JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 13, Number 2, 2006

Pp. 522–553

A Linear-Time Algorithm for the Perfect Phylogeny Haplotyping (PPH) Problem

ZHIHONG DING, VLADIMIR FILKOV, and DAN GUSFIELD

ABSTRACT

Since the introduction of the Perfect Phylogeny Haplotyping (PPH) Problem in *RECOMB 2002* (Gusfield, 2002), the problem of finding a linear-time (deterministic, worst-case) solution for it has remained open, despite broad interest in the PPH problem and a series of papers on various aspects of it. In this paper, we solve the open problem, giving a practical, deterministic linear-time algorithm based on a simple data structure and simple operations on it. The method is straightforward to program and has been fully implemented. Simulations show that it is much faster in practice than prior nonlinear methods. The value of a linear-time solution to the PPH problem is partly conceptual and partly for use in the inner loop of algorithms for more complex problems, where the PPH problem must be solved repeatedly.

Key words: Perfect Phylogeny Haplotyping (PPH) Problem, Haplotype Inference Problem, linear-time algorithm, shadow tree.

1. INTRODUCTION

HAPLOTYPES HAVE RECENTLY BECOME A KEY UNIT of data in genetics, particularly human genetics. The International Haplotype Map Project (Helmuth, 2001; IHMC, 2003) is focused on determining the common SNP haplotypes in several diverse human populations. It is widely expected that correlations between occurrences of specific haplotypes and specific phenotypes (such as certain diseases) will allow the rapid location of genes that influence those phenotypes, and there are already several successful examples of this strategy. However, collecting haplotype data is difficult and expensive, while collecting genotype data is easy and cheap. Hence, almost all approaches collect genotype data and then try to computationally infer haplotype pairs from the genotype data.

1.1. Introduction to the PPH problem

In diploid organisms (such as humans) there are two (not completely identical) "copies" of each chromosome and hence of each region of interest. A description of the data from a single copy is called a *haplotype*, while a description of the conflated (mixed) data on the two copies is called a *genotype*. In complex diseases (those affected by more than a single gene), it is often much more informative to have haplotype data (identifying a set of gene alleles inherited together) than to have only genotype data.

Department of Computer Science, University of California, Davis, CA 95616.

Today, the underlying data that forms a haplotype is usually a vector of values of *m single nucleotide polymorphisms (SNP's)*. A SNP is a single nucleotide site where exactly two (of four) different nucleotides occur in a large percentage of the population. In general, it is not feasible to examine the two haplotypes separately, and *genotype* data rather than haplotype data is usually obtained. Then one tries to infer the original haplotype pairs from the observed genotype data. We represent each of the n input *genotypes* as vectors, each with m sites, where each site in a vector has value 0, 1, or 2. A site i in the genotype vector g has a value of 0 (respectively, 1) if site i has value 0 (or 1) on both the underlying haplotypes that generate g. Otherwise, site i in g has value 2. Note that we do not know the underlying haplotype pair that generates g, but we do know g.

Given an input set of n genotype vectors of length m, the *Haplotype Inference (HI) Problem* is to find a set of n pairs of binary vectors (with values 0 and 1), one pair for each genotype vector, such that each genotype vector is explained (can be generated by the associated pair of haplotype vectors). The ultimate goal is to computationally infer the true haplotype pairs that generated the genotypes. This would be impossible without the implicit or explicit use of some genetic model, either to assess the biological fidelity of any proposed solution, or to guide the algorithm in constructing a solution. The most powerful such genetic model is the population-genetic concept of a *coalescent* (Tavare, 1995; Hudson, 1990). The coalescent model of SNP haplotype evolution says that without recombination the evolutionary history of $2n$ haplotypes, one from each of $2n$ individuals, can be displayed as a rooted tree with $2n$ leaves, where some ancestral sequence labels the root of the tree, and where each of the m sites labels exactly one edge of the tree. A label i on an edge indicates the (unique) point in history where a mutation at site i occurred. Sequences evolve down the tree, starting from the ancestral sequence, changing along a branch e by changing the state of any site that labels edge e. The tree "generates" the resulting sequences that appear at its leaves. In terminology closer to computer science, the coalescent model says that the $2n$ haplotype (binary) sequences fit a *perfect phylogeny*. See Gusfield (2002) for further explanation and justification of the perfect phylogeny haplotype model.

Generally, most solutions to the HI problem will not fit a perfect phylogeny, and this leads to the following.

The Perfect Phylogeny Haplotyping (PPH) Problem: Given an n by m matrix S that holds n genotypes from m sites, find n pairs of haplotypes that generate S and fit a perfect phylogeny.

It is the requirement that the haplotypes fit a perfect phylogeny and the fact that most solutions to the HI problem will not that enforce the coalescent model of haplotype evolution and make it plausible that a solution to the PPH problem (when there is one) is biologically meaningful.

The PPH problem was introduced by Gusfield (2002) along with a solution whose worst-case running time is $O(nm\alpha(nm))$, where α is the extremely slowly growing inverse Ackerman function. This nearly linear-time solution is based on a linear-time reduction of the PPH problem to the *graph realization* problem, a problem for which a near-linear-time method (Bixby and Wagner, 1988) was known for over 15 years. However, the near-linear-time solution to the graph realization problem is very complex (only recently implemented) and is based on other complex papers and methods, and so taken as a whole, this approach to the PPH problem is hard to understand, to build on, and to program. Further, it was conjectured by Gusfield (2002) that a truly linear-time ($O(nm)$) solution to the PPH problem should be possible.

After the introduction of the PPH problem, a slower variation of the graph-realization approach was implemented (Chung and Gusfield, 2003a), and two simpler, but also slower methods (based on "conflict-pairs" rather than graph theory) were later introduced (Bafna *et al.*, 2003; Eskin *et al.*, 2003). All three of these approaches have best- and worst-case running times of $\Theta(nm^2)$. Another paper (Wiuf, 2004) developed similar insights about conflict-pairs without presenting an algorithm to solve the PPH problem. The PPH problem is now well known (for example discussed in several surveys on haplotyping methods [Bonizzoni *et al.*, 2003; Halldórsson *et al.*, 2003a, 2003b; Gusfield, 2004]). Related research has examined extensions, modifications, or specializations of the PPH problem (Kimmel and Shamir, 2004; Halperin and Eskin, 2004; Eskin *et al.*, 2004; Damaschke, 2003, 2004; Barzuza *et al.*, 2004) or examined the problem when the data or solutions are assumed to have some special form (Halperin and Karp, 2004; Gramm *et al.*, 2004a, 2004b). Some of those methods run in linear time, but work only for specializations of the full PPH problem (Gramm *et al.*, 2004a, 2004b) or are correct only with high probability (with some model)

(Damaschke, 2003, 2004). The problem of finding a deterministic, linear-time algorithm for all data has remained open, and a recent paper (Bafna *et al*., 2004) shows that conflict-pairs methods are unlikely to be implementable in linear time.

1.2. Main result

In this paper, we completely solve the open problem, giving a deterministic, linear-time (worst-case) algorithm for the PPH problem, making no assumptions about the form of the data or the solution. The algorithm is graph theoretic, based on a simple data structure and standard operations on it. The linear-time bound is trivially verified, and the correctness proofs are of moderate difficulty. The algorithm is straightforward to implement, and has been fully implemented. Tests show it to be much faster in practice as well as in theory, compared to other existing programs. As in some prior solutions, the method provides an implicit representation of all PPH solutions.

In addition to the conceptual value of our solution, its practical value can be significant. Currently, the full structure of haplotypes in human populations and subpopulations is not known, and there are some genes with high linkage disequilibrium that extends over several hundred kilobases (suggesting very long haplotype blocks with a perfect or near-perfect phylogeny structure). So it is too early to know the full range of *direct* application of this algorithm to long sequences (see Chung and Gusfield [2003b] for a more complete discussion). Moreover, faster algorithms are of practical value when the PPH problem is repeatedly solved in the inner loop of an algorithm. For example, in Chung and Gusfield (2003b) and Wiuf (2004), one finds, from every SNP site, the longest interval starting at that site for which there is a PPH solution. Moreover, there are applications where one may examine *subsets* of sites to find subsets for which there is a PPH solution. In such applications, efficiencies in the inner loop will be significant, even if each subset is relatively small.

2. THE SHADOW TREE

Our algorithm builds and uses a directed, rooted graph, called a *shadow tree* as its primary data structure. There are two types of *edges* in the shadow tree: *tree edges* and *shadow edges*, which are both directed towards the root. Tree and shadow edges are labeled by column numbers from S (with shadow edges having bars over the labels). For each column i in S, there are a tree edge, labeled i, and a shadow edge, labeled $\bar{i}$, in the shadow tree. The end points of each tree and shadow edge are called *connectors* and can be of two types: H or T connectors, corresponding to the head or tail of the edge.

The shadow tree also contains directed *links*. From a graph theory standpoint these are also edges, but we reserve the word *edge* for tree and shadow edges. Links are used to connect certain tree and shadow edges and are needed for linear-time manipulation of the shadow tree. Each link is either *free* or *fixed* and always points away from an H connector. When we say edge E *links to* E', we mean there is a link from the H connector of E to a connector of E'.

Since links can point to either an H or a T connector, the "parent of" relationship between edges is not the same as the "links to" relationship and is defined recursively: if an edge links to the root, then its parent is the root. If an edge E links to the T connector of an edge E_p, then the parent of E, $p(E)$, is defined as E_p. However, if E links to the H connector of an edge E', $p(E)$ is defined to be the same as $p(E')$. For convenience, we define the parent of a connector as the parent of the edge that contains the connector. See Fig. 1 for an illustration of all these elements.

Tree edges, shadow edges, and *fixed* links are organized into *classes*, which are subgraphs of the shadow tree. Every free link connects two classes, while each fixed link is contained in a single class. We will see later that each class in the shadow tree encodes a subgraph that must be contained in **all** solutions to the PPH problem. In each class, if the links (which are all fixed) are contracted, then the remaining edges form two rooted trees (except for the root class, which has only one rooted tree), where if one subtree contains a tree edge the other contains its shadow edge. The roots of the two subtrees are called the *class roots* of this class, and every class root is an H connector. Each class X (except for the root class) attaches to one other unique *parent* class $p(X)$ with two free links. Each link goes from a class root of X to a

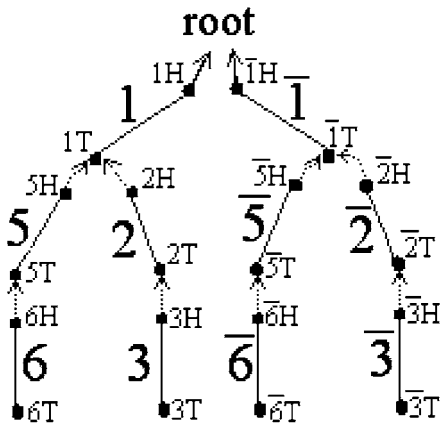

FIG. 1. Edge 1 is the parent of edges 2 and 5. Each pair $(i, \bar{i})$ forms a class. Class 2 attaches to its parent class 1 by linking its class root 2H to join point 1T, and $\bar{2}$H to join point $\bar{1}$T. As a continuing example, edges 4 and $\bar{4}$ will be added later.

distinct connector in $p(X)$. The connectors in $p(X)$ that are linked to are called *join points*. As an example, see Fig. 1.

2.1. Operations on the shadow tree

As the algorithm processes the matrix S, new edges are added to the shadow tree and information about old edges is updated. Three operations are used to update the shadow tree, *edge addition*, *class flipping*, and *class merging*.

An edge is *added* to the shadow tree by creating a single edge class, consisting of the edge and its shadow edge, and then linking both edges to certain connectors in the shadow tree. Both edges of the first class created in the algorithm are linked to the root with fixed links.

A class X can *flip* relative to its parent class $p(X)$ by switching the links that connect X to $p(X)$. A flip does not change any class roots or any join points, but simply switches which of the two class roots links to which of the two join points. See Fig. 2 for an example.

The algorithm may choose to *merge* two classes yielding a larger class. A class X may merge with its parent class $p(X)$, or two classes having the same parent class may merge. No other merges are possible.

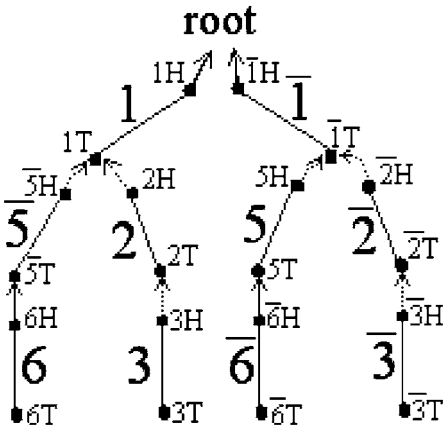

FIG. 2. The result of flipping the class of edges 5 and $\bar{5}$ and flipping the class of edges 6 and $\bar{6}$ in Fig. 1, followed by merging these two classes. Free links are drawn as dotted lines with arrows, while fixed links as solid lines with arrows.

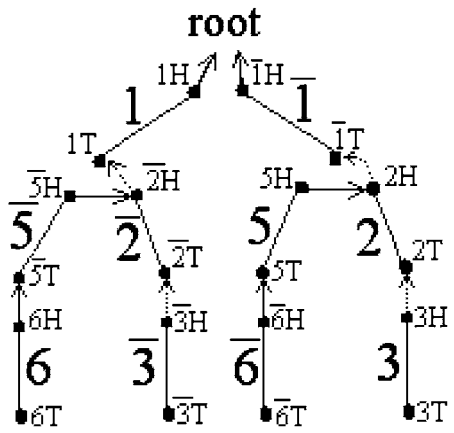

FIG. 3. The result of flipping the class of edges 2 and $\overline{2}$ in Fig. 2, followed by merging it with the class of edges 5 and $\overline{5}$. The class roots of the merged class are 2H and $\overline{2}$H.

In the first case, the free links connecting X to $p(X)$ are changed to fixed links, and the class roots of $p(X)$ become the class roots of the new class. See Fig. 2 for an example. In the second case, when two classes X and X' have the same parent class and *edges that contain class roots of X and X' have same parent edges*, the links from the class roots of X become fixed and are changed to point to the class roots of X' (assuming that column numbers of edges that contain class roots of X' are smaller than those of the class roots of X). After merging, the class roots of X' become the class roots of the new class. See Fig. 3 for an example of this case. Three or more classes can be merged by executing consecutive merges.

The algorithm can *walk up* in the shadow tree by following links from H connectors of tree or shadow edges, until the walk reaches the root. The algorithm can efficiently find class roots and join points of a class by walking up in the shadow tree and checking whether a link encountered is fixed or free.

2.2. Mapping the shadow tree to all PPH solutions

We say that a tree is *contained in* a shadow tree if it can be obtained by flipping some classes in the shadow tree followed by contracting all links and shadow edges. The following is the KEY THEOREM that we establish in this paper. The proof is given in Section 4.5.

Theorem 2.1. *Every PPH solution is contained in the final shadow tree produced by the algorithm. Conversely, every tree contained in the final shadow tree is a distinct PPH solution.*

For example (Fig. 4), by flipping the class of 2, $\overline{2}$, 3 and $\overline{3}$ and then performing the required contractions, we get all PPH solutions for S, which are root(1(2), 3) and root(1(3), 2). Note that flipping the root class results in the same tree. Thus, a final shadow tree with p classes implicitly represents 2^{p-1} PPH solutions.

The KEY THEOREM implies that each class in the final shadow tree encodes a subgraph that is contained in ALL solutions to the PPH problem. In fact, this is true throughout the algorithm. That is, at any point in the algorithm (even in the middle of processing a row of the input), if X is a class in the current shadow tree and G is the graph (consisting of one or two rooted trees) obtained from X by contracting all the links and shadow edges in X, then every solution to the PPH problem contains the (one or) two trees in G.

2.3. Invariant properties

The linear-time PPH algorithm processes the input matrix S one row at a time, starting at the first row. At every step, the algorithm maintains certain properties of the shadow tree which are necessary for the correctness and the running time.

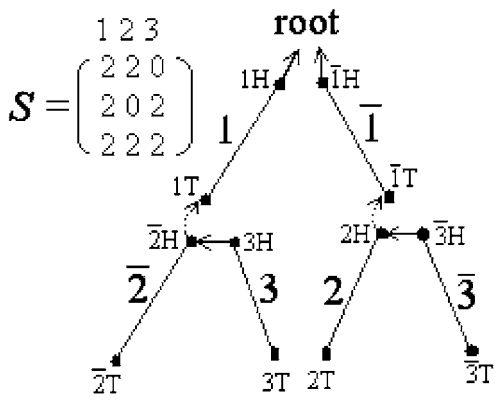

FIG. 4. The final shadow tree after processing the given genotype matrix. It's an implicit representation of all PPH solutions for S.

We define three functions col, te, and se. Function col takes an edge or a connector as input and returns the column number of that edge or the column number of the edge which the connector is part of. Function te (or se) takes a column number or an edge as input and returns the tree edge (or shadow edge, respectively) of that column number or edge. If the input is the root of the shadow tree, then functions col, te, and se each returns the root.

For any column C_i in S, we define the *leaf count* of column C_i as the number of 2's in column C_i plus twice the number of 1's in column C_i. We assume throughout the paper that the columns of S are arranged by decreasing leaf count, with the column containing the largest leaf count on the left.

Theorem 2.2. *The shadow tree has the following invariant properties:*

Property 1. *For any column i in S, the edge labeled by i is in the shadow tree if and only if the shadow edge $\bar{i}$ is also in the shadow tree; i and $\bar{i}$ are in the same class, and are in different subtrees of the class (except for the root class).*

Property 2. *Each class X (except for the root class) attaches to exactly one other class $p(X)$ by two free links, and the two join points j_1 and j_2 are in different subtrees of $p(X)$ unless $p(X)$ is the root class. Links within a class are always fixed links.*

Property 3. *Along any directed path towards the root, the column numbers of the edges (tree or shadow edges) strictly decrease. Also, for any two edges E and E', if E was added to the shadow tree while processing a row k and E' was added when processing a row greater than k, then E' can never be above E on any path to the root in the shadow tree.*

Property 4. *Let X, j_1, and j_2 be as in Property 2. At least one of j_1 and j_2 (say j_1) is the T connector of a tree edge in the parent class. If j_2 is an H connector or the T connector of a shadow edge, then $col(j_2) \leq col(j_1)$.*

Property 5. *For any column C_i, if the parent of $te(C_i)$ (or of $se(C_i)$, respectively) is shadow edge $se(C_j)$, then $te(C_j)$ is on the path from $se(C_i)$ (or $te(C_i)$, respectively) to the root of the shadow tree.*

Property 6. *Two edges that have the same parent in a shadow tree must have been added to the growing shadow tree during the processing of different rows.*

Property 7. *Let TE be a tree edge in the shadow tree, and let SE be its corresponding shadow edge. The union of the edges on the paths from TE and SE to the root of the shadow tree is invariant. Also, the set $\{p(TE), p(SE)\}$ is invariant.*

Proof. It is easy to verify that these properties hold for each tree edge i, shadow edge $\bar{i}$, and the class of edges i and $\bar{i}$, after they are just added to the growing shadow tree by the construction of Procedure NewEntries (see Section 4.4). Next, we prove that the theorem holds after every possible operation to the shadow tree. The only permitted operations that modify the shadow tree are adding new edges, class

flipping, and class merging. Adding new edges to the shadow tree does not change anything originally in the shadow tree. Clearly it will not affect any property. Class flipping switches which class root links to which join point, but it keeps the same roots of the class and the same join points in its parent class. Therefore, these properties hold for edges i, $\bar{i}$, and the class of edges i and $\bar{i}$ after every possible class flipping. Class merging of two classes X_1 and X_2 results in one merged class X, and the class roots and join points of one of classes X_1 and X_2, say X_1, become the class roots and join points of class X. In addition, class merging does not change the parent relation of each edge in the shadow tree. Therefore, these properties hold for edges i, $\bar{i}$, and the class of edges i and $\bar{i}$ after every possible class merging. Thus, these properties are invariant throughout the algorithm. ■

3. SOME DEFINITIONS

We use E_i to denote an edge and C_i to denote a column number (i is an integer between 1 and m). The *class of edge* E_i is defined as the class that contains E_i. The *class root of* E_i is defined as the root of the subtree that contains E_i, in the class of E_i. The class of C_i is defined as the class that contains $te(C_i)$.

For two columns C_i and C_j, $C_j < C_i$ means that column C_j is to the left of column C_i in S. The root is defined as smaller than any column number.

A *2 entry* C_i *in row* k means that the entry at column C_i and row k in S has a value 2. A *new 2 entry* C_i *in row* k means that there is no 2 entry at C_i in rows 1 through $k-1$. An *old 2 entry* C_i *in row* k means there is at least one 2 entry at C_i in rows 1 through $k-1$.

When we say *a PPH solution, restricted to the columns in shadow tree ST*, we mean a tree obtained from a PPH solution after contracting all edges corresponding to columns not in *ST*. We say that a tree T contained in shadow tree *ST* is *in* a PPH solution if T can be obtained from a PPH solution after contracting all edges corresponding to columns not in *ST*. By saying that a column C_i is not in *ST*, we mean that the tree edge and the shadow edge labeled by C_i and $\overline{C_i}$ are not in *ST*.

We define the function *cnt*, which takes a pointer (used in the algorithm) as input and returns the connector to which the pointer points.

4. ALGORITHM

For ease of exposition, in this section, we first describe a linear-time algorithm for the PPH problem where S is assumed to consist of distinct columns which only contain entries of value 0 and 2 and the all-zero sequence is the ancestral sequence in any solution. We will relax these assumptions and solve the general PPH problem in Section 5. The following lemma is immediate (proven by Gusfield [2002]):

Lemma 4.1. *Given S, let T be a solution to the PPH problem for S (if one exists) and let* E_i *be the edge in T labeled by* C_i*, i.e., the edge where site* C_i *mutates. Then the number of leaves in T below* E_i *is exactly the leaf count of column* C_i*. It follows that along any path in T to the root, the successive edges are labeled by columns with strictly increasing leaf counts.*

The algorithm processes the input matrix S one row at a time, starting at the first row. We let $T(k)$ denote the shadow tree produced after processing the first k rows of S. For row $k+1$, the algorithm puts the column numbers of all old 2 entries in row $k+1$ into a list *OldEntryList* and puts column numbers of all new 2 entries in row $k+1$ into a list *NewEntryList*.

The algorithm needs two observations. First, all edges labeled with columns that have 2 entries in row $k+1$ must form two paths to the root in any PPH solution, and no edges labeled with columns that have 0 entries in row $k+1$ can be on either of these two paths. Second, along any path to the root in any PPH solution, the successive edges are labeled by columns with strictly increasing leaf counts (see Lemma 4.1). These two observations are simple, but powerful, and intuitively are the reason why we can achieve linear time, while no such solution exists for the general graph realization problem.

The algorithm processes a row $k+1$ from S using three procedures. The first procedure, *OldEntries*, tries to create two directed paths to the root of $T(k)$ that contain all the tree edges in $T(k)$ corresponding

to columns in OldEntryList by manipulating existing classes in $T(k)$. Those two paths cannot have tree edges in common and may contain some shadow edges. The subgraph defined by those two directed paths is called a *hyperpath*. The process of creating a hyperpath may involve flipping some classes and may also identify classes that need to be merged, fixing the relative position of the edges in the merged class in all PPH solutions. In the second procedure, *FixTree*, the algorithm locates any additional class merges that are required. In the third procedure, *NewEntries*, the algorithm adds the tree and shadow edges corresponding to the columns in NewEntryList and may do additional class merges. The resulting shadow tree is $T(k+1)$.

procedure PPH(S)
{
for $k = 1$ to n {
    Put column numbers of all old 2 entries in row k into OldEntryList;
    Put column numbers of all new 2 entries in row k into NewEntryList;
    Initialize CheckList to an empty list;
    call procedure OldEntries;
    call procedure FixTree;
    call procedure NewEntries;
}}

Procedure OldEntries is divided into two procedures, FirstPath followed by SecondPath. Procedure FirstPath constructs a path (called FirstPath) to the root that consists of tree edges of some column numbers in OldEntryList. The shadow tree produced after this procedure, applied to row $k+1$, is denoted by $T_{FP}(k+1)$.

4.1. Procedure FirstPath for row $k+1$

4.1.1. Procedure FirstPath at a high level. We assume that column numbers in OldEntryList (and other lists used later) are ordered decreasingly, with the largest one, C_i, at the head of the list. By the definition of leaf count, column numbers in OldEntryList are also ordered by increasing leaf count, with C_i having the smallest leaf count. The algorithm performs a front-to-back scan of OldEntryList, starting from C_i, and a parallel walk up in $T(k)$, starting from edge $te(C_i)$. Let C_j denote the next entry in OldEntryList, and let E_p be the parent of $te(C_i)$ in $T(k)$. If E_p and $te(C_i)$ are not in the same class, then let E'_p denote the resulting parent of $te(C_i)$ if we flip the class of C_i. If E'_p is a tree edge and $col(E_p) \leq col(E'_p)$, then the algorithm will flip the class of C_i and set E_p to E'_p (by Property 4 of Theorem 2.2 if E'_p is a shadow tree, then $col(E_p) \geq col(E'_p)$). This class flipping is done to simplify the exposition in the paper and has the effect that the parent of $te(C_i)$ is the tree edge with the larger column number between E_p and E'_p.

The ideal case is that E_p is the tree edge $te(C_j)$, in which case we can move to the next entry in OldEntryList and simultaneously move up one edge in $T(k)$. The ideal case continues as long as the next entries in OldEntryList correspond to the parent edges encountered in the shadow tree and those edges are tree edges. The procedure ends when there is no entry left in OldEntryList, and we move to the root of the shadow tree.

However, there are three cases, besides the ideal case, that can happen. One case is that E_p is a shadow edge, which can only happen when $te(C_i)$ and E_p are in the same class (by Property 4 of Theorem 2.2 and the class flipping above). Then we simply walk past E_p (i.e., let $E_p = p(E_p)$), without moving past entry C_i in OldEntryList. The second case is that E_p is a tree edge (denoted TE_p), but $col(TE_p) < C_j$. This indicates that $te(C_i)$ and $te(C_j)$ can never be on the same path to the root (proven by Lemma 4.3), and the algorithm adds C_j to the head of a list called *CheckList*, to be processed in Procedure SecondPath. The third case is that E_p is a tree edge (denoted TE_p), but $col(TE_p) > C_j$ (and hence $col(TE_p)$ has a 0 entry in row $k+1$). This indicates that edges $te(C_i)$ and TE_p must be on different paths to the root of $T(k+1)$(proven by Lemma 4.2), and the algorithm flips the class that contains $te(C_i)$ to avoid edge TE_p. In that case, the algorithm will also merge the classes containing $te(C_i)$ and TE_p to fix the relative position of those edges in any PPH solution. However, if $te(C_i)$ and TE_p are in the same class when this case occurs, then even flipping the class of $te(C_i)$ will not avoid the problem, and hence the algorithm reports that no PPH solution exists. As an example, see Fig. 5.

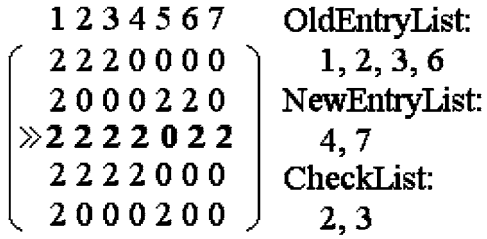

FIG. 5. The shadow tree after processing the first two rows of this genotype matrix is shown in Fig. 1. The shadow tree at the end of Procedure FirstPath for row 3 is shown in Fig. 2. Lists shown are for row 3.

4.1.2. Procedure FirstPath in detail

procedure FirstPath
{
while (there are entries in OldEntryList not processed in this procedure) **do** {
C_i = the largest unprocessed column number in OldEntryList;
C_j = the second largest unprocessed entry in OldEntryList. If C_i is the only unprocessed entry in OldEntryList, then let C_j = root.
/* We use pointer p_1 to point to the end of the FirstPath that is closer to the root. */
Let p_1 point to the H connector of $te(C_i)$;

repeat until encountering an "**exit** Repeat" or "**exit** Algorithm" statement {
/* Use a while loop to achieve the property that if there is a fixed link from $cnt(p_1)$, then it links to a T connector. */
while (there is a fixed link from $cnt(p_1)$ to an H connector) **do** {
Let p_1 point to the H connector which $cnt(p_1)$ links to;
}
if (there is a fixed link from $cnt(p_1)$ to a T connector), **then** {
/* The edge containing $cnt(p_1)$ and its parent edge are in the same class. */
if ($cnt(p_1)$ links to a shadow edge), **then** {
Let p_1 point to the H connector of the shadow edge which $cnt(p_1)$ links to; /* skip the shadow edge */
}
else ($cnt(p_1)$ links to a tree edge), **then** {
TE_p = the tree edge which $cnt(p_1)$ links to;
/* Since $cnt(p_1)$ and TE_p are in the same class, by the definition of a class, TE_p must be on the path from $te(C_i)$ to the root in the shadow tree. */
if ($col(TE_p) == C_j$), **then** {
record that $te(C_i)$ is below TE_p on FirstPath;
mark C_i as processed by this procedure;
exit Repeat;
}
else if ($col(TE_p) < C_j$), **then** {
/* In this case $te(C_j)$ and $te(C_i)$ cannot be on the same path to the root. This is because $te(C_i)$ and TE_p are in the same class, so by the definition of a class no edge can be inserted between them. But placing $te(C_j)$ on a path above TE_p or below $te(C_i)$ would violate Lemma 4.1. So, since $te(C_i)$ is chosen to be on FirstPath, $te(C_j)$ must be on the second path. */
put C_j into CheckList, mark C_j as processed by this procedure;
C_j = the second largest unprocessed entry in OldEntryList. If C_i is the only unprocessed entry in OldEntryList, then let C_j = root;
}

else ($C_j < col(TE_p)$), **then** {
We claim no valid PPH solution exists, report failure and **exit** Algorithm;
/* Because $te(C_i)$ and TE_p are in the same class, TE_p must be on the path from $te(C_i)$ to the root, while $col(TE_p)$ has a 0 entry in this row. */
}
} /* end else ($cnt(p_1)$ links to a tree edge) */
} /* end if (there is a fixed link from $cnt(p_1)$) */

else (there is a free link from $cnt(p_1)$), **then** {
/* $cnt(p_1)$ and its parent are in different classes. Clearly p_1 now points to the class root of $te(C_i)$. */
Let $root_1$ be the class root of $te(C_i)$ and $root_2$ be the class root of $se(C_i)$.
/* Clearly $root_1$ and $root_2$ are different. $root_2$ can be located by following links from $se(C_i)$ to the root, until a free link is encountered. */

if ($root_1$ links to an H connector) or ($root_1$ links to the T connector of a shadow edge), **then** {
flip the class of C_i so that $root_1$ links to the T connector of a tree edge; /* Such a T connector exists by Property 4 of Theorem 2.2. */
}
else if ($root_2$ links to the T connector of a tree edge with a larger column number than that of the tree edge that $root_1$ links to), **then** {
/* When this occurs, $root_1$ links to a T connector of a tree edge. */
flip the class of C_i so that $root_1$ links to the T connector of a tree edge with a larger column number than that of the tree edge that $root_2$ links to;
}
TE_p = the tree edge which $root_1$ links to;
E_q = the edge which $root_2$ links to;

if ($col(TE_p) == C_j$), **then** {
record that $te(C_i)$ is below TE_p on FirstPath;
mark C_i as processed by this procedure;
exit Repeat;
}
else if ($col(TE_p) < C_j$), **then** {
/* We refer to this point in the algorithm as "Sibling Case 1", and say that C_i "places" C_j into CheckList. In this case $te(C_j)$ and $te(C_i)$ cannot be on the same path to the root. This is proven in Lemma 4.3 below. */
put C_j into CheckList, mark C_j as processed by this procedure;
C_j = the second largest unprocessed entry in OldEntryList. If C_i is the only unprocessed entry in OldEntryList, then let C_j = root;
}
else ($C_j < col(TE_p)$), **then** {
/* $col(TE_p)$ has a 0 entry in this row, so TE_p cannot be on the path from $te(C_i)$ to root. */
/* This flip and merge is denoted Flip/Merge Case 1, and is justified in Lemma 4.2. */
flip the class of C_i to walk around TE_p on the FirstPath to the root;
merge the class of C_i with the class of TE_p by setting the links from $root_1$ and $root_2$ as fixed;
}
} /* end else (there is a free link from $cnt(p_1)$) */
} /* end repeat */
}} /* end while, end Procedure FirstPath */

At this point, we prove two lemmas that will be part of our overall proof of correctness of the algorithm.

Lemma 4.2. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm performs a flip/merge in Procedure FirstPath for row $k+1$. Let $T'(k)$ be the shadow tree $T(k)$ after that flip/merge, and note that as a result, $T'(k)$ contains some, but not all, trees contained in $T(k)$. Then, a tree T contained in $T(k)$ is not contained in $T'(k)$ only if T is not in any PPH solution. In looser terms, any tree contained in $T(k)$ that is lost by doing the flip/merge is not in any solution to the PPH problem.*

Proof. When the flip/merge (Flip/Merge Case 1 in the pseudocode) occurs in Procedure FirstPath, column C_i has a 2 entry in row $k+1$, while $col(TE_p)$ has a 0 entry in row $k+1$. So TE_p cannot be on the path from $te(C_i)$ to the root in any PPH solution that explains S. However, before doing the flip/merge, TE_p is on the path from $te(C_i)$ to the root in the shadow tree. So any tree contained in $T(k)$, where the class of C_i is not first flipped relative to the class of TE_p, will not be in any PPH solution. It follows that any tree contained in $T(k)$ that is lost by doing the flip/merge is not in any PPH solution. ■

Lemma 4.3. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. If Sibling Case 1 (in pseudocode of Procedure FirstPath) is reached during the processing of row $k+1$, then $te(C_j)$ and $te(C_i)$ cannot be on the same path to the root in any solution of the PPH problem.*

Proof. Let $root_1$ and $root_2$ be as given in the algorithm when Sibling Case 1 is reached. Let E_r denote the edge that contains $root_1$. By the actions of the algorithm before Sibling Case 1 is reached, either $root_2$ links to an H connector or to a T connector of a shadow edge, or both $root_1$ and $root_2$ link to T connectors of tree edges (three cases). By definition, TE_p (the parent edge of E_r at this point) denotes the edge that $root_1$ is linked to, and E_q denotes the edge that $root_2$ is linked to. By Property 4 of Theorem 2.2, in the first and second cases above, and by the explicit action of the algorithm in the third case, it follows that $col(E_q) \leq col(TE_p)$. At Sibling Case 1, $col(TE_p) < C_j < C_i$ by the explicit actions of the algorithm. Hence, $col(E_q) \leq col(TE_p) < C_j < C_i$.

By the lemma assumption, every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Hence, it suffices to prove that no matter how the classes of $T(k)$ are flipped, edges $te(C_i)$ and $te(C_j)$ are never on the same path to the root in $T(k)$. By Property 2 of Theorem 2.2, the parent of E_r is either edge TE_p or edge E_q no matter how the classes of $T(k)$ are flipped. Hence, $te(C_j)$ can never be the parent of E_r in any way that the classes of $T(k)$ are flipped. But by Property 3 of Theorem 2.2 and the fact that $col(E_q) \leq col(TE_p) < C_j$, $te(C_j)$ cannot be above TE_p or E_q. Similarly, since $C_j < C_i$, $te(C_j)$ cannot be below $te(C_i)$ in any way that the classes of $T(k)$ are flipped. Since E_r and $te(C_i)$ are in the same class, by the definition of a class $te(C_j)$ cannot be in between $te(C_i)$ and E_r. No matter how the classes of $T(k)$ are flipped, edges $te(C_i)$ and $te(C_j)$ are never on the same path to the root in $T(k)$. Hence, $te(C_i)$ and $te(C_j)$ cannot be on the same path to the root in any PPH solution. ■

Lemmas 4.2 and 4.3 together essentially say that when Procedure FirstPath takes any "nonobvious" action, either flipping and merging classes or putting a column number into Checklist, it is "forced" to do so. The algorithm may perform other class flips and merges in other procedures described later. The correctness of those actions will be proven by lemmas similar to Lemma 4.2 and 4.3.

4.2. Procedure SecondPath for row $k+1$

4.2.1. Procedure SecondPath at a high level. At the end of Procedure FirstPath, any columns in OldEntryList, whose corresponding tree edges are not on FirstPath, have been placed into CheckList. In the simple case, Procedure SecondPath tries to construct a second path (called SecondPath) to the root that contains all the tree edges in $T(k)$ corresponding to columns in CheckList. In general, Procedure SecondPath constructs a SecondPath and may modify the FirstPath constructed previously. The goal is that FirstPath and SecondPath together contain all the tree edges in $T(k)$ corresponding to columns in OldEntryList and the two paths have no tree edges in common. The shadow tree produced after this procedure is denoted by $T_{SP}(k+1)$, and it contains a hyperpath for row $k+1$.

Procedure SecondPath: Let C_i be the largest column number in CheckList, and let C_j denote the next entry in CheckList. The algorithm performs a front-to-back scan of CheckList, starting from column C_i, and a parallel walk up in $T_{FP}(k+1)$, starting from edge $te(C_i)$. The parent of $te(C_i)$ in $T_{FP}(k+1)$, denoted E_p, is obtained in the same way as in Procedure FirstPath.

The rest of the algorithm is similar to Procedure FirstPath, with two major differences. First, the second case in Procedure FirstPath (when E_p is a tree edge, and $col(E_p) < C_j$) now causes the algorithm to determine that no PPH solution exists. Second, the third case in Procedure FirstPath (when E_p is a tree edge, denoted TE_p, and $col(TE_p) > C_j$) now indicates two possible subcases. In the first subcase, if $col(TE_p)$ has a 0 entry in row $k+1$, then as in Procedure FirstPath, the algorithm determines that edges $te(C_i)$ and TE_p must be on different paths to the root of $T(k+1)$ and it does a flip/merge as in Procedure FirstPath. In the second subcase, if $col(TE_p)$ is in OldEntryList, but not in CheckList, then it must be that TE_p is on FirstPath. Therefore, SecondPath is about to use a tree edge that is already on FirstPath, and hence some action must be taken to avoid this conflict. In this case, there is a direct way to complete the construction of SecondPath. The algorithm calls Procedure DirectSecondPath and ends Procedure SecondPath.

Procedure DirectSecondPath decides whether TE_p must stay on FirstPath, or whether it must be on SecondPath, or whether it can be on either path to the root (it can be shown that only these three cases yield valid PPH solutions). The procedure also performs the appropriate class flips and merges to ensure that TE_p stays on the path chosen by the algorithm regardless of later class flips, in the first two cases, or that FirstPath and SecondPath have no tree edge in common, in the third case.

Procedure DirectSecondPath: Recall that $te(C_i)$ is the tree edge on SecondPath whose parent edge is TE_p. Let TE_{pc} denote the tree edge on FirstPath whose parent edge is TE_p at the end of Procedure FirstPath. The following tests determine which path to put TE_p on.

Test1: If after flipping the class of C_i and the class of TE_{pc}, TE_p is either on both FirstPath and SecondPath, or on none of them, then no hyperpath exists for row $k+1$ and hence no solution exists for the PPH problem.

Test2: If TE_p is in the same class as TE_{pc} (respectively, $te(C_i)$), then TE_p must be on FirstPath (respectively, SecondPath).

Test3: First try to flip the class of C_i and the class of TE_{pc} so that TE_p is on FirstPath (respectively, SecondPath), but not on SecondPath (respectively, FirstPath). If the try succeeds and there does not exist a hyperpath in the shadow tree after the flip, then TE_p must be on SecondPath (respectively, FirstPath). See Fig. 6 for an example.

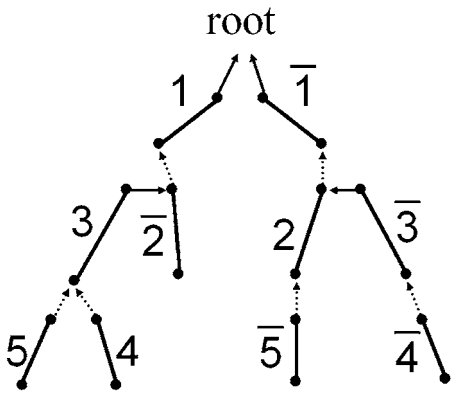

FIG. 6. Suppose this figure shows the shadow tree $T(k)$ for some matrix S. If the OldEntryList of row $k+1$ contains column indices 1, 3, 4, and 5, then in Procedure DirectSecondPath for row $k+1$ the FirstPath consists of tree edges 5, 3, and 1, TE_p is edge 3, TE_{pc} is edge 5, $C_i = 4$, and the algorithm determines that TE_p must be on the path from TE_{pc} to the root in any PPH solution (only flag2 will be set in Procedure DirectSecondPath in detail). If the OldEntryList of row $k+1$ contains column indices 1, 2, 3, 4, and 5, then in Procedure DirectSecondPath for row $k+1$ the FirstPath consists of tree edges 5, 3, and 1, TE_p is edge 3, TE_{pc} is edge 5, $C_i = 4$, and the algorithm determines that TE_p must be on the path from $te(C_i)$ to the root in any PPH solution (only flag1 will be set in Procedure DirectSecondPath in detail).

If the test results indicate that TE_p must be on both FirstPath and SecondPath, then no hyperpath exists for row $k+1$ and hence no solution exists for the PPH problem.

If the test results indicate that TE_p must be on FirstPath (respectively, SecondPath), then flip the class of C_i and/or the class of TE_{pc} so that TE_p is on FirstPath (respectively, SecondPath), but not on SecondPath (respectively, FirstPath), and merge the classes of TE_p, C_i, and TE_{pc}.

If the test results show that TE_p can be on either path, then for concreteness, flip either the class of C_i or the class of TE_{pc} so that TE_p is on FirstPath, but not on SecondPath, and merge the class of C_i with the class of TE_{pc}.

As an example, the shadow tree at the end of Procedure SecondPath for row 3 of the matrix in Fig. 5 is shown in Fig. 3. In this example, the algorithm determines that tree edge 1 can be on either FirstPath or SecondPath.

4.2.2. Procedure SecondPath in detail

```
procedure SecondPath
{
while (there are entries in CheckList not processed in this procedure) do {
    C_i = the largest unprocessed unprocessed column number in CheckList;
    C_j = the second largest unprocessed entry in CheckList. If C_i is the only unprocessed entry in
      CheckList, then let C_j = root.
    /* Pointer p_1 is used to locate the end of the SecondPath that is closer to the root. */
    Let p_1 point to the H connector of te(C_i);

    repeat until encountering an "exit " statement {
        /* Use a while loop to achieve the property that if there is a fixed link from cnt(p_1), it links to a
          T connector. */
        while (there is a fixed link from cnt(p_1) to an H connector) do {
            Let p_1 point to the H connector which cnt(p_1) links to;
        }
        if (there is a fixed link from cnt(p_1) to a T connector), then {
            /* The edge containing cnt(p_1) and its parent edge are in the same class. */
            if (cnt(p_1) links to a shadow edge ), then {
                Let p_1 point to the H connector of the shadow edge which cnt(p_1) links to; /* skip the
                  shadow edge */
            }
            else (cnt(p_1) links to a tree edge), then {
                TE_p = the tree edge which cnt(p_1) links to;
                /* Since cnt(p_1) and TE_p are in one class, by the definition of a class, TE_p must be on
                  the path from te(C_i) to root in the shadow tree. */
                if (col(TE_p) == C_j), then {
                    mark C_i as processed by this procedure;
                    exit Repeat;
                }
                else if (col(TE_p) < C_j or col(TE_p) is not in OldEntryList), then {
                    We claim no PPH solution exists, report failure and exit Algorithm;
                    /* We refer to the above step as Step NoSib 1 and justify it in Lemma 4.5. */
                }
                else (C_j < col(TE_p) and col(TE_p) is in OldEntryList), then {
                    /* col(TE_p) is not in CheckList but is in OldEntryList, so TE_p is on FirstPath. Second-
                      Path cannot also include it and some action must be taken to change one of the two
                      paths. That change is determined in Procedure DirectSecondPath. */
                    call procedure DirectSecondPath;
                    exit procedure SecondPath;
```

```
            }
        } /* end else (cnt(p1) links to a tree edge) */
    } /* end if (there is a fixed link from cnt(p1)) */

    else (there is a free link from cnt(p1) ), then {
        /* cnt(p1) and its parent are in different classes. Clearly p1 points to the class root of te(Ci). */
        Let root1 be the class root of te(Ci) and root2 be the class root of se(Ci).
        if (root1 links to an H connector) or (root1 links to the T connector of a shadow edge), then {
            flip the class of Ci so that root1 links to the T connector of a tree edge; /* Such a
              T connector exists by Property 4 of Theorem 2.2. */
        }
        else if (root2 links to the T connector of a tree edge with a larger column number than that of
          the tree edge that root1 links to), then {
            /* When this occurs, root1 links to a T connector of a tree edge. */
            flip the class of Ci so that root1 links to the T connector of a tree edge with a larger
              column number than that of the tree edge that root2 links to;
        }
        TEp = the tree edge which root1 links to;
        Eq = the edge which root2 links to; /* col(Eq) ≤ col(TEp) */

        if (col(TEp) == Cj), then {
            mark Ci as processed by this procedure;
            exit repeat;
        }
        else if (col(TEp) < Cj), then {
            We claim no PPH solution exists, report failure and exit algorithm;
            /* We refer to the above step as Step NoSib 2 and justify it in Lemma 4.4 */
        }
        else if (Cj < col(TEp), and col(TEp) is not in OldEntryList), then {
            /* This flip and merge is another instance of Flip/Merge Case 1, and is justified by
              Lemma 4.2. */
            flip the class of Ci to walk around TEp on the SecondPath to the root;
            merge the class of Ci with the class of TEp by setting the links from root1 and root2 as
              fixed links;
        }
        else (Cj < col(TEp), and col(TEp) is in OldEntryList), then {
            /* col(TEp) is not in CheckList but is in OldEntryList, so TEp is on FirstPath. SecondPath
              cannot also include it and some action must be taken to change one of the two paths.
              That change is determined in Procedure DirectSecondPath. */
            call procedure DirectSecondPath;
            exit procedure SecondPath;
        }
    } /* end else (there is a free link from cnt(p1)) */
  } /* end repeat */
}} /* end while, end Procedure SecondPath */

procedure DirectSecondPath
{
/* This procedure gives a direct way to finish the construction of the hyperpath for row k + 1 in situations
  where it is called. All variables in this procedure initially have same values as in Procedure SecondPath
  before this procedure is called. */

Mark Ci as processed by Procedure SecondPath;
```

Find the tree edge (denoted TE_{pc}) which is recorded as below TE_p on FirstPath (in Procedure FirstPath) and let C_{pc} denote its column number; /* TE_{pc} may not presently be a child of TE_p. C_i, C_{pc}, and $col(TE_p)$ are all in OldEntryList. */

if ($te(C_i)$ and TE_{pc} are in the same class, and have the same class root), **then** no PPH solution exists, because no hyperpath can contain all the edges $te(C_i)$, TE_{pc}, and TE_p. **exit** Algorithm;

if ($te(C_i)$ and TE_p are in the same class), **then** {
  Set flag1; /* indicating that TE_p must be on the path from $te(C_i)$ to the root in any PPH solution */
}
else if (the set of tree edges on the path from $root_2$ to the root of the shadow tree is NOT identical to the set of tree edges of all unprocessed column numbers in CheckList), **then** {
  Set flag1; /* See Fig. 6 for an example. */
}
if ($te(C_{pc})$ and TE_p are in the same class), **then** {
  Set flag2; /* indicating that TE_p must be on the path from TE_{pc} to the root in any PPH solution */
}
else {
  The class of C_{pc} has two class roots. One of them links to TE_p. Let $root_{pc1}$ be that class root, and let $root_{pc2}$ be the other class root.
  /* $root_{pc1}$ and $root_{pc2}$ can be located by following links from TE_{pc} and $se(C_{pc})$ until two free links are encountered. */
  /* The next condition can be checked in linear time because edges on the path from $root_{pc2}$ to the root of the shadow tree are labeled by decreasing column numbers. */
  if (the set of tree edges on the path from $root_{pc2}$ to the root of the shadow tree is NOT identical to the set of tree edges of all unprocessed column numbers in CheckList), **then** {
    Set flag2; /* See Fig. 6 for an example. */
  }
}

if (both flag1 and flag2 are set) **then** {
  No PPH solution exists (see Lemma 4.7), report failure and **exit** Algorithm;
}
else if (flag1 is set and flag2 is not set) **then** {
  /* This flip and merge is denoted Flip/Merge Case 2, and is justified by Lemma 4.8. */
  flip, if necessary, the class of C_{pc}, so that TE_p is not on the path from TE_{pc} to the root;
  merge the class of C_i with the class of TE_p and the class of C_{pc} by setting links from $root_1$, $root_2$, $root_{pc1}$, $root_{pc2}$ as fixed;
}
else if (flag2 is set and flag1 is not set) **then** {
  /* This flip and merge is another instance of Flip/Merge Case 2, and is justified by Lemma 4.8. */
  flip, if necessary, the class of C_{pc}, so that TE_p is on the path from TE_{pc} to the root;
  flip the class of C_i, so that TE_p is not on the path from $te(C_i)$ to the root;
  merge the class of C_i with the class of TE_p and the class of C_{pc} by setting links from $root_1$, $root_2$, $root_{pc1}$, $root_{pc2}$ as fixed;
}
else (neither flag1 nor flag2 are set, i.e., neither $te(C_i)$ nor TE_{pc} are forced to be on the same path as TE_p to the root), **then** {
  /* In this case the parent edges of the class roots of classes C_i and C_{pc} are the same (see Lemma 4.9) */
  /* This flip and merge is denoted Flip/Merge Case 3, and is justified by Lemma 4.10. */
  flip the class of C_i to walk around TE_p on the second path to the root;
  merge the class of C_i with the class of C_{pc} by removing links from $root_{pc1}$ and $root_{pc2}$, and adding a fixed link from $root_{pc1}$ to $root_2$, and a fixed link from $root_{pc2}$ to $root_1$;
} } /* end Procedure DirectSecondPath */

Lemma 4.4. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. If Step NoSib 2 (in pseudocode of Procedure SecondPath) is reached during the processing of row $k+1$, then there is no solution to the PPH problem.*

Proof. By assumption, every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Hence, it suffices to prove that if Step NoSib 2 is reached in Procedure SecondPath, then no tree contained in $T(k)$ is a PPH solution, restricted to the columns in $T(k)$.

Let $root_1$ and $root_2$ be as given in the algorithm when Step NoSib 2 is reached. Let E_r denote the edge that contains $root_1$. If Step NoSib 2 is reached, then the parent of E_r is either edge TE_p or edge E_q no matter how the classes of $T(k)$ are flipped, E_r and $te(C_i)$ are in the same class, and $col(E_q) \leq col(TE_p) < C_j < C_i$. These are exactly the same conditions where Lemma 4.3 applies. Thus, no matter how the classes of $T(k)$ are flipped, edges $te(C_i)$ and $te(C_j)$ are never on the same path to the root in $T(k)$.

Let C_1 be the column that places C_i into CheckList in Procedure FirstPath. We know that $C_i < C_1$, and by the proof of Lemma 4.3 edges $te(C_i)$ and $te(C_1)$ are never on the same path to the root in $T(k)$, no matter how the classes of $T(k)$ are flipped. If C_1 also places C_j into CheckList in Procedure FirstPath, then by the proof of Lemma 4.3 edges $te(C_j)$ and $te(C_1)$ are never on the same path to the root in $T(k)$, no matter how the classes of $T(k)$ are flipped. Thus, there cannot be a hyperpath containing $te(C_j)$, $te(C_i)$, and $te(C_1)$ in $T(k)$, and hence in any PPH solution. However, there are two entries in columns C_j, C_i, and C_1 in row $k+1$ of S, so every PPH solution must have a hyperpath that contains those three edges. Hence, there is no PPH solution in this case.

Now suppose C_j is placed into CheckList in Procedure FirstPath by column C_2 ($C_2 \neq C_1$). Note that neither C_1 nor C_2 is in CheckList. We know that $C_j < C_2$, and by the proof of Lemma 4.3 edges $te(C_j)$ and $te(C_2)$ are never on the same path to the root in $T(k)$, no matter how the classes of $T(k)$ are flipped. We can observe from Procedure FirstPath that the set of columns placed into CheckList by C_1 (or C_2) is contiguous with C_1 (respectively, C_2) in OldEntryList. Then, since $C_j < C_2$, $C_i < C_1$, and $C_j < C_i$, it follows that $C_j < C_2 < C_i < C_1$. When Step NoSib 2 is reached, $col(E_q) \leq col(TE_p) < C_j < C_2 < C_i$. For column C_2, these are exactly the same conditions where Lemma 4.3 applies. Thus, no matter how the classes of $T(k)$ are flipped, edges $te(C_2)$ and $te(C_i)$ are never on the same path to the root. Thus, there cannot be a hyperpath containing $te(C_j)$, $te(C_i)$, and $te(C_2)$ in $T(k)$, and hence in any PPH solution. However, there are two entries in columns C_j, C_i, and C_2 in row $k+1$ of S, so every PPH solution must have a hyperpath that contains those three edges. Hence, there is no PPH solution in this case. ■

Lemma 4.5. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. If Step NoSib 1(in pseudocode of Procedure SecondPath) is reached during the processing of row $k+1$, then there is no solution to the PPH problem.*

Proof. When Step NoSib 1 is reached in Procedure SecondPath, edge TE_p is the tree edge which $cnt(p_1)$ links to. Let E_c be the edge that contains $cnt(p_1)$. Edge E_c may or may not be the same as $te(C_i)$. However, we know that edges TE_p, E_c, and $te(C_i)$ are in the same class, and E_c is on the path from $te(C_i)$ to the root in $T(k)$. By the definition of a class, TE_p will be on the path from $te(C_i)$ to the root in $T(k)$ no matter how the classes of $T(k)$ are flipped.

Edge TE_p is in $T(k)$, so if $col(TE_p)$ is not in OldEntryList, then there must be a 0 entry in $col(TE_p)$. But, there is a 2 entry in column C_i in row $k+1$, so when $col(TE_p)$ is not in OldEntryList, there can be no solution to the PPH problem.

Now, consider the case that $col(TE_p)$ is in OldEntryList. In that case, it must be that $col(TE_p) < C_j < C_i$. Next we prove that $te(C_i)$ and $te(C_j)$ cannot be on the same path to the root in any PPH solution. By the lemma assumption, every PPH solution, restricted to the columns in $T(k)$, is contained in $T(k)$. Hence, it suffices to prove that no matter how the classes of $T(k)$ are flipped, edges $te(C_i)$ and $te(C_j)$ are never on the same path to the root in $T(k)$. Since TE_p is on the path from $te(C_i)$ to the root in $T(k)$ and both edges are in the same class, edge $te(C_j)$ cannot be between TE_p and $te(C_i)$ no matter how the class of $T(k)$ is flipped. But by Property 3 of Theorem 2.2 and the fact that $col(TE_p) < C_j$, $te(C_j)$ cannot be above TE_p. Similarly, since $C_j < C_i$, $te(C_j)$ cannot be below $te(C_i)$ in any way that the classes of $T(k)$ are flipped. Hence, $te(C_i)$ and $te(C_j)$ cannot be on the same path to the root in $T(k)$ and in any PPH solution.

Further, C_j and C_i are both in Checklist. These are the same facts established in the second paragraph of the proof of Lemma 4.4 and the only facts needed in the third and fourth paragraphs of that proof. Hence, the remainder of the proof of Lemma 4.5 is identical to third and fourth paragraphs of the proof of Lemma 4.4. ■

Lemma 4.6. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. If Procedure DirectSecondPath sets flag1 (respectively, flag2), then TE_p must be on the path from $te(C_i)$ (respectively, TE_{pc}) to the root in any PPH solution.*

Proof. We will explicitly prove the lemma for flag1. The proof for flag2 is symmetric. By the lemma assumption, if $te(C_i)$ and TE_p are in the same class when flag1 is set, then TE_p must be on the path from $te(C_i)$ to the root in any PPH solution.

Next we prove that if TE_p and $te(C_i)$ are not in the same class when flag1 is set (and so certainly are not in the same class in $T(k)$), then TE_p must be on the path from $te(C_i)$ to the root in any PPH solution. By the lemma assumption, every PPH solution, restricted to the columns in $T(k)$, is contained in $T(k)$. So, we can prove the lemma by proving that TE_p must be on the path from $te(C_i)$ to the root in any way that the classes of $T(k)$ can be flipped to obtain a PPH solution, restricted to the columns in $T(k)$. Equivalently, in order for there to be a hyperpath in $T(k)$ for row $k+1$, the classes of $T(k)$ must be flipped so that TE_p is on the path from $te(C_i)$ to the root.

By Property 1 of Theorem 2.2, the two roots of the class of C_i in $T(k)$ link to join points, call them j_1 and j_2, in a single parent class. Since TE_p is the parent of $root_1$ (the class root of $te(C_i)$), when Procedure DirectSecondPath is called, either j_1 or j_2 is the T connector of TE_p (say j_1). The class of TE_p remains the parent class of C_i, and j_1 and j_2 remain the join points of the class of C_i, no matter how the classes of $T(k)$ are flipped. Again by Property 1 of Theorem 2.2, the roots of the class of TE_p link to a single parent class. Let J be the union of the edges on the paths from j_1 and j_2 to the root in $T(k)$. It follows that after any flip of classes in $T(k)$, J remains the union of the edges on the resulting two paths from j_1 and j_2 to the root of the resulting shadow tree. The paths may change, but the union of the edges on those paths cannot change from what it is in $T(k)$.

Let CJ be the set of classes of $T(k)$ that contain the edges in set J. Since Procedure SecondPath works bottom up in $T(k)$, at the time the algorithm calls Procedure DirectSecondPath, no class in CJ has been flipped in Procedure SecondPath, and so the path from j_1 to the root is exactly the subpath of FirstPath from j_1 to the root, and the path from j_2 has not changed since the end of Procedure FirstPath. Hence, if the required hyperpath for row $k+1$ goes through both j_1 and j_2, then the tree edges in J must be exactly the tree edges on FirstPath from j_1 to the root, together with the edges that remain on CheckList at the point flag1 is set. But flag1 is set under the condition that the set of tree edges on the path from $root_2$ (and hence from j_2) to the root of the current shadow tree is not identical to the set of tree edges of all the column numbers in CheckList. Hence, when flag1 is set, J does not have exactly the required set of tree edges, and so the hyperpath for row $k+1$ cannot go through both j_1 and j_2 in order for PPH solutions to exist.

Now, both $col(TE_p)$ and C_i are old entries in row $k+1$, so any hyperpath for row $k+1$ must go through the tree edges TE_p and $te(C_i)$. Further, in any choice of flipping classes in $T(k)$, either TE_p is on the path from $te(C_i)$ to the root or the class root of $te(C_i)$ will link to j_2. In the latter case, the hyperpath will go through both j_1 and j_2. But as shown above, that is impossible when flag1 is set in order for PPH solutions to exist, and this completes the proof of the lemma. ■

Lemma 4.7. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Then (in Procedure DirectSecondPath) TE_p must be on the path from exactly one of $te(C_i)$ and TE_{pc} to the root in any PPH solution, and no valid PPH solution exists if Procedure DirectSecondPath sets both flag1 and flag2.*

Proof. First we prove that no matter how the classes of $T(k)$ are flipped, edges $te(C_i)$ and TE_{pc} cannot be on the same path to the root in $T(k)$. Suppose $C_i < C_{pc}$. The proof for the case that $C_{pc} < C_i$ is symmetric. By Property 3 of Theorem 2.2, TE_{pc} cannot be on the path from $te(C_i)$ to the root in $T(k)$. So we need to prove only that $te(C_i)$ cannot be on the path from TE_{pc} to the root in $T(k)$ in any choice of

flipping classes of $T(k)$. The proof is by contradiction. Suppose that $te(C_i)$ can be on the path from TE_{pc} to the root in $T(k)$ in some way of flipping classes. Edge TE_{pc} is recorded as below TE_p in Procedure FirstPath, and TE_p is the first tree edge on the path from TE_{pc} to the root (on FirstPath). Let E'_p denote the parent of the class root of TE_{pc} after flipping the class of TE_{pc}. By Property 3 of Theorem 2.2 and the fact that $col(TE_p) < C_i$, $te(C_i)$ cannot be above TE_p. Hence $te(C_i)$ is either the same as E'_p or on the path from E'_p to the root. However, as given in the proof for Lemma 4.3, TE_p has a larger column number than that of edge E'_p. This contradicts that $col(TE_p) < C_i$.

Now all of $col(TE_p)$, C_i, and C_{pc} are in OldEntryList, so all three of the *tree* edges TE_p, $te(C_i)$, and TE_{pc} must be on the hyperpath for row $k+1$ in every PPH solution, restricted to the columns in $T(k)$. But a hyperpath must consist of two paths to the root that do not contain tree edges in common. That is impossible when tree edge TE_p is on the paths from both $te(C_i)$ and TE_{pc} to the root, or when TE_p is on neither of those paths, given that $te(C_i)$ and TE_{pc} cannot be on the same path to the root in $T(k)$. Therefore, TE_p must be on exactly one of the paths: $te(C_i)$ to root and TE_{pc} to root, in any PPH solution.

By Lemma 4.6 if Procedure DirectSecondPath sets both flag1 and flag2, then in any PPH solution TE_p must be on the path from both $te(C_i)$ and TE_{pc} to the root. This causes a contradiction, and hence no valid PPH solution exists. ■

Lemma 4.8. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm performs a flip/merge in Flip/Merge Case 2 when processing row $k+1$. Then any tree contained in $T(k)$ that is lost by doing the flip/merge is not in any solution to the PPH problem.*

Proof. By Lemma 4.7, TE_p must be on the path to the root from either $te(C_i)$ or TE_{pc} (denoted E) but not both, in any PPH solution. In Flip/Merge Case 2, exactly one of flag1 and flag2 is set. By Lemma 4.6, we know which one of $te(C_i)$ and TE_{pc} is E. After performing a flip/merge in Flip/Merge Case 2, any tree contained in $T(k)$ that is lost is a tree in which TE_p is not on the path from E to the root, and hence not in any PPH solution. ■

The analysis done so far for Procedure DirectSecondPath implies that when exactly one of flag1 or flag2 is set, or when both are set, the algorithm takes a forced action, either a forced flip and merge, or a forced conclusion that no PPH solution exists. What remains is the case when neither flag1 nor flag2 are set. In this case, it seems reasonable that either the class of C_i or the class of C_{pc} should be flipped, but not both, and that the choice is arbitrary. Furthermore, after the flip, the two classes should be merged to maintain their relative position. This is in fact true, and is proven by Lemma 4.10. The next lemma establishes a fact which is needed to prove Lemma 4.10. Its proof needs familiarity with Procedure NewEntries in Section 4.4 and is given in Section 4.5.

Lemma 4.9. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose that neither flag1 nor flag2 are set in Procedure DirectSecondPath. Then the roots of the classes of C_i and C_{pc} in $T(k)$ have the same parents.*

Lemma 4.10. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm performs a flip/merge in Flip/Merge Case 3 when processing row $k+1$. Then any tree contained in $T(k)$ that is lost by doing the flip/merge is not in any solution to the PPH problem.*

Proof. By Lemma 4.7, TE_p must be on the path from exactly one of $te(C_i)$ and TE_{pc} to the root in any PPH solution. In Flip/Merge Case 3, neither flag1 nor flag2 are set. Therefore TE_p can be on the path from either one of $te(C_i)$ and TE_{pc} to the root. Any tree contained in $T(k)$ that is lost by doing the flip/merge is a tree in which TE_p is not on the path from exactly one of $te(C_i)$ and TE_{pc} to the root, and hence not in any solution to the PPH problem.

By Lemma 4.9, the roots of the classes of C_i and C_{pc} in $T(k)$ have the same parents, and hence these two classes can be merged as described in Flip/Merge Case 3. ■

4.3. Procedure FixTree for row $k+1$

Procedure FixTree finds and merges more classes, if necessary, to remove trees contained in $T_{SP}(k+1)$ that are not in any PPH solutions. It first extends SecondPath with shadow edges whose column numbers are in OldEntryList of row $k+1$. The subgraph defined by FirstPath and the extended SecondPath is called an *extended hyperpath*; it contains the hyperpath found earlier. By utilizing the extended hyperpath, the algorithm can determine which additional classes need to be merged. The shadow tree produced after this procedure is denoted by $T_{FT}(k+1)$.

procedure FixTree
{
/* First we find two edges TE_1 and E_2 which are two ends of the extended hyperpath. */
TE_1 = the tree edge of the largest column number in OldEntryList, i.e., the lowest edge of FirstPath; if OldEntryList is empty, then let TE_1 = root;
Let $SE_1 = se(TE_1)$;

TE_t = the tree edge of the largest column number in OldEntryList whose tree edge is not on FirstPath, i.e., the lowest edge of SecondPath; if no such tree edge exists, then let TE_t = root;

Find a maximal path from TE_t toward leaves in $T_{SP}(k+1)$ consisting of shadow edges whose column numbers are in OldEntryList. We prove in Lemma 4.11 that such a maximal path is unique.
E_2 = the edge that is the lower end of the maximal path found in the previous step; if the path does not contain any edge, then let $E_2 = TE_t$;
/* $col(TE_1) \geq col(E_2)$ */

/* We can check whether TE_1 and E_2 are in the same class by checking whether they have same class roots. */
while (TE_1 and E_2 are in different classes) and (E_2 is not the parent of the class root of SE_1) **do** {
    /* Merge-class cases below are justified by Lemmas 4.13 and 4.14. */
    /* If $E_2 \neq TE_t$, then $te(E_2)$ is on FirstPath and col(the class root of TE_1) > col(the class root of $te(E_2)$) $\geq col$(the class root of TE_t). */
    if (col(the class root of TE_1) > col(the class root of TE_t)), **then**
        merge the class of TE_1 with its attaching class;
    else
        merge the class of TE_t with its attaching class;
}}

See Fig. 7 for an example. The proof of the next lemma needs familiarity with Procedure NewEntries in Section 4.4 and is is given in Section 4.5.

Lemma 4.11. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Then the maximal path found in Procedure FixTree for row $k+1$ is unique.*

Lemma 4.12. *Let ST denote the shadow tree at any stage of the algorithm. Let r_1 and r_2 denote the class roots of class X in ST. Let r_1' and r_2' denote the class roots of class X' in ST. If $max\{col(r_1), col(r_2)\} > min\{col(r_1'), col(r_2')\}$, then flipping class X does not change the position of any edge in class X' in ST.*

Proof. Without loss of generality, we assume that $max\{col(r_1), col(r_2)\} = col(r_1)$ and $min\{col(r_1'), col(r_2')\} = col(r_1')$. Flipping a class X in the shadow tree ST affects only the class itself and all classes that directly or indirectly attach to it. Since X and X' are different classes in ST, we need to prove only that class X' does not attach to class X directly or through a series of classes. The proof is by contradiction. Suppose class X' attaches to class X directly or through a series of classes. Then there is choice of class

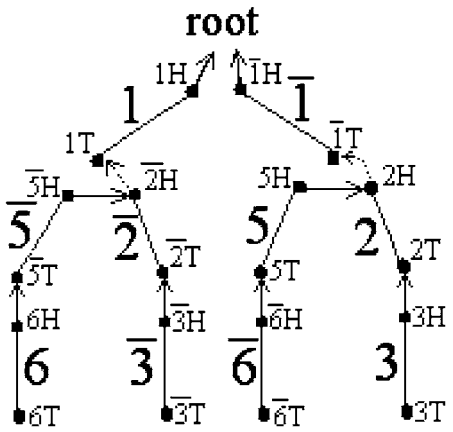

FIG. 7. The shadow tree at the end of Procedure FixTree for row 3 of the matrix in Fig. 5. In Procedure FixTree for this example, $TE_1 = 6$, $TE_t = E_2 = 3$, and the class of edge 3 is merged with the class of edge 2. The class roots of the merged class are 2H and $\bar{2}$H.

flips in the shadow tree such that connector r_1 is on the path from connector r_1' to the root of the shadow tree. But that is contradictory to Property 3 of Theorem 2.2 given that $col(r_1) > col(r_1')$. ■

The proof of the next lemma needs familiarity with Procedure NewEntries in Section 4.4 and is is given in Section 4.5.

Lemma 4.13. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm merges the class of TE_1 with its attaching class in Procedure FixTree for row $k+1$. Then any tree contained in $T(k)$ that is lost by doing the class merge is not in any solution to the PPH problem.*

Lemma 4.14. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm merges the class of TE_t with its attaching class in Procedure FixTree for row $k+1$. Then any tree contained in $T(k)$ that is lost by doing the class merge is not in any solution to the PPH problem.*

Proof. Let TE_1, SE_1, TE_t, and E_2 be the same as in Procedure FixTree. Suppose that the algorithm merges the class of TE_t with its attaching class in Procedure FixTree when processing row $k+1$. This will happen only when TE_1 and E_2 are in different classes, E_2 is not the parent of the class root of SE_1, and col(the class root of TE_1) $\leq col$(the class root of TE_t). The last condition indicates that E_2 is the same as TE_t, because if E_2 is different from TE_t and hence is a shadow edge whose column number is in OldEntryList, then $te(E_2)$ is on FirstPath (in a different class than TE_1) and we must have col(the class root of TE_1) $> col$(the class root of $te(E_2)$) $\geq col$(the class root of TE_t). By Lemma 4.12, the last condition also indicates that flipping the class of TE_t does not affect the class of TE_1 and hence does not change the positions of TE_1 and SE_1 in the shadow tree. We can also deduce that TE_t cannot be on the path from either SE_1 or TE_1 to the root from the last condition. By Property 3 of Theorem 2.2, none of TE_1 or SE_1 can be on the path from TE_t to the root. So TE_1 and TE_t cannot be on the same path to the root in the shadow tree.

Next we do a case analysis. We will prove that in all three cases there does not exist a hyperpath for row $k+1$ if we flip the class of TE_t, and hence any tree contained in $T(k)$ that is lost by doing the class merge is not in any solution to the PPH problem. Let SE_t denote $se(TE_t)$. Let r_1 and r_2 be the class roots of TE_t and SE_t, respectively. Let j_1 and j_2 be the join points which r_1 and r_2 link to. By Property 4 of Theorem 2.2, at least one of j_1 and j_2 is the T connector of a tree edge.

In the first case, j_2 is not the T connector of a tree edge, and hence j_1 must be the T connector of a tree edge, say TE_{j1}. Since TE_t is on SecondPath, TE_{j1} must be on SecondPath and $col(TE_{j1})$ is in OldEntryList. If we flip the class of TE_t, TE_t has to be on FirstPath in order for a hyperpath for row $k+1$

to exist. Edge TE_1 is on FirstPath, but TE_1 and TE_t cannot be on the same path to the root as proven above. Therefore, there does not exist a hyperpath for row $k+1$ if we flip the class of TE_t. In the second case, j_2 is the T connector of a tree edge, say TE_{j2}, whose column number is not in OldEntryList. If we flip the class of TE_t, TE_{j2} is on the path from TE_t to the root, which causes no hyperpath for row $k+1$ to exist. In the third case, j_2 is the T connector of a tree edge, say TE_{j2}, whose column number is in OldEntryList. In this subcase, TE_{j2} is on FirstPath and hence on the path from TE_1 to the root. If we flip the class of TE_t, then TE_{j2} is on the path from both TE_1 and TE_t to the root, because *flipping the class of TE_t does not affect the class of TE_1*. As proven above, TE_1 and TE_t cannot be on the same path to the root. So no hyperpath for row $k+1$ exists.

We have proven that in all three cases any tree contained in $T(k)$ that is lost by doing the class merge is not in any solution to the PPH problem. The proof is complete. ■

4.4. Procedure NewEntries for row $k+1$

4.4.1. Procedure NewEntries at a high level. Procedure NewEntries creates and adds edges corresponding to columns in NewEntryList of row $k+1$ to $T_{FT}(k+1)$. Ideally, it tries to attach new edges to the two ends of the extended hyperpath found in Procedure FixTree. If some new edges cannot be added in this way, the algorithm finds places to attach them. It then merges more classes, if necessary, so that there are two directed paths to the root in $T(k+1)$ containing all the tree edges corresponding to the columns that have 2 entries in row $k+1$, no matter how classes are flipped.

Procedure NewEntries: If NewEntryList is empty, then exit this procedure. Otherwise arrange column numbers in NewEntryList from left to right increasingly, with the largest one on the right end of the list.

Create edges $te(C_i)$ and $se(C_i)$ for each C_i in NewEntryList. Create two free links pointing from the H connector of $te(C_i)$ (respectively, $se(C_i)$) to the T connector of $te(C_j)$ (respectively, $se(C_j)$), for each C_i and its left neighbor C_j in NewEntryList.

Let C_h denote the smallest column number in NewEntryList. At this point, each new edge is attached, using a free link, to one other edge, except for $te(C_h)$ and $se(C_h)$. The algorithm attaches them according to two cases. Let TE_1, TE_t, and E_2 be the same as in Procedure FixTree.

In the first case, when $col(TE_1) < C_h$, $te(C_h)$ and $se(C_h)$ are attached to the two ends of the extended hyperpath. It creates a free link pointing from the H connector of $te(C_h)$ to the T connector of TE_1. It creates a free link pointing from the H connector of $se(C_h)$ to the T connector of E_2, if E_2 is in the class of TE_1, and otherwise, to a connector in the class of TE_1 whose parent is E_2.

In the second case, when $col(TE_1) > C_h$, by Property 3 of Theorem 2.2, none of $te(C_h)$ and $se(C_h)$ can attach to TE_1. If $col(TE_t) > C_h$, then no PPH solution exists no matter where new edges are attached; otherwise, the algorithm finds two edges (TE'_1 and E'_2) to attach $te(C_h)$ and $se(C_h)$, as follows.

Let TE'_1 denote the tree edge of the largest column number in OldEntryList that is less than C_h. Let TE'_t denote the tree edge of the largest column number in OldEntryList that is less than C_h and not on the path from TE'_1 to the root. If TE'_1 or TE'_t does not exist, then let it be the root.

Similarly to Procedure FixTree, the algorithm finds a maximal path from TE'_t toward the leaves in $T_{FT}(k+1)$, consisting of shadow edges whose column numbers are in OldEntryList and less than C_h. Let E'_2 denote the edge that is at the lower end of the maximal path.

If TE'_1 is on the path from TE_1 (respectively, E_2) to the root, then create a free link pointing from the H connector of $se(C_h)$ (respectively, $te(C_h)$) to the T connector of TE'_1, and create a free link pointing from the H connector of $te(C_h)$ (respectively, $se(C_h)$) to the T connector of E'_2 if E'_2 is in the class of TE'_1, otherwise to a connector in the class of TE'_1 whose parent is E'_2.

If there are column numbers in NewEntryList that are larger than $col(TE_1)$, then let C_t denote the smallest one among them ($C_h < col(TE_1) < C_t$). Edge $se(C_t)$ is a new edge that has been attached to an edge by the algorithm. As a special case, the algorithm changes the link from the H connector of $se(C_t)$ to point to the T connector of TE_1.

All new edges are added to $T_{FT}(k+1)$ according to cases 1 and 2. The algorithm then merges the class of C_h with the classes of column numbers in NewEntryList that are less than $col(TE_1)$ and merges the class of C_h with the classes of column numbers in OldEntryList that are larger than C_h.

As an example, Fig. 8 shows the shadow tree T produced by the algorithm after processing the first three rows of the matrix S in Fig. 5. Tree T is also the final shadow tree for S. It can be verified that Theorem 2.1 holds for S and T.

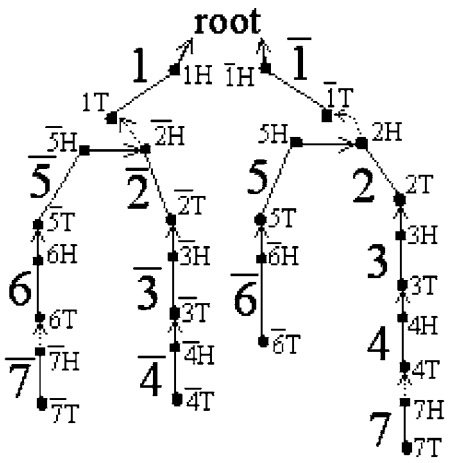

FIG. 8. The shadow tree at the end of Procedure NewEntries for row 3 of the matrix in Fig. 5. In Procedure NewEntries for this example, $TE_1 = 6$, $TE_t = E_2 = 3$, $C_h = 4$, $C_t = 7$, $TE'_1 = 3$, $TE'_t = 1$, $E'_2 = \overline{3}$. Note that edge $\overline{7}$ links to edge 6 instead of edge $\overline{4}$.

4.4.2. Procedure NewEntries in detail

procedure NewEntries
{
if (NewEntryList is empty), **then exit procedure** NewEntries; Otherwise arrange column numbers in NewEntryList from left to right increasingly, with the largest one on the right end of the list.

for each column number C_i in NewEntryList, create edges $te(C_i)$ and $se(C_i)$;
for each C_i and its left neighbor C_j in NewEntryList, let $p(te(C_i)) = te(C_j)$, $p(se(C_i)) = se(C_j)$;

Let TE_1, SE_1, and E_2 be the same edges as in Procedure FixTree;
C_h = the smallest column number in NewEntryList;

if ($col(TE_1) < Ch$), **then** {
    /* Now new edges can be attached directly to the two ends of the extended hyperpath. */
    if (TE_1 and E_2 are in the same class), **then** {
        /* This is denoted Add-new-edges Case 1. */
        $p(te(C_h)) = TE_1$, $p(se(C_h)) = E_2$;
    }
    else (TE_1 and E_2 are in different classes), **then** {
        cr = the class root of SE_1;
        /* By Lemma 4.15, E_2 is the parent of cr. */
        /* This is denoted Add-new-edges Case 2, in which E_2 is the parent of $se(C_h)$. */
        $p(te(C_h)) = TE_1$, let the H connector of $se(C_h)$ link to cr;
    }
}
else ($C_h < col(TE_1)$), **then** {
    /* In this case, $te(C_h)$ cannot link to the T connector of TE_1 by Property 3 of Theorem 2.2. We use the same way as to find TE_1 and E_2 in Procedure FixTree to find two edges TE'_1 and E'_2 whose column numbers are less than C_h. */
    TE'_1 = the tree edge of the largest column number in OldEntryList that is less than C_h; if no such tree edge exists, then let TE'_1 = root;
    $SE'_1 = se(TE'_1)$;
    TE'_t = the tree edge of the largest column number in OldEntryList that is less than C_h, whose tree edge is not on the path from TE'_1 to the root; if no such tree edge exists, then let TE'_t = root;

Find a maximal path from TE'_t toward leaves in $T_{FT}(k)$ consisting of shadow edges whose column numbers are in OldEntryList and less than C_h. By Lemma 4.11, such a maximal path is unique.
E'_2 = the edge that is the lower end of the maximal path found in the previous step. If the path does not contain any edge, then let E'_2 be the same as TE'_t.
/* $col(TE'_1) \geq col(E'_2)$ */

if (TE'_1 and E'_2 are in the same class), **then** {
  if (TE'_1 is on the path from TE_1 to the root), **then** {
    $p(te(C_h)) = E'_2$, $p(se(C_h)) = TE'_1$; /* Add-new-edges Case 3 */
  }
  else (TE'_1 is on the path from E_2 to the root), **then** {
    $p(te(C_h)) = TE'_1$, $p(se(C_h)) = E'_2$; /* Add-new-edges Case 4 */
  }
}
else (TE'_1 and E'_2 are in different classes), **then** {
  cr' = the class root of SE'_1;
  /* By Lemma 4.15, E'_2 is the parent of cr'. */
  if (TE'_1 is on the path from TE_1 to the root), **then** {
    /* Add-new-edges Case 5, in which E'_2 is the parent of $te(C_h)$. */
    $p(se(C_h)) = TE'_1$, let the H connector of $te(C_h)$ link to cr'.
  }
  else (TE'_1 is on the path from E_2 to the root), **then** {
    /* Add-new-edges Case 6, in which E'_2 is the parent of $se(C_h)$. */
    $p(te(C_h)) = TE'_1$, let the H connector of $se(C_h)$ link to cr'.
  }
}

C_t = the smallest column number in NewEntryList that is larger than $col(TE_1)$;
If no column number in NewEntryList is larger than $col(TE_1)$, then let $C_t = col(TE_1)$;
/* The following two merge-class cases are justified by Lemma 4.16. */
merge the class of C_h with classes of column numbers in NewEntryList that are less than C_t;
merge the class of C_h with classes of column numbers in OldEntryList that are larger than C_h;
if ($C_t > col(TE_1)$), **then** let $p(se(C_t)) = TE_1$; /* Add-new-edges Case 7 */
} /* end else ($C_h < col(TE_1)$) */
} /* end Procedure NewEntries */

Lemma 4.15. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Then E_2 is the parent of cr, and E'_2 is the parent of cr' in Procedure NewEntries for row $k+1$.*

Proof. Recall that cr is the class root of SE_1 and cr' is the class root of SE'_1. At the end of Procedure FixTree for row $k+1$, if TE_1 and E_2 are not in the same class, then E_2 is the parent of cr in $T_{FT}(k+1)$. In Procedure NewEntries for row $k+1$ where this lemma applies, TE_1 and E_2 are not in the same class. So E_2 must be the parent of cr.

We next prove that in Procedure NewEntries if TE'_1 and E'_2 are not in the same class, E'_2 must be the parent of cr'. Let E_{r2} be the edge that contains cr'. Assume that TE'_1 is on the path from E_2 to the root in Procedure NewEntries. (The proof for the case where TE'_1 is on the path from TE_1 to the root is similar.) The class of TE'_1 is on the path from E_2 to the root and hence is on the path from SE_1 to the root. Therefore, the class of SE'_1 is on the path from TE_1 to the root; i.e., cr' is on the path from TE_1 to the root. So the parent of cr', say E_{pr2}, is also on the path from TE_1 to the root. We claim that $col(E_{pr2})$ is in OldEntryList. Suppose it is not; then since E_{pr2} is on the path from TE_1 to the root, it cannot be a tree edge. So E_{pr2} must be a shadow edge. Edge E_{pr2} is the parent of E_{r2}. Let $E'_{r2} = te(E_{r2})$ if E_{r2} is a shadow edge, or let $E_{r2'} = se(E_{r2})$ if E_{r2} is a tree edge. By Property 5 of Theorem 2.2, $te(E_{pr2})$ is on the path from E'_{r2} to the root. Since E'_{r2} and TE'_1 are in the same class and have the same class root, $te(E_{pr2})$

is also on the path from TE'_1 to the root, which is contradictory to $col(E_{pr2})$ not in OldEntryList. Thus, $col(E_{pr2})$ is in OldEntryList. Since E_{pr2} is on the path from TE_1 to the root, Procedure NewEntries will either choose E_{pr2} as E'_2 or choose an edge in the class of SE'_1 as E'_2. Since TE'_1 and E'_2 are not in the same class, E_{pr2} must be chosen as E'_2 by the algorithm, and hence E'_2 is the parent of cr'. ■

Lemma 4.16. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm performs a merge-class case in Procedure NewEntries for row $k+1$. Then any tree contained in $T(k)$ that is lost by doing the merge-class case is not in any PPH solution.*

Proof. One critical observation is that running Procedure NewEntries where NewEntryList contains l column numbers is equivalent to running Procedure NewEntries l times where each time NewEntryList contains one increasing column number in the original NewEntryList, starting from the smallest one in the original NewEntryList. Therefore we need to prove the lemma only in the case that NewEntryList of row $k+1$ contains one column number, C_h.

If $col(TE_1) < C_h$ (Add-new-edges Cases 1, 2, and 7), then no merge-class case happens. The merge-class case only happens when $C_h < col(TE_1)$ (Add-new-edges Cases 3-6). In these cases, let C'_h be any column number in OldEntryList of row $k+1$ that is larger than C_h. By the definition of a PPH solution there are two paths to the root (with no tree edge in common) which pass through tree edges corresponding to all columns in OldEntryList of row $k+1$ plus C_h in any PPH solution. We call such two paths $path_1$ and $path_2$. Paths $path_1$ and $path_2$ cannot have tree edges in common. By Property 3 of Theorem 2.2, $te(C'_h)$ cannot be on the path from $te(C_h)$ to the root on either $path_1$ or $path_2$. Since C'_h is an old entry and C_h is a new entry in row $k+1$, there exists at least one row in S where C'_h has a 2 entry and C_h has a 0 entry. So $te(C_h)$ cannot be on the path from $te(C'_h)$ to the root on either $path_1$ or $path_2$ in any PPH solution. Therefore, $te(C_h)$ and $te(C'_h)$ must be on different paths between $path_1$ and $path_2$ in any PPH solution. In Procedure NewEntries, $te(C_h)$ and $se(C_h)$ are added to the shadow tree in such a way that there are two paths to the root (with no tree edge in common) which pass through tree edges corresponding to all columns in OldEntryList of row $k+1$ plus C_h, and $te(C_h)$ and $te(C'_h)$ are on different paths between $path_1$ and $path_2$. Any tree T_l contained in $T(k)$ that is lost by doing the merge-class case is a tree where either $te(C_h)$ and $te(C'_h)$ are both on one of $path_1$ and $path_2$, or there do not exist two paths to the root in T_l (with no tree edge in common) which pass through tree edges corresponding to all columns in OldEntryList of row $k+1$ plus C_h. Therefore T_l is not in any PPH solution. ■

4.5. Correctness and efficiency

For each row of S, the algorithm performs a fixed number of scans of the entries in that row, and in parallel, a fixed number of walk-ups in the shadow tree. There are some steps in the algorithm that require a traversal of the shadow tree (finding a maximal path from an edge, for example), but such operations happen at most once in each procedure, and hence at most once in the processing of each row. However, they can actually be implemented efficiently without traversing the shadow tree (we omit the details). It takes constant time to scan each entry in S, or to walk up one edge in the shadow tree. Each flip or merge is associated with an edge in a walk, and each flip or merge is implemented in constant time. Hence, the time for each row is $O(m)$, and the total time bound is $O(nm)$, where n and m are the number of rows and the number of columns in S.

In the rest of this section, we give proofs for lemmas and theorems that have not been proven in the previous sections.

Lemma 4.9. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose that neither flag1 nor flag2 are set in Procedure DirectSecondPath. Then the roots of the classes of C_i and C_{pc} in $T(k)$ have the same parents.*

Proof. The lemma is trivially true if $te(C_i)$ and TE_{pc} are in the same class. Otherwise, they are in different classes, and since Procedure DirectSecondPath was called, edge TE_p is the first tree edge on the path from TE_{pc} to the root on FirstPath (in Procedure FirstPath) and the first tree edge on the path from $te(C_i)$ to the root on SecondPath. The class of TE_p, call it X, is the parent class of both the classes of

C_i and C_{pc}, while $root_1$ and $root_2$ are the class roots of class C_i, and $root_{pc1}$ and $root_{pc2}$ are the class roots of class C_{pc}. It is clear the TE_p is the parent of both $root_1$ and $root_{pc1}$. Now suppose that $root_2$ and $root_{pc2}$ do not have the same parent edge.

Suppose that one of the paths from $root_2$ and $root_{pc2}$, say from $root_2$, to the root contains a tree edge TE that is not on the path from $root_{pc2}$ to the root in $T(k)$. If $col(TE)$ is not in CheckList, then flag1 should have been set, and if $col(TE)$ is in CheckList, then flag2 should have been set. But neither flag was set, so if either path contains a tree edge, then the two paths must intersect and any tree edge on one of the paths must be in CheckList and must be above the point of intersection. It follows that both paths contain only shadow edges (or the path(s) contain no edges) below their point of intersection.

The two paths must intersect at or before the root of the shadow tree. If the two paths have common edge(s), let E be the first common edge on the two paths from $root_2$ and $root_{pc2}$ to the root. Otherwise, let E denote the root of the shadow tree. We claim that the column number of every (shadow) edge on the paths from $root_2$ and $root_{pc2}$ to E is in OldEntryList. That is, for any edge E'' on those paths, $col(E'')$ has value 2 in row $k+1$ (and certainly it has value 2 in some row before $k+1$ since E'' is in $T(k)$). We prove this explicitly for $root_2$; the proof for $root_{pc2}$ is symmetric. Let SE' be the first (shadow) edge on the path from $root_2$, and assume SE' is not E since otherwise there are no edges from $root_2$ before E, and the claim is is vacuously true.

We now prove that $col(SE')$ is in OldEntryList. Let E_{r2} denote any edge in the class of C_i whose parent is SE'. The simplest such case is that the H connector of E_{r2} is $root_2$, and $root_2$ links to the T connector of SE'. However, this need not be the case, by the definition of "parent," and E_{r2} could be connected to the T connector of SE' via a chain of H connectors. Let $E'_{r2} = te(E_{r2})$ if E_{r2} is a shadow edge, and let $E'_{r2} = se(E_{r2})$ if E_{r2} is a tree edge. By Property 5 of Theorem 2.2 (a simple case analysis based on whether E'_{r2} is a tree or a shadow edge), $te(SE')$ is on the path from E'_{r2} to the root. Edge $te(C_i)$ is in the same class as E_{r2}, and the two edges have different roots in that class, so by Property 1 of Theorem 2.2, $te(C_i)$ and E'_{r2} are in the same class and have the same class root. So $te(SE'))$ is on the path from $te(C_i)$ to the root (and from TE_p to the root), at the point when Procedure DirectSecondPath is called. Hence, $te(SE')$ must be a tree edge on FirstPath, and so $col(SE')$ must be in OldEntryList.

Consider the path from SE' towards E, and recall that all the edges on that path before E are shadow edges. By Property 1 of Theorem 2.2, SE' and $te(SE')$ are not together on any path to the root. Moreover, by repeatedly using Property 5 of Theorem 2.2, all of the tree edges corresponding to shadow edges on the path from SE' to E are on a single path, and $te(SE')$ is the lowest of those tree edges. Edge SE' is in class X or in a class that is an ancestor of X, so by Property 1 of Theorem 2.2, $te(SE')$ is also in X or above. Since Procedure SecondPath works bottom-up in the shadow tree, Procedure SecondPath has not flipped class X or any class that is an ancestor of class X, and no edge in those classes is on SecondPath. Similarly, since edge TE_p is on FirstPath, when Procedure DirectSecondPath is called, the entire path from $root_1$ is on FirstPath. It follows that $te(SE')$ cannot be on SecondPath. Since $col(SE')$ is in OldEntryList, $te(SE')$ must be on FirstPath, and the entire path from $te(SE')$ must be on FirstPath. Therefore, the column number of every tree edge on that path is in OldEntryList. This proves the claim that the columns numbers of all the shadow edges on the path from SE' to E must be in OldEntryList, and their corresponding tree edges must be on FirstPath.

Next, under the continuing assumption that E is not both $p(root_2)$ and $p(root_{pc2})$, there are two cases to consider: either E is none of $p(root_2)$ and $p(root_{pc2})$, or it is one of them.

In the first case, E is the parent of two distinct shadow edges whose column numbers are in OldEntryList. We call those two shadow edges SE_{ii} and SE_{jj}, and assume that $col(SE_{jj}) < col(SE_{ii})$. Let TE_{ii} denote $te(SE_{ii})$, and let TE_{jj} denote $te(SE_{jj})$. As proven above, TE_{ii} and TE_{jj} are on FirstPath, i.e., the path from $root_1$ to the root. Since $col(SE_{jj}) < col(SE_{ii})$, by Property 3 of Theorem 2.2, TE_{jj} must be on the path from TE_{ii} to the root in $T(k)$. Since SE_{ii} and SE_{jj} have the same parent E in the shadow tree, by Property 6 of Theorem 2.2, those two edges were added to the shadow tree during the processing of different rows. Suppose SE_{ii} and TE_{ii} were added to the shadow tree during the processing of row k', $k' < k+1$. Since TE_{jj} is on the path from TE_{ii} to the root, by Property 3 of Theorem 2.2, SE_{jj} and TE_{jj} must have been added to the shadow tree before SE_{ii} was, and so are in $T(k'-1)$.

Procedure NewEntries for row k' finds two edges $TE_{k'1}$ and $E_{k'2}$ that are two ends of an extended hyperpath for row k' (see Section 4.3 for the definition of an *extended hyperpath*). By Property 7 of Theorem 2.2, the union of the edges on the paths from SE_{ii} and TE_{ii} to the root of the shadow tree is

invariant. Since E and TE_{jj} are on paths from SE_{ii} and TE_{ii} to the root in the shadow tree, before doing Flip/Merge Case 3, both E and TE_{jj} are on the hyperpath from $TE_{k'1}$ to $E_{k'2}$ in Procedure NewEntries for row k'. Therefore, $col(TE_{jj})$ is in OldEntryList for row k'.

By Property 7 of Theorem 2.2, the set {p(TE_{ii}), p(SE_{ii})} is invariant. Consider the parent of TE_{ii} before Flip/Merge Case 3, and call it E_{ip}. Then the set {p(TE_{ii}), p(SE_{ii})} is $\{E_{ip}, E\}$. By Property 7 of Theorem 2.2, E_{ip} and E are the $TE_{k'1}$ and $E_{k'2}$ in Procedure NewEntries for row k'. Edge E_{ip} is on the path from TE_{ii} to TE_{jj}. So $col(E_{ip}) \geq col(TE_{jj}) > col(E)$. Therefore, E_{ip} is the $TE_{k'1}$, and E is the $E_{k'2}$ in Procedure NewEntries for row k', which indicates that E is the edge that is the lower end of the maximal path found in Procedure NewEntries for row k'. By Property 7 of Theorem 2.2, the set {p(TE_{jj}), p(SE_{jj})} is invariant. Since TE_{jj} is on the hyperpath from $TE_{k'1}$ to $E_{k'2}$ in Procedure NewEntries for row k', E is the parent of SE_{jj} at that time. Thus, SE_{jj} instead of E should have been found as the lower end of the maximal path in Procedure NewEntries for row k'. That is a contradiction.

In the second case, E is either $p(root_2)$ or $p(root_{pc2})$. Assume E is $p(root_2)$. (The proof for $E = p(root_{pc2})$ is symmetric.) Let E_{r2} denote any edge (in the class of C_i) whose parent is E. Let $E'_{r2} = te(E_{r2})$ if E_{r2} is a shadow edge, and let $E'_{r2} = se(E_{r2})$ if E_{r2} is a tree edge. Edge E is the parent of E_{r2} and a shadow edge on the path from $root_{pc2}$ to the root, say SE_{jj}, whose column numbers is in OldEntryList. Let TE_{jj} denote $te(SE_{jj})$. As proven above, TE_{jj} is on FirstPath, i.e., the path from $root_1$ to the root. So TE_{jj} is on the path from E'_{r2} to the root. Therefore, $col(SE_{jj}) < col(E_{r2})$. (The rest of the proof is similar to the proof for the first case by considering E_{r2} as the SE_{ii} above.)

Since E_{r2} and SE_{jj} have the same parent E in the shadow tree, by Property 6 of Theorem 2.2, those two edges were added to the shadow tree during the processing of different rows. Suppose E_{r2} and E'_{r2} were added to the shadow tree during the processing of row k', $k' < k+1$. Since TE_{jj} is on the path from E'_{r2} to the root before performing Flip/Merge Case 3, we can deduce that SE_{jj} and TE_{jj} are in $T(k'-1)$. Procedure NewEntries for row k' finds two edges $TE_{k'1}$ and $E_{k'2}$, which are two ends of an extended hyperpath for row k'. By Property 7 of Theorem 2.2, the set {p(E_{r2}), p(E'_{r2})} is invariant. Consider the parent of E'_{r2} before Flip/Merge Case 3, and call it E_{ip}. Then the set {p(E_{r2}), p(E'_{r2})} is $\{E_{ip}, E\}$. By Property 7 of Theorem 2.2, E_{ip} and E are the $TE_{k'1}$ and $E_{k'2}$ in Procedure NewEntries for row k'. Edge E_{ip} is on the path from E'_{r2} to TE_{jj}. So $col(E_{ip}) \geq col(TE_{jj}) > col(E)$. Therefore, E_{ip} is the $TE_{k'1}$, and E is the $E_{k'2}$ in Procedure NewEntries for row k', which indicates that E is the edge that is the lower end of the maximal path found in Procedure NewEntries for row k'. By Property 7 of Theorem 2.2, the set {p(TE_{jj}), p(SE_{jj})} is invariant. Since TE_{jj} is on the hyperpath from $TE_{k'1}$ to $E_{k'2}$ in Procedure NewEntries for row k', E is the parent of SE_{jj} at that time. Thus, SE_{jj} instead of E should have been found as the lower end of the maximal path in Procedure NewEntries for row k'. That is a contradiction.

We find a contradiction if the roots of the classes of C_i and C_{pc} in $T(k)$ do not have the same parents. The proof is complete. ■

Lemma 4.11. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Then the maximal path found in Procedure FixTree for row $k+1$ is unique.*

Proof. We prove the lemma by contradiction. Suppose there are two maximal paths from TE_t toward leaves in $T_{SP}(k+1)$ that consist of shadow edges whose column numbers are in OldEntryList. We call these two paths $path_1$ and $path_2$, and these paths must intersect at or before TE_t. If the two paths have common edge(s) before TE_t, let E be the first common edge. Otherwise, let E be the same as TE_t. There are two shadow edges below E whose column numbers are in OldEntryList on $path_1$ and $path_2$. We name the two shadow edges as SE_{ii} and SE_{jj}, and assume that $col(SE_{jj}) < col(SE_{ii})$. Let TE_{ii} denote $te(SE_{ii})$, and let TE_{jj} denote $te(SE_{jj})$. In $T_{SP}(k+1)$, there is a hyperpath that passes through tree edges corresponding to all columns in OldEntryList of row $k+1$. The hyperpath consists of two paths to the root that contain no tree edges in common. In $T_{SP}(k+1)$, every tree edge whose column number is in OldEntryList is either on the path from TE_1 to the root, or on the path from TE_t to the root. Since $col(SE_{ii})$ and $col(SE_{jj})$ are in OldEntryList and TE_{ii} and TE_{jj} are not on the path from TE_t to the root, they must be on the path from TE_1 to the root. Since $col(SE_{jj}) < col(SE_{ii})$, by Property 3 of Theorem 2.2, TE_{jj} must be on the path from TE_{ii} to the root in $T_{SP}(k+1)$. Since SE_{ii} and SE_{jj} have the same parent E in the shadow tree, by Property 6 of Theorem 2.2, those two edges were added to the shadow tree during the processing of different rows. Suppose SE_{ii} and TE_{ii} were added to the shadow tree during the processing

of row k', $k' < k+1$. Since TE_{jj} is on the path from TE_{ii} to the root, by Property 3 of Theorem 2.2, SE_{jj} and TE_{jj} must have been added before SE_{ii}, and so are in $T(k'-1)$.

Procedure NewEntries for row k' finds two edges $TE_{k'1}$ and $E_{k'2}$, which are two ends of the extended hyperpath for row k'. Since E and TE_{jj} are on paths from SE_{ii} and TE_{ii} to the root in $T_{SP}(k+1)$, by Property 7 of Theorem 2.2, both E and TE_{jj} are on the extended hyperpath from $TE_{k'1}$ to $E_{k'2}$ in Procedure NewEntries for row k'. By Property 7 of Theorem 2.2, the set $\{\mathrm{p}(TE_{ii}), \mathrm{p}(SE_{ii})\}$ is invariant. Consider the parent of TE_{ii} in $T_{SP}(k+1)$, and call it E_{ip}. Then the set $\{\mathrm{p}(TE_{ii}), \mathrm{p}(SE_{ii})\}$ is $\{E_{ip}, E\}$. By Property 7 of Theorem 2.2, E_{ip} and E are the $TE_{k'1}$ and $E_{k'2}$ in Procedure NewEntries for row k'. Edge E_{ip} is on the path from TE_{ii} to TE_{jj}. So $col(E_{ip}) \geq col(TE_{jj}) > col(E)$. Therefore, E_{ip} is the $TE_{k'1}$, and E is the $E_{k'2}$ in Procedure NewEntries for row k'. So E is the lower end of the maximal path found in Procedure NewEntries for row k'. By Property 7 of Theorem 2.2, the set $\{\mathrm{p}(TE_{jj}), \mathrm{p}(SE_{jj})\}$ is invariant. Since TE_{jj} is on the extended hyperpath from $TE_{k'1}$ to $E_{k'2}$ in Procedure NewEntries for row k', E is the parent of SE_{jj} at that time. Thus SE_{jj} instead of E should have been found as the lower end of the maximal path in Procedure NewEntries for row k'. That is a contradiction. We find a contradiction if the maximal path found in Procedure FixTree for row $k+1$ is not unique. The proof is complete. ■

Lemma 4.13. *Assume that every PPH solution, restricted to the columns in the shadow tree $T(k)$, is contained in $T(k)$. Suppose the algorithm merges the class of TE_1 with its attaching class in Procedure FixTree for row $k+1$. Then any tree contained in $T(k)$ that is lost by doing the class merge is not in any solution to the PPH problem.*

Proof. Let TE_1, SE_1, TE_t, and E_2 be the same as in Procedure FixTree. Let r_1 and r_2 be the class roots of TE_1 and SE_1, respectively. Let j_1 and j_2 be the join points which r_1 and r_2 link to. The paths from SE_1 and E_2 to the root must intersect at or before the root of the shadow tree. Let E be the first common edge or the root if they intersect at the root.

Suppose that the algorithm merges the class of TE_1 with its attaching class in Procedure FixTree when processing row $k+1$. This will happen only when TE_1 and E_2 are in different classes, E_2 is not the parent of r_2, and col(the class root of TE_1) > col(the class root of TE_t). The last condition indicates that a class flipping of the class of TE_1 does not affect the class of TE_t and hence does not change the position of TE_t in the shadow tree (Lemma 4.12). Next we do a case analysis. We prove that in all three cases there does not exist a hyperpath for row $k+1$ if we flip the class of TE_1, and hence any tree contained in $T(k)$ that is lost by doing the class merge is not in any solution to the PPH problem.

In the first case, TE_t is not on the path from SE_1 to the root. By Property 4 of Theorem 2.2, at least one of j_1 and j_2 is the T connector of a tree edge. We next prove that there does not exist a hyperpath for row $k+1$ if we flip the class of TE_1. The proof has three subcases. In subcase 1, j_2 is not the T connector of a tree edge, and hence j_1 must be the T connector of a tree edge, say TE_{j1}. Since TE_1 is on FirstPath, $col(TE_{j1})$ must be in OldEntryList, and TE_{j1} is on FirstPath too. If we flip the class of TE_1, TE_1 has to be on SecondPath in order for a hyperpath for row $k+1$ to exist. Edge TE_t is on SecondPath and not on the path from SE_1 to the root before flipping, and TE_t is not on the path from TE_1 to the root after flipping, because *flipping the class of TE_1 does not affect the class of TE_t*. We also know that TE_1 cannot be on the path from TE_t to the root by Property 3 of Theorem 2.2. Therefore, there does not exist a hyperpath for row $k+1$ if we flip the class of TE_1 in this subcase. In subcase 2, j_2 is the T connector of a tree edge, say TE_{j2}, whose column number is not in OldEntryList. If we flip the class of TE_1, TE_{j2} is on the path from TE_1 to the root, which causes no hyperpath for row $k+1$ to exist. In subcase 3, j_2 is the T connector of a tree edge, say TE_{j2}, whose column number is in OldEntryList. Edge TE_{j2} is on SecondPath and hence on the path from TE_t to the root. If we flip the class of TE_1, then TE_{j2} is on the path from both TE_1 and TE_t to the root. As proven above, TE_1 and TE_t cannot be on the same path to the root. So no hyperpath for row $k+1$ exists.

In the second case, TE_t is on the path from SE_1 to the root, and E is the parent of r_2 before merging the class of TE_1 with its attaching class. If TE_t is the same as E_2, then E must be the same as E_2, which is contradictory to that E_2 is not the parent of r_2. So E_2 must be a shadow edge whose column number is in OldEntryList and be different from E. We know that if there are any edges on the path from E_2 to TE_t, these edges are shadow edges whose column numbers are in OldEntryList. Since TE_t is on the path from SE_1 to the root, TE_t must be on the path from E to the root. Therefore, every edge on the path from

E_2 to E, including E_2, is a shadow edge whose column number is in OldEntryList. The tree edge of each such shadow edge must be on FirstPath and hence on the path from r_1 to the root. By the same proof for the first case of Lemma 4.9 (paragraphs 7, 8, and 9 in the proof), when the edge containing r_2, say E_{r2}, was added to the shadow tree, E_2 became the parent of E_{r2}. That is a contradiction. Thus this case cannot really happen assuming that every PPH solution, restricted to the columns in $T(k)$, is contained in $T(k)$.

In the third case, TE_t is on the path from SE_1 to the root, and E is not the parent of r_2 before merging the class of TE_1 with its attaching class. Next we prove that there is at least one tree edge on the path from E_{r2} to E before the class merge. The proof is by contradiction. Suppose every edge on the path from E_{r2} to E before the class merge is a shadow edge. Consider the shadow edge below E on the path from E_{r2} to E, say SE'. Edge E is the parent of SE'. By Lemma 4.11 and its proof, $col(SE')$ cannot be in OldEntryList. Next consider the shadow edge below SE' on the path from E_{r2} to E, say SE''. Edge SE' is the parent of SE''. If $col(SE'')$ is in OldEntryList, then by Property 5 of Theorem 2.2, $te(SE')$ is on the path from $te(SE'')$ to the root, which is contradictory to $col(SE')$ not being in OldEntryList. So $col(SE'')$ is not in OldEntryList. By the same reasoning, the parent of r_2, $p(r_2)$, must be a shadow edge whose column number is not in OldEntryList; $p(r_2)$ is the parent of E_{r2}. Let $E'_{r2} = te(E_{r2})$ if E_{r2} is a shadow edge, or let $E'_{r2} = se(E_{r2})$ if E_{r2} is a tree edge. By Property 5 of Theorem 2.2, tree edge $te(p(r_2))$ is on the path from E'_{r2} to the root. Edges E'_{r2} and TE_1 are in the same class and have the same class root. So $te(p(r_2))$ is also on the path from TE_1 to the root, which is contradictory to $col(p(r_2))$ not in OldEntryList. Therefore, there is at least one tree edge, say TE, on the path from E_{r2} to E. Since TE_t is the tree edge on SecondPath that has the largest column number, $col(TE)$ is not in OldEntryList.

Next we prove that no hyperpath exists for row $k+1$ if we flip the class of TE_1 in the third case. The proof has two subcases. In subcase 1, j_2 is not the T connector of a tree edge, and hence j_1 must be the T connector of a tree edge, say TE_{j1}. Since TE_1 is on FirstPath, $col(TE_{j1})$ must be in OldEntryList, and TE_{j1} is on FirstPath. Since TE is on the path from E_{r2} to the root, and hence from SE_1 to the root, by Property 7 of Theorem 2.2, TE must be on the path from either SE_1 or TE_1 to the root at any stage in the algorithm. Since TE and TE_1 are not in the same class before the class merge, TE must be on the path from either j_1 or j_2 to the root at any stage in the algorithm. If we flip the class of TE_1, j_2 is now on the path from TE_1 to the root. As a consequence, TE must be on the path from either TE_1 or TE_{j1} to the root. But in neither case can a hyperpath exist for row $k+1$, as $col(TE)$ is not in OldEntryList. In subcase 2, j_2 is the T connector of a tree edge, say TE_{j2}. Since TE_t is the tree edge on SecondPath that has the largest column number, $col(TE_{j2})$ is not in OldEntryList. If we flip the class of TE_1, TE_{j2} is now on the path from TE_1 to the root, and hence no hyperpath for row $k+1$ exists. The proof is complete. ■

Lemma 4.17. *In every tree contained in $T_{FT}(k+1)$, there are two paths to the root with no edge in common that pass through edges corresponding to all columns in OldEntryList of row $k+1$.*

Proof. Recall that at the end of Procedure FixTree for row $k+1$ there is an *extended hyperpath* in $T_{FT}(k+1)$, whose two ends are edges TE_1 and E_2. The extended hyperpath consists of two directed paths to the root of $T_{FT}(k+1)$ (with no tree edge in common) that contain all the tree edges in $T_{FT}(k+1)$ corresponding to columns in OldEntryList of row $k+1$. Those two paths may also contain some shadow edges. Edges TE_1 and E_2 are either in the same class, or E_2 is the parent of the class root of SE_1.

If TE_1 and E_2 are in the same class, then the extended hyperpath in $T_{FT}(k+1)$ passes through the class roots and the join points of classes of all edges on the extended hyperpath. Let X be a class of an edge on the extended hyperpath. Let r_1 and r_2 be the class roots of X, and let j_1 and j_2 be the join points of X which r_1 and r_2 link to. Assume that r_1 and j_1 are on the path from TE_1 to the root in $T_{FT}(k+1)$. We can divide the extended hyperpath into four paths. Let $path_1$ be the path from TE_1 to r_1, $path_2$ be the path from j_1 to the root, $path_3$ be the path from j_2 to the root, and $path_4$ be the path from E_2 to r_2 in $T_{FT}(k+1)$. Tree edges on $path_1$, $path_2$, $path_3$, and $path_4$ correspond to all columns in OldEntryList of row $k+1$. Flipping class X lets r_1 link to j_2 and r_2 link to j_1. It is clear that there is an extended hyperpath from TE_1 to E_2 consisting of $path_1$, $path_3$, $path_2$, and $path_4$ after flipping class X. Therefore, there is an extended hyperpath in $T_{FT}(k+1)$ after flipping any class X of an edge on the path from TE_1 to the root. By the same reasoning, there exists an extended hyperpath in $T_{FT}(k+1)$ in every way of class flipping in $T_{FT}(k+1)$. It follows that the lemma holds.

If E_2 is the parent of the class root of SE_1, the proof is similar. Let r_1' and r_2' be the class roots of TE_1 and SE_1. Let j_1' and j_2' be join points which r_1' and r_2' link to. We can divide the extended hyperpath into three paths. Let $path_1'$ be the path from j_1' to the root, $path_2'$ be the path from j_2' to the root, and $path_3'$ be the path from TE_1 to r_1'. When flipping classes of edges on $path_1'$ or $path_2'$ in $T_{FT}(k+1)$, the proof is the same as above. When flipping the class of TE_1, it is the same as letting r_1' link to j_2' and letting r_2' link to j_1'. Now E_2 becomes the parent of r_1' after the flipping. It is clear that there is an extended hyperpath consisting of $path_2'$, $path_1'$, and $path_3'$ in $T_{FT}(k+1)$ after flipping the class of TE_1. Therefore, there exists an extended hyperpath in $T_{FT}(k+1)$ in every way of class flipping in $T_{FT}(k+1)$. It follows that the lemma holds. ■

Lemma 4.18. *In every tree contained in $T(k+1)$, there are two paths to the root with no edge in common that pass through edges corresponding to all columns that have a 2 entry in row $k+1$.*

Proof. At the end of Procedure NewEntries for row $k+1$, there are two directed paths to the root in $T(k+1)$ (with no tree edge in common) that pass through tree edges corresponding to all columns that have a 2 entry in row $k+1$. Let C be the largest column number in NewEntryList of row $k+1$. Then $te(C)$ and $se(C)$ are the two ends of those two paths. Since $te(C)$ and $se(C)$ are in the same class, the two paths from $te(C)$ and $se(C)$ to the root pass through the class roots and the join points of classes of all edges on the paths from $te(C)$ and $se(C)$ to the root. By the proof of Lemma 4.17, there exist two directed paths to the root in $T(k+1)$ (with no tree edge in common) that pass through tree edges corresponding to all columns that have a 2 entry in row $k+1$ in every way of class flipping in $T(k+1)$. It follows that the lemma holds. ■

Theorem 2.1. *Every PPH solution is contained in the final shadow tree produced by the algorithm. Conversely, every tree contained in the final shadow tree is a distinct PPH solution.*

Proof. The theorem has two parts. We prove the second part first.

All new edges corresponding to new entries in row $i+1$ are attached to leaves of $T(i)$. Any tree contained in $T(i+1)$, restricted to the columns in $T(i)$, is contained in $T(i)$. By Lemma 4.18, in every tree contained in $T(i+1)$, there are two paths to the root with no edge in common that pass through edges corresponding to all columns that have a 2 entry in row $i+1$. Thus, in every tree contained in the final shadow tree, there are two paths *for each row* to the root with no edge in common that pass through edges corresponding to all columns that have a 2 entry *in that row*. In addition, by Property 3 of Theorem 2.2, along any directed path towards the root in every tree contained in the final shadow tree, the successive edges are labeled by columns with strictly increasing leaf counts. Therefore, every tree contained in the final shadow tree is a solution to the PPH problem. Since each distinct choice of class flipping, followed by the required shadow edge and link contractions, leads to a distinct tree, the second part of the theorem is proven.

Next we prove the first part of the theorem by induction.

We first prove that every PPH solution, restricted to the columns in the shadow tree $T(1)$, is contained in $T(1)$. All 2 entries in the first row of S are new entries. Procedure NewEntries runs the simplest case: create a path to the root that consists of tree edges of columns that have 2 entries in this row, and create a path to the root that consists of shadow edges of these columns. All links between edges are free links. In every PPH solution, restricted to the columns in $T(1)$, there must be two paths to the root that pass through edges corresponding to all new entries in the first row. It is easy to verify that $T(1)$ contains all possible trees that satisfy this constraint. Therefore, every PPH solution, restricted to the columns in the shadow tree $T(1)$, is contained in $T(1)$.

Assume that every PPH solution, restricted to the columns in the shadow tree $T(i)$, is contained in $T(i)$. To complete the induction, we next prove that every PPH solution, restricted to the columns in the shadow tree $T(i+1)$, is contained in $T(i+1)$.

Three operations that modify the shadow tree in the algorithm are class flipping, class merging, and edge addition. Class flipping does not change the set of trees contained in the shadow tree. By adding new edges and corresponding new classes to $T(i)$, the number of choices of class flipping increases; i.e., the number of trees contained in the shadow tree increases. Every time a new tree edge and its corresponding

shadow edge are added to $T(i)$, a new class is created, and hence the number of trees contained in the shadow tree is doubled. The increase of the number of trees contained in the shadow tree by class addition is larger than or equal to the maximum possible increase of the number of PPH solutions, restricted to the columns in $T(i)$, to the number of solutions restricted to the columns in $T(i+1)$. Thus, all possible solutions have been included. Class merging removes some trees from the set of trees contained in the shadow tree. However by Lemmas 4.2, 4.8, 4.10, 4.13, 4.14, and 4.16, any tree contained in the shadow tree that is lost by doing the merge-class in the algorithm for row $i+1$ is not in any solution to the PPH problem. Thus, no PPH solution, restricted to the columns in the shadow tree $T(i+1)$, is lost from $T(i+1)$ by class merging. Based on the analysis above, we can conclude that every PPH solution, restricted to the columns in the shadow tree $T(i+1)$, is contained in $T(i+1)$. This completes the induction. ■

5. GENERAL PPH PROBLEM

Now we solve the general PPH problem for S with entries of value 0, 1, and 2. We assume that the rows of S are arranged by the position of rightmost 1 entry in each row decreasingly, with the first row containing the rightmost 1 entry in S. It is easy to prove that if there exists PPH solution(s) for S, then entries of value 1 are to the left of entries of value 2 in each row of S.

To solve the general PPH problem, we need to first build an *initial perfect phylogeny* T_i for S. The initial perfect phylogeny is described in detail by Gusfield (2002) and is built as follows. Let C_1 (respectively, R_1) denotes the set of columns (respectively, rows) in S that each contain at least one entry of value 1. We build T_i by first creating, for each row i in R_1, an ordered path to the root consisting of edges labeled by columns that have entries of value 1 in row i, with the edge of the smallest column label attaching to the root. We can then simply merge the identical initial segments of all these paths to create T_i. As shown by Gusfield (2002), T_i can be built in linear time and must be in every PPH solution for S.

We build an *initial shadow tree* ST_i based on T_i by changing each edge in T_i into a tree edge in ST_i, creating an H connector and a T connector for each tree edge in ST_i, and creating a fixed link pointing from the H connector of each tree edge, corresponding to an edge E in T_i, to the T connector of the tree edge whose corresponding edge in T_i is the parent of E. There are no shadow edges in ST_i, and the tree edges in ST_i form one class.

5.1. Algorithm with entries of value 1

The underlying idea of the algorithm is that in any PPH solution for S all the edges labeled with columns that have entries of value 2 in row $k+1$ must form two paths toward an edge in the initial tree. From that edge, there is a path to the root consisting of edges labeled with columns that have entries of value 1 in row $k+1$.

The algorithm for the PPH problem with entries of value 1, denoted as the algorithm with 1 entries, is very similar to the algorithm in Section 4. There are three differences. First, the algorithm with 1 entries builds and uses an initial shadow tree ST_i. Second, we now call an entry C_i an *old 2 entry* C_i *in row* $k+1$ if there is at least one entry of either value 2 or 1 at C_i in rows 1 through k. The third difference is the most important one. In the algorithm with 1 entries, whenever we use the term *root* during the processing of row $k+1$, we mean the *root for row* $k+1$. The root for row $k+1$ is defined as the T connector of the tree edge in the initial shadow tree ST_i whose column number has the rightmost 1 entry in row $k+1$. If there is no entry with value 1 in row $k+1$, then the root for row $k+1$ is defined as the root of ST_i. Every new edge attached to the root for row $k+1$ becomes part of the same class as the root of ST_i. This is a simple generalization of the earlier algorithm, since earlier, the root for each row is the root of the whole shadow tree.

5.2. Remaining issues

Identical columns: We use an example to demonstrate how to deal with identical columns. Suppose that after arranging columns of the matrix S by decreasing leaf count, columns 5, 6, and 7 are identical. We first remove columns 6, 7 from S and obtain a new matrix S' with distinct columns. Note that we use the same column indices of S to label columns in S'; i.e., column 5 of S' has a column label 5, but column 6

Table 1. Comparison of the Running Time Measured in Seconds on a P4 3 GHz Machine

Sites (m)	*Individuals (n)*	*Number of test cases*	*Average running time*	
			DPPH	*Our program*
5	1000	20	0.01	0.006
50	1000	20	0.20	0.07
100	1000	20	1.06	0.11
500	250	30	5.72	0.13
1000	500	30	45.85	0.48
2000	1000	10	467.18	1.89

of S' has a column label 8. Then we solve the PPH problem on S' by using our previous algorithm. Once a final shadow tree T' for S' is constructed, we can get a final shadow tree T for S according to two cases.

In the first case, the class of column 5 in T' consists of just edge 5 and $\overline{5}$. We then split tree edge 5 into three tree edges 5, 6, 7, and split shadow edge $\overline{5}$ into $\overline{5}$, $\overline{6}$, $\overline{7}$ in T. The result is equivalent to saying that 7H free links to 6T, 6H free links to 5T, and the links that link to 5T in T' now link to 7T in T. The same idea holds for shadow edges. In the second case, the class of column 5 in T' consists of edges other than 5 and $\overline{5}$. Then we want 7H to link to 6T with a fixed link, and 6H to fix link to 5T, and the links that link to 5T in T' now link to 7T in T. The same idea holds for shadow edges.

Unknown ancestral sequence: As mentioned by Gusfield (2002), the PPH problem with unknown ancestral sequence can be solved by using the *majority sequence* as the root sequence and then applying our algorithm. See Gusfield (2002) for more details.

6. RESULTS

We have implemented our algorithm for the general PPH problem in *C* and compared it with existing programs for the PPH problem. DPPH (Bafna *et al.*, 2003) was previously established as the fastest of the existing programs (Chung and Gusfield, 2003b). Some representative examples are shown in Table 1. In the case of $m = 2{,}000$ and $n = 1{,}000$, our program is about 250 times faster than DPPH, and the linear behavior of its running time is clear. This result is an average of 10 test cases. As in Chung and Gusfield (2003b), our test data is generated by the program *ms* (Hudson, 2002). That program is the widely used standard for generating sequences that reflect the coalescent model of SNP sequence evolution.The cases of 50 and 100 sites and 1,000 individuals are included because they reflect the sizes of subproblems that are of current interest in larger genomic scans. In those applications, there may be a huge number of such subproblems that will be examined. Our program can be downloaded at *wwwcsif.cs.ucdavis.edu/~gusfield/lpph/*.

ACKNOWLEDGMENTS

The authors thank Chuck Langley for helpful discussions. This research is partially supported by grant EIA-0220154 from the National Science Foundation.

REFERENCES

Bafna, V., Gusfield, D., Hannenhalli, S., and Yooseph, S. 2004. A note on efficient computation of haplotypes via perfect phylogeny. *J. Comp. Biol.* 11(5), 858–866.

Bafna, V., Gusfield, D., Lancia, G., and Yooseph, S. 2003. Haplotyping as perfect phylogeny: A direct approach. *J. Comp. Biol.* 10, 323–340.

Barzuza, T., Beckmann, J.S., Shamir, R., and Pe'er, I. 2004. Computational problems in perfect phylogeny haplotyping: Xor-genotypes and tag SNP's. *Proc. of CPM 2004.*

Bixby, R.E., and Wagner, D.K. 1988. An almost linear-time algorithm for graph realization. *Mathematics of Operations Research* 13, 99–123.

Bonizzoni, P., Vedova, G.D., Dondi, R., and Li, J. 2003. The haplotyping problem: Models and solutions. *J. Comput. Sci. Technol.* 18, 675–688.

Chung, R.H., and Gusfield, D. 2003a. Perfect phylogeny haplotyper: Haplotye inferral using a tree model. *Bioinformatics* 19(6), 780–781.

Chung, R.H., and Gusfield, D. 2003b. Empirical exploration of perfect phylogeny haplotyping and haplotypers. *Proc. 9th Int. Conf. on Computing and Combinatorics*, *LNCS* 2697, 5–9.

Damaschke, P. 2003. Fast perfect phylogeny haplotype inference. *14th Symp. on Fundamentals of Comp. Theory FCT 2003*, *LNCS* 2751, 183–194.

Damaschke, P. 2004. Incremental haplotype inference, phylogeny and almost bipartite graphs. *2nd RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*, preproceedings, 1–11.

Eskin, E., Halperin, E., and Karp, R.M. 2003. Efficient reconstruction of haplotype structure via perfect phylogeny. *J. Bioinformatics and Comp. Biol.* 1(1), 1–20.

Eskin, E., Halperin, E., and Sharan, R. 2004. Optimally phasing long genomic regions using local haplotype predictions. *Proc. 2nd RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*.

Gramm, J., Nierhoff, T., Tantau, T., and Sharan, R. 2004a. On the complexity of haplotyping via perfect phylogeny. Presented at the *2nd RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*. Proceedings to appear in *LNBI*.

Gramm, J., Nierhoff, T., and Tantau, T. 2004b. Perfect path phylogeny haplotyping with missing data is fixed-parameter tractable. Accepted for the *1st Int. Workshop on Parametrized and Exact Computation (IWPEC)*. Proceedings to appear in *LNCS*.

Gusfield, D. 2002. Haplotyping as perfect phylogeny: Conceptual framework and efficient solutions (extended abstract). *Proc. of RECOMB '02*, 166–175.

Gusfield, D. 2004. An overview of combinatorial methods for haplotype inference, *in* Istrail, S., Waterman, M., and Clark, A., eds., *Computational Methods for SNPs and Haplotype Inference*, *LNCS* 2983, 9–25, Springer, New York.

Halldórsson, B.V., Bafna, V., Edwards, N., Lippert, R., Yooseph, S., and Istrail, S. 2003a. A survey of computational methods for determining haplotypes. *Proc. 1st RECOMB Satellite on Computational Methods for SNPs and Haplotype Inference*, *LNBI* 2983, 26–47.

Halldórsson, B.V., Bafna, V., Edwards, N., Lippert, R., Yooseph, S., and Istrail, S. 2003b. Combinatorial problems arising in SNP and haplotype analysis, *in* Calude, C., Dinneen, M., and Vajnovski, V., eds., *Discrete Mathematics and Theoretical Computer Science, Proc. of DMTCS*, *LNCS* 2731.

Halperin, E., and Eskin, E. 2004. Haplotype reconstruction from genotype data using imperfect phylogeny. *Bioinformatics* 20, 1842–1849.

Halperin, E., and Karp, R.M. 2004. Perfect phylogeny and haplotype assignment. *Proc. of RECOMB '04*, 10–19.

Helmuth, L. 2001. Genome research: Map of the human genome 3.0. *Science*, 293(5530), 583–585.

Hudson, R. 1990. Gene genealogies and the coalescent process. *Oxford Survey of Evolutionary Biology* 7, 1–44.

Hudson, R. 2002. Generating samples under the Wright–Fisher neutral model of genetic variation. *Bioinformatics* 18(2), 337–338.

International HapMap Consortium. 2003. The HapMap Project. *Nature* 426, 789–796.

Kimmel, G., and Shamir, R. 2004. The incomplete perfect phylogeny haplotype problem. Presented at the *2nd RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*. To appear in *J. Bioinformatics and Comp. Biol.*

Tavare, S. 1995. Calibrating the clock: Using stochastic processes to measure the rate of evolution, *in* Lander, E., and Waterman, M., eds., *Calculating the Secrets of Life*, National Academy Press, Washington, DC.

Wiuf, C. 2004. Inference on recombination and block structure using unphased data. *Genetics* 166(1), 537–545.

Address correspondence to:
Zhihong Ding
Department of Computer Science
University of California
Davis, CA 95616

E-mail: dingz@cs.ucdavis.edu