HAL open science

A Resolution of the Static Formulation Question for the Problem of Computing the History Bound

Julia Matsieva, Steven Kelk, Celine Scornavacca, Chris Whidden, Dan Gusfield

► To cite this version:

Julia Matsieva, Steven Kelk, Celine Scornavacca, Chris Whidden, Dan Gusfield. A Resolution of the Static Formulation Question for the Problem of Computing the History Bound. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2017, 14 (2), pp.404-417. 10.1109/TCBB.2016.2527645 . hal-02154874

HAL Id: hal-02154874
https://hal.science/hal-02154874

Submitted on 16 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Resolution of the Static Formulation Question for the Problem of Computing the History Bound

Julia Matsieva, Steven Kelk, Celine Scornavacca, Chris Whidden, and Dan Gusfield

Abstract—Evolutionary data has been traditionally modeled via phylogenetic trees; however, branching alone cannot model conflicting phylogenetic signals, so networks are used instead. Ancestral recombination graphs (ARGs) are used to model the evolution of incompatible sets of SNP data, allowing each site to mutate only once. The model often aims to minimize the number of recombinations. Similarly, incompatible cluster data can be represented by a reticulation network that minimizes reticulation events. The ARG literature has traditionally been disjoint from the reticulation network literature. By building on results from the reticulation network literature, we resolve an open question of interest to the ARG community. We explicitly prove that the History Bound, a lower bound on the number of recombinations in an ARG for a binary matrix, which was previously only defined procedurally, is equal to the minimum number of reticulation nodes in a network for the corresponding cluster data. To facilitate the proof, we give an algorithm that constructs this network using intermediate values from the procedural History Bound definition. We then develop a top-down algorithm for computing the History Bound, which has the same worst-case runtime as the known dynamic program, and show that it is likely to run faster in typical cases.

Index Terms—rooted phylogenetic networks, clusters, reticulate evolution, parsimony, computational complexity, algorithms.✦

1 Introduction

Many types of biological data can be intuitively represented by labeled, directed trees, which model the diversity observed in a set of specimens, or *taxa*, with a sequence of branching events. However, many real-world data sets cannot be perfectly represented by a tree, because the biological signals conflict. To handle these cases, the field has allowed "joining events" as an additional mechanism for increasing diversity, which leads to interest in the study of phylogenetic networks [1]. 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 minimize certain properties of the nodes with in-degree > 1, at which point the problems become computationally hard [1].

An ancestral recombination graph, or ARG, is a type of phylogenetic network designed to model the evolution of ordered sequence data from a single starting sequence [2], [3]. The input to the problem is a set of binary sequences of the same length, and the objective is the construction of a DAG with one source node designated as the *root* and labeled with the ancestral sequence, and edge transitions modeling mutations along the edges, subject to the restriction that each site is only allowed to mutate once. However, sequences are also allowed to *recombine*, taking a prefix from one sequence and a suffix from another, to generate a new sequence of the same length. Usually, we aim to construct minARGs, or ARGs with the minimum possible number of recombinations. However, the problem of constructing minARGs is known to be NP-hard [4], [5], [6], so research has focused on computing lower bounds on R_{min}, the number of recombination nodes in a minARG.

A variety of methods have been developed for computing lower bounds on R_{min}. These methods all operate on the input matrix of binary sequences, but they generally also have a combinatorial interpretation. One particularly effective but computationally intensive lower bound, the History Bound, was originally defined as the output of an algorithm [7], and did not have a static interpretation. 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 to exponential time [8]. However, we also know from [8] that the problem of computing the History Bound is NP-hard. Nevertheless, tools like integer linear programming are often shown to be efficient in practice for instances of NP-hard problems of meaningful sizes, so a static formulation of the History Bound as an optimization problem is desirable.

In contrast to ARGs, *reticulation* networks (or *hybridiza-*

- *J. Matsieva and D. Gusfield are with the Department of Computer Science, University of California, Davis, Davis, 95616. E-mail: {jmatsieva, dmgusfield}@ucdavis.edu*
- *S. Kelk is with the Department of Knowledge Engineering (DKE), Maastricht University, Maastricht, Netherlands. E-mail: steven.kelk@maastrichtuniversity.nl*
- *C. Whidden is with the Fred Hutchinson Cancer Research Center, Seattle, WA, 98109. E-mail: cwhidden@fhcrc.org*
- *C. Scornavacca is with the Institut des Sciences de l'Evolution de Montpellier (ISEM), Montpellier, France E-mail: celine.scornavacca@univ-montp2.fr*

tion networks) model a set of clusters (also called clades), which are subsets of the taxa (although trees or triplets are also used in other problem settings, as summarized in [9] and [1]). 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, formally defined in Section 2.

In the past, the literature on ARGs has traditionally been disjoint from the work on reticulation networks, and the two communities have engaged in limited communication. In a previous paper aimed at the reticulation network community, the authors of [10] develop a theory to describe and analyze structures in cluster input data. The theory given there sheds light on the underlying meaning of the number computed by the History Bound algorithm. In fact, it is claimed that the construction given in [10] is equivalent to the History Bound. The first result in this paper is an explicit proof of this claim.

The main result in this paper is a further understanding of the relationship between the procedural definition of History Bound given in [7] and the structural properties of reticulation networks. Building on concepts developed in [10], we prove that the History Bound is equal to the minimum number of reticulation nodes in a reticulation network that represents the clusters encoded by an input matrix [10], solving the open problem posed in [11]. This result is not stated there and the paper was unlikely to have been noticed by members of the ARG community. Here, we give stylistically different definitions and algorithms to explicitly prove this result. However, the arguments given in Section 4 map closely onto the constructions in [10]; therefore, we will point out the correspondence between the arguments when it occurs.

We then use the structural insights from Section 4 to develop a top-down algorithm for computing the History Bound. We demonstrate that this algorithm runs faster in the typical case than the dynamic programming approach from [8].

2 Preliminaries

In this section, we will discuss standard definitions and relevant theorems from the phylogenetic networks literature. We will then recall definitions from [10], which are used later.

2.1 Ancestral Recombination Graphs

A *genealogical network* is a network constructed to model the derivation of a set of sequences originating from a single source sequence via the genetic processes of mutation and recombination. Thus, the underlying graph is a rooted DAG where the root is labeled with the ancestral sequence, and the observed or input sequences appear at the leaves. Formally, a genealogical network is created from a collection of n input sequences of length m over a *binary* alphabet representing single nucleotide polymorphisms (SNPs), usually given in a $n \times m$ binary matrix M, with each row representing the binary sequence for a taxon. A network N that represents M has exactly n leaves, each labeled with a sequence from M according to the rules discussed below.

The edges of N can be either labeled or unlabeled, with a distinction made between *tree edges*, edges into a node with in-degree one, or *recombination edges* which are edges into a *recombination node*, which has in-degree two. A tree edge may be given an integer label in the range $\{1, \ldots, m\}$ to indicate the site in M at which a mutation occurs, or be unlabeled to indicate no change. Thus, the sequence labeling a tree node can be read from its path from the root by interpreting each edge as a binary state change. Under the single-crossover model, the sequence labeling a recombination node takes a prefix from one parent sequence and a suffix from the other, resulting in a sequence of length m. An integer in $\{1, \ldots, m\}$ specifying the length of the prefix is usually indicated on the recombination node.

A genealogical network N is an *ancestral recombination network* (or ARG) if each site is allowed to mutate at most once. That is, an ARG N has at most one tree edge with label c for all $c \in \{1, \ldots, m\}$. ARGs are heavily studied in population genetics [12]; a detailed motivation for their use appears in [13]. It is especially desirable to construct a minARG, i.e. 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 [13].

2.2 The History Bound

The algorithm that defines the History Bound on R_{min} was given by Myers and Griffiths in [7]. It operates on a binary input matrix M and computes a lower bound on the minimum number of recombination nodes that must appear in an ARG that represents M. We give the version of the algorithm as explained in [13], where it is split into two procedures, procedure CHB for generating a *candidate* for the History Bound, which is shown as Algorithm 2.1, and procedure CHB_Branch, which selects the minimum value over all the candidates. This version assumes that the ancestral sequence for the data is the all-zero sequence. Procedure CHB reduces the input matrix by running the Clean procedure, which iterates the following operations:

Dr: which collapses identical rows together
Dc: which removes columns from M with at most one 1

These operations are performed by Clean until neither is possible. At this point the CHB procedure performs operation **Dt**, choosing an arbitrary row to delete from the matrix. The CHB procedure iterates these operations until the matrix is empty. The number of row deletions

Algorithm 2.1 CHB algorithm that computes a candidate for the History Bound as given in [13].

```
1: procedure CLEAN(M)
2:     while possible do
3:         Dr: collapse together duplicate rows of M
4:         Dc: remove columns of M with at most one 1
5:     return M
6: end procedure
7:
8: procedure CHB(M)
9:     set M~ = M, H = 0
10:    while M~ is not empty do
11:        M~ = CLEAN(M~)
12:        Dt: remove an arbitrary row r from M~
13:        set H = H + 1
14:    return H
15: end procedure
16:
17: procedure CHB_BRANCH(M)
18:    return min_{all possible executions} CHB(M)
19: end procedure
```

by rule **Dt**, shown on line 14 of Algorithm 2.1, is a candidate for the History Bound; the smallest candidate is then computed by procedure CHB_BRANCH, executions of CHB. This smallest value is defined to be the History Bound.

2.3 Reticulation Networks and Clusters

Critical background for this paper focuses on the development of reticulation networks in the cluster setting. A *reticulation network* on a finite set X of taxa 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 called a *reticulation node* and incoming edges into such a node are called *reticulation edges*. The literature on reticulation networks is often concerned with counting reticulation "events", which are given by the *reticulation number* of a network $N = (V, E)$, defined as follows:

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

where $d_{in}(v)$ denotes the in-degree of node v [10]. Then, the reticulation number $r(\mathscr{C})$ of a set of clusters $\mathscr{C}$ is the minimum of the reticulation numbers over all the reticulation networks that represent, or model, $\mathscr{C}$. However, in this paper, we will be concerned with counting the *number of reticulation nodes* in a network N, denoted $nr(N)$.

A cluster is a subset of X. 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 $\mathscr{C}$ of clusters is compatible if all the clusters in $\mathscr{C}$ are pairwise compatible. A collection $\mathscr{C}$ of clusters can be converted to a matrix M by choosing an arbitrary ordering on X and $\mathscr{C}$, filling the matrix by placing 1 into the row corresponding to taxon x and the column

$$\mathscr{C} = \begin{array}{c|cccc} C_1 & 1 & 2 & 3 & \\ C_2 & 1 & 2 & 4 & 5 \\ C_3 & 1 & 3 & & \\ C_4 & 1 & 4 & 5 & \\ C_5 & 3 & 4 & 5 & \\ C_6 & 5 & & & \end{array} \qquad \text{M} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

(a) (b)

Fig. 1. (a) An example set of clusters $\mathscr{C}$ over $X = \{1, 2, 3, 4, 5\}$. C_6 is a singleton cluster. Such clusters, and their corresponding columns, are considered *uninformative*. (b) The matrix equivalent of cluster set $\mathscr{C}$. The ith row of M represents taxon i of X. An entry with value 1 in row i of column j corresponds to membership of taxon i in cluster C_j.

corresponding to cluster C, if $x \in C$, and placing 0 there otherwise (see Figure 1). We can also convert a matrix M into a set of clusters $\mathscr{C}$ by considering the element encoded by a row of M to be a member of cluster $C \in \mathscr{C}$ if there is a 1 in the column corresponding to C in M. 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 presence of all three pairs is referred to in [13] as the Three Gametes condition.

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 for x in M. See [13] for a more complete discussion. This definition assumes that T is constructed with an all-zero ancestral sequence, which we will assume throughout this paper. 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 perfect phylogeny that represents the data, when there is at least one pair of incompatible columns in M. Equivalently, a set of clusters can be represented by a perfect phylogeny, if and only if the clusters are compatible [13].

In contrast to the ARG setting, the input to the reticulation network problem is a set X of taxa and an *unordered* collection $\mathscr{C}$ of clusters, where each $C \in \mathscr{C}$ 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) in N such that the set of leaf descendants of v is exactly equal to C. N represents a cluster C in the *softwired* sense if there is a rooted spanning subtree T of N such that T represents C in the hardwired sense. That is, if N contains an edge e such that C is equal to the set of all leaf descendants of e after all but one incoming edge at each reticulation

node of N is removed [1]. The operation of turning on one reticulation edge at each reticulation node and turning all others off is referred to as the construction of a *switching* of N in [10]. A network N represents a collection of clusters $\mathscr{C}$ if it represents each $C \in \mathscr{C}$ in the softwired sense.

The problem in this setting then consists of constructing reticulation networks to represent $\mathscr{C}$ subject to some optimality criterion, such as minimizing the reticulation number, $r(\mathscr{C})$. In this paper, we will usually discuss networks with the minimum *number of reticulation nodes*, so we define a network N to be *ret-minimum* for a set of clusters $\mathscr{C}$ if N has the fewest number of reticulation nodes over all networks that represent $\mathscr{C}$ in the softwired sense. We use $nr(\mathscr{C})$ to denote that number. Therefore, a reticulation network N that represents $\mathscr{C}$ is *ret-minimum* if and only if $nr(N) = nr(\mathscr{C})$.

Figures 6 and 7 in the Appendix, which can be found on the Computer Society Digital Library, show an example of a reticulation network and an ARG for the data in Figure 1. According to the definitions in this section, an ARG is also a reticulation network that represents the clusters encoded by M in the softwired sense. However, the internal structure of ARGs tends to have more complexity due to the requirements of the model, which is designed to describe the derivation of ordered sequence data.

2.4 ST-Set Sequences

In this paper, we build on the ST-set concept from [10]. An ST-set is a subset of the taxa with special, "treelike" properties. Informally, an ST-set of $\mathscr{C}$ can be thought of as a union of compatible clusters or compatible subsets of clusters. Given a subset $S \subseteq X$ of taxa, let $\mathscr{C} \setminus S$ denote the result of removing S from every $C \in \mathscr{C}$, and let $\mathscr{C} \mid S$ denote the restriction of $\mathscr{C}$ to S (i.e. $\mathscr{C} \setminus (X - S)$). Formally, $S \subseteq X$ is an ST-set with respect to a collection $\mathscr{C}$ of clusters if

1) S is compatible with $\mathscr{C}$.
2) all pairs of clusters $C_1, C_2 \in \mathscr{C} \mid S$ are compatible.

An ST-set S is *maximal* if there is no other ST-set S' such that $S \subset S'$. It is shown in [10] that all of the maximal ST-sets of a collection of clusters can be computed in time polynomial in the size of the input.

A sequence $\mathscr{S} = \{S_1, S_2, \ldots, S_p\}$ is an *ST-set sequence* if each $S_i \in \mathscr{S}$ is an ST-set of $\widetilde{\mathscr{C}} = \mathscr{C} \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})$; $\mathscr{S}$ is a *maximal* ST-set sequence if each S_i is a maximal ST-set of $\widetilde{\mathscr{C}}$. $\mathscr{S}$ is a maximal ST-set *tree* sequence, if all the clusters contained in $\mathscr{C} \mid I$ where

$$I = X - (S_1 \cup \ldots \cup S_p)$$

are pairwise compatible. By the Perfect Phylogeny Theorem, this means that all the taxa in $\mathscr{C} \mid I$ can be represented by a perfect phylogeny.

The length p of the shortest maximal ST-set tree sequence that can be computed for $\mathscr{C}$ is defined in [10] to be the MST lower bound on the reticulation *number* $r(\mathscr{C})$ and it is shown to be a true lower bound. But, in this paper, we are more concerned with the minimum number of reticulation nodes in a network that represents $\mathscr{C}$. In Section 3, we show that the length of the shortest maximal ST-set tree sequence is equal to $nr(\mathscr{C})$, the minimum number of reticulation nodes in a reticulation network that represents $\mathscr{C}$ in the softwired sense.

3 Bound Equivalence

Throughout this section, we will use the conversion between clusters and matrices described in Section 2.3 and demonstrated in Figure 1. Using the concepts developed in [10], we formally prove the following claim:

Claim 3.1: Given a binary matrix M, the minimum H computed by execution of the CHB algorithm is equal to the length of the shortest maximal ST-set tree sequence for the set of clusters $\mathscr{C}$ encoded by M.

This claim was suggested but not explicitly stated in [10]. 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 conversely that every maximal ST-set tree sequence corresponds to an execution of algorithm CHB.

Lemma 3.1: Let $\widetilde{M_i}$ denote the state of the matrix after iteration i of CHB and let $\{r_1, \ldots, r_p\}$ be the sequence of rows that are deleted by operation **Dt** during an execution of CHB. For each r_i, let R_i denote the set of rows that were collapsed together with r_i by rule **Dr** before r_i was deleted by a **Dt** operation. 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.

Proof: In order to prove Lemma 3.1, we must first establish that every set S_i is a maximal ST-set of $\widetilde{\mathscr{C}_i} = \mathscr{C} \setminus (S_1 \cup \ldots \cup S_{i-1})$, which means we must prove that:

1) S_i is compatible with $\widetilde{\mathscr{C}_i}$, the set of clusters encoded by $\widetilde{M_i}$.
2) any pair of clusters $C_1, C_2 \subset S_i$ that are in $\widetilde{\mathscr{C}_i}$ are pairwise compatible.
3) there does not exist another element e in $\widetilde{\mathscr{C}_i}$ such that $S_i \cup \{e\}$ is also an ST-set.

Suppose toward a contradiction, that S_i is not compatible with $\widetilde{\mathscr{C}_i}$. This implies that there must exist some cluster $C \in \widetilde{\mathscr{C}_i}$ 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 must be identical when ignoring uninformative columns at iteration i. However, z is also a member of C, which cannot correspond to an uninformative column, because it also contains another distinct element y. But then x and z are not identical on informative columns. Therefore, in order for the algorithm to place x and z into S_i, x must agree with z on column C, and so it must also be a member of C, which contradicts the assumption that $x \notin C$, and therefore (1) holds.

Similarly, suppose that there are two incompatible clusters $C_1, C_2 \in \widetilde{\mathscr{C}_i} \mid S_i$, the restriction of $\mathscr{C}_i$ to S_i. We know that all the elements in S_i must have been found by CLEAN to be identical; however, the assumed incompatibility of C_1, C_2 implies that there exist three elements x, y, z such that $x, z \in C_1$ and $x \notin C_2$; and $y, z \in C_2$ and $y \notin C_1$. Both columns C_1 and C_2 are informative because they contain two rows with 1 entries. This means that x and y are not identical after iteration i, so they would not have been placed in S_i, a contradiction. Therefore C_1 and C_2 must be compatible, and (2) holds. This shows that S_i is an ST-set of $\mathscr{C}_i$.

Furthermore, S_i is a maximal ST-set. Suppose that S_i is not maximal, so there exists a row e such that $S'_i = S_i \cup \{e\}$ is also an ST-set of C_i. If e were identical to r_i with respect to informative columns, then it would have been collapsed 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

(a) either there exists an informative cluster $C \in \widetilde{\mathscr{C}_i}$ such that $e \in C$ and $r_i \notin C$,
(b) or there exists an informative cluster $C \in \widetilde{\mathscr{C}_i}$ such that $e \notin C$ and $r_i \in C$.

In case (a), in order for S'_i to still be compatible with $\widetilde{\mathscr{C}_i}$, cluster C *must* be a singleton cluster that only contains e. Otherwise, if C contains some other element $d \in S'_i$, its existence will give rise to a violation of the Three Gametes condition in C and S'_i. This is 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 uninformative in $\widetilde{M_i}$ 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 $C \subseteq S_i$, because, if C contained an element $d \notin S_i$, then the taxa r_i, e and d would create a violation of the Three Gametes condition 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 is a subset of S_i, then it corresponds to an uninformative column in $\widetilde{M_i}$, because it has 1 entries in rows that are contained in S_i and since $S_i = R_i \cup \{r_i\}$, all of the rows in R_i have already been removed. Thus, the column encoding C has only one 1 entry, at row r_i, and is therefore an uninformative column at the end of iteration i. So we conclude that there is no $e \notin R_i$ that can be added to S_i and maintain its ST-set properties. Therefore, S_i is maximal, so property (3) holds as well.

In order to show that $\{S_1, \ldots, S_p\}$, as defined in the statement of Lemma 3.1, is a maximal ST-set *tree* sequence, it remains to show that all of the clusters contained in $I = X - (S_1 \cup \ldots \cup S_p)$ are compatible. We know that after r_p is deleted by rule **Dt**, the **Dr** and **Dc** operations remove all remaining rows and columns in M. Otherwise, it would be necessary to apply **Dt** again and r_p would not have been the last row in the sequence. All the clusters contained in I must be compatible, because otherwise there would be two clusters $C_1, C_2 \subseteq I$ with a non-trivial intersection. Then, neither rules **Dc** nor **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. □

Lemma 3.1 implies that the History Bound, the minimum possible value produced by CHB, 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 the following Lemma.

Lemma 3.2: For every maximal ST-set tree sequence $\mathscr{S} = \{S_1, \ldots, S_p\}$, there exists an execution of CHB 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: We prove this by induction on the iterations of CHB. We assume that when rule **Dr** combines two rows together, it leaves the row with the smaller index in M and removes the row with the larger index.

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 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 in $\widetilde{M_i}$ when rule **Dt** executes in CHB.

Suppose the row sequence $R = \{r_1, \ldots, r_{j-1}\}$ is chosen for deletion by rule **Dt** in the first $j - 1$ iterations of an execution of CHB, such that each row $r_i \in R$ 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 jth 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 for deletion. Row r could not have been deleted by rule **Dr** in $\widetilde{M_{j-1}}$, since if it were, it would be identical to a row r' that has a lower index in M. However, then all of the elements of S_j would also be found to be identical to r' before iteration j. This contradicts the assumption that r has the lowest index in S_j, so r cannot have been removed by rule **Dr** in M_j.

Next, suppose that, at iteration $i < j$ of CHB, row r was deleted by rule **Dt**. However, we inductively assumed that the algorithm removes each maximal ST-set $S_i \in \mathscr{S}$ at iteration $i < j$ and, since S_i is maximal, $S_i \cup \{r\}$ cannot also be an ST-set of $\widetilde{\mathscr{C}_i}$. Furthermore, it is shown in [10] that the maximal ST-sets of $\mathscr{C}$ partition X, so r cannot belong to both S_i and S_j.

Therefore, the row r in S_j with the lowest index cannot be missing in $\widetilde{M_j}$ during iteration j of CHB. Hence, $r_j = r$ and the induction argument is complete. This proves Lemma 3.2.

□

Since we have proved Lemmas 3.1 and 3.2, every run of CHB corresponds to a maximal st-set tree sequence and every maximal-st-set corresponds to an execution of

Algorithm 4.1 Algorithm that constructs a reticulation network N with $\leq p$ nodes when given a valid maximal ST-set tree sequence $\mathscr{S}$ of length p.

```
1:  procedure NETWORK.BUILD(𝒞, 𝒮)
2:      N = graph()
3:      N.add_nodes(root)
4:      set X = ⋃_{C∈𝒞} C;  set I = X − ⋃_{S∈𝒮} S;
5:
6:      set Subsets = {C ∈ 𝒞 | C ⊂ S_i for some S_i ∈ 𝒮};
7:      set nonSubsets = 𝒞 − Subsets
8:      for C ∈ 𝒞 do
9:          𝒞̃ = 𝒞̃ ∪ {C.restrict_to(I)}
10:     set N = TREE.BUILD(N, root, nonSubsets)
11:     set 𝒮 = 𝒮.reverse()
12:     for S_i ∈ 𝒮 do
13:         for C ∈ 𝒞̃ do
14:             C.restrict_to(C.restriction ∪ S_i)
15:         N = NETWORK.ADD(N, 𝒞̃, S_i)
16:     return N
17: end procedure
```

Algorithm 4.2 Algorithm for adding a tree to network N that represents a set of compatible clusters $\widetilde{\mathscr{C}}$.

```
1:  procedure TREE.BUILD(N, inputRoot, 𝒞̃)
2:      set 𝒞̃ = a list of clusters in 𝒞̃ sorted by size
3:      for C ∈ 𝒞̃ do
4:          N.add_nodes(newNode)
5:          set Σ = the smallest cluster that is a superset of C
6:          if Σ exists in 𝒞̃ then
7:              (u, v) = Σ.tree_edge
8:              set treeEdge = (v, newNode)
9:              for x ∈ C do
10:                 N.delete_edge(v, x)
11:         else
12:             set treeEdge = (inputRoot, newNode);
13:             for x ∈ C do
14:                 N.add_nodes(x)
15:         for x ∈ C do
16:             N.add_edge(newNode, x)
17:         N.add_edge(treeEdge)
18:         C.tree_edge= treeEdge
19: end procedure
```

CHB, it follows that an execution of CHB that produces the minimum value corresponds to a shortest maximal ST-set tree sequence, Claim 3.1 is proved.

4 DEFINING THE HISTORY BOUND

In this section, we explicitly state the main result of the paper:

Claim 4.1: Let M be a binary matrix and let $\mathscr{C}$ be the set of clusters encoded by M. Then, the History Bound for M is equal to $nr(\mathscr{C})$, the minimum number of reticulation nodes that must be present in any network that represents $\mathscr{C}$ in the softwired sense.

Although this claim is not stated explicitly in [10], many of the algorithms presented in this section are extensions of the constructions presented there, and are used here to obtain bounds on $nr(\mathscr{C})$ rather than $r(\mathscr{C})$. We will explicitly point out such extensions in the results that follow.

To prove Claim 4.1, we first give an algorithm that takes as input the set of clusters $\mathscr{C}$ and a maximal ST-set tree sequence $\mathscr{S} = \{S_1, \ldots, S_p\}$ for $\mathscr{C}$ and constructs a reticulation network N that represents $\mathscr{C}$ with at most p reticulation nodes. We show that this algorithm would produce a network with exactly p reticulation nodes when executed on a sequence $\mathscr{S}$ 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. Combined with Claim 3.1, this proves Claim 4.1.

4.1 Network.Build

In this section, we describe in detail 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 $\mathscr{S}$ of length p. The explicit pseudocode for the NETWORK.ADD procedure of the NETWORK.BUILD algorithm is given in Algorithm 4.3. NETWORK.ADD inserts a ST-set S into a network N that represents $\mathscr{C} \setminus S$ such that the resulting network represents $\mathscr{C}$. NETWORK.ADD performs the same operations as the procedure outlined in the proof of Lemma 10 of [10]. The explicit procedures are included here in the interest of making the paper self-contained, as well as for greater procedural clarity and accessible implementation. The details of the NETWORK.ADD procedure are also used in the proof of Lemma 4.2.

As mentioned in Section 2, 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 $C \in \mathscr{C}$, 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 N might have reticulation edges that are not on the path from C.`tree_edge` to a leaf. However, any reticulation nodes in the subtree of C.`tree_edge` will have no incoming edges from outside the subtree. In the algorithms shown, we will use C.`restrict_to(`S`)` to indicate that the cluster C has been restricted to just the elements present in the set S. Therefore, all subsequent set-theoretic operations will be executed on the taxa present in $C \mid S$. We will use the C.`restriction` statement to refer to the most recent set S that C has 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 4.1. 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

Algorithm 4.3 Procedure that inserts ST-set S_i into N, maintaining the property that N represents $\widetilde{\mathscr{C}_i}$.

```
1:  procedure NETWORK.ADD(N, C̃_i, S_i)
2:      N.add_nodes(internalNode)
3:      set Disjoint = {C ∈ C̃_i | S_i ∩ C = ∅}
4:      set Subsets = {C ∈ C̃_i | S_i ⊃ C}
5:      set Supersets = {C ∈ C̃_i | S_i ⊆ C}
6:      set MaxDownstream = {};  set IsUpstreamFrom = {};
7:      for (C, K) ∈ Supersets × Supersets do
8:          if K is downstream from C then
9:              MaxDownstream.remove(C)
10:             MaxDownstream.add(K)
11:             IsUpstreamFrom[K].add(C)
12:         else if C is downstream from K then
13:             MaxDownstream.remove(K)
14:             MaxDownstream.add(C)
15:             IsUpstreamFrom[C].add(K)
16:     N = TREE.BUILD(N, internalNode, Subsets)
17:     for x ∈ (S_i − ⋃_{C∈Subsets}) do
18:         N.add_nodes(x)
19:         N.add_edge(internalNode, x)
20:     for C ∈ MaxDownstream do
21:         set (u, v) = C.tree_edge
22:         N.add_edge(v, internalNode)
23:         for K ≠ C ∈ MaxDownstream do
24:             K.off_edges.add(v, internalNode)
25:             for Q ∈ IsUpstreamFrom[K] do
26:                 Q.off_edges.add(v, internalNode)
27:     for (D, C) ∈ Disjoint × MaxDownstream do
28:         if C is downstream from D then
29:             set (u, v) = C.tree_edge
30:             D.off_edges.add(v, internalNode)
31:             if D is upstream from all of Supersets then
32:                 N.add_edge(root, internalNode)
33:                 for Q ∈ IsUpstreamFrom[C] do
34:                     Q.off_edges.add(root, internalNode)
35:     return N
36: end procedure
```

two node arguments. An edge can be removed from a graph G using G.delete_edge.

The outer abstraction layer of NETWORK.BUILD is shown as Algorithm 4.1. Since the input $\mathscr{S}$ is a maximal ST-set tree sequence, we know that after we perform the restriction to I on lines 7-9, all the clusters in $\widetilde{\mathscr{C}}$ will be pairwise compatible. So $\widetilde{\mathscr{C}}$ can be represented by a tree, built by the algorithm shown in the pseudocode as TREE.BUILD in Algorithm 4.2. It is equivalent to the solution to the Perfect Phylogeny problem in Section 2.1.2 of [13]. We will also use Algorithm 4.2 to extend a network N by adding a tree to N at a specified node. We give a specific implementation in Algorithm 4.2; it takes as input a network N, a node inputRoot of N, and a set of compatible input clusters $\widetilde{\mathscr{C}}$; it outputs the updated network N with a new subtree rooted at inputRoot that represents $\widetilde{\mathscr{C}}$. More importantly, it also sets the tree_edge values for all the input clusters, even those whose restriction is empty. Thus, after line 11 of Algorithm 4.2 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 the set of taxa $C \mid I$. After building the initial tree for $\widetilde{\mathscr{C}} = \mathscr{C} \mid I$, NETWORK.BUILD reverses the maximal ST-set tree sequence $\mathscr{S}$ and iteratively adds ST-sets into N by calling the procedure NETWORK.ADD, shown in Algorithm 4.3.

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 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.

At each iteration, NETWORK.BUILD will insert an ST-set S_i of $\mathscr{S}$ into the network N, maintaining the invariant that, after each iteration, N represents

$$\widetilde{\mathscr{C}_i} = \mathscr{C} \mid (I \cup S_p \cup S_{p-1} \cup \ldots \cup S_i).$$

When processing a maximal ST-set S_i, the program NETWORK.ADD 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 $\widetilde{\mathscr{C}_i}$ because S_i is an ST-set and must be compatible with all the clusters in $\widetilde{\mathscr{C}_i}$; 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 $P \subseteq C$ 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 it 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 $\mathscr{K}$ of clusters where each is a superset of S_i, whose tree edges are maximally downstream of all the superset clusters. $\mathscr{K}$ is computed on lines 7-15 of Algorithm 4.3 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 $K \in \mathscr{K}$, the set of clusters that are upstream from K in N. Since the clusters in $\mathscr{K}$ contain S_i, the procedure adds an edge from the endpoint of K.tree_edge of each $K \in \mathscr{K}$ to internalNode, so the subtree that represents S_i is on a directed path from K.tree_edge to all S_i. After this step is performed, any cluster $C \supseteq S_i$ that is upstream of a cluster in $K \in \mathscr{K}$ will contain the taxa in S_i as leaf

descendants. This is because C has a directed path to K and the endpoint of K.tree_edge 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 $K \in \mathscr{K}$, and adds all the incoming edges of internalNode to C.off_edges except the edge incident to K.tree_edge. This step is shown in lines 21-28 of Algorithm 4.3. Therefore, after line 28 executes, all clusters $C \supseteq S_i$ will have a directed path to internalNode, and will have all the taxa in S_i as leaf descendants.

At this point, there are still some clusters that are disjoint from S_i but might be upstream in N from a cluster in $K \in \mathscr{K}$. This is a problem because it means that after lines 21-28 of Algorithm 4.3 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 $K \in \mathscr{K}$, 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 $\mathscr{K}$, 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 of the clusters 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 Algorithm Properties

It is clear from the pseudocode of Algorithm 4.1 that it generates a DAG using the input data. Therefore, we first prove:

Claim 4.2: The NETWORK.BUILD procedure produces a valid reticulation network that represents the input clusters.

Proof: Deferred to the Appendix, which can be found on the Computer Society Digital Library. □

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 is a reticulation node, meaning it has in-degree ≥ 2. It turns out that there are some valid maximal ST-set tree sequences that produce internal nodes with in-degree 1. An example of such a sequence is shown in Figure 8, which can be found on the Computer Society Digital Library.

Lemma 4.1: If NETWORK.BUILD creates a network N with $m < p$ reticulation nodes when executed on a maximal ST-set tree sequence $\mathscr{S}$ of length p, then there exists a maximal ST-set tree sequence $\mathscr{S}'$ of length m.

Proof: Deferred to the Appendix which can be found on the Computer Society Digital Library. □

ret-minimum with respect to $\mathscr{C}$ if it has the fewest number of reticulation nodes over all the networks that represent $\mathscr{C}$. To complete the proof of Claim 4.1, we show that

Lemma 4.2: The NETWORK.BUILD algorithm constructs a ret-minimum network for $\mathscr{C}$ when given as input a shortest maximal ST-set tree sequence.

The proof of Lemma 4.2 uses an argument that relies on pruning the subtrees of a network until the network is a tree. This argument appears in [10] in the proof of Lemma 7.

Proof: Let N be the network constructed by NETWORK.BUILD for a set of clusters $\mathscr{C}$ on $\mathscr{S}$, a shortest maximal ST-set tree sequence for $\mathscr{C}$, of length p. Suppose toward a contradiction that there exists a network N' with $m < p$ reticulation nodes that is ret-minimum for $\mathscr{C}$. 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 $\mathscr{C}$. Therefore, 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 let G_1 be the subgraph composed of r_1, 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. Repeat until N has no remaining reticulation nodes. We argue that the sequence $\mathscr{S} = \{\Sigma_m, \ldots, \Sigma_1\}$ is a maximal ST-set tree sequence; specifically

1) The graph $N - G_1 - \ldots - G_m$ is a tree.
2) Each $\Sigma_j \in \mathscr{S}$ is an ST-set of $\mathscr{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$, which means Σ_j is compatible with all clusters $C \in \mathscr{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$ and all clusters $C \subset \Sigma_j$ are pairwise compatible.
3) Each $\Sigma_j \in \mathscr{S}$ 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.

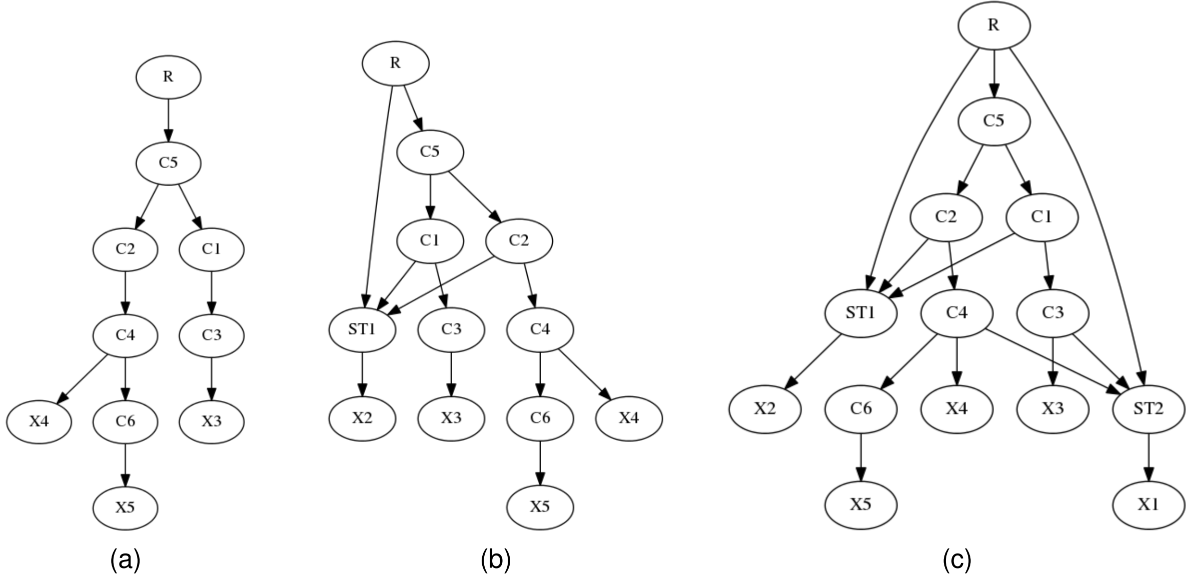


Fig. 2. The intermediate states of the NETWORK.BUILD procedure when building the network in Figure 1 to represent $\mathscr{C} = \{\{1,2,3\}, \{1,2,4,5\}, \{1,3\}, \{1,4,5\}, \{3,4,5\}, \{5\}\}$ with input maximal ST-set sequence $\mathscr{S} = \{\{1\}, \{2\}\}$. (a) Tree constructed using TREE.BUILD to represent $\widetilde{\mathscr{C}} = \mathscr{C} \mid I$ where $I = \{3,4,5\}$. The C_5 node is highest in the tree because $C_5 \mid I$ has the largest size of all $C \in \widetilde{\mathscr{C}}$. (b) Result of calling NETWORK.ADD on the last ST-set $\{2\}$ of $\mathscr{S}$ and the tree from Fig 4.1(a). The procedure adds node X2 as a leaf child of the new `internalNode` labeled ST1. The set $\mathscr{K}$ of most downstream clusters containing $\{2\}$ is just $\{C_1\}$. There is an edge from the root R to ST1, because otherwise the switching for C_5 would be disconnected from the network since C_5 does not contain 2. This edge is added on lines 34-35 of NETWORK.ADD. (c) Result of calling NETWORK.ADD on the ST-set $\{1\}$ of $\mathscr{S}$ and the network from Fig. 4.1. (a) NETWORK.ADD creates new internal node ST2 with leaf child X1 and adds edges to ST2 from the tree edges of C_2 and C_3 because the clusters C_1, C_2, C_3, C_4 all contain $\{1\}$, which means they are the contents of the `Supersets` structure in this iteration. $\mathscr{K} = \{C_3, C_4\}$ because C_1, C_2 are upstream from C_4, C_3, respectively so they obtain $\{1\}$ automatically from the clusters in $\mathscr{K}$. C_5 is in the `Disjoint` structure because it does not contain $\{1\}$, so there is another edge added on lines 34-35 of NETWORK.ADD from the root to ST2, as in Fig. 4.1(b), such that the switching representing C_5 is connected and contains no paths through ST2.

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 $\Sigma_j \in \mathscr{S}$ is not compatible with $\widetilde{\mathscr{C}_j} = \mathscr{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$, which means there is some cluster $C \in \widetilde{\mathscr{C}_j}$ 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 $\mathscr{C}$, 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 $\widetilde{\mathscr{C}_j}$ and so property (2) holds.

It remains to show that $\mathscr{S}$ is a maximal ST-set *tree* sequence. Suppose that Σ_j is not a maximal ST-set of $\mathscr{C} \setminus (\Sigma_1 \cup \ldots \Sigma_{j-1})$ 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 $\mathscr{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1})$, 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 been incompatible, since C would contain elements not in Σ_j and Σ contains x. Let $\mathscr{K}$ be the set of clusters $C \supset \Sigma$. The tree edges of the clusters in $\mathscr{K}$ must lie on several different paths from the root of N to r_j, because N is assumed to be ret-minimum for $\mathscr{C}$, so it cannot have unnecessary reticulation nodes. If the clusters in $\mathscr{K}$ 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 $\mathscr{K}$ do not lie on a single path, but all the clusters in $\mathscr{K}$ contain $x \notin \Sigma_j$, there must be another reticulation node r' such the tree edge endpoint of each cluster in $\mathscr{K}$ has a path to r, and the subtree S of r contains x. However, this means that all the clusters in $\mathscr{K}$ contain $\Sigma_j \cup S$, which means that $\Sigma_j \cup S$ is compatible with $\mathscr{K}$. This means that we can modify N' to create a network N'' with one fewer reticulation nodes by removing the subtree of r and its incoming edges from N and then calling TREE.BUILD

Algorithm 5.1 Dynamic programming approach by [8] that computes that History Bound, as given in [13]. Procedure CLEAN is given in Algorithm 2.1.

```
1:  procedure HB_DP(M)
2:      for k = 2, ..., n do
3:          for each subset K of k rows of M do
4:              set M_K = submatrix of M with rows in K
5:              set M~ = CLEAN(M_K)
6:              set K~ = set of rows of M~
7:
8:              set H[M_K] = min_{r ∈ M~} (1 + H[M~_{K~ - r}])
9:      return H[M]
10: end procedure
```

on cluster set $\mathscr{C} \mid (\Sigma_j \cup S)$ with input root r_j. However, this is a contradiction since N' is ret-minimum for $\mathscr{C}$; 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 $\mathscr{C}$, then we can construct maximal a ST-set tree sequence $\mathscr{S}'$ of length m. However, this is a contradiction, since we claimed that the sequence $\mathscr{S}$ of length p that we passed as input to the NETWORK.BUILD algorithm was already a shortest maximal ST-set tree sequence for $\mathscr{C}$. Therefore NETWORK.BUILD produces a ret-minimum network when executed on a shortest maximal ST-set tree sequence for $\mathscr{C}$. □

In summary, the proof of Lemma 4.2 concludes the proof of Claim 4.1. This resolves the open problem posed in [11], formulating the value computed by the History Bound algorithm as a statement about network structure.

5 Searching ST-set Sequence Space

In addition to the original method of computing the History Bound by branching over all possible executions of CHB, there is a dynamic programming solution that computes the History Bound more efficiently [8]. We give the pseudocode for the dynamic program from [13] in Algorithm 5.1. CLEAN refers to the procedure given in Algorithm 2.1. The dynamic programming method computes values of the History Bound for successively larger subsets of the taxa in the input matrix M by minimizing over previously-computed values for smaller subsets.

We also know from [8] that the problem of computing the History Bound is NP-hard and APX-hard, so we do not expect to find a polynomial-time algorithm. However, dynamic programming solutions often depend only on the optimal substructure property, instead of exploiting other structural insights. A common downside of dynamic programming is pessimism: algorithms using this technique solve the problem for *all* smaller problem instances, including those that may not be subproblems of the given one. To circumvent these inefficiencies, dynamic programs are often transformed into top-down algorithms with memoization, which cache intermediate results for look-up when they are encountered again.

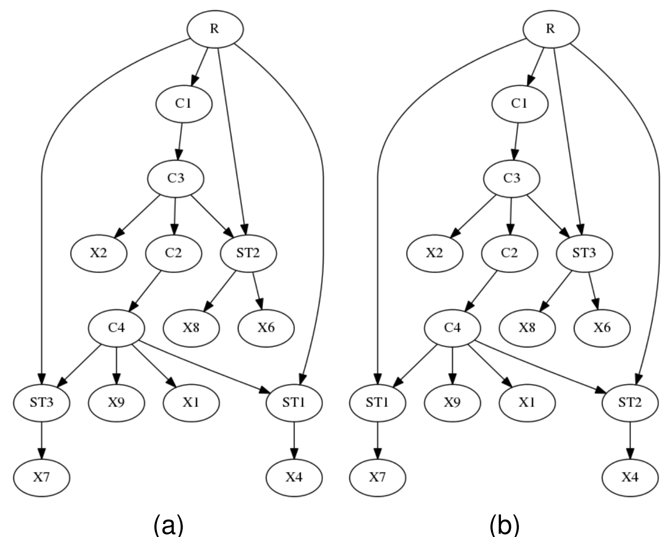


Fig. 3. A pair of isomorphic networks created using NETWORK.BUILD on the input clusters from Figure 8. Note that the only difference in the two figures are the labels on the reticulation nodes. (a) Network created using ST-set sequence $\{\{4\}, \{6, 8\}, \{7\}\}$. (b) Network created using ST-set sequence $\{\{7\}, \{4\}, \{6, 8\}\}$. These maximal ST-set sequences both have $I = \{1, 2, 9\}$ and thus are permutations of each other by Theorem 5.1.

In this section, we develop a top-down method to compute the History Bound whose efficiency will come from the structural insights of previous sections. We will identify equivalent subproblems and avoid examining them more than once. This allows us to compute the History Bound more efficiently on data generated by a standard coalescent approach with the infinite sites assumption.

More specifically, we observe that the concept of ST-set sequences iteratively *removes* taxa from X at each step. Furthermore, the NETWORK.ADD procedure adds ST-sets into the network independently of their position in the sequence. Experimenting with this property reveals that certain networks produced by NETWORK.BUILD are isomorphic. In the following section, we give an equivalence relation for maximal ST-set sequences that cause NETWORK.BUILD to produce isomorphic networks. Then, in Section 5.2, we leverage this search space reduction to give a top-down algorithm for computing the History Bound that performs better in the "typical" case than Algorithm 5.1.

5.1 Equivalence of Maximal ST-set Sequences

There are distinct maximal ST-set sequences that produce isomorphic networks when passed as input to NETWORK.BUILD. An example of two such sequences is shown in Figure 3. In Theorem 5.1, we give the criteria that cause NETWORK.BUILD to create isomorphic networks from two distinct maximal ST-sequences. First, we prove the following technical lemma.

Lemma 5.1: Let S be an ST-set of a set of clusters $\mathscr{C}$. Then S is an ST-set of $\widetilde{\mathscr{C}_k} = \mathscr{C} \setminus (S_1 \cup \ldots \cup S_k)$ where $\mathscr{S} = \{S_1, \ldots S_k\}$ is a maximal ST-set sequence of $\mathscr{C}$ and no $S_i \in \mathscr{S}$ contains any elements of S.

Proof: Deferred to the Appendix, which can be found on the Computer Society Digital Library. □

Note that S will not necessarily be maximal in $\widetilde{\mathscr{C}_k}$ even if it is maximal in $\mathscr{C}$.

Theorem 5.1: Let $\mathscr{S} = \{S_1, \ldots, S_p\}$ and $\Sigma = \{\Sigma_1, \ldots, \Sigma_q\}$ be two maximal ST-set sequences that have the same initial set; that is, if

$$I = X - \bigcup_{1 \le i \le p} S_i = X - \bigcup_{1 \le i \le q} \Sigma_i = I'$$

then $p = q$ and the sequences $\mathscr{S}$ and Σ are permutations of each other.

Proof: Let S_i be a maximal ST-set in sequence $\mathscr{S}$ and let Σ_j be the first ST-set in Σ to contain any elements of S_i. By Lemma 5.1, we know that S_i is an ST-set of

$$\widetilde{\mathscr{C}_j} = \mathscr{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{j-1}),$$

but since Σ is a maximal ST-set sequence, Σ_j is a *maximal* ST-set of $\widetilde{\mathscr{C}_j}$, which means that that $S_i \subseteq \Sigma_j$. In order to show that Σ is a permutation of $\mathscr{S}$, we must show that Σ_j cannot contain any other elements.

Suppose toward a contradiction that there is at least one $x \in \Sigma_j$ that isn't in S_i. Then x must belong to some set of clusters of $\widetilde{\mathscr{C}_j}$ that are entirely contained in Σ_j (because Σ_j is compatible with $\widetilde{\mathscr{C}_j}$) but are not contained in S_i. Let $Q \neq \emptyset$ be the union of those clusters. We can write $\Sigma_j = S_i \cup Q$, but we emphasize that that $Q \neq \emptyset$ contains all the elements of S_j that are not in S_i, meaning that Q and S_i are disjoint.

We argue that the entirety of Q must appear in some maximal ST-set of $\mathscr{S}$. Suppose that this is not the case and there are ST-sets $S_{k_1} \supseteq Q_1$ and $S_{k_2} \supseteq Q_2$ in $\mathscr{S}$ such that $Q = Q_1 \cup Q_2$, where Q_1 and Q_2 are disjoint. That is, each of these ST-sets contains a fragment of Q but not its entirety. Without loss of generality, if $k_1 < k_2$, meaning S_{k_1} occurs earlier in $\mathscr{S}$, then S_{k_1} is not maximal because Q is a union of compatible clusters, so if S_{k_1} containing Q_1 is an ST-set, then $S_{k_1} \cup Q_2$ is a larger ST-set and thus contradicts the assumption that S_{k_1} is maximal. Therefore, there must be some $S_k \in \mathscr{S}$ that contains Q.

We argue that S_k must occur before S_i in sequence $\mathscr{S}$ (meaning $k < i$), because S_i is maximal for

$$\widetilde{\mathscr{C}_i} = \mathscr{C} \setminus (S_1 \cup \ldots \cup S_{i-1})$$

and Q is disjoint from S_i, which means that all clusters $C \subseteq Q$ are compatible with S_i. If all of these elements were still present in $\widetilde{\mathscr{C}_i}$, then S_i would not be maximal because $S_i \cup Q$ would be a larger ST-set of $\widetilde{\mathscr{C}_i}$; therefore, S_k occurs before S_i in $\mathscr{S}$.

Let Σ_ℓ be the first ST-set of Σ to contain any elements of S_k. By Lemma 5.1, S_k is an ST-set of

$$\widetilde{\mathscr{C}_\ell} = \mathscr{C} \setminus (\Sigma_1 \cup \ldots \cup \Sigma_{\ell-1}).$$

Suppose $\ell < j$. This means that Σ_ℓ contains elements of $S_k - q$ but does not contain any element of q, so they must be contained in an ST-set Σ_j that occurs later in the sequence Σ. However, Σ_ℓ cannot contain the elements of $S_k - Q$ without also containing all of Q by maximality, because the elements of Q are still present in $\widetilde{\mathscr{C}_\ell}$. Therefore, ℓ cannot come before j.

So, suppose $\ell > j$ and consider $\widetilde{\mathscr{C}_j}$. Since Σ_ℓ is the first element of Σ to contain any elements of S_k and $\ell > j$, S_k is still an ST-set of $\widetilde{\mathscr{C}_j}$. Therefore, $\widetilde{\mathscr{C}_j}$ has two ST-sets Σ_j and S_k that both contain Q, so by Lemma 4 of [10] $\Sigma_j \cup S_k$ is also an ST-set. This contradicts the assumption that $\Sigma_j = S_i \cup Q$ with $Q \neq \emptyset$ is not maximal, so it must be the case that $Q = \emptyset$ and $S_i = \Sigma_j$. And since S_i was an arbitrary ST-set of $\mathscr{S}$, then it holds for any ST-set; therefore, $\mathscr{S}$ and Σ are permutations. □

Note that Theorem 5.1 does not imply that the elements of any given maximal ST-set sequence $\mathscr{S}$ can be permuted arbitrarily to create another equivalent maximal ST-set sequence. This is because some S_i may be a maximal ST-set of $\widetilde{\mathscr{C}_i}$ but still an incompatible set of $\widetilde{\mathscr{C}_{j<i}}$, because certain clusters are still incompatible in $\widetilde{\mathscr{C}_j}$. The result merely states that if two maximal ST-set sequences with the same initial set *do* exist, then they are permutations.

Theorem 5.1 demonstrates that the maximality criterion on the tree-like components of reticulation networks is very strong, meaning that when the compatible set I of two sequences $\mathscr{S}$ and Σ is the same, maximality forces the elements not in I to be "packaged" together into the same maximal ST-sets. These two factors make the networks produced by NETWORK.BUILD isomorphic: the algorithm first constructs the perfect phylogeny for I, which is the same for both sequences, then it calls NETWORK.ADD on each of the ST-sets in the sequence *independently*. Theorem 5.1 shows that the ST-sets of $\mathscr{S}$ and Σ are the same, so NETWORK.ADD inserts each ST-set into the network without making any decisions based on previous NETWORK.ADD calls, or the structure of the network. Therefore, only the node labels differ between the networks produced from two maximal ST-set tree sequences that are permutations of each other.

Corollary 5.1: If $\mathscr{S}$ and Σ are two maximal ST-set sequences over X then the following are equivalent:

(a) $\mathscr{S}$ and Σ are permutations of each other.
(b) $X - \bigcup_{S \in \mathscr{S}} S = X - \bigcup_{\sigma \in \Sigma} \sigma$.

Proof: Since $\mathscr{S}$ and Σ are collections of the same sets, just ordered differently, their unions are equal, so (a) implies (b). Theorem 5.1 gives that (b) implies (a). □

Theorem 5.1 implies that certain ST-set sequences are equivalent, because they consist of the same maximal ST-sets and cause NETWORK.BUILD to produce structurally identical reticulation networks. Therefore, if two maximal ST-set sequences $\mathscr{S}$ and Σ over X are permutations of each other, then we define them to be *p-equivalent*. In the context of computing the History Bound, the p-equivalence relation reduces the search space of maximal

ST-set tree sequences, since, if an algorithm examines a maximal ST-set tree sequence $\mathscr{S}$ of length p, then all other sequences that are p-equivalent to $\mathscr{S}$ will be permutations of $\mathscr{S}$ with length p and do not need to be examined. We will use this idea in the development of a top-down History Bound algorithm.

5.2 A Top-Down History Bound Algorithm

In this section, we develop a top-down algorithm that computes the History Bound. This method also enumerates all of the shortest maximal ST-set tree sequences that are distinct up to p-equivalence for a matrix M.

To use Theorem 5.1, we first define an order on the maximal ST-sets of a set of clusters $\mathscr{C}$ over X as follows:

$$S_1 \prec S_2 \quad \text{if} \quad \begin{array}{l} |S_1| < |S_2|. \\ |S_1| = |S_2| \text{ and } \min\limits_{x \in S_1} x < \min\limits_{y \in S_2} y. \end{array} \tag{1}$$

This is a total order, because the maximal ST-sets of $\mathscr{C}$ partition X [10], so the minimum elements of two maximal ST-sets of $\mathscr{C}$ will be distinct. We use this ordering to restrict the search to only examine maximal ST-set sequences $\mathscr{S}$ whose ST-sets are in increasing order: that is, those sequences $\mathscr{S}$ such that if $S_i, S_j \in \mathscr{S}$ and $i < j$ then $S_i \prec S_j$. In other words, the sequences whose ST-sets are increasing with respect to $\prec$ are taken as representatives of their p-equivalence classes.

The pseudocode for the top-down algorithm is shown in Algorithm 5.2. Given a matrix M with n rows, HB_TopDown first initializes a queue with the input matrix M, and initial level $\ell = 0$. It also initializes the variable p with an overestimate for the length of the shortest maximal ST-set tree sequence. On line 2, it sets $p = n-2$, because all size-2 subsets of X are compatible, so the shortest maximal ST-set tree sequence will have length $n - 2$ in the worst case. The procedure then uses the queue to process subproblems in a breadth-first order, growing each known maximal ST-set sequence by one maximal ST-set, and using the level variable ℓ to keep track of the length of all the currently computed ST-set sequences.

At each iteration, HB_TopDown extracts the triple $(\ell, \widetilde{M}, S)$ from the queue, where $\widetilde{M}$ is the subproblem matrix, ℓ is the current level, and S is the maximal ST-set S whose removal produced $\widetilde{M}$. The procedure then runs Clean on $\widetilde{M}$: if the resulting matrix is empty, then $\widetilde{M}$ is compatible, meaning the end of a maximal ST-set tree sequence has been reached. It must be a *shortest* maximal ST-set tree sequence, because HB_TopDown examines sequences of increasing lengths starting from $\ell = 0$. Further, $(\ell, \widetilde{M}, S)$ is also the endpoint of the *first* shortest maximal ST-set sequence to be encountered by HB_TopDown if the variable p is still an overestimate. If this is the case, HB_TopDown updates p with the length of the sequence ending with S, which is the value of ℓ for this subproblem. As given in Algorithm 5.2, the HB_TopDown inspects the remaining items in the queue that also have level $\ell = p$ before terminating; however, it can easily be modified to terminate immediately after it finds the first shortest maximal ST-set tree sequence.

Algorithm 5.2 Top-down algorithm that computes the History Bound for binary matrix M. It can be modified to terminate after finding the first shortest sequence. Procedure Clean is given in Algorithm 2.1.

1: **procedure** HB_TopDown(M)
2: $p = |M| - 2$
3: Q = `Queue()`
4: Q`.put((0,` M`,` $\emptyset$`))`
5: **while** Q is not empty **do**
6: (ℓ, $\widetilde{M}$, S) = Q`.get()`
7: **if** $\ell > p$ **then**
8: break
9: **if** $\widetilde{M}$ is compatible **then**
10: **if** p == $|M|$ **then**
11: set $p = \ell$
12: **if** ℓ == p **then**
13: $\widetilde{M}$ is the endpoint of an optimal sequence
14: **if** $p \geq \ell$ **then**
15: $\widetilde{M}$ = Clean($\widetilde{M}$)
16: **for** $i \in \{1, \ldots, |\widetilde{M}|\}$ **do**
17: S_i = the ST-set encoded by row r_i of $\widetilde{M}$
18: **if** $S \prec S_i$ **then**
19: Q`.put(`$\ell + 1$, $\widetilde{M} - r_i$, S_i`)`
20: **return** p
21: **end procedure**

Meanwhile, if HB_TopDown extracts triple $(\ell, \widetilde{M}, S)$ from the queue and $\widetilde{M}$ is not compatible, then, for every ST-set S_i encoded by row $r_i \in M$ such that $S \prec S_i$, the triple $(\ell + 1, \widetilde{M} - r_i, S_i)$ is inserted into the queue. Checking ST-sets against the $\prec$ ordering ensures that only the representative maximal ST-set sequence in each p-equivalence class is considered during the computation. Note that $\prec$ is designed so that ST-sets with larger cardinality follow smaller ones. As taxa are removed from M by HB_TopDown, existing ST-sets combine to create new ones that have not been considered in previous execution steps. So, they come later in the $\prec$ order than those with lower cardinality so that they will be chosen for removal in subsequent subproblems.

5.3 Comparing Algorithm Performance

We can see from the pseudocode in Algorithm 5.1 that the dynamic programming procedure to compute the History Bound runs in time

$$\sum_{k=2}^{n} \binom{n}{k} (k + T_{\texttt{Clean}}(k, m)) \in O(nm2^n) \tag{2}$$

where $|X| = n$, $|\mathscr{C}| = m$ and $T_{\texttt{Clean}}(n, m)$ is the runtime of Clean on an $n \times m$ matrix. Expression (2) gives the runtime of the dynamic program *regardless* of the specific problem instance. We can read expression (2) from the pseudocode without knowing anything about the internal structure of the maximal ST-sets of the clusters encoded by M. This is because HB_DP looks

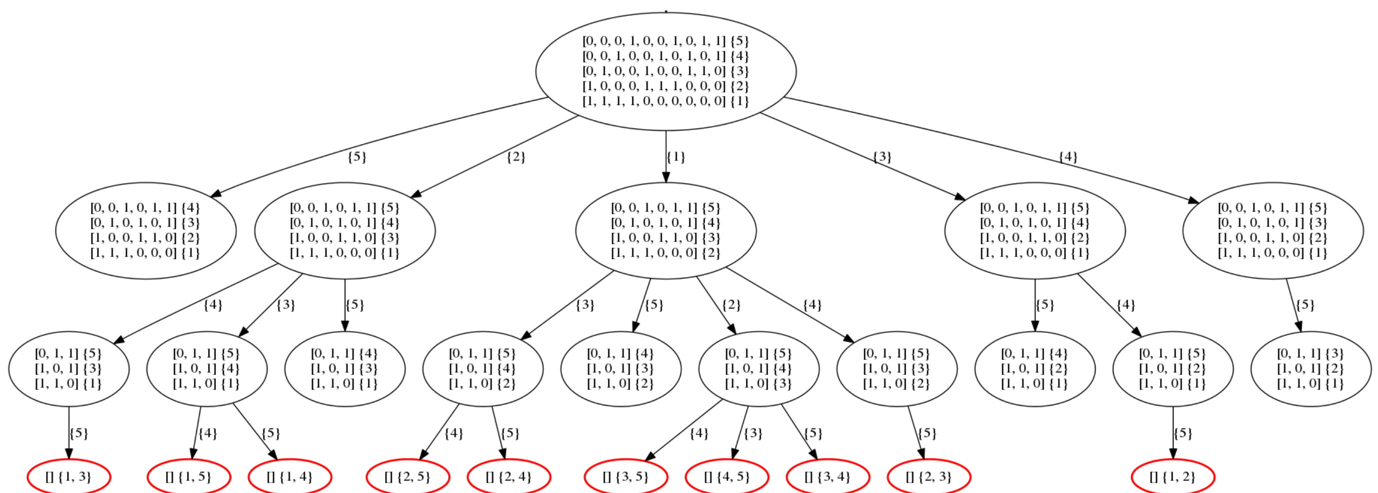


Fig. 4. Execution of HB_TopDown illustrated as a search tree. The input set $\mathscr{C}$ are all size-2 subsets of $X = \{1, \ldots, 5\}$. The nodes are labeled with the matrix $\widetilde{M}$ that gets processed at that step and the edges are labeled with the maximal ST-set that is removed to obtain the child node. The edge labels along each path from the root to a leaf are increasing according to the $\prec$ ordering. Note that, for this problem instance, the number of nodes at each level ℓ of the tree is equal to $\binom{n}{\ell}$.

at *all* $\binom{n}{k}$ subsets of the rows of the input matrix M for all subset sizes $k \leq n-2$, even though not all subsets of taxa may be removed to construct a valid maximal ST-set sequence, not to mention a shortest one, for many inputs M.

This raises the following question: for which problem instances is it *necessary* for either algorithm to examine all size > 2 subsets of X to compute all of the shortest maximal ST-set tree sequences for M? We give one such instance in Figure 4, which shows the execution of HB_TopDown illustrated as a search tree. The cluster set for this example is all size-2 subsets of X. For this data, the number of matrices processed at each level ℓ is equal to $\binom{n}{\ell}$. This occurs because, at every step of the execution of HB_TopDown, a maximal ST-set with cardinality 1 is removed, but the removal never causes rule **Dr** to trigger in procedure Clean, so all subsequent maximal ST-sets have cardinality 1 as well. Thus, each level ℓ of the search tree has $\binom{n}{\ell}$ nodes, and execution continues until ℓ reaches $n-2$, when X only has two taxa left and every set of size 2 is compatible. This means that all maximal ST-set tree sequences of length $n-2$ are optimal for this problem instance. Therefore, those subproblems that HB_DP examines must *all* be looked at by HB_TopDown in the extreme case where it is impossible to trigger rule **Dr** by removing fewer than $n-2$ taxa. Thus, Fig. 4 shows that the worst-case runtime of HB_TopDown is also given by expression (2).

However, in more typical situations, HB_TopDown out-performs HB_DP, as it is able to examine fewer subproblems before reaching the answer. In the HB_TopDown search tree, the number of nodes at each level ℓ is equal to the number of ways to remove ℓ maximal ST-sets from the data. In less extreme problem instances, removing taxa from X does trigger rule **Dr**, which identifies maximal ST-sets containing more than one taxon. These larger maximal ST-sets cause the paths leading to compatible subproblems to be shorter than $n-2$, because there are fewer ST-sets to remove. One such problem instance with $n = 6$ is shown in Figure 5 of the Appendix, which can be found on the Computer Society Digital Library. On this example, HB_TopDown only examines 23 subproblems, compared to the 57 that are inspected HB_DP, demonstrating an advantage in efficiency.

We also tested both History Bound algorithms on biological data.

1) We used the 43 polymorphic columns of Kreitman's 1983 data of the alcohol dehydrogenase locus from 11 chromosomes of *Drosophila melanogaster* that were transformed into binary sequences and studied in [14]. The value of the History Bound for this data is 3, with two shortest maximal ST-set tree sequences, distinct up to p-equivalence. HB_TopDown examined 127 nodes while HB_DP looked at 502[1].
2) On the data from double-stranded RNA in fungi studied in [15] with seven taxa and 229 sites, the History Bound is 3, with three shortest maximal ST-set tree sequences. HB_TopDown examined 63 nodes, which is about half of the 120 nodes examined by HB_DP. This result is particularly optimistic, because the construction in Fig. 4 achieves the worst-case condition with $\binom{n}{2} = 21$ sites, and yet all 229 sites from this biological sample do not conflict that strongly.

1. Running Clean on the input matrix collapses three rows into one, which is why the number of nodes examined by HB_DP is close to 2^9 and not 2^{11}.

Therefore, unlike the example data in Figure 5, which was contrived for illustrative purposes, these results give us reason to expect HB_TopDown to outperform HB_DP on on biological data typically used to compute the History Bound.

6 Conclusions

We explicitly proved that the History Bound from a matrix M is equal to the length of the shortest maximal ST-set sequence for the clusters encoded by M. Then, we proved that the History Bound counts the minimum number of reticulation nodes in a network that represents the clusters encoded by M in the softwired sense, resolving the open question from [11]. We then developed a top-down algorithm for computing the History Bound, and demonstrated that it has the same worst-case runtime as HB_DP [8], but outperforms HB_DP on some biological data.

7 Acknowledgements

J. Matsieva and D. Gusfield are supported by the Foundation NSF grant CCF-1017580. C. Whidden is a Simons Foundation Fellow of the Life Sciences Research Foundation and supported by the NSF grant DMS-1223057. This publication is contribution No 2015-243 of the Institut des Sciences de l'Evolution de Montpellier (UMR 5554 UM CNRS IRD).

References

[1] D. H. Huson, R. Rupp, and C. Scornavacca, *Phylogenetic networks: concepts, algorithms and applications*. Cambridge Univ Pr, 2011.

[2] R. C. Griffiths and P. Marjoram, "Ancestral inference from samples of DNA sequences with recombination," *JCB*, vol. 3, pp. 479–502, 1996.

[3] ——, "An ancestral recombination graph," in *Progress in Population Genetics and Human Evolution*, P. Donnelly and S. Tavare, Eds. IMA Volumes in Mathematics and Its Applications, vol. 87, 1997, pp. 257–270.

[4] M. Bordewich and C. Semple, "Computing the minimum number of hybridization events for a consistent evolutionary history," *Discrete Applied Mathematics*, vol. 155, no. 8, pp. 914 – 928, 2007.

[5] ——, "On the computational complexity of the rooted subtree prune and regraft distance." *Annals of Combinatorics*, vol. 8, no. 4, 2005.

[6] L. Wang, K. Zhang, and L. Zhang, "Perfect phylogenetic networks with recombination," *Journal of Computational Biology*, vol. 8, no. 1, pp. 69–78, 2001.

[7] S. R. Myers and R. C. Griffiths, "Bounds on the minimum number of recombination events in a sample history," *Genetics*, vol. 163, no. 1, pp. 375–394, 2003.

[8] V. Bafna and V. Bansal, "Inference about recombination from haplotype data: lower bounds and recombination hotspots," *Journal of Computational Biology*, vol. 13, no. 2, pp. 501–521, 2006.

[9] L. van Iersel and S. Kelk, "When two trees go to war," *Journal of Theoretical Biology*, vol. 269, no. 1, pp. 245 – 255, 2011.

[10] S. Kelk, C. Scornavacca, and L. van Iersel, "On the elusiveness of clusters." *IEEE/ACM Trans. Comput. Biology Bioinform.*, vol. 9, no. 2, pp. 517–534, 2012.

[11] Y. Wu and D. Gusfield, "A new recombination lower bound and the minimum perfect phylogenetic forest problem." *Proceedings of the 13th Annual International Conference on Combinatorics and Computing*, vol. 4598, pp. 16–26, 2007.

[12] J. Wakeley, *Coalescent Theory: An Introduction*. Roberts & Company Publishers, 2009.

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

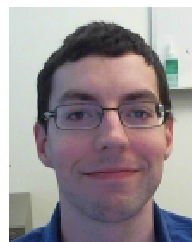
[14] Y. Song and J. Hein, "Parsimonious reconstruction of sequence evolution and haplotype blocks," in *Algorithms in Bioinformatics*, ser. Lecture Notes in Computer Science, G. Benson and R. Page, Eds. Springer Berlin Heidelberg, 2003, vol. 2812, pp. 287–302.

[15] N. D. Charlton, I. Carbone, S. M. Tavantzis, and M. A. Cubeta, "Phylogenetic relatedness of the m2 double-stranded rna in rhizoctonia fungi," *Mycologia*, vol. 100, no. 4, pp. 555–564, 2008.

Julia Matsieva Julia Matsieva obtained a joint degree in mathematics and computer science from Harvey Mudd College in Claremont, California in 2011 and earned her M.S. from UC Davis in 2014. She is currently a Ph.D. student at UC Davis, working on problems related to phylogenetic networks and computational biology.

Steven Kelk Steven Kelk is an Assistant Professor at the Department of Knowledge Engineering (DKE) at Maastricht University in the Netherlands. Between 2004 and 2010, he was a postdoc at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam. He obtained his PhD in late 2003 from the University of Warwick. His main research interests are phylogenetics, graph theory, fixed parameter tractability, approximation algorithms and combinatorial optimization within computational biology.

Celine Scornavacca Celine Scornavacca graduated in mathematical engineering at University of Roma 2 in 2006 and received her Ph.D. in Computer Science from Montpellier University in 2009. After a postdoctoral position at the University of Tbingen, she joined the Institut des Sciences de l'Evolution de Montpellier (ISE-M) as research associate (CNRS) in October 2011. Her research interests are parameterized complexity, supertree and network methods for phylogenetics, combinatorics and bioinformatics.



Chris Whidden Chris Whidden received his Ph.D. degree in computer science from Dalhouse University, Halifax, Canada in 2013. He is currently a postdoctoral research fellow at the Fred Hutchinson Cancer Research Center, Seattle, WA. His research interests include developing efficient algorithms and software for solving NP-hard problems, with a particular focus on inferring and comparing phylogenetic trees and networks.

Dan Gusfield Dan Gusfield received his Ph.D. in 1980 from UC Berkeley, working with Richard Karp, and was an Assistant Professor at Yale University from 1980 to 1986, and has been on the faculty of UC Davis since then. He is the co-author of the book "The Stable Marriage Problem: Structure and Algorithms" (MIT press, 1988), with Rob Irving; the author of "Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology" (Cambridge Press, 1997); and of "ReCombinatorics: The Algorithmics of Ancestral Recombination Graphs and Explicit Phylogenetic Networks", (MIT Press, 2014).

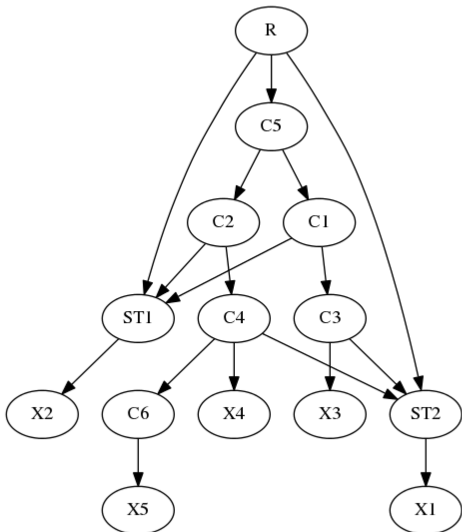


Fig. 6. A reticulation network that represents the cluster set $\mathscr{C}$ in the softwired sense from the example in Figure 1. It has 2 reticulation nodes and reticulation number 4.

APPENDIX

Claim A.1 (4.2): The NETWORK.BUILD procedure produces a valid reticulation network that represents the input clusters.

Proof of Claim 4.2: We prove the claim by induction on iterations of NETWORK.BUILD. The set of clusters $\mathscr{C} \mid I$ is correctly represented by N by the correctness of the known TREE.BUILD algorithm. Suppose that, when NETWORK.ADD is called on inputs N and S_i, the network correctly represents the set of clusters $\mathscr{C} \mid (I \cup S_p \cup S_{p-1} \cup \ldots \cup S_{i+1})$. We must show that, after iteration i, N correctly represents every cluster in

$$\widetilde{\mathscr{C}_i} = \mathscr{C} \mid (I \cup S_p \cup S_{p-1} \cup \ldots \cup S_i).$$

We prove the claim by showing the following two conditions.

First, we need to show that after NETWORK.ADD finishes processing ST-set S_i, cluster $C \in \widetilde{\mathscr{C}_i}$ 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 $C \in \mathscr{K}$ then the algorithm will add an edge from the endpoint of C.`tree_edge` to the `internalNode`, so x will be a leaf descendant of C.`tree_edge`. If the algorithm instead attaches `internalNode` to the tree edges of a set of clusters $\mathscr{K}$ such that $C \in \mathscr{K}$, then by lines 9-17 there is some other cluster $C \in \mathscr{K}$ 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`. On line 32, the algorithm adds the edges into the `internalNode` to Ds set of `off_edges`. This is a book-keeping step to illustrate that N still displays the tree that represents $\widetilde{\mathscr{C}_i}$. Finally, it is possible that all of the edges into the `internalNode` are downstream of C. This would mean that the switching of N that represents $\widetilde{\mathscr{C}_i}$ will be disconnected. Therefore, on line 34, NETWORK.ADD adds 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 $\widetilde{\mathscr{C}_i}$. □

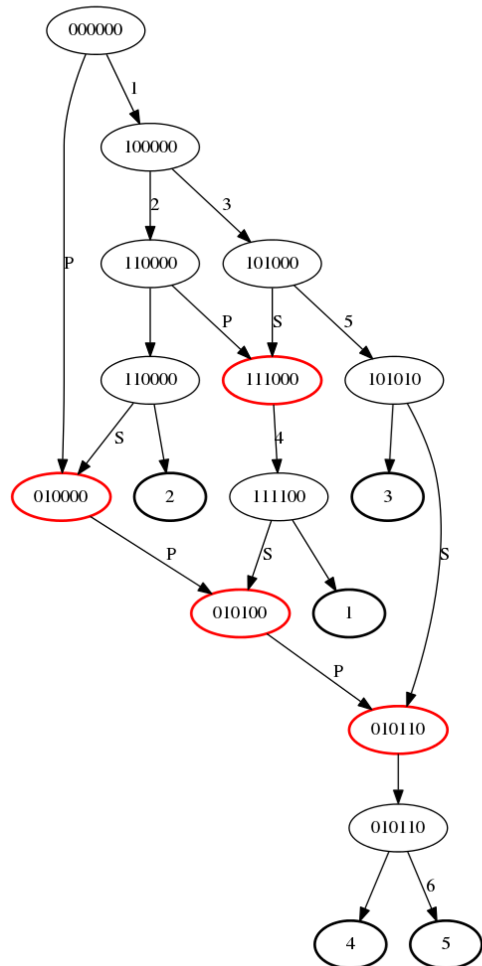


Fig. 7. A minARG for the data in Figure 1 has 4 recombination nodes.

Lemma A.1 (4.1): If NETWORK.BUILD creates a network N with $m < p$ reticulation nodes when executed on a maximal ST-set tree sequence $\mathscr{S}$ of length p, then there exists a maximal ST-set tree sequence $\mathscr{S}'$ of length m.

Proof of Lemma 4.1: 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 $\mathscr{K}$, 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

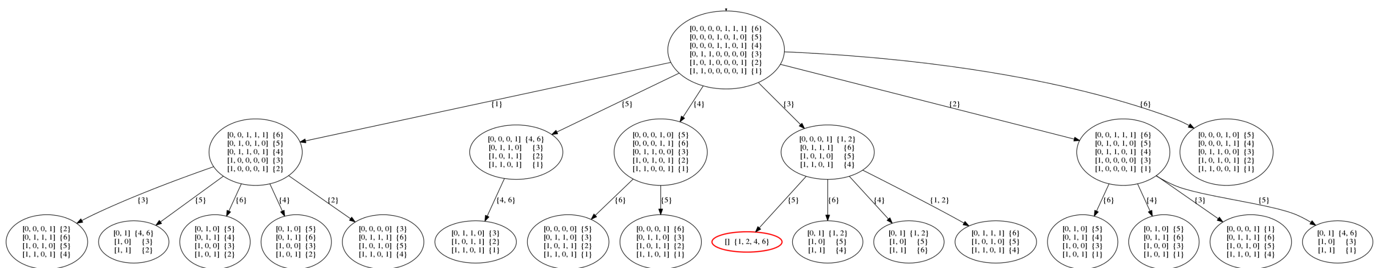


Fig. 5. Execution of HB_TopDown on inputs $X = \{1, \ldots, 6\}$ and $\mathscr{C} = \{\{1,2\}, \{1,3\}, \{2,3\}, \{4,5\}, \{4,6\}, \{5,6\}, \{1,2,4,5\}\}$. This figure may be too small to read closely; it is intended to illustrate search tree structure. $\mathscr{C}$ has only one shortest maximal ST-set tree sequence of length 2 with the endpoint shown in red. It shows rule **Dr** generating maximal ST-sets of size > 2, which allows HB_TopDown to find the shortest sequence faster than if it were starting the search from the bottom of the search tree of size $\approx 2^6$. This method examines 23 nodes, while HB_DP looks at 57 nodes when executed on $\mathscr{C}$.

of all clusters $C \supseteq I$ 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, the set of clusters $C \mid (I \cup S_i)$ can be arranged in a tree. Therefore, all the clusters in $C \mid (I \cup S_i)$ 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 of length $p - 1$. Thus, if a sequence $\mathscr{S}$ of length p is the shortest maximal ST-set tree sequence for a set of clusters $\mathscr{C}$, then each `internalNode` will have in-degree at least two. Hence, Network.Build will produce a reticulation network N with exactly p reticulation nodes. □

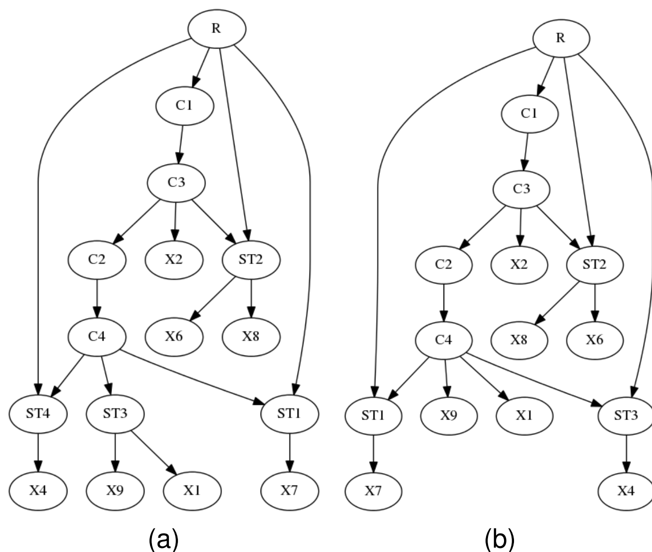


Fig. 8. Two almost-isomorphic networks produced by Network.Build on the input cluster set $\{1,9,2,4\}$, $\{1,9,2\}$, $\{1,9,2,6,8\}$, $\{1,9,4,7\}$. (a) Network produced using the ST-set sequence $\{\{4\}, \{1,9\}, \{6,8\}, \{7\}\}$. The node labeled ST3 has only one incoming edge. (b) Network produced using the ST-set sequence $\{\{4\}, \{6,8\}, \{7\}\}$.

Lemma A.2 (5.1): Let S be an ST-set of a set of clusters $\mathscr{C}$. Then S is an ST-set of $\widetilde{\mathscr{C}_k} = \mathscr{C} \setminus (S_1 \cup \ldots \cup S_k)$ where $\mathscr{S} = \{S_1, \ldots S_k\}$ is a maximal ST-set sequence of $\mathscr{C}$ and no $S_i \in \mathscr{S}$ contains any elements of S.

Proof of Lemma 5.1: In order to show that S is an ST-set of $\widetilde{\mathscr{C}_k}$, we must show that

1. S is compatible with all $C \in \widetilde{\mathscr{C}_k}$.
2. all clusters $C_1, C_2 \subseteq S$ are pairwise compatible.

We know that S is compatible with $\mathscr{C}$, which means that for all $C \in \mathscr{C}$, either $C \subseteq S$, $S \subseteq C$ or $S \cap C = \emptyset$. But since no elements of S are missing from $\widetilde{\mathscr{C}_k}$, then all the clusters $C \subseteq S$ are still in $\widetilde{\mathscr{C}_k}$, so they are still compatible with S. Similarly, the clusters corresponding to those that were disjoint from S in $\mathscr{C}$ contain the same or fewer elements in $\widetilde{\mathscr{C}_k}$, so they are still disjoint from S. Finally, we know that none of the $S_i \in \mathscr{S}$ contain any elements of S, which means that any $C \in \mathscr{C}$ such that $S \subseteq C$ will not have any elements of S missing in $\widetilde{\mathscr{C}_k}$. Therefore, for those clusters, it will still be true that $S \subseteq C$, so all of the compatibility conditions for S and $\widetilde{\mathscr{C}_k}$ are met, so condition (1) is true. Condition (2) is true as well because no elements of S are removed in $\mathscr{S}$ so all the clusters that are contained in S are still pair-wise compatible in $\widetilde{\mathscr{C}_k}$. Thus, S is an ST-set of $\widetilde{\mathscr{C}_k}$. □