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

Zhihong Ding*
Computer Science Department
University of California, Davis
dingz@cs.ucdavis.edu

Vladimir Filkov
Computer Science Department
University of California, Davis
filkov@cs.ucdavis.edu

Dan Gusfield*
Computer Science Department
University of California, Davis
gusfield@cs.ucdavis.edu

Abstract

Since the introduction of the Perfect Phylogeny Haplotyping (PPH) Problem [14] in Recomb 2002, the problem of finding a linear-time (deterministic, worst-case) solution has remained open, despite broad interest in the problem and a series of papers on various aspects of the PPH problem. 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 it is much faster in practice than prior methods.

In addition to the conceptual value of our solution, and new ideas and applications it will lead to, its practical value can be significant. The difference in speed between our linear-time solution and prior solutions is dramatic as the length of the sequences increase (see Sec. 6). Currently, most of the applications in human SNP data are to intervals of under one hundred SNPs, but 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). There is enormous diversity in biology with very little known about haplotype structure in most organisms or subpopulations, so it is too early to know the full range of *direct* application of this algorithm to PPH problems on long sequences (see [7] 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. One example is in studying the genotype/haplotype structure when recombination has influenced the underlying haplotypes [7], and another example is in searching for recombination hotspots and low-recombination blocks given genotype data [25]. In both of these cases, one finds, from every SNP site, the longest interval starting at that site for which there is a PPH solution. When applied on a genomic scale (as is anticipated), even a ten fold increase in speed has significant impact. Moreover, there are applications where one may examine *subsets* of sites, rather than contiguous intervals of sites, to find subsets for which there is a PPH solution. This is partly due to missing data or errors in the data, or small departures from the perfect phylogeny model, but it is also motivated by "dispersed haplotype blocks" that are now being observed, where the sites in the haplotypes are not contiguous, but are interlaced with other SNPs that are not part of that haplotype. The lengths of these dispersed haplotypes are not known. When solving the PPH problem repeatedly on a huge number of subsets of sites, efficiencies in the inner loop will be significant, even if each subset is relatively small.

* Research partially supported by grant EIA-0220154 from the National Science Foundation. Thanks to Chuck Langley for helpful discussions.

1 Introduction

Haplotypes have recently become a key unit of data in genetics, particularly human genetics. The international Haplotype Map Project [20] is focussed 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.

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* [24, 21]. 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 more computer science terminology, the coalescent model says that the $2n$ haplotype (binary) sequences fit a *perfect phylogeny*. See [14] 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 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 in [14] 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 [4] was known for over fifteen 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 in [14] 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 graph-realization approach was implemented [6], and two simpler, but also slower methods (based on "conflict-pairs" rather than graph theory) were later introduced [2, 10]. All three of these approaches have best and worst-case running times of $O(nm^2)$. Another paper [25] 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 [5, 16, 17, 15]). Related research has examined extensions, modifications or specializations of the PPH problem [23, 18, 11, 8, 9, 3], or examined the problem when the data or solutions are assumed to have some special form [19, 12, 13]. Some of those methods run in linear time, but only work for specializations of the full PPH problem [12, 13], or are only correct with high probability (with some model) [8, 9]. The problem of finding a deterministic, linear-time algorithm for all data has remained open, and a recent paper [1] 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. It is relatively easy to understand, 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 (the program is available). Tests show it to be much faster in practice as well as in theory, compared to the other existing programs. As in some prior solutions, the method provides an implicit representation of all the PPH solutions. 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.

The algorithm builds and uses a directed, rooted graph, called a "*shadow tree*", as its primary data structure, described in the next section.

2 The Shadow Tree

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 distinction). For each tree edge, i, there is a shadow edge, $\bar{i}$, 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 (arrow) 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*" edge 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 "link 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

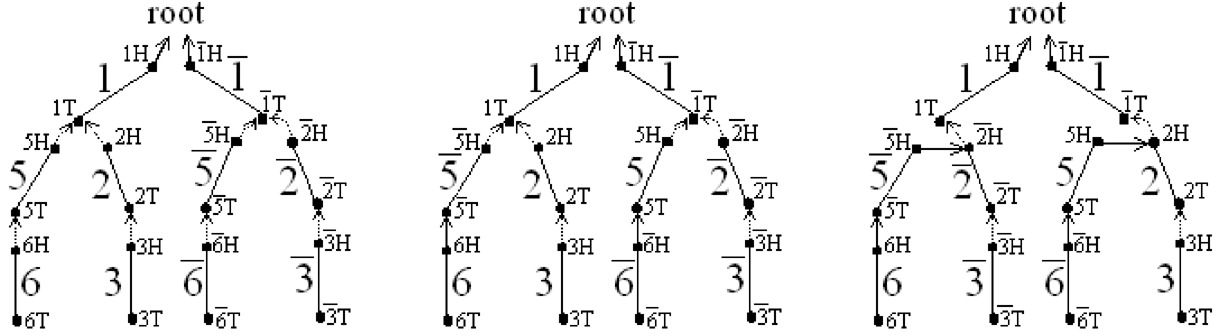

Figure 1: Edge 1 is the parent of edges 2 and 5. Each pair $(i, \bar{i})$ forms a class. Class 2 attaches to its parent class 1 by linking its class root 2H to join point 1T, and $\overline{2}$H to join point $\overline{1}$T. As a continuing example, edges 4 and $\overline{4}$ will be added later.

Figure 2: The result of flipping the class of edges 5 and $\overline{5}$, and flipping the class of edges 6 and $\overline{6}$ 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.

Figure 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.

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 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 $p(X)$ by using two free links. Each link goes from a class root of X to a 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 and information about old edges is updated. Three operations are used to modify 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.

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 resulting in a larger class. A class X may merge with its parent class $p(X)$, or two classes with the same parent may merge. No other merges are possible. 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 same parent, 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 cases. 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

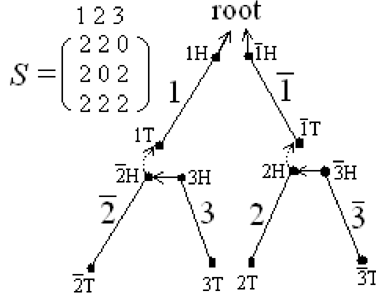

Figure 4: The final shadow tree after processing the given genotype matrix. It's an implicit representation of all PPH solutions for S.

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 if 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 Appendix A.

Theorem 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, in Fig. 4, by flipping the class of edges 2, $\bar{2}$, 3, and $\bar{3}$, and then doing 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. Therefore a final shadow tree with p classes implicitly represents 2^{p-1} PPH solutions.

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.

Theorem 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; 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 (except for the root class) attaches to exactly one other class, and the two join points are in different subtrees of the parent class unless it is the root class.*

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 a path to the root in the shadow tree.*

When each edge is added to the growing shadow tree, the algorithm ensures that these properties are satisfied. None of operations to the shadow tree later changes these properties.

3 Some Definitions

We use E_i to denote an edge, and C_i to denote a column number (i could be any 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.

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 belongs to. 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 function *col*, *te*, and *se* each returns the root. 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 also say that a tree T contained in 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.

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.

4 Algorithm

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. For ease of exposition, in this section we first describe a linear time algorithm for the PPH problem where S is assumed to 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 Sec. 5.

The algorithm processes the input matrix S one row at at 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. These two observations are simple, but powerful and are intuitively why we can achieve linear time, while no such solution exists for the general graph realization problem.

The algorithm processes a row $k+1$ using three procedures. In the first procedure, *OldEntries*, it 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. Those two paths may also 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 $T(k+1)$.

4.1 Procedure OldEntries

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 is denoted by $T_{FP}(k)$.

Procedure FirstPath: Assume that column numbers in OldEntryList (and lists used later) are ordered decreasingly, with the largest one, C_i, at the head of the list. The algorithm does 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 (it can be proven that 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 proofs of correctness of the algorithm.

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

$$
\begin{array}{r}
\mathbf{1\ 2\ 3\ 4\ 5\ 6\ 7} \\
\left(\begin{array}{l}
2\,2\,2\,0\,0\,0\,0 \\
2\,0\,0\,0\,2\,2\,0 \\
\gg \mathbf{2\,2\,2\,2\,0\,2\,2} \\
2\,2\,2\,2\,0\,0\,0 \\
2\,0\,0\,0\,2\,0\,0
\end{array}\right)
\end{array}
$$

OldEntryList:
1, 2, 3, 6
NewEntryList:
4, 7
CheckList:
2, 3

Figure 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.

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. Then we simply walk past E_p (i.e. let $E_p = p(E_p)$), without moving past entry C_i in OldEntryList. A second case is that E_p is a tree edge, but $col(E_p) < C_j$. This indicates that $te(C_i)$ and $te(C_j)$ can never be on the same path to the root, 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, but $col(E_p) > C_j$ (and hence $col(E_p)$ has a 0 entry in row $k+1$). This indicates that edges $te(C_i)$ and E_p must be on different paths to the root of $T(k+1)$, and the algorithm flips the class that contains $te(C_i)$ to avoid edge E_p. In that case, the algorithm will also merge the classes containing $te(C_i)$ and E_p to fix the relative position of those edges in any PPH solution. However, if $te(C_i)$ and E_p are in the same class when this case occurs, then even flipping the class of $te(C_i)$ won't avoid the problem, and hence the algorithm reports that no PPH solution exists. □

As an example, see Fig. 5. Procedure FirstPath performs a flip/merge-class in the third case. Let $T'(k)$ denote 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. This is proven as Lemma 3 in Appendix A. The algorithm also puts column number C_j into CheckList in the second case during processing C_i, which indicates that $te(C_j)$ and $te(C_i)$ cannot be on the same path to the root in any PPH solution. This is proven as Lemma 4 in Appendix A. Lemmas 3 and 4 together essentially say that when Procedure FirstPath takes any "non-obvious" 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 can be proven by lemmas similar to Lemma 3 and 4.

At the end of Procedure FirstPath, any columns in OldEntryList, whose corresponding tree edges are not on FirstPath, have been placed into CheckList. 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. The shadow tree produced after this procedure is denoted by $T_{SP}(k)$, 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 does a front to back scan of CheckList, starting from column C_i, and a parallel walk up in $T_{FP}(k)$, starting from edge $te(C_i)$. The parent of $te(C_i)$ in $T_{FP}(k)$, denoted as 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, $col(E_p) < C_j$) 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, $col(E_p) > C_j$), now indicates two possible subcases. In the first subcase, if $col(E_p)$ has a 0 entry in row $k+1$, then as in Procedure FirstPath, the algorithm determines that edges $te(C_i)$ and E_p must be on different paths to the root of $T(k+1)$, and it does a flip/merge-class as in Procedure FirstPath. In the second subcase, if $col(E_p)$ is in OldEntryList, but not in CheckList, then it must be that E_p is on FirstPath. 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. □

When SecondPath is about to use a tree edge, E_p, that is on FirstPath, Procedure DirectSecondPath is called to decide whether E_p must stay on FirstPath, or whether it must be on SecondPath, or if it can be on either path to root (it can be shown that only these three cases yield valid PPH solutions). The algorithm also performs the appropriate class flips and merges to ensure that E_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 E_p. Let E_k denote the tree edge on FirstPath whose parent edge is E_p at the end of Procedure FirstPath. The following tests determine which path to put E_p on.

Test1: If after flipping the class of C_i and the class of E_k, E_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 E_p is in the same class as E_k (respectively $te(C_i)$), then E_p must be on FirstPath (respectively SecondPath).

Test3: If after flipping the class of C_i and the class of E_k so that E_p is on FirstPath (respectively SecondPath), but not on SecondPath (respectively FirstPath), and there doesn't exist a hyperpath in the shadow tree after the flip, then E_p must be on SecondPath (respectively FirstPath).

If the test results indicate that E_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 E_p must be on FirstPath (respectively SecondPath), then flip the class of C_i and the class of E_k so that E_p is on FirstPath (respectively SecondPath), but not on SecondPath (respectively FirstPath), and merge the classes of E_p, C_i, and E_k.

If the test results show that E_p can be on either path, then for concreteness, flip one of the class of C_i and the class of E_k so that E_p is on FirstPath, but not on SecondPath, and merge the class of C_i with the class of E_k. □

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 Procedure FixTree

Procedure FixTree finds and merges more classes, if necessary, to remove trees contained in $T_{SP}(k)$ 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 call 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 $T_{FT}(k)$.

Procedure FixTree: Let E_1 denote the tree edge of the largest column number in OldEntryList, i.e., the lowest edge of FirstPath. Let E_2 denote the tree edge of the largest column number in OldEntryList whose tree edge is not on FirstPath, i.e., the lowest edge of SecondPath.

Find a maximal path from E_2, towards the leaves in $T_{SP}(k)$, consisting of shadow edges whose column numbers are in OldEntryList. It can proven that such a maximal path is unique. Let E_2' denote the edge that is the lower end of the maximal path; if the path doesn't contain any edge, then let E_2' be the same as E_2.

Repeat the following process until either E_1 and E_2' are in the same class, or E_2' is the parent of the class root of $se(E_1)$: if the column number of the edge containing the class root of E_1 is larger than the column number of the edge containing the class root of E_2, then merge the class of E_1 with its attaching class, otherwise merge the class of E_2 with its attaching class. □

As an example, see Fig. 6.

4.3 Procedure NewEntries

Procedure NewEntries creates and adds edges corresponding to columns in NewEntryList of row $k+1$ to $T_{FT}(k)$. Ideally it tries to attach new edges to the two ends of the extended hyperpath constructed in Procedure FixTree. If some new

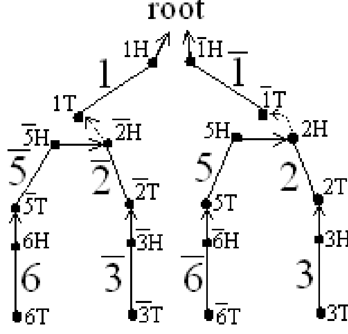

Figure 6: The shadow tree at the end of Procedure FixTree for row 3 of the matrix in Fig. 5. In Procedure FixTree for this example, $E_1 = 6$, $E_2' = E_2 = 3$, the class of edge 3 is merged with the class of edge 2. The class roots of the merged class are 2H and $\overline{2}$H.

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 any 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.

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 E_1, E_2, and E_2' be the same as in Procedure FixTree. Let C_h denote the smallest column number in NewEntryList.

In the first case, when $col(E_1) < C_h$, $te(C_h)$ 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 E_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 E_1, and otherwise to a connector in the class of E_1 whose parent is E_2'.

In the second case, when $col(E_1) > C_h$, by Property 3 of Theorem 2, none of $te(C_h)$ and $se(C_h)$ can attach to E_1. If $col(E_2) > C_h$, then no PPH solution exists no matter where new edges are attached; otherwise the algorithm finds two edges (E_{h1} and $E_{h2'}$) to attach $te(C_h)$ and $se(C_h)$, as follows.

Let E_{h1} denote the tree edge of the largest column number in OldEntryList that is less than C_h. Let E_{h2} denote the tree edge of the largest column number in OldEntryList that is less than C_h, and not on the path from E_{h1} to the root. If E_{h1} or E_{h2} doesn't exist, then let it be the root.

Similar to Procedure FixTree, the algorithm finds a maximal path from E_{h2} toward the leaves in $T_{FT}(k)$, consisting of shadow edges whose column numbers are in OldEntryList and less than C_h. Let $E_{h2'}$ denote the edge that is at the lower end of the maximal path.

If E_{h1} is on the path from E_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 E_{h1}, and create a free link pointing from the H connector of $te(C_h)$ (respectively $se(C_h)$) to the T connector of $E_{h2'}$ if $E_{h2'}$ is in the class of E_{h1}, otherwise to a connector in the class of E_{h1} whose parent is $E_{h2'}$.

If there are column numbers in NewEntryList that are larger than $col(E_1)$, then let C_t denote the smallest one among them ($C_h < col(E_1) < C_t$). $se(C_t)$ 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 E_1.

All new edges are added to $T_{FT}(k)$ according to case 1 and 2. The algorithm then merges the class of C_h with the classes of column numbers in NewEntryList that are less than $col(E_1)$, and merges the class of C_h with the classes of column numbers in OldEntryList that are larger than C_h. □

See Fig. 7 for an example. The shadow tree T shown in Fig. 7 is produced by the algorithm after processing the first three rows of the matrix S in Fig. 5. It is also the final shadow tree for S. It can be verified that the KEY THEOREM holds for S and T. The following lemma establishes that our

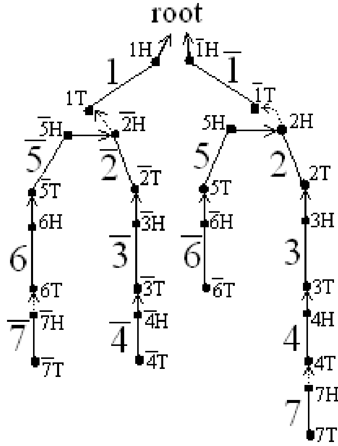

Figure 7: The shadow tree at the end of Procedure NewEntries for row 3 of the matrix in Fig. 5. In Procedure NewEntries for this example, $E_1 = 6$, $E'_2 = E_2 = 3$, $C_h = 4$, $C_t = 7$, $E_{h1} = 3$, $E_{h2} = 1$, $E_{h2'} = \overline{3}$. Note that edge $\overline{7}$ links to edge 6 instead of $\overline{4}$. The class of edge 4 is merged with the class of edge 2.

algorithm accomplishes the first observation made at the beginning of Sec. 4.

Lemma 1. *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 $k+1$.*

4.4 Correctness and Efficiency

Due to space limitations we cannot include proofs for all lemmas and theorems. We give the proofs of two lemmas, and the proof of the KEY THEOREM in Appendix A. An implementation of the linear time algorithm in pseudo code is given in Appendix B. For each row of S, the algorithm does a fixed number of scans of the entries in that row, and a fixed number of parallel 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 one entry in S, or to walk up one edge in the shadow tree. Each flip and/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.

5 General PPH Problem

Now we solve the general PPH problem for genotype matrix 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. The following lemma is immediate:

Lemma 2. *If there exists any PPH solution 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 in [14], 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 in [14], 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 is 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 $k+1j$ 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 Sec. 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, 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 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 $\overline{5}$, i.e., this class has never been merged with other classes. 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 7T free links to 6H, 6T free links to 5H, and the links that link to 5H in T' now link to 7H 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 7T to link to 6H with a fixed link, and 6T to fix link to 5H, and the links that link to 5H in T' now link to 7H in T. The same idea holds for shadow edges.

Unknown Ancestral Sequence: As mentioned in [14], the PPH problem with unknown ancestral sequence can be solved by using *majority sequence* as the root sequence, and then applying our algorithm. See [14] 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 problems. Program DPPH [2] was previously established as the fastest of the existing programs [7] (about two times faster than HPPH [10] and three times faster than GPPH [6]). Some representative examples are shown in the table below. In the case of $m = 2000, n = 1000$, 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 [7], our test data is generated by the program ms [22]. 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 1000 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 http://wwwcsif.cs.ucdavis.edu/~gusfield/lpph/.

			Average Running Time (seconds)	
Sites (m)	Individuals (n)	# test cases	DPPH	Our program
50	1000	20	0.20	0.07
100	1000	20	1.06	0.11
300	150	30	1.07	0.05
500	250	30	5.72	0.13
1000	500	30	45.85	0.48
1000	1000	10	92.20	0.95
2000	1000	10	467.18	1.89

References

[1] V. Bafna, D. Gusfield, S. Hannenhalli, and S. Yooseph. A note on efficient computation of haplotypes via perfect phylogeny. *J. Computational Biology*, 2004. To appear.

[2] V. Bafna, D. Gusfield, G. Lancia, and S. Yooseph. Haplotyping as perfect phylogeny: A direct approach. *J. Computational Biology*, 10:323–340, 2003.

[3] T. Barzuza, J.S. Beckmann, R. Shamir, and I. Pe'er. Computational Problems in Perfect Phylogeny Haplotyping: Xor-Genotypes and Tag SNP's. In *Proceedings of CPM* 2004.

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

[5] P. Bonizzoni, G. D. Vedova, R. Dondi, and J. Li. The haplotyping problem: Models and solutions. *Journal of Computer Science and Technology*, 18:675–688, 2003.

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

[7] R.H. Chung and D. Gusfield. Empirical Exploration of Perfect Phylogeny Haplotyping and Haplotypers. In *Proceedings of the 9'th International Conference on Computing and Combinatorics*, volume 2697 of *LNCS*, pages 5–9, 2003.

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

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

[10] E. Eskin, E. Halperin, and R.M. Karp. Efficient Reconstruction of Haplotype Structure via Perfect Phylogeny. *J. Bioinformatics and Computational Biology*, 1(1):1–20, 2003.

[11] E. Eskin, E. Halperin, and R. Sharan. Optimally Phasing Long Genomic Regions using Local Haplotype Predictions. In *Proceedings of the Second RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*, Pittsburg, USA, Feburary 20–21, 2004.

[12] J. Gramm, T. Nierhoff, T. Tantau, and R. Sharan. On the Complexity of Haplotyping Via Perfect Phylogeny. Presented at the *Second RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*, February 20–21, Pittsburgh, USA. Proceedings to appear in LNBI, Springer, 2004.

[13] J. Gramm, T. Nierhoff, and T. Tantau. Perfect Path Phylogeny Haplotyping with Missing Data is Fixed-Parameter Tractable. Accepted for the *First International Workshop on Parametrized and Exact Computation (IWPEC 2004)*, Bergen, Norway, September 2004. Proceedings to appear in LNCS, Springer.

[14] D. Gusfield. Haplotyping as perfect phylogeny: Conceptual framework and efficient solutions (extended abstract). In *Proceedings of RECOMB 2002: The Sixth Annual International Conference on Research in Computational Molecular Biology*, pages 166–175, 2002.

[15] D. Gusfield. An overview of combinatorial methods for haplotype inference. In S. Istrail, M. Waterman, and A. Clark, editors, *Computational Methods for SNPs and Haplotype Inference*, volume 2983 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 2004.

[16] B.V. Halldórsson, V. Bafna, N. Edwards, R. Lippert, S. Yooseph, and S. Istrail. A survey of computational methods for determining haplotypes. In *Proceedings of the First RECOMB Satellite on Computational Methods for SNPs and Haplotype Inference*, Springer Lecture Notes in Bioinformatics, LNBI 2983, pages 26–47, 2003.

[17] B. Halldórsson, V. Bafna, N. Edwards, R. Lipert, S. Yooseph, and S. Istrail. Com-

binatorial problems arising in SNP and haplotype analysis. In C. Calude, M. Dinneen, and V. Vajnovski, editors, *Discrete Mathematics and Theoretical Computer Science. Proceedings of DMTCS 2003*, volume 2731 of *Springer Lecture Notes in Computer Science.*

[18] E. Halperin and E. Eskin. Haplotype reconstruction from genotype data using Imperfect Phylogeny. *Bioinformatics*, 20:1842–1849, 2004.

[19] E. Halperin and R.M. Karp. Perfect Phylogeny and Haplotype Assignment. In *Proceedings of RECOMB 2004: The Eighth Annual International Conference on Research in Computational Molecular Biology*, pages 10–19, 2004.

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

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

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

[23] G. Kimmel, and R. Shamir. The Incomplete Perfect Phylogeny Haplotype Problem. Presented at the *Second RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes*, February 20–21, 2004, Pittsburgh, USA. To appear in *J. Bioinformatics and Computational Biology.*

[24] S. Tavare. Calibrating the clock: Using stochastic processes to measure the rate of evolution. In E. Lander and M. Waterman, editors, *Calculating the Secretes of Life.* National Academy Press, 1995.

[25] C. Wiuf. Inference on Recombination and Block Structure Using Unphased Data. *Genetics*, 166(1):537–545, January 2004.

Appendix A: Proofs

Space doesn't allow all lemmas and their proofs to be presented, but we want to present proofs of Lemma 3 and 4 and of the Key Theorem, to introduce some of the main ideas that are used in all of the proofs.

Lemma 3. *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-class in Procedure FirstPath for row $k+1$. 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-class occurs in Procedure FirstPath, column C_i has a 2 entry in row $k+1$, while $col(E_p)$ has a 0 entry in row $k+1$. So E_p can't be on the path from $te(C_i)$ to the root in any PP tree that explains S. However, before doing the flip/merge-class, E_p is on the path from $te(C_i)$ to the root in the shadow tree. So any tree contained in $T(k)$ that does not first flip the class of C_i relative to the class of E_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-class is not in any PPH solution. □

Lemma 4. *Assume that every PPH solution, restricted to the columns in the shadow tree T(k), is contained in T(k). If the algorithm puts a column number C_j into CheckList when processing column C_i in Procedure FirstPath for row $k+1$, then $te(C_j)$ and $te(C_i)$ cannot be on the same path to the root in any solution to the PPH problem.*

Proof. By assumption of the lemma, every PPH solution, restricted to the columns in the shadow tree T(k), can be obtained by flipping classes in $T(k)$ and then contracting all links and shadow edges. 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 each class in the shadow tree attaches to one unique class, the parent of $te(C_i)$ is either edge E_p or edge E'_p no matter how the classes in $T(k)$ are flipped (E_p and E'_p are defined in Procedure FirstPath). Hence $te(C_j)$ can never be the parent of $te(C_i)$ in any way that the classes of $T(k)$ are flipped. But by Property 3 of Theorem 2, and the fact that $col(E'_p) < col(E_p)$ and $col(E_p) < C_j$, $te(C_j)$ can't be above E_p or E'_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 any solution to the PPH problem. □

Theorem. 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 1, 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 and the fact above, 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 flips, followed by the required 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 doesn't 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 needed solutions have been included. Class merging removes some trees from the set of trees contained in the shadow tree. However by Lemma 3 and similar lemmas not shown here, any tree contained in the shadow tree that is lost by doing the flip/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 and Lemma 1 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. $\square$

Appendix B: Pseudo Code for the Linear Time Algorithm

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;
Call Procedure FirstPath;
Call Procedure SecondPath;
Call Procedure FixTree;
Call Procedure NewEntries;
} □

Procedure FirstPath:

While there are entries in OldEntryList not processed in this procedure, do the following {
Let C_i be the largest unprocessed column index in OldEntryList, and let C_j denote the second largest unprocessed entry in OldEntryList, if C_i is the only unprocessed entry in OldEntryList, then let C_j = root.
Let $E_r = te(C_i)$.
Repeat until encountering an "Exit" statement {
If E_r and $p(E_r)$ are in the same class, then {
If $p(E_r)$ is a shadow edge, then let $E_r = p(E_r)$, and run the repeat loop again;
Else ($p(E_r)$ is a tree edge), then {
Let $TE_p = p(E_r)$.
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, and Exit repeat loop;
Else if $col(TE_p) < C_j$, then {
We claim in this case $te(C_j)$ and $te(C_i)$ can't be on the same path to the root. As a result, we put C_j into CheckList, and mark C_j as processed by this procedure. Then we let C_j denote the second largest unprocessed entry in OldEntryList, if C_i is the only unprocessed entry in OldEntryList, then let C_j = root. Run repeat loop again. }
Else ($C_j < col(TE_p)$), then {
We claim no valid PPH solution exists, Report failure and Exit Algorithm.
}
} (end else $p(E_r)$ is a tree edge)
}
Else (E_r and $p(E_r)$ are in different classes), then {
Let $root_1$ be the class root of $te(C_i)$, and let $root_2$ be the class root of $se(C_i)$. Compare the parent edges of $root_1$ and $root_2$. If $p(root_1)$ is a shadow edge, then flip the class of C_i. If both $p(root_1)$ and $p(root_2)$ are tree edges, and $col(p(root_1)) < col(p(root_2))$, then flip the class of C_i.
By the property of the shadow tree (not stated here) and the operations in the previous step, $p(root_1)$ is a tree edge now, and $col(p(root_1)) > col(p(root_2))$. We use TE_p to denote $p(root_1)$.
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, and Exit repeat loop;
Else if $col(TE_p) < C_j$, then {
We put C_j into CheckList, and mark C_j as processed by this procedure. Then we let C_j denote the second largest unprocessed entry in OldEntryList, if C_i is the only unprocessed entry in OldEntryList, then let C_j = root. Run repeat loop again.
}
Else ($C_j < col(TE_p)$), then {

Flip the class of C_i, and merge the class of C_i with the class of TE_p. Run repeat loop again.
}
} (end else E_r and $p(E_r)$ are in different classes)
} (end repeat)
} (end while) □

Procedure SecondPath:

While there are entries in CheckList not processed in this procedure, do the following {
Let C_i be the largest unprocessed column index in CheckList, and let C_j denote the second largest unprocessed entry in CheckList, if C_i is the only unprocessed entry in OldEntryList, then let C_j = root.
Let $E_r = te(C_i)$.
Repeat until encountering an "Exit" statement {
If E_r and $p(E_r)$ are in the same class, then {
If $p(E_r)$ is a shadow edge, then let $E_r = p(E_r)$, and run the repeat loop again;
Else ($p(E_r)$ is a tree edge), then {
Let $TE_p = p(E_r)$.
If $col(TE_p) == C_j$, then mark C_i as processed by this procedure, and Exit repeat loop.
Else if $col(TE_p) < C_j$, or $col(TE_p)$ isn't in OldEntryList, then {
We claim no valid PPH solution exists, report failure and Exit Algorithm. }
Else ($C_j < col(TE_p)$, and $col(TE_p)$ is in OldEntryList), then {
Call Procedure DirectSecondPath, and Exit Procedure SecondPath. }
} (end else $p(E_r)$ is a tree edge)
}
Else (E_r and $p(E_r)$ are in different classes), then {
Let $root_1$ be the class root of $te(C_i)$, and let $root_2$ be the class root of $se(C_i)$. Compare the parent edges of $root_1$ and $root_2$. If $p(root_1)$ is a shadow edge, then flip the class of C_i. If both $p(root_1)$ and $p(root_2)$ are tree edges, and $col(p(root_1)) < col(p(root_2))$, then flip the class of C_i.
By the property of the shadow tree (not stated here) and the operations in the previous step, $p(root_1)$ is a tree edge now, and $col(p(root_1)) > col(p(root_2))$. We use TE_p to denote $p(root_1)$.
If $col(TE_p) == C_j$, then mark C_i as processed by this procedure, and Exit repeat loop.
Else if $col(TE_p) < C_j$, then {
We claim no valid PPH solution exists, report failure and Exit Algorithm. }
Else if $C_j < col(TE_p)$, and $col(TE_p)$ isn't in OldEntryList, then {
Flip the class of C_i, and merge the class of C_i with the class of TE_p. }
Else ($C_j < col(TE_p)$, and $col(TE_p)$ is in OldEntryList), then {
Call Procedure DirectSecondPath, and Exit Procedure SecondPath. }
} (end else E_r and $p(E_r)$ are in different classes)
} (end repeat)
} (end while) □

Procedure DirectSecondPath:

Mark C_i as processed by Procedure SecondPath.
Find the tree edge which is recorded as below TE_p on FirstPath (in Procedure FirstPath), and let C_{pc} denote its column number.
If $te(C_i)$ and $te(C_{pc})$ are in the same class, and have the same class root, then no PPH solution exists, because no hyperpath can contain all the edges $te(C_i)$, $te(C_{pc})$, and TE_p. Report failure and Exit algorithm.

If $te(C_i)$ and TE_p are in the same class, then {
Set flag1; ** Flag1 indicates that TE_p must be on the path from $te(C_i)$ to the root in any solution to PPH($k+1$).
}
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;
}
If $te(C_{pc})$ and TE_p are in the same class, then {
Set flag2; ** Flag2 indicates that TE_p must be on the path from $te(C_{pc})$ to the root in any solution to PPH($k+1$).
}
Else {
Let $root_{pc1}$ be the class root of the class of C_{pc} that links to TE_p, and let $root_{pc2}$ be the other class root of the class of C_{pc}.
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; }
}
If both flag1 and flag2 are set, then {
no PPH solution exists, report failure and exit algorithm;
}
Else if flag1 is set and flag2 is not set, then {
Flip, if necessary, the class of C_{pc}, so that TE_p is not on the path from $te(C_{pc})$ to the root; flip, if necessary, the class of C_i, so that TE_p is on the path from $te(C_i)$ to the root; and merge the class of C_i with the class of TE_p and the class of C_{pc}.
}
Else if flag2 is set and flag1 is not set, then {
Flip, if necessary, the class of C_{pc}, so that TE_p is on the path from $te(C_{pc})$ to the root; flip, if necessary, the class of C_i, so that TE_p is not on the path from $te(C_i)$ to the root; and merge the class of C_i with the class of TE_p and the class of C_{pc}.
}
Else (none of flag1 and flag2 is set), then {
Flip, if necessary, the class of C_{pc}, so that TE_p is on the path from $te(C_{pc})$ to the root; flip, if necessary, the class of C_i, so that TE_p is not on the path from $te(C_i)$ to the root; and merge the class of C_i with the class of C_{pc}.
} □

Procedure FixTree:

Let TE_1 denote the tree edge of the largest column number in OldEntryList; if OldEntryList is empty, then let TE_1 be the root. Let $SE_1 = se(TE_1)$.
Let TE_t denote the tree edge of the largest column number in OldEntryList whose tree edge is not on the path from TE_1 to root; if no such tree edge exists, then let TE_t be the root.
Find a maximal path from TE_t toward leaves in $T_{SP}(k)$ consisting of shadow edges whose column numbers are in OldEntryList.

Let E_2 denote the edge that is the lower end of the maximal path found in the previous step; if the path doesn't contain any edge, then let E_2 be the same as TE_t.

Repeat until either TE_1 and E_2 are in the same class, or E_2 is the parent of the class root of SE_1 {
  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.
} (end repeat) □

Procedure NewEntries:

If NewEntryList is empty, then there is no new edge for row $k+1$, exit Procedure NewEntries.
Create edges $te(C_i)$ and $se(C_i)$ for each column number C_i in NewEntryList.
Let $p(te(C_i)) = te(C_j)$, $p(se(C_i)) = se(C_j)$, for each C_i in NewEntryList and its left neighbor C_j in NewEntryList (column numbers in NewEntryList are ordered by their leaf counts, with the leftmost entry having the largest leaf count).
Let TE_1, SE_1, and E_2 be the same edges as in Procedure FixTree.
Let C_h = the smallest column number in NewEntryList.
If $col(TE_1) < Ch$, then {
  If TE_1 and E_2 are in the same class, then {
    Let $p(te(C_h)) = TE_1$, $p(se(C_h)) = E_2$. }
  Else (TE_1 and E_2 are in different classes), then {
    Let $p(te(C_h)) = TE_1$, and let the H connector of $se(C_h)$ link to the class root of SE_1. }
}
Else ($C_h < col(TE_1)$), then {
  Let TE'_1 denote 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 be the root. Let $SE'_1 = se(TE'_1)$.
  Let TE'_t denote 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 be the 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.
  Let E'_2 denote the edge that is the lower end of the maximal path found in the previous step, if the path doesn't contain any edge then let E'_2 be the same as TE'_t.
  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 {
      Let $p(te(C_h)) = E'_2$, $p(se(C_h)) = TE'_1$. }
    Else (TE'_1 is on the path from E_2 to the root), then {
      Let $p(te(C_h)) = TE'_1$, $p(se(C_h)) = E'_2$. }
  }
  Else (TE'_1 and E'_2 are in different classes), then {
    If TE'_1 is on the path from TE_1 to the root, then {
      Let $p(se(C_h)) = TE'_1$, and let the H connector of $te(C_h)$ link to the class root of SE'_1. }
    Else (TE'_1 is on the path from E_2 to the root), then {
      Let $p(te(C_h)) = TE'_1$, and let the H connector of $se(C_h)$ link to the class root of SE'_1. }
  }
  Let 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 $C_t = col(TE_1)$;
  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 $p(se(C_t)) = TE_1$.
} (end else $C_h < col(TE_1)$) □