary, M. S. Bansal, A. Wehe, D. Fernández-Baca, and O. Eulenstein. iGTP: a software package for large-scale gene tree parsimony analysis. *BMC Bioinformatics*, 11:574, 2010.

[5] D. H. Huson, R. Rupp, and C. Scornavacca. *Phylogenetic networks: concepts, algorithms and applications*. Cambridge University Press, 2010.

[6] H. T. Lin, J. G. Burleigh, and O. Eulenstein. Consensus properties for the deep coalescence problem and their application for scalable tree search. *BMC Bioinformatics*, 13 Suppl 10:S12, 2012.

[7] W. P. Maddison and L. L. Knowles. Inferring phylogeny despite incomplete lineage sorting. *Syst Biol*, 55(1):21–30, 2006.

[8] D. Robinson and L. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1981.

[9] C. Than and L. Nakhleh. Species tree inference by minimizing deep coalescences. *PLoS Comput Biol*, 5(9):e1000501, 2009.

[10] Y. Yu, R. M. Barnett, and L. Nakhleh. Parsimonious inference of hybridization in the presence of incomplete lineage sorting. *Systematic Biology*, 62(5):738–751, 2013.