A New Algorithm for the Reconstruction of Near-Perfect Binary Phylogenetic Trees

Kedar Dhamdhere[1,2]**, Srinath Sridhar**[1,2]**, Guy E. Blelloch** [2]**, Eran Halperin** [3]
R. Ravi [4] **and Russell Schwartz** [5]

March 17, 2005
CMU-CS-05-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In this paper, we consider the problem of reconstructing near-perfect phylogenetic trees using binary characters. A perfect phylogeny assumes that every character mutates at most once in the evolutionary tree. The algorithm for reconstructing a perfect phylogeny for binary characters is computationally efficient but impractical in most real settings. A near-perfect phylogeny relaxes this assumption by allowing characters to mutate a constant number of times. We show that if the number of additional mutations required by the near-perfect phylogeny is bounded by q, then we can reconstruct the optimal near-perfect phylogenetic tree in time $2^{O(q^2)}nm^2$ where n is the number of taxa and m is the number of characters. This is a significant improvement over the previous best result of $nm^{O(q)}2^{O(q^2r^2)}$ where r is the number of states per character (2 for binary). This improvement could lead to the first practical phylogenetic tree reconstruction algorithm that is both computationally feasible and biologically meaningful. We finally outline a method to improve the bound to $q^{O(q)}nm^2$.

[1]Equally contributing authors
[2]Email: {kedar, srinath, blelloch}@cs.cmu.edu, Computer Science Department, CMU
[3]Email: heran@icsi.berkeley.edu, ICSI, Berkeley
[4]Email: ravi@cmu.edu, Tepper School of Business, CMU
[5]Email: russells@andrew.cmu.edu, Dept of Biological Sciences, CMU
Supported in part by NSF grant CCR-0105548 and ITR grant CCR-0122581 (The ALADDIN project).

Keywords: Phylogenetic trees, Parsimony, Near-perfect phylogeny

1 Introduction

One of the classical problems of computational biology is to reconstruct an evolutionary tree for a set of taxa based on character data. Parsimony is one of the widely used metrics to solve the problem. It has been known to be particularly useful in the case when the evolutionary tree reconstructed is over a short period of time.

We borrow some of the following definitions and notations from [5]. The input to the problem is generally represented by a matrix I where rows R are strings of states corresponding to taxa. The columns $C = \{1, \cdots, m\}$ are referred to as characters. The set of states corresponding to any character c is denoted by $\mathcal{A}_c$, therefore every taxon $s \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$. In a phylogenetic tree, each vertex v corresponds to a taxon and has an associated label $l(v) \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$. We use the terms phylogeny, phylogenetic tree or just tree interchangeably.

Definition 1: A *phylogeny* for a set of n taxa R is a tree $T(V, E)$ with the following properties:

1. if a taxon $s \in R$ then $s \in l(V(T))$
2. for all $(u, v) \in E(T)$, $H(l(u), l(v)) = 1$ where H is the hamming distance

Definition 2: The *length of a phylogeny* T, $length(T) = |E(T)|$.
Definition 3: The penalty of a phylogeny T is defined as

$$penalty(T) = length(T) - \sum_{c \in C}(|A_c| - 1)$$

Minimizing the length of a phylogeny is the problem of finding the most parsimonious tree, a well known NP-complete problem [6]. A phylogenetic tree T constructed on input I, is called a perfect phylogeny if $penalty(T) = 0$. As summarized by [5], reconstructing a perfect phylogeny was proved to be NP-hard independently by Bodlaender et al. [2] and Steel [10]. This led researchers to work on either sophisticated heuristics (for e.g [7], [3]) or solve optimally for fixed parameter versions of the problem (for e.g, [1], [9]). Gusfield considered an important special case of the perfect phylogeny problem, when the number of states is bounded by 2. We call such a tree as a binary perfect phylogeny. Gusfield showed that binary perfect phylogenies can be reconstructed in linear time [8]. Lagergren and Fernandez-Baca, considered the problem of reconstructing near-perfect phylogenies [5]. The assumption of a 'near'-perfect phylogeny is that $penalty(T)$ is small for the most parsimonious tree. Their algorithm runs in time $nm^{O(q)}2^{O(q^2r^2)}$, where r is the number of states per character, q is the penalty, n is the number of taxa and m is the number of characters.

Our Results: In our work, we consider an important special case of the problem when $r = |\mathcal{A}_c| = 2$. The case when $r = 2$ is primarily important because Single Nucleotide Polymorphisms (SNPs) are bi-allelic. We can therefore use the algorithm for reconstructing trees where the taxa are DNA sequences and the characters are SNP markers. We show that if the penalty of the most parsimonious phylogeny is bounded by q, then we can reconstruct the phylogenetic tree in time $2^{O(q^2)}nm^2$. In section 4 we briefly describe how this time-bound can be improved to $q^{O(q)}nm^2$. More details on the improved algorithm will be available in [4]. Our algorithm is almost entirely self-contained and its understanding requires only some fundamental theorems on phylogenetic trees. Although some existential proofs are hard, the algorithm itself is not very complicated to implement. We also expect the algorithm to perform significantly better in practice than the worst case bounds.

1. create a conflict graph G s.t there exists a bijection $c : V(G) \mapsto C$ and $(u, v) \in E(G)$ iff $|G_{c(u),c(v)}| = 4$
2. find Q with $|Q| \leq 2q$ s.t Q is a set of columns corresponding to the vertex cover of G, return nil if none exists
   (a) for all trees T s.t there exists an onto function $\mu : E(T) \mapsto Q$
      i. for each vertex $v \in V(T)$
         A. find the subset of rows $b(v)$ 'compatible' to v
         B. build a perfect phylogeny T_v for the set $b(v)$
         C. replace v with tree T_v
         D. complete the tree by linking all the T_v's
      ii. if $cost(T) < min$ then $T_f = T, min = cost(T_f)$
   (b) return T_f

Figure 1: Overview of the algorithm

2 Overview of the Algorithm

We first sketch the outline of the algorithm and then explain the details of each step in subsequent sections. Specifically, given an input I and a penalty q the algorithm finds most parsimonious phylogeny with penalty at most q or *nil* if none exists. The following definitions are useful:

Definition 1 *The set of gametes $G_{i,j}$ restricted to two characters i, j is defined as: $G_{i,j} = \{(k, l) | \exists r \in R, r[i] = k, r[j] = l\}$.*

Definition 2 *Two columns $i, j \in C$ are said to contain (all) four gametes when $|G_{i,j}| = 4$.*

A high level pseudo-code of the algorithm is given in 1. The following sections elaborate on each of the steps of this pseudo-code which we will refer to as the main pseudo-code.

Pre-processing It is well known that the most parsimonious phylogeny reconstructed is imperfect if and only if the input I contains the four-gamete property. Before running our algorithm, we pre-process the input as follows. We remove all the columns that that have only single state (i.e. they do not mutate at all). We then look for the pairs of input columns i, j for which $|G_{i,j}| = 2$. In such a case, both the columns contain the same information. We remove one of the two columns from the input. We show in the Appendix (Theorem 12) that this preprocessing steps does not alter the overall running time or the correctness.

At the end of pre-processing, we have the input I that satisfies $|G_{i,j}| \geq 3$ for all columns i, j.

Conflict graph G and the vertex cover-columns Q: A conflict graph G corresponding to input I is constructed as follows. Each vertex $v \in V(G)$ of the graph represents a character $c(v) \in I$. An edge (u, v) is added if and only if all four gametes are present in $c(u)$ and $c(v)$. The function $\mu : E(T) \mapsto C$ maps mutation events to the character that mutates. Note that in any phylogeny if $\mu(u, v) = c$ then $l(u)[c] \neq l(v)[c]$.

To find the vertex cover, we use the classical 2-approximation algorithm. We set Q to be the set of columns corresponding to the vertex cover returned by the algorithm. We assert that there exists no q-near perfect phylogeny if $|Q| > 2q$.

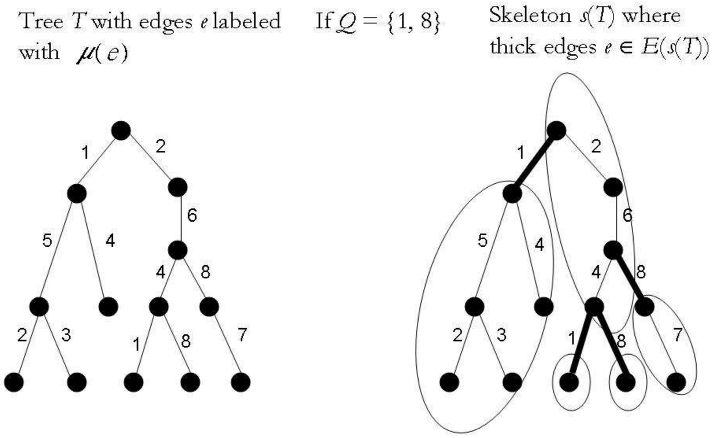

Figure 2: A phylogenetic tree and its associated skeleton. The bit vectors of the species are left out for simplicity.

Lemma 1 *There exists no q-near-perfect phylogeny if* $|Q| > 2q$.

Proof: First notice that for any pair of characters (i, j) s.t $|G_{i,j}| = 4$ the tree T should contain either two edges e, e' such that $\mu(e) = \mu(e') = i$ or $\mu(e) = \mu(e') = j$. If $|Q| > 2q$ then there exists no vertex cover of size q. This implies that there exists at least one edge $(u, v) \in E(G)$ s.t neither $c(u)$ nor $c(v)$ mutates multiple times in T, a contradiction. $\square$

For the step 2a of the pseudo-code, we enumerate by brute force all possible trees T that mutate the columns in Q at least once.

Definition 3 *A vertex* v *is bad w.r.t a pair of characters* (i, j) *and a set of columns* Q*, if* $i, j \notin Q$ *and* $(l(v)[i], l(v)[j]) \notin G_{i,j}$.

From now on we adhere to the following naming convention. The tree T (used in step 2a that mutates the columns in the vertex cover Q is referred to as the **skeleton tree**. The vertices of the a skeleton tree T are referred to as **super nodes**, since an entire tree replaces each of these nodes in a final phylogenetic tree. Figure 2 shows an example of a phylogenetic tree along with its skeleton.

3 Details

3.1 Finding a compatible set for all vertices

This is the most complicated step of the algorithm. We want to partition the rows of the input matrix such that each super node v is assigned a subset of 'compatible' rows $b(v)$. By arbitrarily rooting the tree T, the structure of the tree defines the bits for the characters in Q. We can think of every super node v in the tree being *tagged* with the states $b_1, b_2, \cdots, b_{|Q|}$ and use the notation $t(v)$ to denote the tag. Note that $t(v)$ is the same as $l(v)$ restricted to the columns of Q. We can therefore partition the set of rows if the tags $t(v)$ are all unique. The problem arises when two super nodes say v_1, v_2 are both assigned the same tags - and therefore the same subset of rows S. We need to further partition S into two sets S_1 and S_2 one for each v_1 and v_2. We need several lemmas to determine how to make this partition.

Definition 4 *We use the notation $\mathcal{P}_{v_1,v_2}(T)$ to be the path between (and including) vertices (or super nodes) v_1 and v_2 of any tree T.*

Note that if v_1, v_2 are super nodes and T is the phylogeny (not skeleton), then the notation $\mathcal{P}_{v_1,v_2}(T)$ is used to denote the path in the tree T that connects super nodes v_1 and v_2 of the corresponding skeleton $s(T)$.

The proof for the following theorem is given in Appendix 4.

Theorem 2 *If there exists a q-near-perfect phylogeny that has bad vertices (w.r.t Q) then there exists a q-near-perfect phylogeny that does not contain any bad vertices (w.r.t Q).*

For the rest of the paper, we fix an optimal phylogeny T_{opt} that does not contain any bad vertices. We will impose additional restrictions on T_{opt} later.

Corollary 3 *In T_{opt}, if there exists two edges $e = (v_1, v_2)$ and $e' = (v_3, v_4)$ such that $\mu(e) = \mu(e')$ then for all columns $j \notin Q$, $|\{e \in \mathcal{P}_{v_2,v_3}(T_{opt})|\mu(e) = j\}|$ is even and therefore $l(v_1)[i] = l(v_4)[i]$ for all $i \notin Q$.*

For any phylogeny T, the skeleton of T, $s(T)$, with respect to a set of columns Q (vertex-cover) is the tree of super-nodes where each super node is attached to edges $e \in T$ s.t $\mu(e) \in Q$. A super node can be viewed as a set of vertices of the final phylogenetic tree and therefore gives us the following obvious properties of containment. For a super node $v_1 \in s(T)$, we say that edge $e = (u, v) \in v_1$ iff $u \in v_1, v \in v_1$ and $(u, v) \in E(s(T))$. Similarly, we say that column $j \in v_1$ if $\exists e \in v_1$ s.t $\mu(e) = j$.

Definition 5 *Two super nodes v_1 and v_2 of $s(T)$ are tag-adjacent if*

1. *$t(v_1) = t(v_2)$ and*
2. *there exists no super node $v_3 \in \mathcal{P}_{v_1,v_2}(s(T))$ such that $t(v_3) = t(v_1)$*

Definition 6 *Two super nodes v_1 and v_2 of $s(T)$ are paired-tag-adjacent if there exists no super node $v_3 \in \mathcal{P}_{v_1,v_2}(s(T))$ such that $t(v_3) = t(v_1)$ or $t(v_3) = t(v_2)$*

Definition 7 *An edge $e \in E(T)$ or column $\mu(e)$ is distinguishing w.r.t a phylogeny T if for any two tag-adjacent super nodes v_1 and v_2, $|\{e' \in \mathcal{P}_{v_1,v_2}(T)|\mu(e') = \mu(e)\}|$ is odd.*

It is easy to see that for the optimal near-perfect phylogeny T_{opt} there exists a distinguishing edge for every pair of tag-adjacent super nodes.

3.1.1 Partition

We can define equivalence classes $V_{t(v)}$ containing super nodes based on the equality of tags $t(v)$ of super nodes. It is straightforward to partition the rows R for each equivalence class of super nodes. This is performed by just examining the tags of each equivalence class. The pseudo-code in Figure 3 provides the details of the steps necessary to compute the partitions. At any point in time, the partition algorithm maintains a partition $P(V_{t_i})$ for each equivalence class. Each recursive call refines one of the partitions of one of the equivalence classes. Just to avoid excessive notations we will use $P(V_{t_i}) = \{S_1, \cdots, S_k\}$ for all i. It should be obvious from the context as to which equivalence class S_i belongs to. We say that a column i partitions a set S_j if there exists two rows $r_1, r_2 \in S_j$ such that $r_1[i] = 0$ and $r_2[i] = 1$. The partition can be written as $S_{j_0} = \{r_1 \in S_j | r_1[i] = 0\}$ and $S_{j_1} = \{r_2 \in S_j | r_2[i] = 1\}$.

An overview of the approach is to first discover the distinguishing columns that partition two different equivalence classes but only mutates once in T_{opt}. In the second step we guess the distinguishing columns that partition two different equivalence classes and mutates multiple times. The columns guessed are added to the skeleton so that the skeleton expands in size.

The value $\mathbf{mark(v)}$ denotes our guess of whether or not the super node v contains any real vertices in T_{opt}. The value is set to 'real' if we guess that there exists at least one real vertex in v and 'steiner' otherwise. A partition $P(V_{t_i})$ is complete when the number of sets in $P(V_{t_i})$ is equal to the number of super nodes v in V_{t_i} with $mark(v)$ = real, that is: $|P(V_{t_i})| = |\{v \in V_{t_i} | mark(v) = \text{real}\}|$.

The partition function initially finds the set D_2 consisting of columns that refines the partition in two different equivalence classes. If $|D_2| > q+1$, then the function picks an arbitrary set of $q+1$ columns and performs $2(q+1)$ recursive calls on partition induced by each column in the set on two different equivalence classes that the column refines. If $|D_2| \leq q$, then the function guesses all the mutations of the columns in D_2 and adds them to $s(T)$. Note that each added edge splits a single super node into two new super nodes.

At the end of the function partition, we add mutations and expand the skeleton. Before the expansion the set D_2 was empty. The following Lemma proves that it remains empty even after the skeleton expands. Moreover, it remains empty throughout the execution of function use-interface which will be described shortly.

Lemma 4 *If D_2 is empty at any time, then it remains empty when expanding the skeleton by the addition of non-vertex cover columns (specifically in steps 3a, 3c of function partition described in Figure 3 and step 5 of function use-interface described in Figure 4).*

Proof: For the sake of contradiction, assume that D_2 was empty and later became non-empty. Let c be a column in D_2. Let t_1 and t_2 be the two tags that are split by c. Note that if the t_1 and t_2 are different on the columns in Q, then the column c would have been found in function partition (Figure 3) itself. Therefore it must be the case that t_1 and t_2 are identical on columns in Q. Since t_1 and t_2 are identical on columns in Q, the algorithm found a column d that distinguishes them. Now note that c and d together must form 4 gametes. This is a contradiction to the fact that neither c nor d belongs to Q.

□

Correctness:

A partition $P(V_{t_i}) = \{S_1, \cdots, S_k\}$ is defined to be *good*, if it can be refined by further partitions (no merges/unions) into that of T_{opt}. The set of partitions $\{P(V_{t_1}), \cdots, P(V_{t_p})\}$ is good if every partition is good.

The correctness can be proved inductively by making the claim that if the argument set of partitions is good for a function call of *partition* then the arguments for at least one of the recursive calls is good. Consider any one function call to partition and assume inductively that its argument is good. We recursively perform partition in step 4(b)ii. If $|D_2| \geq q+1$ then there exists at least one column $i \in D_2$ that mutates only once in T_{opt}. A column i that mutates only once induces a partition on $P(V_{t_j})$ only when two super nodes (of the same equivalence class V_{t_j}) contain e with $\mu(e) = i$ in the path connecting them or if i mutates inside a super node $v \in V_{t_j}$. We know however that i partitions at least two different equivalence classes $V_{t_j}, V_{t_{j'}}$. Since i mutates once, it can mutate inside the super node of at most one of V_{t_j} or $V_{t_{j'}}$ and lie in the path connecting two super nodes of the other equivalence class. We recurse on both the partition induced on $P(V_{t_j})$ and $P(V_{t_{j'}})$. Moreover, since the arguments of partition is good, the partition induced by using i on at least one of $S_j \in P(V_{t_j})$ or $S_j \in P(V_{t_{j'}})$ is good. This completes the recursive case. The other case

function partition ($P(V_{t_1}), \cdots, P(V_{t_p})$ **, int list selectedCols)**

1. if $\forall i$, $P(V_{t_i})$ is complete then
   (a) guess the assignments of the partitions S_i to the super nodes of the skeleton
   (b) expland the skeleton into S' st columns in selectedCols mutate once and the expanded tags of the super nodes are compatible with the rows assigned
   (c) return S'
2. let D_2 be the set of columns st, $\forall\ i \in D_2$, $\exists_{>1} j$, $\exists k$ st i partitions $S_k \in P(V_{t_j})$
3. if $|D_2| \leq q+1$
   (a) guess multiple mutating columns among set D_2 and add to skeleton
   (b) guess the assignments of the partitions S_i to the super nodes of the skeleton
   (c) expland the skeleton into S' such that columns in selectedCols mutate once and the expanded tags of the super nodes are compatible with the rows assigned
   (d) guess $mark(v)$ for all super nodes $v \in S'$ corresponding to the expanded skeleton
   (e) use-interface (S')
4. else
   (a) consider any $D_2' \subseteq D_2$ s.t $|D_2'| = q+1$
   (b) $\forall\ i \in D_2'$ and for any two j s.t $\exists S_k \in P(V_{t_j})$, i partitions S_k
      i. let $P(V_{t_j}) = (P(V_{t_j}) \backslash \{S_k\}) \cup \{S_{k_0}\} \cup \{S_{k_1}\}$
      ii. partition ($P(V_{t_1}), \cdots, P(V_{t_p})$, selectedCols $\cup \{i\}$)

Figure 3: Algorithm to partition and assign rows of the input to super nodes

function use-interface (skeleton S)

1. let D_1 be the set of columns s.t, $\forall\, i \in D_1$, $\exists_1 j$, s.t i partitions $R(V_j)$.
2. select some V_j that is incomplete
3. select a lowest pair $v_1, v_2 \in V_j$ of tag-adjacent super nodes, such that $mark(v_1) = mark(v_2) =$ real
4. if $\exists h \in D_1, v_3 \in V_{t'}$ and $v_4 \in V_{t''}$, for some $t' \neq t''$ and $t \neq t', t''$ such that
   (a) $mark(v_3) = mark(v_4) = real$
   (b) h partitions $R(V_{t'}) \cup R(V_{t''})$
   (c) $\mathcal{P}_{v_1,v_2}(S)$ and $\mathcal{P}_{v_3,v_4}(S)$ share a super-node in S
   (d) (v_1, v_3) and (v_2, v_4) are paired-tag-adjacent then
5. guess extension S' of skeleton S by mutating column h (once or multiple) times
6. guess $mark(v)$ for super nodes adjacent to mutation(s) of h
7. use-interface (S')

Figure 4: Algorithm to find interface edges and extend the skeleton

is when we guess all multiple mutations among columns i that partition at least two different equivalence classes in step 3a. Since the size of the set of columns from which we guess is bounded by q, this step can clearly be performed in $O(2^q)$ time to find the columns mutating multiple times and $2^{O(q)}q^{O(q)}$ to place the edges in the skeleton tree. Note that we also guess the number of times a column mutates.

The step 1b expands the skeleton by adding columns of selectedCols.

As before, the edges of the expanded skeleton define the tags associated with each super node. We now assign for each super node a partition S_i. While making this assignment it could be the case that the number of partitions is less than the the total number of super nodes. Specifically, two super nodes of the same tag are assigned the same partition. However, the number of distinct tags in the skeleton should be equal to the number of partitions. This step can be naively implemented by once again enumerating all skeletons using the columns of the current skeleton and selectedCols. We can then discard any skeleton that is not compatible to the set of partitions. The running time for extending the skeleton and assigning the rows can be bounded as follows. The recursion tree has height q, and any function call to *partition* makes at most $2(q+1)$ recursive calls. Therefore the number of leaves of the recursion tree is bounded by $2^q(q+1)^q$. Each leaf node contains at most q columns in its list *selectedCols*. Since the number of trees with $2q$ vertices is bounded by $2^{O(q)}q^{O(q)}$, step 1b can be implemented in $2^{O(q)}q^{O(q)}$ time. This assures that the total time spent by partition function is bounded by $2^{O(q)}q^{O(q)}$.

3.1.2 Interface

For discovering distinguishing edges, we are just left with the last case - distinguishing columns that mutate once or more but partition only one equivalence class. We define $R(V_t) \subseteq R$ for any equivalence class V_t to be the set of rows that match tag t. A vertex v of phylogeny T is called *real* if $l(v) \in R$ - informally if the label of the vertex is present in the input (equivalently, the vertex is non-steiner).

Consider the partition defined by T_{opt} on any one equivalence class V_t of super nodes. We say that a column i is an internal column if the partition induced by i on $R(V_t)$ is not good (similar to the definition used in function partition). Informally, this implies that edge e with $\mu(e) = i$ is present inside some super node $v \in V_t$ and there exists real vertices $r, r' \in v$ s.t $r[i] = 0$ and $r'[i] = 1$. We define a column that is not internal as interface. Note that an interface column induces a good partition. Informally a distinguishing interface edge is present in the path between two connected components of real vertices of the super nodes $u, v \in V_t$ s.t there exists real vertex $r \in u, r[i] = 0$ and real vertex $r' \in v, r'[i] = 1$.

Definition 8 *A pair of tag adjacent super nodes v_1 and v_2 is defined to be a* **lowest pair** *in $s(T_{opt})$ if there exists a vertex x such that for all super nodes $v_3 \in s(T_{opt})$ with $t(v_3) = t(v_1)$, $x \in \mathcal{P}_{v_1,v_3}(T_{opt})$ and $x \in \mathcal{P}_{v_2,v_3}(T_{opt})$. The vertex x is called the branch point of v_1 and v_2.*

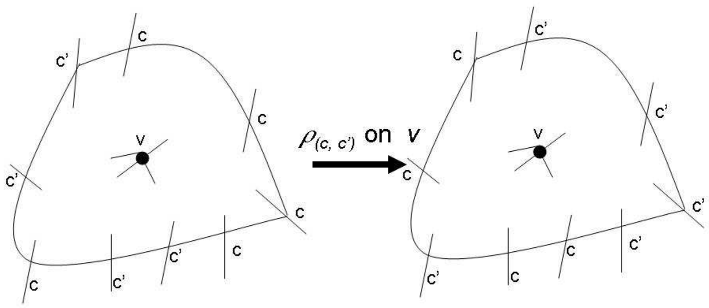

Figure 5: Application of transform ρ. Note that the bits of steiner vertex v changes at positions c and c' after the transform.

Transform $\rho_{c,c'}$

We now describe a very simple transform $\rho_{c,c'}$ that operates on any tree T and is defined over two columns c, c' and a steiner vertex v. To apply transform $\rho_{c,c'}$ on steiner vertex v, all vertices v' such that v' is the only real vertex in $\mathcal{P}_{v,v'}(T)$ should satisfy the following property. There exists edge $e \in \mathcal{P}_{v,v'}(T)$ s.t $\mu(e) = c$ or $\mu(e) = c'$. An example of when such a transform can be applied is shown in Figure 5. For all such paths from v to v', the transform replaces the first mutation of c with c' and vice-versa. The result after applying the transform is shown in Figure 5. Clearly, the transform does not change the cost of the tree.

Lemma 5 *In T_{opt}, for every lowest pair super nodes $v_1, v_2 \in V_t$ with branch point x such that every distinguishing column of v_1 and v_2 is an interface column, there exists at least one distinguishing interface column h with the following properties:*

1. *column h partitions $R(V_t)$*
2. *$\exists v_3 \in V_{t'}, v_4 \in V_{t''}$, $t \neq t', t''$ s.t v_1, v_3 and v_2, v_4 are paired-tag-adjacent and h partitions $R(V_{t'}) \cup R(V_{t''})$ and*

3. *the paths $\mathcal{P}_{v_1,v_2}(s(T_{opt}))$ and $\mathcal{P}_{v_3,v_4}(s(T_{opt}))$ share a super-node*

Proof: First note that the column h that partitions $R(V_{t'}) \cup R(V_{t''})$ can only partition it into $\{R(V_{t'}), R(V_{t''})\}$. This is because h induces a partition only in one equivalence class V_t and $t \neq t', t''$. For the same reason $t' \neq t''$. Throughout the proof of the lemma, the distance between two vertices (or edges) on a tree is simply the number of edges in the path connecting the vertices (or edges).

We know that for an optimal phylogeny, there has to exist at least one distinguishing edge h for v_1 and v_2. Let u_1 and u_2 be the vertices in v_1 and v_2 respectively, that lie on either ends of $\mathcal{P}_{v_1,v_2}(T_{opt})$. Therefore h has to mutate odd number of times in exactly one of $\mathcal{P}_{u_1,x}(T_{opt})$ and $\mathcal{P}_{x,u_2}(T_{opt})$. Without loss of generality assume that it mutates an odd number of times in $\mathcal{P}_{u_1,x}$. Consider the edge e closest to u_1 such that $\mu(e) = h$. In T_{opt} if the path from e to u_1 consists entirely of degree 2 steiner vertices, then the mutation of e can be incrementally swapped with the mutation of the neighboring edge, until the mutation of h is moved inside super node v_1. Note that swapping adjacent edges of a degree 2 steiner vertex is just a trivial application of ρ on the vertex. After several applications of ρ, column h is no longer distinguishing between v_1 and v_2 in the new optimal solution.

If the path from e to u_1 contains a vertex w of degree greater than 2, then every branching path from w leads to a real vertex (since steiner vertices can not be a leaf). Consider any such real vertex r, the super node in which r lies cannot have tag t since it contradicts the assumption that v_1 and v_2 are a lowest pair. Now consider the case when for all such real vertices r, the path $\mathcal{P}_{w,r}(T_{opt})$ contains a mutation of h. Using transform ρ, the mutation of h can be swapped with neighboring edges until it is adjacent to w. Once again let e be the edge adjacent to w, st $\mu(e) = h$. Let $e' \in \mathcal{P}_{u_1,x}, e \neq e'$ be another edge adjacent to w. We can now apply transform $\rho_{\mu(e'),h}$ on vertex w. The result of the transform is that the mutations of h and $\mu(e')$ swap positions. We can therefore keep applying transform ρ to move the mutation of h closer to u_1. There are only two cases when we can not apply the transform any further. In both the cases described below, we find super node v_3 that contains a real vertex.

Case 1: If the path from e to u_1 contains a real vertex r, that belongs to a super node v_3 say, then $t(v_3) \neq t(v_1)$, since we assumed that v_1 and v_2 are tag-adjacent.

Case 2: If the path from e to u_1 contains a vertex w of degree greater than 2, that contains a branch leading to a real vertex $r \in v_3$, where $t(v_3) \neq t(v_1)$ such that the path from w to r does not contain a mutation of h. Note that $t(v_3) \neq t(v_1)$ since v_1 and v_2 are a lowest pair.

Now consider the edge $e \in \mathcal{P}_{u_1,x}(T_{opt})$ such that $\mu(e) = h$ and the distance of e from v_1 is the largest. Using a series of transforms ρ we can move the mutation of h towards x. Once again with a similar argument we can either claim that there exists a real vertex r' in a super node v_4 that either lies in the path from e to x or is in the induced tree of w which lies in the path from e to x. If this is the case, then r and r' are two real vertices separated by an odd number of mutations of h. Also, $t(v_1) \neq t(v_3), t(v_4)$. This satisfies all the properties of the lemma.

Now consider the case when the mutation of h can be moved so that it is adjacent to x. Now consider any vertex cover column c that mutates an odd number of times between x and v_2. There has to exist at least one such edge since x can not belong to a super node of tag t as v_1 and v_2 are tag-adjacent. Consider the edge e' such that $\mu(e') = c$ and the distance from e' to v_2 is maximum. First assume that all the vertices in $\mathcal{P}_{u_2,x}(T_{opt})$ are of degree 2 and steiner. Clearly now, using the trivial ρ transform, the mutation of c can be moved so that it is adjacent to x. We can now perform transform $\rho_{c,h}$ on x to obtain a new optimal tree where the mutations of h and c adjacent to x have switched places. Note that the condition of $\rho_{c,h}$ requires that every path from x to a real vertex contains either a mutation of h or a mutation of c. This can be proved as follows for the optimal tree T_{opt}. Consider any path from x to the first real vertex r' that is in a super node v_4. If $t(v_4) = t(v_2)$ then the path from x to v_4 should contain an odd number of mutations of c, since the path from x to v_2 contains an odd number of mutations of c. If $t(v_4) \neq t(v_2)$, then the path from x to v_4

should contain an odd number of mutations of h, otherwise r' and r that lie in v_3 and v_4 respectively satisfy the properties of the Lemma. After applying $\rho_{c,h}$, we can move the mutation of h inside v_2 since all the vertices are degree 2 the result of which is that h is no longer distinguishing between v_1 and v_2.

The last case that is left to analyze is when the path from x to e' contains either real vertices or steiner vertices of degree greater than 2. If it contains a real vertex r', then r' along with r satisfy the properties of the Lemma. If there exists a vertex w that has degree greater than 2, then the induced subtree rooted at w contains real vertices, and all paths to real vertices should contain a mutation of h (otherwise we can use the real vertex along with r to satisfy the properties of the lemma). Therefore $\not{e}xistse''' \in \mathcal{P}_{x,u_2}$ with $\mu(e''') = h$, since otherwise we can move the mutation of h using a series of ρ transforms so that it is adjacent to x. This is a contradiction to optimality since we already have e adjacent to x with $\mu(e) = h$. Therefore, we can directly apply transform $\rho_{h,c}$ on vertex x. We can then move the mutation of h into u_2 with a series of ρ transforms.

Putting everything together, we have the fact that if for any distinguishing column h there exists no super nodes that satisfy the properties of the lemma, then the edge corresponding to the mutation of h can be moved such that it is no longer distinguishing. Furthermore, no new column becomes distinguishing in this process. We can continue this argument for every distinguishing column. If none satisfy the properties of the lemma, then eventually the first and last vertices of the path $\mathcal{P}_{v_1,v_2}(T_{opt})$ are identical, a contradiction to optimality. □

Lemma 5 proves that for every pair of tag-adjacent super nodes, there exists at least one distinguishing interface edge that has the properties that the pseudo-code uses to identify them in step 4 of function *use-interface*. However the converse that only distinguishing interface edges have the property is not true. We will however prove that only internal edges that mutate multiple times could satisfy the properties. Intuitively, this proves that not too many internal columns are falsely identified as interface columns. Note that at step 4 of the pseudo-code column h partitions V_t and also partitions $R(V_{t'}) \cup R(V_{t''})$. If h is an internal column, then there exists internal edge e'' in some super node $v \in V_t$ s.t $\mu(e'') = h$. Super node $v \notin \mathcal{P}_{v_3,v_4}$ because we assumed that (v_1, v_3) and (v_2, v_4) are paired-tag adjacent. Assume that h lies in $\mathcal{P}_{u_3,u_4}$ in T_{opt} where $u_3 \in V_{t'}$ and $u_4 \in V_{t''}$. Note that for h to partition $R(V_{t'}) \cup R(V_{t''})$ it has to be the case that both u_3 and u_4 contain at least one real vertex. Note that one of u_3 or u_4 could be v_3 or v_4 respectively.

Consider the two paths $\mathcal{P}_{v_3,u_3}$ and $\mathcal{P}_{v_4,u_4}$ (shown in Figure 6). If h mutates just once, then it should be the case that the mutation of h, edge e'', should lie in one of the two paths. This is however a contradiction since h now partitions two equivalence classes V_t and one of $V_{t'}$ or $V_{t''}$ (since v_3, u_3, v_4 and u_4 each contain at least one real vertex in T_{opt}). Therefore, it is not possible to add an internal column to the set I unless it mutates more than once in T_{opt}.

At each recursive call, we either find a column that mutates only once thereby partitioning rows into two super nodes or discover multiple mutant edges. Therefore the depth of the recursion is bounded by $O(q)$. Since the degree of any node of the recursion tree is bounded by $2^q q^q$, a naive analysis gives a bound of $2^{O(q^2)} q^{O(q^2)}$ number of calls to the function use-interface.

At the end of the recursive calls, one of the leaves of the recursion tree corresponds to the expanded skeleton of T_{opt}. **Note:** We showed that the depth in the recursion tree of the leaf that corresponds to the expanded skeleton of T_{opt} is bounded by cq for some constant c. Therefore we can safely terminate the recursion at depth cq.

3.2 Linking perfect phylogenies

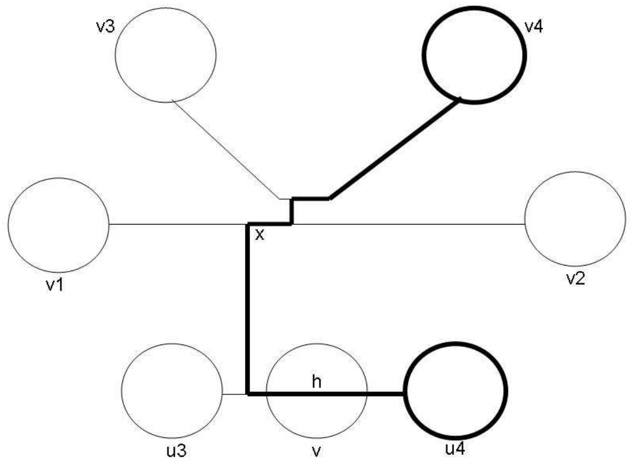

Figure 6: Proof that column h has to mutate multiple times to be an internal column that also satisfies the properties of a distinguishing interface column.

function link-trees (skeleton S)

1. For each leaf super-node S_i of S do
   (a) build perfect phylogeny on the vertices in S_i
   (b) let $P_i \leftarrow$ the set of vertices of S_i
   (c) for each mutation in S_i do
      i. if $S \setminus S_i$ contains only a single state (say 0) on column c, then remove all in P_i that contain 1 on column c
   (d) "guess" a vertex w from P_i, and let w' be obtained from w by mutating the vertex cover between S_i and $p(S_i)$.
   (e) add w' to $p(S_i$ and remove the leaf S_i from S
2. repeat step 1 until S is empty

Figure 7: Algorithm to build the tree using the skeleton and partition

We now show how to complete the the near-perfect phylogeny once the correct partition on the super nodes have been computed (step 2(a)iD of the main pseudo-code). **Note:** Let the initial skeleton built using the vertex cover columns in step 2a of the main pseudo-code be S. Assume that S is isomorphic to $s(T_{opt})$. Since $s(T_{opt})$ does not contain any bad vertices and is optimal, all the supernodes contain perfect phylogenies.

Lemma 6 *Every super node u of the skeleton $s(T_{opt})$ contains a perfect phylogeny.*
Proof: For the sake of contradiction, assume not. Consider edges $e, e' \in u$ with $\mu(e) = \mu(e') = c$. Consider

the path $\mathcal{P}$ connecting edges e and e'. There exists no column $c' \notin Q$ that mutates an odd number of times in $\mathcal{P}$ according to Corollary 3.1. By the definition of the super-nodes and skeleton we know that there exists no edge $e \in u$ with $\mu(e) = c$ if $c \in Q$. For optimality, we can not have two vertices v and v' with $l(v) = l(v')$. Therefore there can not exists edges e and e' as stated. □

Using the partitions of the expanded skeleton T, we can find the partitions for the skeleton S. For the rest of the paper, we will work with skeleton S rather than T. We lose some information in this step, but it keeps the analysis simple. We now show how to construct a perfect phylogeny for the rows assigned to a super node of S. For any pair of columns that mutate within a super node, there can be at most three gametes. We can now reconstruct a unique perfect phylogeny within each super node as follows. If two columns induce two gametes, then we can arbitrarily remove one of them. If two columns induce just one gamete then we can remove both the columns. After this step, every pair of columns contain exactly three gametes. Therefore a unique perfect phylogeny can be built. We can now add back columns i that were removed since (i, j) induced only two gametes. This is done by adding the mutation of i adjacent to the mutation of j. The relative ordering of i and j are determined based on which of the third gamete between i and j is present in the input (and absent from the rows of this super node). Hence we can build a unique perfect phylogeny that does not contain any bad vertices.

The notation $p(v)$ is used to denote the parent of a vertex v in a directed tree. In the following section the term subtree T (say rooted at v) of the rooted tree T_{opt}, is used to refer to the induced tree T rooted at vertex v obtained by removing the edge $(v, p(v))$ from T_{opt}.

Lemma 7 *In the rooted tree T_{opt}, consider any induced subtree T'. If $T_{opt} \backslash T'$ contains vertices v and v' such that $l(v)[c] = 0$ and $l(v')[c] = 1$ then there exists at least one edge e s.t $\mu(e) = c$ in $T_{opt} \backslash T'$*

Proof: Assume not, then the mutation of column c occurs in T'. But T' and $T_{opt} \backslash T'$ are connected by a single edge. Therefore, $T_{opt} \backslash T'$ contains only one of either 0 or 1 in column c, contradiction.

Corollary 8 *If both T' and $T_{opt} \backslash T'$ each contain a pair of vertices v, v' s.t $l(v)[c] = 0$ and $l(v')[c] = 1$ for some column c then there are at least 2 edges $e, e' \in T_{opt}$ s.t $\mu(e) = \mu(e') = c$.*

Note that in the rooted T_{opt}, the perfect phylogeny inside each super node u can be decomposed into a minimal connected component of perfect phylogeny on real vertices T_u^r, called core tree. The edges in $T_u \backslash T_u^r$ are called peripheral edges. There could be several connected components of peripheral edges inside each super node. Note that all interface edges are peripheral edges. At this point, it is easy to compute the unique core tree, which is just the perfect phylogeny on the rows that have been partitioned to any super node. For this last part of the proof, we will be considering the optimal tree T_{opt} with the following additional property: no super node u (except the root) contains a degree two vertex that is adjacent to both - a peripheral edge and the vertex cover edge connecting u to super node $p(u)$.

Lemma 9 *There exists an optimal tree T_{opt} that contains no vertex v in super node u with the property that degree(v) = 2 and v is adjacent to both an peripheral edge and the vertex cover edge connecting u to super node p(u).*

Proof: Assume not, then consider the optimal tree T_{opt} that contains the minimum number of vertices v with the above property. One edge adjacent to v is the vertex cover edge say e_v, with say $\mu(e_v) = k$. The second edge connects to a vertex v' inside super node u and let $\mu(v, v') = l$. Now we can flip the ordering of mutations of e_v and (v, v') such that $\mu(e_v) = l$ and $\mu(v, v') = k$ to obtain a new tree T'_{opt}. Note that this is just a trivial operation of transform $\rho_{k,l}$ on steiner vertex v. It is easy to see that the only pair of columns in v of T'_{opt} that could contain a new gamete not present in the original tree T_{opt} contains one column of the vertex cover. Therefore in T'_{opt}, all the vertices are good (assuming they were in the original tree). The number of vertices in the super node $p(u)$ increased by one and the number of super nodes in u decreased by one since v was previously part of u and now is part of $p(u)$. Therefore the number of vertices with the property given in the claim decreased by one, a contradiction to the assumption.

□

In $s(T_{opt})$, consider a super node u, that has children super nodes $c_1, \cdots, c_k$. The goal is to discover all the edges of T_{opt} bottom-up. Let S_i be the tree present inside the super node c_i in T_{opt}. Note that we are using the super node S_i to refer to the super node of both the skeleton S and that of $s(T_{opt})$. Let S'_i be the subtree of T_{opt} rooted on a vertex of c_i, obtained by removing the vertex cover edge connecting c_i to u. Note that the tree S'_i restricted to the nodes of c_i is T_i.

Assume inductively that we have computed the trees inside super nodes S_i descendants and all the of S_i. In otherwords the subtrees S'_i of T_{opt} have been found. Now we proceed to construct the tree inside super node u. Note that this amounts to building the core tree of u and connecting vertices of u to vertices in S_i as in T_{opt} (which is equivalent to computing the peripheral edges). Consider one by one each of the trees S_i. We start by adding all vertices of S_i into a selected set P. At any time if $|P| < q$, then we have completed processing S_i. If not, then for each mutation of column c in S_i, we check if $S \backslash S'_i$ contains both states 0 and 1 on c. If so, we ignore c. By claim 7, the fact that there are at most q multiple mutations for any character c and that S_i is a perfect phylogeny, we ignore at most q mutations of S_i. If not, then wlog say 0 is the only state in column c that occurs in $S \backslash T'_i$. If any vertex in T_i contains a 1 in column c and is present in P, then we remove it from P.

Claim 10 *Only the selected vertices in P can be vertices of T_{opt} that connects c_i to u.*

Proof: There exists exactly one edge $e = (v_0, v_1)$, s.t $\mu(e) = c$ in S_i. The edge e partitions S_i into S_i^0 and S_i^1 based on the value on the column c and assume $v_0 \in S_i^0$ and $v_1 \in S_i^1$. For the sake of contradiction assume that in T_{opt}, a vertex v from S_i^1 connects to u via the vertex cover edge. Clearly every path to a real vertex in $T_{opt} \backslash S'_i$ from v contains an edge $e' = (v'_1, v'_0)$ with $\mu(e') = c$.

Consider the edges e' that is the first mutation of column c in every path from e to a real vertex in $T_{opt} \backslash S'_i$. Let M be the set of columns that mutate in the path connecting v_0 and v. Since there are no bad vertices, all mutations of columns except vertex cover, between e and e' occur even number of times. Therefore, specifically v_0 and v'_0 are identical in all the columns in M. Therefore, connecting v_0 instead of v to u via vertex cover and deleting all mutations of M that occurs in the path between e and e' results in a tree of smaller cost, a contradiction to optimality. □

We continue by finding sets P for each of S_i. Now, using exhaustive search, we pick one by one every possible way of selecting a vertex from each of the selected sets P for each of S_i. One would be the correct set of vertices as used in T_{opt}. For each v selected from S_i, we mutate the column corresponding to the vertex cover edge between super nodes c_i and u and add to set A. We now add A to the super node u and re-construct the perfect phylogeny of u. It follows from the claim that we have now completely identified the tree in u. It is easy to implement the above procedure in $q^{O(q)}$ time.

Theorem 11 *The new algorithm solves the q-near-perfect phylogeny problem in $O(2^{O(q^2)}(nm + n^2))$ time.*

Proof: The proof follows from the above lemmas and Theorem 2. □

4 Discussion

We now give a quick summary of how the running time can be improved to $q^{O(q)}nm^2$. The only step that needs to be improved is `use-interface`. Intuitively, instead of building the skeleton top-down, we can construct it bottom-up as follows. Assume that we have a skeleton T_1 and we complete the use-interface step as described above to obtain a tree where the super nodes have unique tags. Let this final skeleton be T_2. We are now going to show that we can arrive at tree T using a modified routine of use-interface. It is easy to guess the final topology of the skeleton T_2 in time $q^{O(q)}$ even before executing the function `use-interface` since T_2 has $O(q)$ edges. Let this unlabeled tree be T'. We can guess the labels $\mu(e)$ for all edges $e \in T$ where $\mu(e)$ is either a vertex cover column or a column found in function partition. This gives us a partially labelled tree T', where the only unlabeled edges are those that were found using the old `use-interface` function. Note that in the current tree T', the connected components of unlabeled edges are exactly the super nodes of the skeleton T_1. We can now execute the `use-interface` function as described above, except that we do not add mutations of columns h into an existing skeleton (as in step 5). Instead we guess all unlabeled edges $e \in T'$ such that $\mu(e) = h$. If we guess that column h mutates i number of times, then this step can be performed in time $\binom{p}{i}$, where p is the number of unlabeled edges currently in the tree. As before we perform `use-interface` on each guess recursively. The following recursion bounds the number of calls to `use-interface` when the current tree has p unlabeled edges:

$$T(p) \leq \sum_{i=1}^{p} p^i T(p-i)$$

This recursion can be solved for $T(p) = q^{2q}$. Note that the new function `use-interface`, exactly explores the same trees as the old function. This is just a tighter analysis of the running time obtained by making sure that we don't repeatedly explore the same tree several times. Such a bottom-up construction can be extended to include the partition function as well. The algorithm now becomes simple to implement, since we can guess the final skeleton's topology up front and just guess the labels of the skeleton as the algorithm progresses.

References

[1] R. Agarwala and D. Fernandez-Baca. A Polynomial-Time Algorithm for the Perfect Phylogeny Problem when the Number of Character States is Fixed. In: *SIAM Journal on Computing*, 23 (1994). pp 1216-1224.

[2] H. Bodlaender, M. Fellows and T. Warnow. Two Strikes Against Perfect Phylogeny. In *proc. 19th International Colloquium on Automata, Languages and Programming, LNCS*, (1992). pp 273-283.

[3] M. Bonet, M. Steel, T. Warnow and S. Yooseph. Better Methods for Solving Parsimony and Compatibility. In: *Journal of Computational Biology*, 5(3), (1992). pp 409-422.

[4] K. Dhamdhere, S. Sridhar, G. E. Blelloch, E. Halperin, R. Ravi and R. Schwartz. A Faster Reconstruction of Near-Perfect Binary Phylogenetic Trees. *Manuscript in preparation.*

[5] D. Fernandez-Baca and J. Lagergren. A Polynomial-Time Algorithm for Near-Perfect Phylogeny. In: *SIAM Journal on Computing*, 32 (2003). pp 1115-1127.

[6] L. R. Foulds and R. L. Graham. The Steiner problem in Phylogeny is NP-complete. In: *Advances in Applied Mathematics* (3), (1982). pp 4

[7] G. Ganapathy, V. Ramachandran and T. Warnow. Better Hill-Climbing Searches for Parsimony. In: *WABI* (2003).

[8] D. Gusfied. Efficient Algorithms for Inferring Evolutionary Trees. In: *Networks*, 21 (1991). pp 19-28.

[9] S. Kannan and T. Warnow. A Fast Algorithm for the Computation and Enumeration of Perfect Phylogenies. In *SIAM Journal on Computing*, 26 (1997). pp 1749-1763.

[10] M. A. Steel. The Complexity of Reconstructing Trees from Qualitative Characters and Subtrees. In *J. Classification*, 9 (1992). pp 91-116.

Appendix A: Proof of Theorem 12

Theorem 12 *It is possible to construct a q-near-perfect phylogeny on the original input (before preprocessing) using the above algorithm.*

Proof: Let I be the original input and I' be the pre-processed input. If there is a tree with penalty q on I, then there exists a tree with penalty at most q on I'. Since the above algorithm goes through all trees with less than q penalty, we can add back the deleted columns and check if the penalty is still less than q. In order to show that such an algorithm will find an optimal tree, we need to prove that there always exists an optimal tree with this structure.

Consider any optimal tree T on the input I. Consider two identical columns a and b in I. We claim that we can modify the optimal tree in such a way that only the mutations of a or b are changed and the resulting tree will have all the mutations of a and b adjacent to each other. Suppose a mutation of a doesn't have a mutation of b adjacent to each other, then the vertices between them are steiner. Therefore, we can apply the transform τ_a to the neighborhood of steiner vertices. As a result of the transform, we get the mutations of a and b adjacent to each other. This also shows that the number of mutations of a and b in tree T is exactly same. □

Appendix B: Proof of Theorem 2

In this section, we prove Theorem 2. Recall that a vertex v is *bad* w.r.t a pair of columns (b, c) if it has a gamete for that pair which doesn't appear in the input. Theorem 2 asserts that there is an optimal tree which doesn't have any bad vertices, i.e. all its vertices have gametes that appear in the input.

The main idea is to take any optimum solution and to transform it into one which has the above property. This is achieved through a series of transformations that reduce the number of bad vertices. We first prove some simple facts about the OPT solution.

Definition 9 *We call a column c* clean *in a tree T, if all the vertices of T are good w.r.t (c, x) for all columns x.*

Fact 13 *If the column x is clean in the tree T and y is any other column, then the following properties hold:*

1. *Between any two mutations of column x, there is an even number of mutations of column y.*
2. *Between any two mutations of column y there is an even number of mutations of x.*

Definition 10 *Let u be a bad vertex w.r.t (b, c). We define the* bad neighborhood *of the vertex u w.r.t (b, c) to be*

$$N_{b,c}(u) = \{v \mid v \textit{ is bad w.r.t } (b, c) \textit{ \& all vertices on the path } u \to v \textit{ are bad w.r.t } (b, c)\}.$$

Definition 11 *The* boundary *of $N_{b,c}(u)$ is the set of edges that connect vertices in $N_{b,c}(u)$ to the rest of the tree T.*

Fact 14 *$N_{b,c}(u)$ is the connected component of bad vertices of tree T containing u.*

Fact 15 *The boundary of $N_{b,c}(u)$ consists of only the mutations of columns b and c.*

Claim 16 *The optimal tree contains equal number of mutations of columns b and c.*

Proof: If not then the following transformation reduces the number of mutations in the tree. This contradicts the optimality of the tree. □

Transform τ We now describe our transform that reduces the number of bad vertices in the optimum tree. Consider all the vertices in the optimum tree that are bad w.r.t (b, c). Let $V_1, V_2, \dots, V_k$ denote the bad neighborhoods w.r.t (b, c). The transform $\tau_{\mathbf{b}}$ (resp. $\tau_{\mathbf{c}}$) changes $V_1, \dots, V_k$ simultaneously as follows: delete the mutations of column b (resp. c) from the boundaries of the bad components and replace every mutation of c on the boundaries by a mutation of c followed by a mutation of b (resp. replace b by bc). Let T' denote the tree obtained by applying the transform τ_b on tree T. Note that both T and T' are optimal trees.

We prove the following two properties about the transform τ.

Lemma 17 *After applying the transform τ_b (or τ_c), all vertices of the tree T' are good w.r.t (b, c).*

Proof: The vertices that were good w.r.t (b, c) stay good even after the transform τ_b. All the bad vertices w.r.t (b, c) were replaced by good vertices during transform τ_b. So no bad vertices are left. □

Fact 18 *If the tree T does not have any bad vertices w.r.t (c, x), then the resulting tree T' after applying transform τ_b will not have any bad vertices w.r.t (c, x).*

Lemma 19 *If the column x is clean, then the transform τ_b will not create any bad vertices w.r.t (b, x).*

Proof of Theorem 2: Consider a lexicographic order on the columns $c_1, c_2, \dots$. We start with an optimum tree T. We first make column c_1 clean by considering bad vertices w.r.t (c_1, c_i) for each i and applying transform τ_{c_i}. Note that the transform τ_{c_i} will only affect pairs $(c_i, c_{i'})$. It does not create bad vertices w.r.t $(c_1, c_{i'})$. From Lemma 17, it follows that at the end of this step column c_1 will be clean.

Now inductively assume that columns $c_1, \dots, c_j$ are clean. Next we look at all the bad vertices w.r.t (c_{j+1}, c_i) (for $i > j + 1$) for each i and apply transform τ_{c_i}. Note that the transform τ_{c_i} doesn't create any bad vertices w.r.t $(c_j, c_{i'})$. Moreover, by Lemma 19, we can say that the resulting tree has columns $c_1, \dots, c_j$ clean. Moreover, it follows from Lemma 17 that at the end of this step the column c_{j+1} is also clean.

Therefore, in the end all the columns are clean. In other words, there are no bad vertices in the tree. □

We prove the Lemma 19 through the following series of claims.

Claim 20 *If the column x is clean in the tree T, then mutations of x in T are all outside $\cup_i V_i$ or all inside.*

Proof: For the sake of contradiction, assume that there is a mutation of x inside V_1 and another one outside $\cup_i V_i$. Consider a path between these two mutations. It is easy to see that either b or c mutates odd number of times between the two mutations of x. This contradicts the fact that x is clean. □

Claim 21 *Transform τ_b does not create odd number of mutations of b between two mutations of a clean column x.*

Proof: Using the result of Claim 20, we consider following two cases.

case 1 (*All x's are outside*) Consider a pair of mutations of x. If it doesn't have any mutation of b or c between them, then their parity will not change. If there were an even number of mutations of b in between, then in transform τ_b, an even number of mutations (twice the number of bad neighborhoods on the path) will get deleted and hence the parity won't change. If there were an even number of c's, then two mutations of b will be added for every bad neighborhood on the path. Overall, parity does not change.

case 2 (*All x's are inside*) In this case, every c gets replaced by cb. Since there were even number of b's and c's an even number of b's get created and an even number of b's get deleted between any pair of x's. Thus the parities do not change. □

Remark 1 *Claim 21 is required only if the column x mutates multiple times in the optimal tree.*

Note that in transform τ_b only b's are deleted or added. So a similar claim for c holds automatically.

Claim 22 *The tree T' resulting from transform τ_b does not leave odd mutations of a clean column x's between two mutations of b's.*

Proof: We show that between every adjacent pair of mutations of b in tree T', there are even number of mutations of x. For the sake of contradiction assume that a pair of b's has odd number of x's in tree T' resulting after the transform τ_b was applied. Let b_1 and b_2 denote the two mutations of b that sandwich an odd number of mutations of x. Note that both b_1 and b_2 cannot be preset in the tree T. Without loss of generality, assume that b_1 was created in tree T'. Therefore, there is a mutation of column c next to b_1. Call it c_1.

In the tree T, the column x was clean. Since the transform τ_b does not change in mutations of c, no mutation of x is sandwiched between two mutations of c in the new tree T'.

If b_2 is a new mutation added to tree T', then let c_2 be a mutation of the column c which is next to b_2 in the tree T'. In this case, the odd number of mutations of x are sandwiched between c_1 and c_2 which is not possible. Therefore b_2 is present in the tree T.

We use Claim 20 to separate following two cases.

case 1 (*All x's are outside*) Let $N_{b,c}$ denote the bad neighborhood where b_1 was created. Let $\hat{b}_1$ be a mutation of b that was deleted from the boundary of $N_{b,c}$. Then the path from $\hat{b}_1$ to c_1 in tree T contains no mutations of x, while the path from c_1 to b_2 contains an odd number of mutations of x. Therefore, in tree T there are an odd number of mutations of x between $\hat{b}_1$ and b_2, which contradicts the fact that x was clean in tree T.

case 2 (*All x's are inside*) Note that b_2 could not have been on the boundary of a bad neighborhood in the tree T, otherwise it would have been removed in the tree T'. Since all the mutations of x were inside bad neighborhoods, at least one of the neighborhoods on the path from b_1 to b_2 had an odd number of mutations. If this bad neighborhood is the one where b_1 was created in the transform, then the path b_1 to b_2 includes the mutation c_1. In T', all the vertices are good w.r.t (b, c). Hence there must be another mutation of c (call it c_2) between b_1 and b_2. Note that the path c_1 and c_2 contains an odd number of mutations of x. This is a contradiction. Therefore, the bad neighborhood containing odd number of mutations of x is not the one where b_1 was created. But in that case, consider the path from b_1 to b_2 in the old tree T. It must have entered and exited the bad neighborhood via mutations of b. Thus the odd number of mutations of x were sandwiched between two mutations of b in the old tree T. This is a contradiction to the fact that x was clean. $\square$