Title
A Static Formulation of the History Bound Problem

Author
Matsieva, Julia

Publication Date
2014-07-01

A Static Formulation of the History Bound Problem

By

JULIA MATSIEVA
B.S. (Harvey Mudd College) 2011

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Daniel M. Gusfield, Chair

Todd J. Green

Zhendong Su

Committee in Charge

2014

Contents

Abstract iii

1. Introduction 1

1.1. Recombination Graphs and Reticulation Networks 2

1.2. Lower Bounds on Numbers of Recombination Nodes 4

1.3. Research Contribution 5

1.4. Organization 6

2. Definitions 7

2.1. Reticulation Networks and Clusters 7

2.2. The History Bound 10

2.3. ST-Set Sequences 11

3. Bound Equivalence 14

4. A Static Definition of the History Bound 20

4.1. Network.Build 20

4.2. Correctness 27

4.3. Minimizing Reticulations 29

5. A DP Algorithm for Constructing Ret-Minimum Networks 35

6. Conclusion 39

6.1. Future work 39

Appendix A. Source of `Network.Build` 43

References 52

Abstract

Biologists working with DNA or character data are often interested in modeling the evolution of biological change present in their samples. The biological processes that create diversity are traditionally represented as branching events in the lineages of species. However, branching alone is insufficient to model all real-world data, hence there is an interest in constructing phylogenetic networks, or rooted DAGs, that simultaneously display multiple evolutionary trees or sometimes model biological processes, such as recombination. We often require that these networks fulfill an optimization criteria, such as having a minimum number of recombination or reticulation nodes.

In this work, we prove that the History Bound, a value previously computed as a lower bound on the minimum number of recombinations in a set of DNA data, is such an optimization criterion on phylogenetic networks; specifically, it is the minimum number of reticulation nodes in a particular type of network. This was stated earlier without a proof by other researchers. We also give an algorithm for constructing the network with this number of nodes and show that a network of this type with fewer reticulation nodes cannot exist.

1. Introduction

Many types of biological data can be intuitively represented by labeled, directed trees that model the diversity observed in a set of specimens, or *taxa*, with a sequence of branching events, in a way that is consistent with our current understanding of biological change. More specifically, phylogenetic trees are used to model the string of mutations that cause a single ancestor DNA sequence to change incrementally to finally generate an input set of observed DNA sequences. It must be possible to label internal edges in the tree with the index of the character at which mutations occur, such that the resulting DNA sequences represent the input data. More generally, phylogenetic trees can be used to model the evolution history of any set of *characters*, or observed traits and sequences, in a collection of taxa. The inputs to the problem of constructing a phylogenetic tree are several subsets of the taxa, each consisting of those taxa that share a certain trait. These subsets, called clusters or clades, must be the leaf descendants of some edge in the phylogenetic tree that represents the data. On the surface, the sequence and cluster problems appear quite similar, and indeed it is known that they are almost equivalent in the instances where a solution exists. These problems are formally defined in Section 2.1.

However, many real-world data sets cannot be represented by a tree, i.e. there is no perfect phylogeny, or tree where each character is allowed to mutate only once, that represents the data (cf. Section 2.1). This is because the perfect phylogeny model is not general enough to represent all possible data sets. Indeed, a known result in the literature gives a narrow set of criteria that a set of data must exhibit for a solution to exist [2]. To handle these cases, the requirement that the data be placed on a tree is relaxed to allow *joining* events, in addition to branching, as a mechanism for increasing diversity, which

results in the creation of phylogenetic networks. In a phylogenetic network, two lineages are allowed to combine to form a lineage that could not be obtained by branching events alone. With no further restrictions, the problem of finding a network to represent a set of data can be computed in polynomial time; however, our current understanding of biology suggests that these joining, or reticulation, events occur relatively infrequently in nature. Thus, we often seek to construct plausible networks that have a minimum number of nodes with in-degree > 1, at which point the problems become computationally hard.

The creation of such networks is not without justification, as many biological mechanisms exhibit this sort of merging behavior. In the DNA setting, the merging of lineages is consistent with the biological process of recombination in meiosis, or the creation of gametes, in which two corresponding, but not necessarily identical, chromosomal sequences recombine to create a new sequence consisting of alternating segments. In the general character setting, the converging branches can indicate the point of independent evolution of similar observed traits in different species or the creation of hybrid species. Due to the differences in biological models and the additional complexity involved in the development of networks, the objectives, terminology and techniques involved in these problems differ in subtle but often significant ways. We discuss these differences in more detail in the following section.

1.1. **Recombination Graphs and Reticulation Networks**

An ancestral recombination graph, or ARG, is a particular kind of network constructed to represent the derivation of a set of DNA sequences. ARGs are created from a collection of input sequences over a binary alphabet representing single nucleotide polymorphisms (SNPs). This set is compactly represented in a binary matrix, with each row representing the DNA sequence of each taxon. The model assumes that the input set of sequences was

generated from a single ancestral sequence; thus, the goal is to generate a rooted DAG with the ancestral sequence (usually all zero) at the root, whose leaves are labeled with the input sequences that abides by the following rules: Each character in the ARG is allowed to mutate exactly once at each *site*, or index, and a mutation is represented in the graph by a labeled edge. Typically such an edge has a parent node with two outgoing edges. The DNA sequences are also allowed to *recombine*, resulting in nodes of in-degree two. Under the single-crossover model, which will be the default assumption for the rest of this thesis, the resulting DNA sequence is only allowed to take a prefix from one sequence and a suffix from the other. It is also possible to construct ARGs using multiple-crossover; under this assumption, intervals of the parent sequences can be interwoven multiple times. (Note that, under both assumptions, ARGs have recombination nodes with in-degree 2.) It is especially desirable to construct a *minARG*, or an ARG with the fewest number of recombination nodes, for a given set of data. This problem is known to be NP-hard, so research has focused on computing and evaluating lower bounds on R_{min}, the number of recombination nodes in a minARG.

In contrast, reticulation networks (also sometimes referred to as hybridization or phylogenetic networks) do not usually model actual sequences. Rather, the input is a set of *clusters* (also called clades), which are subsets of the taxa, although sets of trees or triplets are also used in other problem settings, as summarized by van Iersel and Kelk [7]. The objective is to construct a network such that the taxa in each cluster are the leaf descendants of some tree topologically embedded in the network, and that the network has the fewest number of *reticulation events*, which will be formally defined in Section 2.1. An instance of the reticulation network problem, and its conversion to a matrix is shown in Figure 1. One

important difference from the ARG scenario, whose inputs are binary sequences, meaning the sites have a fixed ordering, is that the inputs to this problem are usually *unordered*. However, it is easy to see that a set of clusters can be converted into an input matrix by imposing an arbitrary ordering on the clusters and using 1's to denote membership in a cluster and 0's to denote exclusion. Thus, an ARG for a set of DNA sequences will be a valid reticulation network for the underlying clusters, but it is likely to have reticulation nodes that are only necessary due to the choice of ordering. Reticulation networks also do not place any restrictions on the in-degrees of the *reticulation nodes* in the network.

1.2. **Lower Bounds on Numbers of Recombination Nodes**

As mentioned in the previous section, numerous methods have been developed for computing lower bounds on R_{min}, such as the Haplotype Bound, the Forest Bound, etc. [2]. These methods all operate on the input matrix of binary SNP sequences, but generally have a combinatorial interpretation. For example, the Forest bound is characterized as the solution to the Minimum Perfect Phylogenetic Forest (MPPF) Problem, which is defined as the smallest partition of the input matrix such into subsets with perfect phylogenies [2]. Unlike the Forest bound, the History Bound was originally formulated as the output of an algorithm [6], and did not have a static interpretation [2]. The algorithm that computes a candidate for the History Bound works on the input matrix by first successively removing columns that contain at most a single 1 and collapsing identical rows until neither operation is possible. Then, the algorithm removes a row arbitrarily, and restarts the previous procedure (see Algorithm 1). The resulting candidate for the History Bound is the number of times an arbitrary row removal occurs; the History Bound was originally defined as the minimum over all the candidate values [6].

As given, the algorithm to compute the History Bound runs in time that is super-exponential in the size of the input, but a dynamic programming version speeds up the computation time to exponential [1]. However, we also know that the problem of computing the History Bound is NP-hard [1], so the development of an asymptotically more efficient algorithm is not expected. Nevertheless, tools like integer linear programming are repeatedly shown to be efficient in practice for NP-hard problems, so a static formulation of the History Bound as an optimization problem is desirable for use on real-life biological data.

1.3. **Research Contribution**

In Kelk et al. [4], the authors develop a theory to describe and analyze structures in cluster input data. Their analysis sheds some light on the underlying meaning of the number computed by the History Bound algorithm. In fact, they claim that the construction given in their work is equivalent to the algorithm that computes the History Bound. The main contribution of this thesis is the detailed proof that this is indeed the case.

The main contribution of this thesis is a proof of an orally stated conjecture by the same authors that the History Bound is the minimum number of *reticulation nodes* in a reticulation network that represents the clusters encoded by an input matrix [5]. In this thesis, we prove this conjecture by using the equivalence from Kelk et al. [4] to give an algorithm that constructs the corresponding reticulation network when given as input the intermediate values computed by the history bound algorithm. We then show this network to have the fewest number of reticulation nodes possible when run on the intermediate values of the minimum run of the candidate algorithm. This result naturally extends to an exponential-time dynamic programming algorithm for constructing the network with the minimum number of reticulation nodes.

1.4. **Organization**

Section 2 gives formal definitions of the relevant mathematical objects involved in the study of phylogenetic networks, and outlines the relevant concepts developed in [4]. These are then used in Section 3 to explicitly prove a claim from [4], and as in Section 4 to prove the main conjecture outlined in Section 1.3. We give the dynamic programming solution for constructing reticulation networks in Section 5. Conclusion and future work are in Section 6. The Python source code for the algorithm from Section 4 is given in the Appendix.

2. Definitions

Section 2.1 discusses standard definitions and relevant theorems from the phylogenetic networks literature that are needed in this thesis. In Section 2.2, we describe the algorithm that defines the History bound. In Section 2.3, we recall definitions from Kelk et al. [4], which we will use in later sections to prove the main results of this thesis.

2.1. **Reticulation Networks and Clusters**

The motivating work for this thesis focuses on the development of reticulation networks in the cluster setting, so we start with the general concepts related to that problem. A *reticulation network* on a finite set X of taxa (specimens described by the data) is a rooted, connected DAG with no nodes having both in-degree and out-degree 1, whose leaves are the elements of X. A node of in-degree of 2 or greater is a *reticulation node* and incoming edges into such a node are called *reticulation edges*. This branch of the literature is often concerned with counting reticulation "events", which is why the *reticulation number* (also sometimes called *hybridization number*) of a network $N = (V, E)$ is defined in [4] as

$$r(N) = \sum_{v \in V : d_{in}(v) > 0} (d_{in}(v) - 1) = |E| - |V| + 1$$

where $d_{in}(v)$ denotes the in-degree of a node v. However, throughout this thesis, we will be more concerned with counting the *number of reticulation nodes* in a network N, with no regard for the degree of each reticulation node.

A *cluster* $C \subseteq X$ is a subset of the taxa. We say that two clusters C_1 and C_2 are *compatible* if either $C_1 \subseteq C_2$, $C_2 \subseteq C_1$ or $C_1 \cap C_2 = \emptyset$, and incompatible otherwise. A collection $\mathcal{C}$ of clusters is compatible if all the clusters in $\mathcal{C}$ are pairwise compatible. Then,

according to the matrix equivalence mentioned in Section 1.1: a collection $\mathcal{C}$ of clusters can be converted to a matrix M by choosing an arbitrary ordering on X and $\mathcal{C}$, filling the matrix by placing 1 into the row corresponding to taxon x and the column corresponding to cluster C, if $x \in C$, and placing 0 there otherwise. Thus, in the matrix setting, when the ancestral sequence is the all-zero sequence, two columns C_1, C_2 are incompatible if the set of pairs constructed by pairing column entries in corresponding rows contains all of the pairs $(1, 0)$, $(0, 1)$ and $(1, 1)$.

The concept of compatibility relates directly to the perfect phylogeny problem, which is the problem of finding a tree that represents the data encoded in a binary matrix M. In order to represent M, a tree T must have the properties that all of its leaves correspond to the rows of M, each column of M labels exactly one edge of T, every interior edge of T is labeled by at least one column of M, and for each leaf x in T, the edge labels on the path from the root to x must correspond to those columns that have state 1 in M [2]. This definition assumes that T is constructed with an all-zero ancestral sequence, which we will assume throughout this thesis. The conditions for when a solution to the problem exists are well-understood; the Perfect Phylogeny theorem states that the problem does not have a solution, that is, a tree that represents the data, in the presence of the so-called Three Gametes condition, which occurs when there is at least one pair of incompatible columns in M, and recombination must be used to generate the data [2]. Equivalently, if a set of clusters can be represented by a tree, then all of the clusters must be compatible. The Perfect Phylogeny Theorem also states the converse: if a no pair of columns of M is incompatible, then there does exist a perfect phylogeny that represents the data.

(A) A set of subsets of X. C_5 is a singleton cluster because it only contains one taxon. Such clusters, and their corresponding columns, are sometimes called *uninformative*.

$$\mathcal{C} = \begin{array}{c|ccc} C_1 & 1 & 2 & 3 \\ C_2 & 1 & 3 & \\ C_3 & 1 & 4 & 5 \\ C_4 & 3 & 4 & 5 \\ C_5 & 5 & & \end{array}$$

(B) Matrix equivalent of cluster set $\mathcal{C}$. Each row of M represents a taxon of X and each column represents a cluster in C. A entry with value 1 in row x of column i correspond to membership of x in cluster C_i.

$$M = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

FIGURE 1. An example set of clusters $\mathcal{C}$ on the taxon set $X = \{1, 2, 3, 4, 5\}$ and its corresponding matrix representation.

In contrast, the input to the reticulation network problem is a set X of taxa and a collection $\mathcal{C}$ of clusters, where each $C \in \mathcal{C}$ is a proper subset of X. A network N represents a cluster C in the *hardwired sense* if there exists a tree edge (u, v) such that the set of leaf descendants of v is exactly equal to C. N represents a cluster C in the *softwired* sense if N contains an edge e such that $C \in C(e)$, where $C(e)$ is formed as follows: For each reticulation node, choose one incoming edge to remain on and turn all others "off"; then, for each set of choices, a cluster is in $C(e)$ if it is exactly equal to the set of taxa accessible from e without following any edges that have been turned off [3]. The operation of turning on one reticulation edge and turning all others off is referred to as the construction of a *switching* of N in [4]. In this thesis, we will say that a network N represents a collection of clusters $\mathcal{C}$ if it represents each $C \in \mathcal{C}$ in the softwired sense, unless otherwise specified. These networks are usually subject to some optimality criterion, such as the minimization of reticulation number. In this thesis, we will often seek networks with the minimum number of reticulation nodes, so we define a network N to be *ret-minimum* for a set $\mathcal{C}$ of clusters if N has the fewest number of reticulation nodes over all networks that represent $\mathcal{C}$ (in the softwired sense).

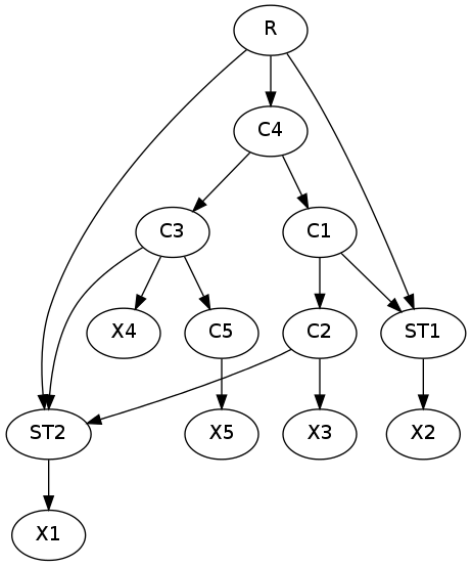


FIGURE 2. A reticulation network that represents the cluster set $\mathcal{C}$ in the softwired sense from the example in Figure 1. It has 2 reticulation nodes and reticulation number 3.

2.2. **The History Bound**

The algorithm that defines the History Bound was first given by Myers and Griffiths in [6]. This algorithm operates on a binary input matrix M and computes a lower bound on R_{min}, the number of recombination events necessary to represent M with an ARG. We give the version of the algorithm as explained in [2], where it is split into two procedures, one for generating a *candidate* for the History Bound, which is shown as Algorithm 1, and another procedure (referred to as CHB-Branch) for selecting the minimum value over all the candidates. The first algorithm reduces the matrix by iterating the following procedure: remove columns with at most one 1 entry and collapse identical rows together until neither is possible, then choose an arbitrary row to delete. The algorithm applies this procedure repeatedly until the matrix is empty. The number of row deletions shown on line 8 of Algorithm 1 is the candidate for the History Bound; the smallest candidate over all possible executions of CHB is defined as the History Bound.

Algorithm 1: CHB algorithm that computes a candidate for the History Bound as given in [2].

Input: A binary matrix M
Output: A candidate value H for the History Bound

```
set M̃ = M
set chb = 0
while M̃ is not empty do
    while possible do
        Dr: collapse together any duplicate rows of M̃
        Dc: remove any columns with at most 1 entry from M̃
    end
    Dt: remove an arbitrary row r from M̃
    H = H + 1
end
return H
```

Since the binary matrix input to the algorithm can be thought of as representing a set of clusters $\mathcal{C}$, we can describe how the algorithm would operate directly on a set of clusters. The column removal operation corresponds to the removal of singleton or empty clusters from $\mathcal{C}$, while the row removal can be thought of as collapsing two taxa with identical cluster memberships into a single meta-taxon. Furthermore, the History Bound algorithm will produce the same result for any ordering of the rows and columns of the input matrix, because the value is computed as the minimum over all possible executions of CHB. This property suggests a closer relationship of the History Bound to the clusters represented by M rather than the specific binary sequences that M contains.

2.3. **ST-Set Sequences**

Kelk et al. [4] define a central concept called an *ST-set*, which is a subset of the taxa with some special, "treelike" properties. Informally, an ST-set of $\mathcal{C}$ can be thought of as a union of some compatible clusters of $\mathcal{C}$ or the compatible subsets of clusters.

Given a subset S of taxa, let $\mathcal{C} \setminus S$ denote the result of removing S from each cluster $C \in \mathcal{C}$, and let $\mathcal{C} \mid S$ denote the restriction of $\mathcal{C}$ to S (or $\mathcal{C} \setminus (X \setminus S)$). Formally, $S \subseteq X$ is an ST-set with respect to a collection $\mathcal{C}$ of clusters if

(1) S is compatible with $\mathcal{C}$.

(2) all pairs of clusters C_1, $C_2 \in \mathcal{C} \mid S$ are compatible.

An ST-set S is *maximal* if there is no other ST-set S' such that $S \subsetneq S'$. It is shown in [4] that the maximal ST-sets of a collection of clusters can be computed in time polynomial in the size of the input.

A sequence $\mathcal{S} = \{S_1, S_2, \ldots, S_p\}$ is an *ST-set sequence* if each $S_i \in \mathcal{S}$ is an ST-set of $\widetilde{\mathcal{C}} = \mathcal{C} \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})$; $\mathcal{S}$ is a *maximal* ST-set sequence if each S_i is a maximal ST-set of $\widetilde{\mathcal{C}}$. $\mathcal{S}$ is a maximal ST-set *tree* sequence, if all the clusters in $I = \mathcal{C} \setminus (S_1 \cup \ldots \cup S_p)$ are pairwise compatible. By the Perfect Phylogeny Theorem, this means that all the taxa in I can be represented by a tree. Kelk et al. then define the *MST lower bound* on the

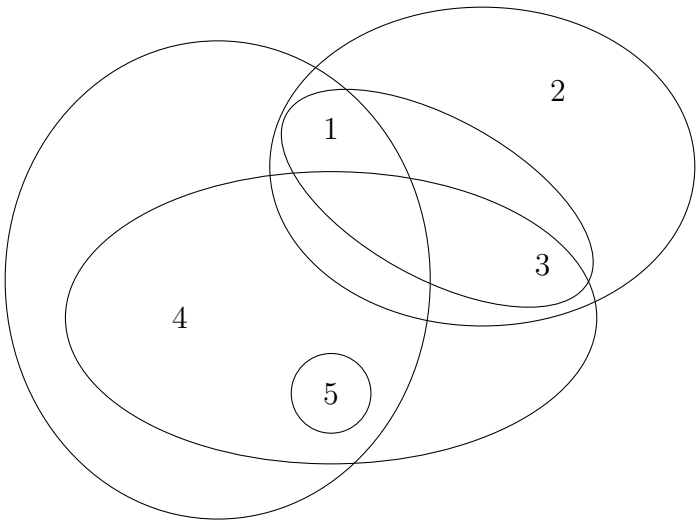


FIGURE 3. The clusters set $\mathcal{C}$ from Figure 1 has only one non-singleton, non-trivial maximal ST-set $\{4, 5\}$. The other maximal ST-sets are $\{1\}, \{2\}$ and $\{3\}$; neither of them can be combined together into a set that has a non-trivial intersection with all clusters in $\mathcal{C}$. They are described in [4] as "islands of laminarity". We can see that 4 and 5 can be grouped together with no intersections.

reticulation number $r(N)$ of a set of clusters $\mathcal{C}$ as the length p of the shortest maximal ST-set tree sequence that can be computed for $\mathcal{C}$, and show that it is a true lower bound. In Section 4, we show that this value is equal to the minimum *number of reticulation nodes* in a reticulation network that represents $\mathcal{C}$ in the softwired sense.

3. Bound Equivalence

After defining the concepts of maximal ST-set tree sequences and the MST lower bound, the length of the shortest maximal ST-set tree sequence, the authors of [4] make the claim that these concepts can be used to elaborate on the History Bound. They write

> We highlight that the phylogenetic network model described [here] is in a strong sense identical to the recombination network model under the assumption of an all-0 root, the infinite sites model and multiple crossover recombination. The computational lower bound described in Algorithm 3 of [18] is, taking this equivalence into account, essentially identical to the MST lower bound.

where Algorithm 3 references the original History Bound algorithm presented by Myers and Griffiths [6]. This claim can be restated as follows:

Claim 3.1. *Given a collection of clusters $\mathcal{C}$, the minimum execution of the Candidate History Bound algorithm is equal to the length of the shortest maximal ST-set tree sequence for $\mathcal{C}$.*

This claim was not explicitly stated in [4]. We prove it here by demonstrating that *every* run of the Candidate History Bound algorithm generates, through its intermediate values, a valid maximal ST-set tree sequence, and that every maximal ST-set sequence corresponds to an execution of CHB.

Lemma 3.2. *Let $\{r_1, \ldots, r_p\}$ be the sequence of rows that are deleted by an execution of Algorithm 2. For each r_i, let R_i denote the set of rows that were collapsed together with r_i by rule* **Dr** *during previous iterations of the algorithm. Let $S_i = R_i \cup \{r_i\}$. Then, $\{S_1, \ldots, S_p\}$ is a maximal ST-set tree sequence of the cluster set encoded by M.*

To make the proofs clearer, we slightly modify Algorithm 1 to save the relevant intermediate states. The modified version is restated in Algorithm 2. When rule **Dt** selects row r

Algorithm 2: An equivalent formulation of CHB, which removes all collapsed rows from $\widetilde{M}$ at the same time as the corresponding row removal of r_i.

```
   Input: A binary matrix M
   Output: A candidate value H for the History Bound
1  set M̃ = M; set H = 0
2  set X' = rows(M)
3  set C' = cols(M)
4  while M̃ is not empty do
5      while possible do
6          if two unmarked rows r_i and r_j are identical and i < j then
7              Dr: mark X'[r_j] = r_i
8              /* update rows found identical in previous iteratons to always be
                  marked with the lowest-index marker                          */
9              for all rows r such that X'[r] == r_j do
10                 update the marker to X'[r] = r_i
11             end
12         end
13         if an unmarked column c contains only one 1 then Dc: mark C'[c] = 1 ;
14     end
15     if C' and X' aren't completely marked then
16         Dt: delete an arbitrary unmarked row r from M̃
17         delete all rows marked r in X' from M̃
18         delete all marked columns in C (now empty)
19         H = H + 1
20     end
21 end
22 return H
```

for deletion, we remove at that moment but not before, all the other rows that were labeled r in previous iterations. However, it should be clear that the modified algorithm is equivalent to the original, since all the rules still operate on the unmarked portion of the matrix $\widetilde{M}$, which is the same as having deleted those rows and columns immediately. The set of rows in X' marked r_i in Algorithm 2, when rule **Dt** executes, corresponds to R_i from the definition in Lemma 3.2; therefore, we can see that after iteration i of Algorithm 2, the original matrix M has been modified by the algorithm to encode the set of clusters $\widetilde{\mathcal{C}}_i = \mathcal{C} \setminus (S_1 \cup \ldots \cup S_{i-1})$, since all the members of set S_i are removed at the end of iteration i. We use $\widetilde{M}_i$ to denote the state of the matrix after iteration i.

In order to prove Lemma 3.2, we must first establish that every set S_i is a maximal ST-set of $\widetilde{\mathcal{C}}_i$, which means we must prove that:

(1) S_i is compatible with $\widetilde{\mathcal{C}}_i$, the set of clusters encoded by $\widetilde{M}_i$.

(2) any pair of clusters $C_1, C_2 \subset S_i$ in $\widetilde{\mathcal{C}}$ are pairwise compatible.

(3) there does not exist another element e in $\widetilde{\mathcal{C}}_i$ such that $S_i \cup \{e\}$ is also an ST-set.

Proof. Suppose toward a contradiction that S_i is not compatible with $\widetilde{\mathcal{C}}_i$. This would imply that there must exist some cluster $C \in \widetilde{\mathcal{C}}_i$ and at least three distinct elements x, y, z such that $x, z \in S_i$ but $x \notin C$, and y, z are in cluster C but $y \notin S_i$. Since, z and x are in S_i they are identical when ignoring marked columns. However, z is also a member of C, which cannot be a marked column because it also contains another distinct unmarked element y. But then x and z are not identical on unmarked columns. Therefore, in order for the algorithm to place x and y into S_i, x must agree on column C and so it must also be a member of C, which contradicts the assumption that $x \notin C$, so (1) holds.

Similarly, suppose that there are two incompatible clusters $C_1, C_2 \in \widetilde{\mathcal{C}}_i \mid S_i$, the restriction of $\widetilde{\mathcal{C}}_i$ to S_i. We know that all the elements in S_i must have been found by Algorithm 2 to be identical when ignoring marked columns; however, the assumed incompatibility of C_1, C_2 implies that there exist at least three elements x, y, z such that $x, z \in C_1$ and $x \notin C_2$; and $y, z \in C_2$ and $y \notin C_1$. Neither column C_1 nor C_2 would have been marked by the algorithm because they both contain two rows with 1 entries, so x and y are not identical with respect to unmarked columns, so they would not have been placed in S_i, a contradiction. Therefore C_1 and C_2 must be compatible and (2) holds, so S_i is an ST-set of $\widetilde{\mathcal{C}}_i$.

Furthermore, S_i is a maximal ST-set. Suppose that S_i is not maximal, so there exists an element e such that $S_i' = S_i \cup \{e\}$ is also an ST-set of $\widetilde{\mathcal{C}}_i$. If e was identical to r_i with

respect to unmarked columns, then it would have had to be collapsted together with r_i and so would be in S_i. So if e is not identical to r_i when r_i is removed, then either

(a) there exists a cluster $C \in \widetilde{\mathcal{C}}_i$ such that $e \in C$ and $r_i \notin C$,

(b) or there exists a cluster $C \in \widetilde{\mathcal{C}}_i$ such that $e \notin C$ and $r_i \in C$.

In case (a), in order for S'_i to still be compatible with $\mathcal{C}_i$, cluster C *must* be a singleton cluster that only contains e. Otherwise, if C contains some other element $d \notin S'_i$ then its existence will give rise to the Three Gametes condition in C and S'_i, because $r_i \in S'_i$ but not in C, $d \in C$ but not in S'_i and e is in both. So if C is not a singleton cluster, then C and S'_i are incompatible. But if C is a singleton cluster, then its column will be marked in $\widetilde{M}_i$ by rule **Dc** before r_i is removed and so case (a) leads to a contradiction. In case (b), since S'_i is assumed to be compatible with C, and $r_i \in C$, this means that $C \subset S'_i$ because if C contained an element $d \notin S'_i$, then the taxa r_i, e and d would create the Three Gametes condition in $\widetilde{M}_i$ in sets C and S_i. To see this, note that e is in S'_i but not in C, and d is in C but not in S'_i and r_i is in both, so S'_i and C would be incompatible. But if $C \subset S_i$, then C would be a column with a single 1 in row i of $\widetilde{M}_i$ because, at that point, all of the taxa in S_i are collapsed together into r_i. So C would have been marked before r_i was removed, a contradiction. So we conclude that there is no $e \neq r_i$ that can be added to S_i and maintain its ST-set properties and S_i is maximal. Thus, property (3) holds as well.

In order to show that $\{S_1, \ldots, S_p\}$, as defined in the statement of Lemma 3.2, is a maximal ST-set *tree* sequence, it remains to show that all the clusters in $I = \mathcal{C} \setminus (S_1 \cup \ldots \cup S_p)$ are compatible. We know that after r_p is deleted by rule **Dt** then the **Dr** and **Dc** operations mark all remaining rows and columns in $\widetilde{M}$. Otherwise, it would be necessary to apply **Dt** again and r_p would not have been the last row the sequenece. All the clusters in I must

be compatible, because otherwise there would be two clusters $C_1, C_2 \subseteq I$ with a non-trivial intersection, and neither rules **Dc** and **Dr** could have applied again since neither cluster can be a singleton and there are elements in these clusters whose memberships are not identical. Therefore, $\{S_1, \ldots, S_p\}$ is a maximal ST-set tree sequence of the clusters encoded by M. $\square$

This implies that the History Bound, the minimum value produced by the CHB algorith, is greater than or equal to the length of the shortest maximal ST-set sequence. But there may exist ST-set sequences that do not have corresponding executions of CHB. In order to prove Claim 3.1, we must also prove:

Lemma 3.3. *For every maximal ST-set tree sequence $\mathcal{S} = \{S_1, \ldots, S_p\}$, there exists an execution of Algorithm 2 that removes row sequence $\{r_1, \ldots, r_p\}$ by rule* **Dt** *such that $r_i \in S_i$, for $i \in \{1, \ldots, p\}$.*

Proof. For this proof, we refer to the version of CHB shown in Algorithm 2. Note that in this version of the algorithm, when rule **Dr** combines two rows together, it leaves the row with the smaller index in M unmarked (shown on line 6) and marks the row with the larger index with the index of the smaller. Since rule **Dt** chooses rows arbitrarily, there exists an execution of CHB that chooses a row $r \in \widetilde{M_i}$ at iteration i, provided that r is not marked or deleted at that point. Therefore, in order to show that there exists an execution of CHB that chooses rows $\{r_1, \ldots, r_p\}$ with each $r_i \in S_i$, we need to show that the row of S_i with the lowest index in M is still present and unmarked in $\widetilde{M_i}$ when rule **Dt** executes in Algorithm 2.

Let $\mathcal{S} = \{S_1, \ldots, S_p\}$ be a maximal ST-set tree sequence and suppose the row sequence $\mathcal{R} = \{r_1, \ldots, r_{j-1}\}$ is chosen for deletion by rule **Dt** in the *first* $j-1$ iterations of an execution of Algorithm 2, such that each row $r_i \in \mathcal{R}$ for all $1 \leq i \leq j-1$ is the row of S_i with the

lowest index in M. We want to show that, before rule **Dt** executes on the j^{th} iteration of CHB, the row r in S_j with the lowest index in M is available in $\widetilde{M}_{j-1}$ to be chosen by for deletion. Therefore, we must show that, right before rule **Dt** executes in iteration j of CHB, row r is neither *marked* nor *deleted* in $\widetilde{M}_{j-1}$. Row r cannot be marked in M_{j-1}, since if it were, it would be marked by CHB with a row r' that has a lower index in M. However, this means that *all* the elements of S_j would also be marked with r', as that would imply that $r' \in S_j$ has a lower index than r because CHB updates all the markers to the lowest index (shown in lines 8-10). This contradicts the assumption that r has the lowest index in S_j, so r cannot be marked in $\widetilde{M}_j$. Next, suppose that, at iteration j of Algorithm 2, row r is not present in $\widetilde{M}_j$. This means that r was deleted in a previous iteration i of the algorithm. However, we already inductively assumed that the algorithm removes each maximal ST-set $S_i \in \mathcal{S}$ at iteration $i < j$ and, since S_i is maximal, $S_i \cup \{r\}$ cannot also be an ST-set of $\widetilde{\mathcal{C}}_i$. Furthermore, it is shown in [4] that the maximal ST-sets of $\mathcal{C}$ partition X, so r cannot belong to both S_i and S_j. Therefore, the row r in S_j with the lowest index, could not be missing or marked in iteration j of Algorithm 2; hence, $r_j = r$ and the induction argument is complete. This proves Lemma 3.3. □

Since we have proved Lemmas 3.2 and 3.3, every run of the CHB algorithm corresponds to a maximal ST-set tree sequence and every maximal-ST-set corresponds to an execution of CHB, it follows that an execution of CHB that produces the minimum value corresponds to a shortest maximal ST-set tree sequence, so Claim 3.1 is proved.

4. A Static Definition of the History Bound

In this section, we will prove a conjecture [5] using the properties of maximal ST-set tree sequences developed in [4].

Conjecture 4.1. *Given a set of clusters $\mathcal{C}$ over a set of taxa X, the History Bound is the minimum number of reticulation nodes that must be present in a network that represents $\mathcal{C}$ in the softwired sense.*

To prove this conjecture, we first give an algorithm that takes as input the set of clusters $\mathcal{C}$ and a maximal ST-set tree sequence $\mathcal{S} = \{S_1, \ldots, S_p\}$ and constructs a reticulation network N with *at most* p reticulation nodes that represents the clusters. We show that this algorithm would produce a network with *exactly* p reticulation nodes when executed on a sequence $\mathcal{S}$ of minimum length. We then show that there cannot exist a network with fewer reticulation nodes than the length of shortest maximal ST-set tree sequence.

4.1. **Network.Build**

In this section, we describe the `Network.Build` algorithm, which will iteratively construct a network to represent a set of input clusters, guided by an input maximal ST-set tree sequence $\mathcal{S}$ of length p. At each iteration, `Network.Build` will insert an ST-set S_i of $\mathcal{S}$ into the network N, maintaining the invariant that, after each iteration, N represents $\widetilde{\mathcal{C}}_i = \mathcal{C} \mid (I \cup S_p \cup S_{p-1} \cup \ldots \cup S_i)$. As mentioned in Section 2.1, network N represents a cluster C in the *softwired* sense if some *switching* of N contains a tree edge whose set of leaf descendants is exactly the set of taxa in C. Thus, it helps to introduce a cluster data structure that, in addition to storing the subset of taxa of a cluster $C \in \mathcal{C}$, will also keep track of its `tree_edge` in N and a set of `off_edges`. When the set C.`off_edges` is removed

from the network, the subtree of C.`tree_edge` will form a tree and the leaf descendants of C.`tree_edge` will be exactly the set of taxa in C. The graph $N - C$.`off_edges` might not be a tree because some reticulation edges may not be on a path from C.`tree_edge` to a leaf. However, any reticulation nodes in the subtree of C.`tree_edge` will have no other incoming edges except the one from C.`tree_edge`. Additionally, each cluster C will contain its original taxa in the variable C.`taxa`, but it will also have a C.`restrict_to(`S`)` procedure that takes as input $S \subseteq X$ and causes C to return $C \mid S$ as its taxa when C participates in set operations. The C.`restriction` procedure will output the most recent set S that C been restricted to.

We will also assume that a data structure representing an empty graph is initialized using `graph`, as shown on line 2 of Algorithm 3. A graph G is populated using procedures G.`add_nodes` which takes either a list or a single node as input, and G.`add_edge` which requires two node arguments. An edge can be removed from a graph G using G.`remove_edge`.

The outer abstraction layer of `Network.Build` is shown as Algorithm 3. Since the input $\mathcal{S}$ is a maximal ST-set tree sequence, we know that after we perform the restriction to I on line 6, all the clusters in $\widetilde{\mathcal{C}}$ will be pairwise compatible. So $\widetilde{\mathcal{C}}$ can be represented by a tree, built by the well-known algorithm shown in the pseudocode as `Tree.Build` in Algorithm 4. It is equivalent to the solution to the Perfect Phylogeny problem in Section 2.1.2 of [2]. We will also use Algorithm 4 to extend a network N by adding a tree to N at a specified node. We give a specific implementation in Algorithm 4, which takes as input a network N, a node *inputRoot* of N, and a set of compatible input clusters $\mathcal{C}$, and outputs the updated network N with a new subtree rooted at *inputRoot* that represents $\mathcal{C}$. More important, it also sets the `tree_edge` values for all the input clusters, even those whose restriction is empty. Thus,

Algorithm 3: Algorithm that constructs a reticulation network N with $\leq p$ nodes when given a valid maximal ST-set tree sequence $\mathcal{S}$ of length p.

```
1  Function Network.Build(C, S)
       Input: A set of clusters C and a maximal ST-set tree sequence S of length p
       Output: A network N that represents C with ≤ p reticulation nodes
2      N = graph(); N.add_nodes(root);
3      X = ⋃_{C∈C} C
4      I = X \ ⋃_{S∈S} S
5      set nonSubsets = {C ∈ C | C ⊄ S_i for any S_i ∈ S}
6      /* create C̃ that contains the clusters restricted to the taxa in I     */
7      C̃ = {}
8      for C ∈ C do
9          C̃ = C̃ ∪ { C.restrict_to(I) }
10     end
11     set N = Tree.Build(N, root, nonSubsets);
12     set S = S.reverse();
13     for S_i ∈ S do
14         /* widen the restriction for all the clusters                       */
15         for C ∈ C̃ do C.restrict_to (C.restriction ∪ S_i) ;
16         /* add ST-set S_i into the network                                  */
17         N = Network.Add(N, C, S_i);
18     end
19     return N;
20 end
```

after line 11 of Algorithm 3 executes, all the clusters in *nonSubsets* will have `tree_edge` values set, which means that for each C in *nonSubsets*, the subtree of C.`tree_edge` in N will be be the set of taxa $C \mid I$. After building the initial tree for $\mathcal{C} \mid I$, `Network.Build` reverses the maximal ST-set tree sequence $\mathcal{S}$ and iteratively adds ST-sets into N by calling the procedure `Network.Add`, shown in Algorithm 5.

After a set of clusters has been processed by `Tree.Build`, each of the processed clusters C will have a non-null C.`tree_edge` value. Thus, we can describe a cluster C_1 to be *downstream* from cluster C_2 in a network N if there is a directed path from C_2.`tree_edge`

Algorithm 4: Algorithm for adding a tree to network N that represents a set of compatible clusters $\mathcal{C}$.

```
1  Function Tree.Build(N, root, C)
       Input: A network N, a node inputRoot in N and a collection of compatible clusters C
       Output: The updated network N where the node inputRoot has a subtree below it
               that represents C
2      /* sort the clusters by size in decreasing order                    */
3      C = C.sort();
4      for C ∈ C do
5          N.add_nodes(newNode)
6          Σ = the smallest cluster that is a superset of C
7          if Σ exists then
8              (u, v) = Σ.tree_edge
9              for x ∈ C do
10                 N.delete_edge(v, x)
11             end
12             set treeEdge = (v, newNode)
13         else
14             /* C is disjoint from all the processed clusters            */
15             set treeEdge = (inputRoot, newNode);
16             for x ∈ C do
17                 N.add_nodes(x)
18             end
19         end
20         /* hang the taxa in C off the new node                          */
21         for x ∈ C do
22             N.add_edge(newNode, x)
23         end
24         N.add_edge(treeEdge)
25         set C.tree_edge = treeEdge
26     end
27     return N
28 end
```

to C_1.`tree_edge` in the graph $N - C_2$.`off_edges`. Naturally, C_2 is *upstream* from C_1 in N if C_1 is downstream from C_2.

When processing maximal ST-set S_i, the `Network.Add` program works by first partitioning the input clusters into three subsets – the set of clusters *Supersets* that contain the current ST-set S_i, the set of clusters *Disjoint* that are disjoint from S_i, and the clusters *Subsets* that are proper subsets of S_i. This creates a partition of $\widetilde{\mathcal{C}}$ because S_i is an ST-set and

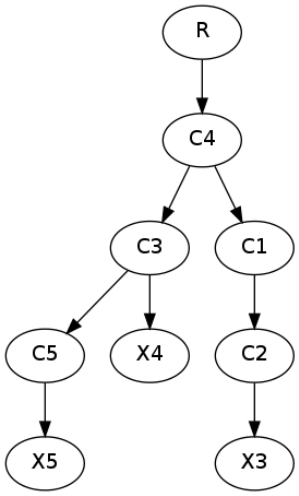
Algorithm 5: Procedure that inserts ST-set S_i into N, maintaining the property that N represents $\widetilde{\mathcal{C}}_i$.

Function `Network.Add`$(N, \mathcal{C}, S_i)$
  Input: A network N, a set of clusters $\mathcal{C}$ and a maximal ST-set S_i
  Output: Modified network N such that N represents $\mathcal{C}$ containing S_i
  N`.add_nodes(`*internalNode*`)`
  set Disjoint $= \{C \in \mathcal{C} \mid S_i \cap C = \emptyset\}$
  set Supersets $= \{C \in \mathcal{C} \mid S_i \subseteq C\}$
  set Subsets $= \{C \in \mathcal{C} \mid C \subset S_i\}$
  MaxDownstream $= \{\}$; IsUpstreamFrom $= \{\}$;
  for $(C, K) \in$ *Supersets* $\times$ *Supersets* **do**
    if K *is downstream from* C **then**
      MaxDownstream`.add(`K`)`
      IsUpstreamFrom$[K]$`.add(`C`)`
    else if C *is downstream from* K **then**
      MaxDownstream`.add(`C`)`
      IsUpstreamFrom$[C]$`.add(`K`)`
    end
  end
  $N =$ `Tree.Build(`N, *internalNode, Subsets*`)`
  for $x \in \left(S_i - \bigcup_{C \epsilon Subsets}\right)$ **do**
    N`.add_nodes(`x`)`
    N`.add_edge(`*internalNode,* x`)`
  end
  for $C \in$ *MaxDownstream* **do**
    set $(u, v) = C$`.tree_edge`
    N`.add_edge(`*internalNode,* x`)`
    for $K \neq C \in$ *MaxDownstream* **do**
      K`.off_edges .add(`v, *internalNode*`)`
      for $Q \in$ *IsUpstreamFrom[K]* **do** Q`.off_edges .add(`v, *internalNode*`)` ;
    end
  end
  for $(D, C) \in$ *Disjoint* $\times$ *MaxDownstream* **do**
    if C *is downstream from* D **then**
      set $(u, v) = C$`.tree_edge`
      D`.off_edges .add(`v, *internalNode*`)`
      if D *is upstream from all* $C \in$ *Supersets* **then**
        if $(v, internalNode) \notin N$ **then** N`.add_edge(`*root, internalNode*`)` ;
        for $Q \in$ *IsUpstreamFrom[C]* **do** Q`.off_edges .add(`*root, internalNode*`)` ;
      end
    end
  end
  return N
end

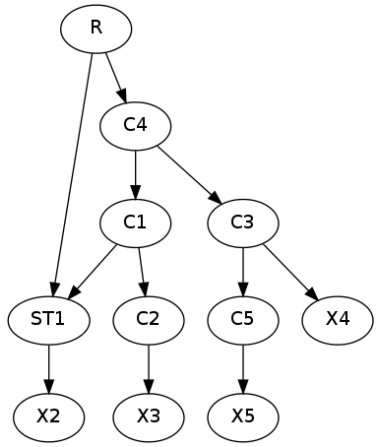
must be compatible with all the clusters in $\widetilde{\mathcal{C}}$; that is, no cluster has a non-trivial intersection with S_i. The procedure then adds a node called *internalNode* to the network and attaches the taxa contained in S_i to *internalNode* in the following steps: Let $P \subseteq \mathcal{C}$ be the set of clusters that are properly contained in S_i. All clusters in P are guaranteed to be pairwise compatible, by the definition of ST-set. So is safe to call the `Tree.Build` procedure on P, with *inputRoot set* to be the newly created *internalNode*. The program then processes any remaining taxa in $L = S_i \setminus \bigcup_{C \in P} C$ by creating an individual node x for each taxon in L and inserting it into N by adding an edge from *internalNode* to each $x \in L$, directed by lines 17-19 of `Network.Add`. Therefore, at this point, N represents all clusters $C \subset S_i$.

After the set P of taxa is processed, `Network.Add` uses depth-first search to efficiently compute the set $\mathcal{K}$ of clusters where each is a superset of S_i, whose tree edges are *maximally downstream* of all the superset clusters. $\mathcal{K}$ is computed in lines 7-15 and the clusters are stored in the *MaxDownstream* data structure. During this computation, `Network.Build` also computes the set of clusters *isUpstreamFrom[K]* for each $K \in \mathcal{K}$, the set of clusters that are upstream from K in N. Since the clusters in $\mathcal{K}$ contain S_i, the procedure adds an edge from the endpoint of K.`tree_edge` to *internalNode* of each $K \in \mathcal{K}$, so the subtree that represents S_i is on a a directed path from C.`tree_edge` of all $C \supseteq S_i$. After this step is performed, any cluster $C \supseteq S_i$ that is upstream of a cluster in $K \in \mathcal{K}$ will now contains the taxa in S_i as leaf descendants, because C has a directed path to K and the endpoint of K.`tree_edge` now has an edge to *internalNode* whose subtree has leaf descendants S_i. Finally, `Network.Build` iterates through all clusters C in *isUpstreamFrom[K]* for each $K \in \mathcal{K}$, and adds all the incoming edges of *internalNode* to C.`off_edges` *except* for the edge incident to K.`tree_edge`. This step is shown in lines 21-28. Therefore, after line 28 executes, all clusters

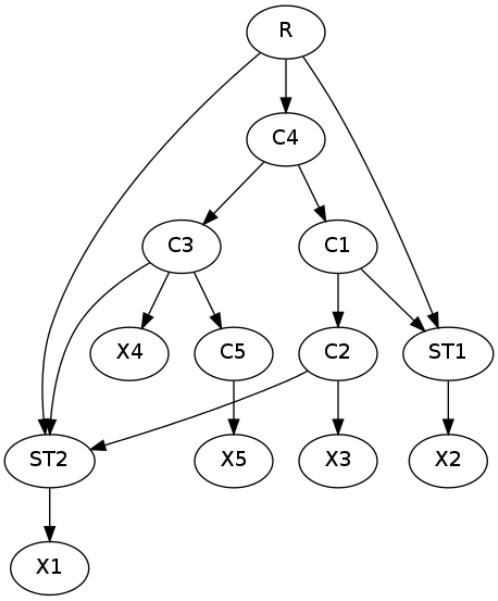
FIGURE 4. The intermediate states of the `Network.Build` procedure when building the network in Figure 2 to represent $\mathcal{C} = \{\{1,2,3\},\{1,3\},\{1,4,5\},\{3,4,5\},\{5\}\}$ with input maximal ST-set sequence $\mathcal{S} = \{\{1\},\{2\}\}$.



(A) Tree constructed to represent $\mathcal{C} \mid I$ where $I = \{3,4,5\}$ using `Tree.Build`. The C_4 node is highest in the tree because $C_4 \mid I$ has the largest size of all $C \in \widetilde{\mathcal{C}}$.



(B) Result of calling `Network.Add` on the last ST-set $\{2\}$ of $\mathcal{S}$ and the tree from Fig.4A. The procedure adds node X2 as a leaf child of the new *internalNode* labeled ST1. The set $\mathcal{K}$ of most downstream clusters containing $\{2\}$ is just $\{C_1\}$ There is an edge from the root R to ST1 because, without it, the tree below the C_4 node would be disconnected since C_4 does not contain 2.



(C) Result of calling `Network.Add` on the ST-set $\{1\}$ of $\mathcal{S}$ and the network from Fig. 4B. `Network.Add` creates new internal node ST2 with leaf child X1 and adds edges to ST2 from the `tree_edge`s of C_2 and C_3 because the clusters C_1, C_2, C_3 all contain $\{1\}$, which means they are the contents of the *Supersets* structure in this iteration. $\mathcal{K} = \{C_2, C_3\}$ because C_1 is upstream from C_2, so it obtains $\{1\}$ automatically from C_2. Meanwhile, C_4 is in *Disjoint* because it does not contain $\{1\}$, so there is another edge from the root to ST2, as in Fig. 4B to give C_4 a connected switching that bypasses the path through ST2.

$C \supseteq S_i$ will have a directed path to *internalNode*, and will have all the taxa in S_i as leaf descendants.

However, there are still some clusters that are disjoint from S_i but might be upstream in N from a cluster in $K \in \mathcal{K}$. This is a problem, because it means that after lines 21-28 execute, all of these disjoint clusters have had the taxa of S_i added to them as leaf descendants through the directed path from K.`tree_edge` to *internalNode*. Therefore, for each cluster $D \in$ *Disjoint* that is upstream from a cluster in $K \in \mathcal{K}$, the edge e from the endpoint of K.`tree_edge` to *internalNode* is added to the set D.`off_edges`, to make sure that there exists a switching of N that does not contain a directed path from D.`tree_edge` to *internalNode*. However, it is possible for a disjoint cluster D to be upstream from *all* the clusters in $\mathcal{K}$, which means that adding *all* of the incoming edges incident upon *internalNode* to D.`off_edges` will disconnect the subtree of D.`tree_edge` from the network. In this case, `Network.Build` creates a new edge from the *root* of N to *internalNode*, and then adds this edge to C.`off_edges` for all clusters $C \in$ *Supersets* that are downstream of D on lines 34-35. After this step, no cluster D that is disjoint from S_i has a directed path to *internalNode* in the graph $N - D$.`off_edges`, which is now guaranteed to be connected.

4.2. **Correctness**

It is clear from the pseudocode of Algorithm 3 that it generates a DAG using the input data. Therefore, we first prove:

Claim 4.2. *The* `Network.Add` *procedure produces a valid reticulation network that represents the input clusters.*

Proof. The set I is correctly represented by N by the correctness of the known `Tree.Build` algorithm. Suppose that, when `Network.Add` is called on input N and S_i, the network correctly represents the clusters $\widetilde{\mathcal{C}} = \mathcal{C} \mid (I \cup S_p \cup S_{p-1} \cup \ldots \cup S_{i+1})$. Let C be a clusters

in $\widetilde{\mathcal{C}}$. First, we need to show that after `network.add` finishes processing ST-set S_i, cluster C contains all of its taxa. A cluster $C \subset S_i$ will be correctly represented by N after `network.add` runs by the correctness of `Tree.Build`, and if a taxon $x \in C$ is not in S_i, then it is already a leaf descendant of C.`tree_edge` by the induction hypothesis. Similarly, if $x \in S_i$ and $C \in \mathcal{K}$ then the algorithm will add an edge from the endpoint of C.`tree_edge` to the *internalNode*, so x will be a leaf descendent of C.`tree_edge`. If the algorithm instead attaches *internalNode* to the tree edges of a set of clusters $\mathcal{K}$ such that $C \notin \mathcal{K}$, then by lines 9-17 there is some other cluster $C' \in \mathcal{K}$ downstream of C. By the definition of downstream, this means that N has a directed path from C.`tree_edge` to C'.`tree_edge`, so x will be a leaf descendant of C.`tree_edge`.

Next, we show that if C does not contain any taxa in S_i, then the algorithm does not force any $x \in S_i$ to be a leaf descendant of C.`tree_edge`. Indeed, on line 32, the algorithm adds the edges into the *internalNode* to D's set of `off_edges`. In some ways, this is just a book-keeping step to illustrate that N still displays the tree that represents $\mathcal{C}$; however, it is possible that *all* of the edges into the *internalNode* are downstream of C. This would mean that creating the switching that represents C from N would require disconnecting the graph. Therefore, on line 34, `Network.Build` takes care of this possibility by adding an edge from the root to the *internalNode*, which keeps the switching graph intact. Thus, even if C does not contain any taxa in S_i, the `Network.Add` procedure maintains the property that N is a connected network that represents C. □

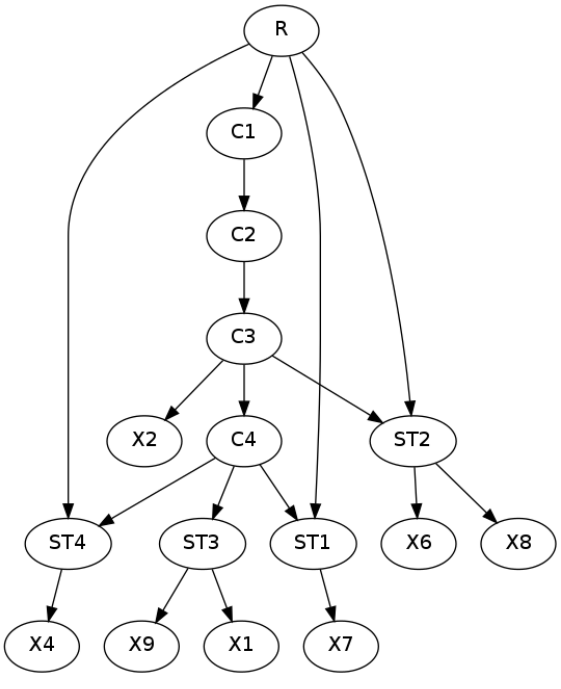
Therefore, we have shown that when the `Network.Build` is run on a set of clusters $\mathcal{C}$ and a valid maximal ST-set tree sequence $\mathcal{S}$, it produces a connected reticulation network N that properly represents $\mathcal{C}$ in the softwired sense.

4.3. **Minimizing Reticulations**

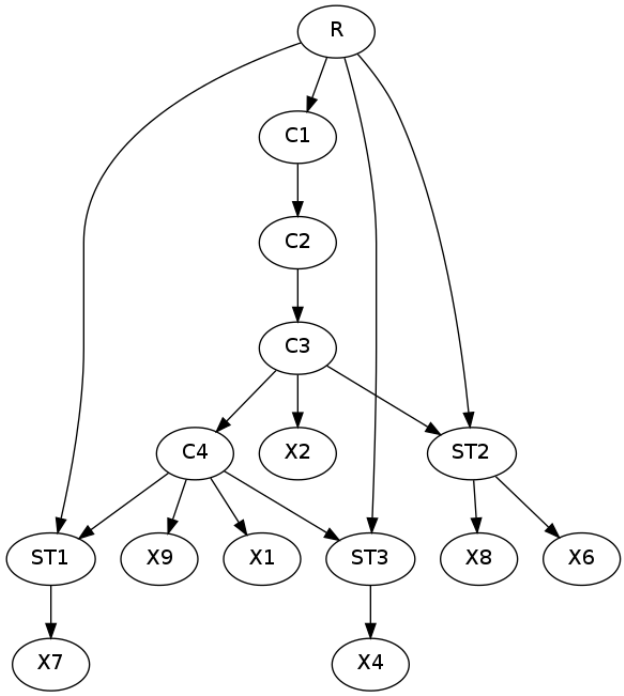
It is clear from the pseudocode and description of the `Network.Add` procedure that `Network.Build` adds only one non-leaf node, called *internalNode*, to the network in each iteration. It remains to demonstrate that *internalNode* really is a reticulation node, meaning it has in-degree of at least two. It turns out that there *are* some valid maximal ST-set tree sequences that produce internal nodes with in-degree equal to one. An example of such a sequence is shown in Figure 5. However, we will show:

Lemma 4.3. *If* `Network.Build` *creates a network* N *with* $m < p$ *reticulation nodes when executed on a maximal ST-set tree sequence* $\mathcal{S}$ *of length* p*, then there exists a maximal ST-set tree sequence* $\mathcal{S}'$ *of length* m*.*

FIGURE 5. Two almost-isomorphic networks produced by the `Network.Build` algorithm on the input cluster set $\{1,9,2,4\},\{1,9,2\},\{1,9,2,6,8\},\{1,9,4,7\}$.



(A) Network produced using the ST-set sequence $\{\{4\},\{1,9\},\{6,8\},\{7\}\}$. The node labeled ST3 has only one incoming edge.



(B) Network produced using the ST-set sequence $\{\{4\},\{6,8\},\{7\}\}$

Proof. Suppose that, after the `Network.Add` procedure processes an ST-set S_i, the *internalNode* added during its execution has in-degree one. This happens if there is only one cluster in $\mathcal{K}$, the set of maximally downstream clusters that contain S_i, meaning that the tree edges of all clusters $C \supseteq S_i$ lie on a single path in N. Since the tree edges of all clusters $C \subseteq I$ were placed by the `Tree.Build` procedure and the taxa in S_i were added to the tree constructed by `Tree.Build` via an edge from C.`tree_edge` to the *internalNode*, which we assumed to have in-degree one, then the set of clusters $\mathcal{C} \mid (I \cup S_i)$ can be arranged a tree. Therefore, all the clusters in $\mathcal{C} \mid (I \cup S_i)$ are pairwise compatible by the Perfect Phylogeny Theorem. This means that it is unnecessary to include S_i in the ST-set tree sequence, because the sequence $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_p\}$ is a maximal ST-set tree sequence with length $p-1$. Thus, if a sequence $\mathcal{S}$ of length p is the shortest maximal ST-set tree sequence for a set of clusters $\mathcal{C}$, then each *internalNode* will have in-degree of at least two. Hence, `Network.Build` will produce a reticulation network N with exactly p reticulation nodes. □

A reticulation network N that represents a set of clusters $\mathcal{C}$ is called *ret-minimum* with respect to $\mathcal{C}$ if it has the fewest number of reticulation nodes over all the networks that represent $\mathcal{C}$. To complete the proof of Conjecture 4.1, we show that

Lemma 4.4. *The* `Network.Build` *algorithm constructs a ret-minimum network for* $\mathcal{C}$ *when given as input a shortest maximal ST-set tree sequence.*

Proof. Let N be the network constructed by `Network.Build` for a set of clusters $\mathcal{C}$ on $\mathcal{S}$, a shortest maximal ST-set tree sequence for $\mathcal{C}$, of length p. Suppose toward a contradiction that there exists a network N' with $m < p$ reticulation nodes that is ret-minimum for $\mathcal{C}$. If N' had a reticulation node r such that all paths from r to a leaf go through another reticulation node r', then we could remove r and redirect the incoming edges of r into r' thus creating

a network N'' with *fewer* reticulation nodes. This operation would not modify the taxa, so N'' would still represent $\mathcal{C}$. Therefore, a every reticulation node in a ret-minimum network N' must have at least one path to a leaf that does not go through another reticulation node.

Let r_1 be a reticulation node of N' such that no path from r_1 to a leaf goes through another reticulation node. We know that at least one such reticulation node must exist in N', because otherwise N' would either be a tree or would contain a directed cycle. Let Σ'_1 be the set of taxa that are direct descendants of r_1 and G_1 be the subgraph composed of r_j, its subtree and its incoming edges. Then, inductively define r_j to be a reticulation node of $N' - G_1 - \ldots - G_{j-1}$ such that no path from r_j to a leaf contains any other reticulation nodes, where G_j is defined to be the subtree containing r_j, its subtree and its incoming edges, as before. Let S_j be the set of leaf descendants of r_j. Do this until N' has no remaining reticulation nodes. We argue that the sequence $\mathcal{S}' = \{\Sigma_m, \ldots, \Sigma_1\}$ is a maximal ST-set tree sequence; specifically,

(1) The graph $N' - G_1 - \ldots - G_m$ is a tree.

(2) Each $\Sigma_j \in \mathcal{S}'$ is an ST-set of $\mathcal{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$, which means Σ_j is compatible with all clusters $C \in \mathcal{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$ and all clusters $C \subset \Sigma_j$ are pairwise compatible.

(3) Each $\Sigma_j \in \mathcal{S}'$ is a maximal ST-set.

Property (1) follows from the construction of G_j; no more reticulation nodes remain after all m reticulation nodes are removed from N', so all remaining taxa are arranged on a tree.

We also know that Σ_j was arranged on a tree rooted at r_j before it was removed from N', which means that all the clusters $C \subset \Sigma_j$ are pairwise compatible by the Perfect Phylogeny Theorem. Suppose that some set $\Sigma_j \in \mathcal{S}'$ is not compatible with $\widetilde{\mathcal{C}} = \mathcal{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$, which means there is some cluster $C \in \widetilde{\mathcal{C}}$ in N' that has a nontrivial intersection

with Σ_j. Since N' represents C, the tree edge corresponding to C cannot be downstream of r_j in N', since this would indicate that $C \subset \Sigma_j$ and thus compatible with Σ_j. Therefore, the tree edge corresponding to C would either be upstream from r_j in the network or not on any path from the tree edge of C. In order for N' to represent C, there would have to be a path from the endpoint of C in N' to the subtree containing the taxa in $C \cap \Sigma_j$, which would mean that there is *another* reticulation node r' in G_j, which is a contradiction. Therefore Σ_j must be compatible with all the clusters in $\widetilde{\mathcal{C}}$ and so property (2) holds as well.

It remains to show that $\mathcal{S}'$ is a *maximal* ST-set tree sequence. Suppose that Σ_j is not a maximal ST-set of $\mathcal{C} \setminus (\Sigma_1 \cup \ldots \Sigma_{j-1})$ and that there exists some taxon $x \notin \Sigma_j$ that such that $\Sigma = \Sigma_j \cup \{x\}$ is also an ST-set. This means that Σ must be compatible with all the clusters in $\mathcal{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$, which means that all the clusters $C \supseteq \Sigma_j$ must either be equal to Σ_j or must also contain x. Otherwise, a cluster $C \supset \Sigma_j$ and Σ would have be incompatible, since C would contain elements not in Σ_j and Σ contains contain x. Let $\mathcal{K}$ be the set of clusters $C \supset \Sigma$. The tree edges of the clusters in $\mathcal{K}$ must lie on several different paths from the root of N' to r_j, because N' is assumed to be ret-minimum for $\mathcal{C}$, so it cannot have "unnecessary" reticulation nodes. If the clusters in $\mathcal{K}$ did lie on the same path, then r_j would have in-degree one and would not be a reticulation node.

Since the tree edges of the clusters in $\mathcal{K}$ *do not* lie on a single path, but all the clusters in $\mathcal{K}$ contain $x \notin \Sigma_j$, then there must be *another* reticulation node r such the tree edge endpoint of each cluster in $\mathcal{K}$ has an path to r, and the subtree S of r contains x. However, this means that all the clusters in $\mathcal{K}$ contain $\Sigma_j \cup S$, which means that $\Sigma_j \cup S$ is compatible with $\mathcal{K}$. This means that the reticulation node r is unnecessary and we can modify N' to create a network N'' with one fewer reticulation node, by removing the subtree of r and its

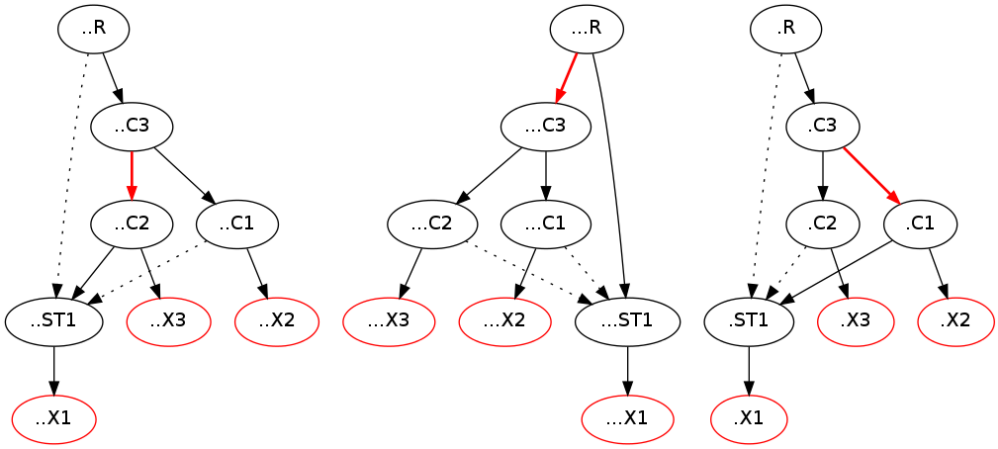
incoming edges from N' and then calling `Tree.Build` on cluster set $\mathcal{C} \mid \Sigma_j \cup S$ with input root r_j. However, this is a contradiction since N' is ret-minimum for $\mathcal{C}$; therefore, Σ_j must be a maximal ST-set and so property (3) also holds.

Therefore, if a network N' with $m < p$ is ret-minimum for $\mathcal{C}$, then we can construct maximal ST-set tree sequence $\mathcal{S}'$ of length m. However, this is a contradiction, since we claimed that the sequence $\mathcal{S}$ of length p that we passed as input to the `Network.Build` algorithm was already a shortest maximal ST-set tree sequence for $\mathcal{C}$. Therefore `Network.Build` produces a ret-minimum network when executed on a shortest maximal ST-set tree sequence for $\mathcal{C}$. □

We conclude the proof of Conjecture 4.1, which states that the History Bound algorithm computes the minimum number of reticulation nodes needed for a reticulation network to represent a set of clusters $\mathcal{C}$ in the softwired sense. We give a procedure that uses the execution history of the CHB algorithm with output R to construct a reticulation network N, which represents $\mathcal{C}$ in the softwired sense and has at most R reticulation nodes. Then, using the theory developed in [4] we argue that the procedure generates a ret-minimum network when given as input the execution history of CHB that produces the minimum value, or the History Bound, for $\mathcal{C}$.

However, we note that the number of reticulation nodes necessary for a network N to represent a set of clusters $\mathcal{C}$ in the *hardwired* sense is greater than or equal to the History Bound. Consider the set of taxa $X = \{1, 2, 3\}$ and cluster set $\mathcal{C}$ containing $\{1, 2\}, \{1, 3\}$ and $\{2, 3\}$. $\mathcal{C}$ is incompatible, so it cannot be represented by a tree, so at least one reticulation node will have to be present in any network that represents $\mathcal{C}$. Only one reticulation node is required for a network that represents $\mathcal{C}$ in the softwired sense, because any singleton subset

FIGURE 6. Network constructed by `Network.Build` for $\mathcal{C}$ with tree edges shown for each cluster.



of X is a maximal ST-set. Without loss of generality, if we choose $\{1\}$ to be the ST-set we remove, then the remaining cluster set $\widetilde{\mathcal{C}}$ is $\{2\}, \{3\}$ and $\{2,3\}$, which is compatible, so no more reticulation nodes are necessary for N. The network representing this input is shown in Figure 6.

Suppose there exists a network N with leaf set X that has a single reticulation node r. We will show that it cannot represent $\mathcal{C}$ in the hardwired sense. If two of the leaves (for example, 1 and 2), are leaf descendants of the r, then *all* upstream edges will have $\{1,2\}$ as leaf children, because we can no longer "switch off" edges when representing clusters in the hardwired sense. This means that no upstream edges can be the tree edges that correspond to clusters $\{1,3\}$ or $\{2,3\}$. Therefore, N does not represent $\mathcal{C}$. Otherwise, suppose one leaf child (taxon 1) is a descendant of the reticulation node in N, then none of the upstream edges can be tree edges of cluster $\{2,3\}$, because all upstream edges of r have leaf descendant $1 \notin \{2,3\}$. This shows that N cannot represent $\mathcal{C}$ in the hardwired sense in this case either, which means that a network requires at least two reticulation nodes to represent $\mathcal{C}$ in the hardwired sense.

5. A DP Algorithm for Constructing Ret-Minimum Networks

Now that we have established via Lemmas 4.3 and 4.4 that `Network.Build` produces a ret-minimum network N when executed on a shortest maximal ST-set tree sequence, whose length is equal to the History Bound, we can combine the two known algorithms to produce a procedure that will generate a ret-minimum network for a set of clusters $\mathcal{C}$, or an equivalent matrix M, in exponential time.

The original DP formulation of the History Bound algorithm was given by [1], and it runs in time $O(mn2^n)$ where $|X| = n, |\mathcal{C}| = m$. The presentation in Algorithm 6 is the same as the one given in [2]. The dynamic programming variant of the algorithm computes the History Bound for all subsets $K \subseteq X$ of size k. The key insight is that the History Bound value for K is one more than the minimum value of the History Bound over all subsets of K that have size $k - 1$.

Once again, in order to extract the ST-set sequence from the intermediate values computed by Algorithm 6, we need to save all the rows that are collapsed by rule **Dr** into

Algorithm 6: The dynamic programming equivalent of the History Bound algorithm.

Input: A binary matrix M representing cluster set $\mathcal{C}$ on taxon set X
Output: The value of the History Bound for M

```
for x ∈ X do set H[{x}] = 0;
for k ∈ {2, ..., |X|} do
    for each subset K ⊆ X of size k do
        set M_K to be the submatrix of M with row set K
        while M_K contains duplicate rows or single-entry columns do
            Dr: collapse together any duplicate rows of M_K
            Dc: remove any columns with a single 1 entry from M_K
        end
        /* choose which row removal minimizes the score for K ⊆ X      */
        set H[K] = min_{x∈K} (1 + H[K − {x}])
    end
end
return H[X]
```

Algorithm 7: An equivalent DP algorithm to compute the History Bound that saves away relevant history; namely the ST-sets that are removed to achieve each value.

Input: A binary matrix M representing cluster set $\mathcal{C}$ on taxon set X
Output: A table containing History Bound value for all subsets of X and the ST-set deletions that resulted in that value, for backtracking.

```
/* initialize the table H : P(X) → ℕ × P(X) for subsets of size 1        */
for x ∈ X do set H[{x}] = (0, ∅);
for k ∈ {2, ..., |X|} do
    for each subset K ⊆ X of size k do
        /* initialize the X' list for marking duplicate taxa for deletion  */
        for x ∈ X' do set X'[r] = ∅;
        set M_K to be the submatrix of M with row set K
        while M_K contains duplicate rows or single-entry columns do
            Dr:
            if two unmarked rows r_1 and r_2 are identical then
                mark X'[r_2] = r_1 for all rows r such that X'[r] == r_2 do
                    update the marker to X'[r] = r1
                end
            end
            Dc: remove any columns with a single unmarked 1 entry from M_K
        end
        /* collect all the marked rows into sets indexed by their markers  */
        for x ∈ X do
            set S[x] = {x} ∪ {r ∈ X such that X'[x] = r}
        end
        set f(x) = 1 + H[K − S[x]].first()
        set (h, r) = ( min_{x∈K} f(x), argmin_{x∈K} f(x) ) ;   /* save the row that minimizes f */
        set H[K] = (h, S[r]) ;          /* save ST-set corresponding to argmin f */
    end
end
return H
```

the DP table. Therefore, we make a change similar to the the one made to Algorithm 1. This is shown in Algorithm 7, and it will allow us to backtrack once the minimum value of H is computed. Note that the table H no longer maps from the subset of X to the integer value of the History Bound for that subset of X, but to pairs that save the ST-set that whose removal achieves the optimal value, in addition to the value itself.

Algorithm 8: The backtracking algorithm for constructing a ret-minimum network using the intermediate values of the History Bound computation.

Input: Table H containing the History Bound value for all subsets of X and the ST-set deletions that resulted in that value.
Output: Ret-minimum network N representing cluster set $\mathcal{C}$ on taxon set X

```
1  /* extract the ST-set sequence S in reverse from the DP table          */
2  set K = X; set S = {} ;
3  while K ≠ ∅ do
4  |  get (_, S_i) = H[K] ;
5  |  S.append(S_i) ;
6  |  set K = K − S_i
7  end
8  set nonSubsets = {C ∈ C | C ⊄ S_i for any S_i ∈ S} ;
9  set I = X − ⋃_{S∈S} S ;
10 /* restrict all the clusters to the initial set of taxa               */
11 for C ∈ C do  C.restrict_to(I);
12 N = graph();
13 N.add_nodes(root) ;
14 N = Tree.Build(N, root, nonSubsets) ;
15 for S_i ∈ S do
16 |  for C ∈ C do C.restrict_to (C.restriction ∪ S_i);
17 |  N = Network.Add(N, C, S_i);
18 end
19 return N
```

The new algorithm also changes the way subsets are mapped by H. In the previous procedure, the clean-up rules **Dr** and **Dc** were allowed to run on all the rows and columns of the matrix M_K, while in the modified algorithm, we only explicitly *remove* the rows from M_K after a taxon is chosen that minimizes the History Bound score. Nevertheless, these algorithms are equivalent, because the clean-up rules are deterministic [2], so there is a bijection between collapsing *all* other maximal ST-sets into meta-taxa and leaving all but one uncollapsed.

In Algorithm 8 we show the procedure for extracting the maximal ST-set tree sequence from the values recorded in H, and for using those to construct the ret-minimum network for the clusters encoded by M. The sequence is assembled from the second value of the pair

recorded at $H[K]$ for each $K \subset X$, starting from $K = X$. Otherwise, this algorithm is very similar to `Network.Build`, given in Algorithm 3; it first constructs a tree from the clusters restricted to I and then iteratively inserts each ST-set into the tree using `Network.Add`.

6. Conclusion

One of the interesting things about this work is the way our understanding of the Reticulation-Network problem progressed almost entirely backwards from the way NP-hard problems are typically presented in classrooms and textbooks. That is, the algorithm to compute a lower bound on R_{min}, was known first. Although the algorithm appeared to be performing some common-sense operations on the underlying networks that represent the data, a clear understanding of the combinatorial property of reticulation networks counted by algorithm was not known. The hardness of computing this value was shown by a nontrivial reduction that did not involve the underlying network.

To summarize, we explicitly proved the statement from [4] that every maximal ST-set tree sequence corresponds to an execution of the Candidate History Bound algorithm, and conversely that every execution of CHB generates a maximal ST-set tree sequence, thereby demonstrating that the length of the shortest maximal ST-set tree sequence for a set of clusters is equal to the value of the History Bound. Then, we proved the conjecture that the History Bound algorithm counts the minimum number of reticulation nodes in a network that represents the input clusters in the softwired sense. This was done by giving an algorithm that constructs the network in question and showing that a network with fewer reticulation nodes does not exist. We argued that this result does not apply to networks that represent clusters in the hardwired sense.

6.1. **Future work**

This new understanding of the combinatorial properties of the History Bound raises additional questions about its relationship to other desirable properties of reticulation networks and ARGs. The static formulation also allows us to propose standard analyses that

are typically used to grapple with NP-hard problems, such as ILP formulations, heuristics and approximation algorithms. We describe some of these questions in more detail in the following sections.

6.1.1. *Relationship to Reticulation Number.* In Section 1.1, we mentioned that computational biologists are often interested in minimizing the number of *reticulation events* in a network – a computation that also involves the edges of the resulting network. This thesis does not currently address any relationship between between the History Bound and reticulation *number*; however, the `Network.Build` algorithm does try to create "nice" reticulation nodes that do not have obviously redundant incoming edges. The iteration on lines 7-15 of Algorithm 3 attaches the new node to only the most downstream set of clusters, even though the network would still correctly represent the input if the reticulation node had incoming edges from the tree edges of all the relevant clusters, not just the downstream ones. We do not claim that this choice by `Network.Build` produces a network with minimum reticulation *number*. However, this raises the question of whether a ret-minimum network N also has minimum reticulation number $r(N)$, or if there are networks with lower reticulation number that contain more reticulation nodes than N.

6.1.2. *Relationship to R_{min}.* As described in Section 1.2, the History Bound algorithm originated in the ARG setting as a way to compute a lower bound on R_{min}. Although this work has explained the relationship of the History Bound to reticulation networks, it remains unclear whether the value has any direct combinatorial relationship to the minARGs that represent a binary matrix. Thus, a natural extension of this work would be to investigate the following conjecture by Gusfield:

Conjecture 6.1. *There exists a minARG that represents a binary matrix M whose number of visible recombination nodes is equal to the History Bound.*

where a recombination node is *visible* in an ARG if no path from that node to a leaf goes through another recombination node. We know that every ARG is a (softwired) reticulation network; perhaps there is a way to combine recombination nodes until none are left and show that the resulting graph is a ret-minimum network.

6.1.3. *Hardwired vs Softwired.* We point out that the History Bound counts the number of reticulation nodes that represents a set of clusters in the *softwired* sense. It appears that the flexibility provided by the softwired definition is what allows the `Network.Add` procedure to use one reticulation node per ST-set. Similarly, it also allows the tree edges of the initial clusters to be arranged in a somewhat careless order, because a cluster that does not contain an ST-set S that is located upstream in the network from a cluster that does contain S can always "bypass" S using the root-edge trick described in lines 33-35 of Algorithm 5 and shown in Figure 4. In contrast, the hardwired definition requires that *all* the leaf children of a cluster's tree edge be considered, not just those in a particular switching, so the "bypassing trick" does not work in that setting. Nevertheless, it is possible that we can use the insights developed in this work to develop an algorithm that will count the minimum number of reticulation nodes needed to represent a set of clusters in the hardwired sense, and investigate the relationship of this value to R_{min} and reticulation number.

6.1.4. *Approximation Algorithms and Kernels.* Another area of exploration related to this topic is the development of more efficient approximation algorithms that are guaranteed to compute a value that can be bounded by a constant factor of optimum. Indeed, from a human perspective, there are some situations where it seems that certain taxa are obviously

better to remove than others during the execution of the CHB algorithm, so it be desirable to formalize how well those heuristics would behave in practice, or whether they only work on some data sets. The structure of the networks produced by `Network.Build` also suggests that the order of the ST-sets in the sequences might not be as important as currently stated, and there may re-orderings of ST-set sequences that produce equivalent networks. Similarly, the example in Figure 5 suggests that some ST-set sequences are equivalent, so partitioning the space of ST-sequences into equivalence classes to shrink the search space may be desirable as well.

Appendix A. Source of `Network.Build`

This appendix contains the Python source of the `Network.Build` algorithm, and all its related data structures. Below, we show the data structure for encoding clusters as described in Section 4.1.

Listing 1. Python implementation of the Cluster data structure.

```
#! /usr/bin/python

class Cluster:
    # need to have the taxa for initialization
    def __init__(self, name, S):
        self.name = name
        self.taxa = S
        self.restriction = set()
        self.ct = self.taxa & self.restriction
        # network things
        self.off_edges = set()
        self.cut_edge  = None

    def __repr__(self):
        return self.name + "(" + str(list(self.ct)) + ")"

    def __str__(self): return self.__repr__()

    def size(self): return len(self.ct)

    def is_empty(self): return self.size() == 0

    def single(self):
        if self.size() == 1:
            return list(self.ct)[0]
        return None

    def current_taxa(self): return (self.ct)

    def restrict_to(self, S):
        self.restriction  = S
        self.ct = self.taxa & self.restriction
```

Below is the code for the `Tree.Build` algorithm, which takes a set of compatible clusters and creates a tree.

LISTING 2. Python implementation of the `Tree.Build` algorithm.

```
#! /usr/bin/python

import cluster

from copy import copy
from pygraph.classes.digraph import digraph as graph

# build a tree of out a set of compatible clusters and a subset of the taxa
# that displays C
def build(T, clusters, root):

    # process largest to smallest
    C = sorted(clusters, key = lambda c: c.size(), reverse = True)

    for i in range(0, len(C)):
        c = C[i]

        node = c.name
        leaves = ["X" + x for x in c.current_taxa()]

        print "\n=====> TREE: Processing " + node

        # check the already processed clusters if they're a superset of c
        superset = None
        for j in range(0, i):
            p = C[j]
            if c.current_taxa() <= p.current_taxa():
                superset = p # grab the smallest superset of c
            elif not c.current_taxa().isdisjoint(p.current_taxa()):
                exit("ERROR: clusters " + str(c) + " and " + str(p) + "are not
    compatible.")

        if superset != None:
            (u, v) = superset.cut_edge

            # otherwise, put a new node and hook the leaves onto it instead
            for n in leaves:
                print "deleting edge (" + v + ", " + n + ")"
                T.del_edge((v,n))
            edge = (v, node)
        else:
```

```
41              # disjoint from all the processed clusters -- new edge from the root
42              edge = (root, node)
43              T.add_nodes(leaves)
44
45          T.add_node(node)
46          for x in leaves: T.add_edge((node, x))
47
48          T.add_edge(edge); c.cut_edge = edge
49          print "Cluster " + node + " now has cut edge " + str(c.cut_edge)
50
51      return T
```

The following listing gives the code that adds a single ST-set into a reticulation network. This code contains some helper functions, such as `verify` function that draws all the trees displayed by a given network that represent a set of clusters. Although this function allows the networks to be human-verified, it can be easily modified to verify the networks automatically.

LISTING 3. Python implementation of the `Network.Add` algorithm, and helper functions.

```
#! /usr/bin/python
import sys
sys.path.append('..')
sys.path.append('/usr/lib/graphviz/python/')
sys.path.append('/usr/lib64/graphviz/python/')
import gv

# my files
import cluster
import tree

from sys import argv

from pygraph.classes.digraph import digraph as graph
from pygraph.algorithms.sorting import topological_sorting as topsort
from pygraph.algorithms.searching import depth_first_search as dfs
from pygraph.readwrite.dot import write

def draw(G, name):
    dot = write(G)
    pic = gv.readstring(dot)
    gv.layout(pic, "dot")
    gv.render(pic, "png", name + ".png")

def personalize_dfs(N, C):
    D = {}
    for c in C:
        for e in c.off_edges: N.del_edge(e)
        (_, pre, post) = dfs(N, root="R")
        D[c.name] = (pre, post)
        for e in c.off_edges: N.add_edge(e)

    return D

def verify(N, clusters):
```

```
36      V = graph()
37      for i in range(0, len(clusters)):
38          c = clusters[i]
39          prefix = "." * (i+1)
40          color  = "red"
41
42          for n in N.nodes():
43              attrs = []
44              if n[0] == "X" :  attrs += [("color", color )]
45              V.add_node(prefix + n, attrs)
46          for e in N.edges():
47              attrs = []; (x, y) = e
48              if e in c.off_edges: attrs += [("style","dotted")]
49              if e == c.cut_edge:  attrs += [("style", "bold"), ("color", color)]
50
51              new_edge = (prefix + x, prefix + y)
52              V.add_edge(new_edge, attrs=attrs)
53      return V
54
55  # given c1's traversal, is c1's cut edge downstream from c1?
56  def downstream(c1, c2, traversal):
57      (pre, post) = traversal
58      (_, v) = c1.cut_edge
59      (_, u) = c2.cut_edge
60
61      # this means v is a descendant (downstream) of u
62      return pre.index(u) < pre.index(v) and post.index(v) < post.index(u)
63
64  def build(N, clusters, st_set, label):
65
66      print "\n=====> NETWORK " + label + ": Processing " + str(st_set)
67
68      # compute the traversals for each cluster's tree
69      traversals = personalize_dfs(N, clusters)
70
71      supersets  = [] # most downstream clusters that contain S
72      subsets    = [] # clusters that are subsets of S
73      disjoint   = [] # clusters that don't contain S
74      upstream   = {} # clusters that contain S but don't get any edges added
75
76      print "Sifting clusters that contain set " + str(st_set)
77      for c in clusters:
78          upstream[c.name] = []
79          if st_set <= c.current_taxa():
80              supersets.append(c)
81              for k in supersets:
```

```
82                if k != c:
83                    if downstream(c, k, traversals[k.name]):
84                        print "  Removing " + k.name + " since it is upstream of
    " + c.name
85                        supersets.remove(k)
86                        upstream[c.name] += [k]
87                    elif downstream(k, c, traversals[c.name]):
88                        print "  Removing " + c.name + " since it is upstream of
    " + k.name
89                        supersets.remove(c)
90                        upstream[k.name] += [c]
91                        break
92
93        elif st_set.isdisjoint(c.current_taxa()):
94            disjoint.append(c)
95        elif c.current_taxa() < st_set:
96            subsets.append(c)
97        else:
98            exit("The " + str(st_set) + " is not a valid ST-set.\nIt has a
    nontrivial intersection with " + str(c) + ".\nQuitting ...")
99
100    # add the new node and leaves
101    rec_node = "ST" + label
102    N.add_node(rec_node)
103
104    # these subset guys don't have cut edges, so we need to set them first
105    N = tree.build(N, subsets, rec_node)
106
107    for x in st_set:
108        n = "X" + x
109        if not N.has_node(n):
110            N.add_node(n)
111            N.add_edge((rec_node, n))
112
113    for d in disjoint:
114        print "Processing disjoint cluster " + d.name
115
116        # if this guy has no cut edge yet, that means it's the subset of
117        # some other st-set further in the sequence, so it hasn't even been
    added yet
118        if d.cut_edge == None: continue
119
120        # find out if d is disjoint from all -- then we need to add a root edge
121        d_upstream = []
122        for c in supersets: d_upstream += [downstream(c, d, traversals[d.name])]
123        d_upstream_from_all = all(d_upstream)
```

```
124
125         root_edge = ("R", rec_node)
126         if d_upstream_from_all and not N.has_edge(root_edge):
127             # need an edge from the root to create the tree that doesn't break c
128             print "  Upstream from all; adding root edge " + str(root_edge)
129             N.add_edge(root_edge)
130
131         # see if d conflicts with any of the clusters containing S
132         for c in supersets:
133             (_, has_s) = c.cut_edge
134
135             if downstream(c, d, traversals[d.name]):
136                 # it's bad for the containing cluster to be downstream of the
    non-containing
137                 # one because then the non-containing one automatically includes
    the new st-set
138                 print "  Adding OFF edge " + str((has_s, rec_node)) + " to " +
    d.name
139                 d.off_edges.add((has_s, rec_node))
140
141                 # if cluster is disjoint, N will be disconnected when all its
    off edges are turned off.
142                 if d_upstream_from_all:
143                     print "  Adding OFF edge " + str(root_edge) + " to " + c.name
144                     c.off_edges.add(root_edge)
145                     for q in upstream[c.name]: q.off_edges.add(root_edge)
146
147     for c in supersets:
148         print "Adding recombination edge to " + c.name
149         (u, v) = c.cut_edge
150         N.add_edge((v, rec_node))
151
152         # edge that we just added needs to be turned off in the other clusters
    that have this node
153         for k in supersets:
154             if k != c:
155                 print "  Adding OFF edge " + str((v, rec_node)) + " to " + k.name
156                 k.off_edges.add((v, rec_node))
157                 for q in upstream[k.name]: q.off_edges.add((v, rec_node))
158     return N
```

Finally, this listing gives the code for `Network.Build`, which iteratively adds ST-sets from an input sequence into a tree to create a reticulation networks.

LISTING 4. Python implementation of the `Network.Build` algorithm.

```
#! /usr/bin/python

import cluster
import tree
import network

import os
from sys import argv
from pygraph.classes.digraph import digraph as graph

if len(argv) != 2:
    exit("Usage: process.py input")

filename = argv[1]
f = open(filename)

clusters = []; st_set_sequence = []; X = set(); i = 1
for line in f:
    line = line.strip()
    if line == '': break
    name = "C" + str(i); i += 1
    clusters += [cluster.Cluster(name, set(line.split(" ")))]

for line in f:
    line = line.strip()
    st_set_sequence += map(lambda x: set(x.split(" ")), line.split(", "))
f.close()

# the set of taxa is the union of all the clusters.
for c in clusters: X = X | c.taxa
print "X: " + str(X)

for c in clusters: c.restrict_to(X)
print "clusters:   " + str(clusters)

# initial set of taxa we need to build a tree for
I = X
for st_set in st_set_sequence: I = I - st_set

print "I " + str(I)

```

```
42 # grab the clusters that aren't strictly contained in any ST-seti
43 # or are disjoint from all ST-sets
44 non_subsets = []
45 for c in clusters:
46     c.restrict_to(I)
47     subset_of_s     = map(lambda s: c.taxa < s, st_set_sequence)
48     disjoint_from_s = map(lambda s: s.isdisjoint(c.taxa), st_set_sequence)
49 
50     if not any(subset_of_s) or all(disjoint_from_s):
51         non_subsets.append(c)
52 
53 st_set_sequence.reverse()
54 
55 # dump this into the folder with the input name
56 folder = filename.split(".")[0] + "/"
57 if not os.path.isdir(folder): os.mkdir(folder)
58 
59 # make a tree
60 N = graph(); N.add_node("R")
61 N = tree.build(N, list(non_subsets), "R")
62 
63 network.draw(N, folder + "network0")
64 
65 for i in range(0, len(st_set_sequence)):
66     st_set = st_set_sequence[i]
67     label  = str(i+1)
68 
69     for c in clusters: c.restrict_to(c.restriction | st_set)
70 
71     N = network.build(N, clusters, st_set, label)
72     V = network.verify(N, clusters)
73     network.draw(N, folder + "network" + label)
74     network.draw(V, folder + "verify" + label)
75 
76     print
77 
78     # for c in clusters:
79     #     print str(c) + " with cut edge " + str(c.cut_edge)
80     #     print "OFF: " + str(list(c.off_edges)) + "\n"
```

References

[1] V. Bafna and V. Bansal. Inference about recombination from haplotype data: Lower bounds and recombination hotspots. *Journal of Computational Biology*, 13(1):501–521, 2006.

[2] Dan Gusfield. *ReCombinatorics: The Algorithmics of Ancestral Recombination Graphs and Explicit Phylogenetic Networks*. MIT Press, 2014.

[3] D.H. Huson, R. Rupp, and C. Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Phylogenetic Networks: Concepts, Algorithms and Applications. Cambridge University Press, 2010. ISBN 9781139492874.

[4] Steven Kelk, Celine Scornavacca, and Leo van Iersel. On the elusiveness of clusters. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 9(2):517–534, March 2012. ISSN 1545-5963.

[5] Steven Kelk, Leo van Iersel, and Christopher Whidden. Personal communication, October 2012.

[6] S. R. Myers and R. C. Griffiths. Bounds on the minimum number of recombination events in a sample history. *Genetics*, 163(1):375–394, 2003.

[7] Leo van Iersel and Steven Kelk. When two trees go to war. *Journal of Theoretical Biology*, 269(1):245 – 255, 2011. ISSN 0022-5193.