An Efficient Algorithm for Perfect Phylogeny Haplotyping

Ravi VijayaSatya and Amar Mukherjee
{rvijaya,amar}@cs.ucf.edu
School of Computer Science, University of Central Florida 32816-2362

Abstract

The Perfect Phylogeny Haplotyping (PPH) problem is one of the many computational approaches to the Haplotype Inference (HI) problem. Though there are many $O(nm^2)$ solutions to the PPH problem, the complexity of the PPH problem itself has remained an open question. In this paper, We introduce the FlexTree data structure that represents all the solutions for a PPH instance. We also introduce row-ordering that arranges the genotypes in a more manageable fashion. The column ordering, the FlexTree data structure and the row ordering together make the $O(nm)$ OPPH algorithm possible. We also present some results on simulated data which demonstrate that the OPPH algorithm performs quiet impressively when compared to the earlier $O(nm^2)$ algorithms.

1. Introduction

Loci in the human genome in which a considerable percentage of the population varies from the rest are called SNPs(Single Nucleotide Polymorphisms). As the human genome is diploid, an SNP in an individual can be homozygous or heterozygous, depending on whether the two copies of the chromosome have the same allele or not. The sequence of alleles on a single copy of the chromosome is called a haplotype. Obtaining the haplotypes present in the population is essential in disease association studies. However, the haplotype information is expensive to obtain experimentally. Hence, the genotype information, which gives the unordered pair of alleles at each SNP site, is obtained experimentally.

Given a set of n genotypes over m SNP sites, the Haplotype Inference (HI) problem is to find a set of haplotypes such that each input genotype can be expressed as a combination of a pair of haplotypes. If k SNP sites are heterozygous in a given genotype, 2^{k-1} distinct pairs of haplotypes are possible that result in the same genotype. In other words, the genotype can have 2^{k-1} possible *explanation*s. The HI problem deals with finding the most 'accurate' explanation out of all these possible explanations. Initial approaches [3] to the HI problem were based on parsimony. However, the formulation presented in [3] was proven to be NP-hard by Gusfield [7]. Studies have shown that the actual observed diversity within any region of a chromosome is much lower than what we can expect from the number of SNPs covered by that region. Specifically, Daly et. al. [4] have shown that the human genome can be divided into blocks within which no recombinations are possible. This, together with the standard infinite sites assumption, implies that each block can be explained by a coalescent, or a phylogenetic tree. The coalescent model assumes that the evolutionary history of all the haplotypes in the population can be explained by a rooted tree, where each haplotype labels a vertex in the tree. This formulation of the haplotype inference problem is called as the PPH (Perfect Phylogeny Haplotyping) problem. Several $O(nm^2)$ algorithms have been presented for the PPH problem [8, 1, 6, 11].

1.1. The Perfect Phylogeny Haplotyping Problem

An $n \times m$ matrix A over the alphabet $\{0, 1, 2\}$ is given, in which the rows represent the genotype vectors, and the columns represent the SNP sites. The two alleles are indicated by 0 and 1. A 0(1) in a genotype vector indicates that the SNP site is homozygous with the 0-allele(1-allele). A 2 in the genotype vector indicates that the corresponding SNP site is heterozygous. The problem is to find a $2n \times m$ binary matrix B which has the following properties (1) Every row in the matrix A is explained by a pair of rows in the matrix B. (2) There is a perfect phylogeny T for the matrix B. Each row in the matrix B represents a haplotype. Since all the rows in B label the nodes in T, the evolutionary history of the haplotypes is a perfect phylogeny.

In any perfect phylogeny for the matrix A, let h_i and k_i be the vertices representing the two haplotypes for a row i in A. Gusfield[8] made the following important observations - (1)All the columns that are '1' in the row i of the matrix A must be in the path from the root to the LCA (lowest common ancestor) of h_i and k_i. (2) All the columns that are '2' in row i must be in the path from the LCA to one of

the vertices h_i or k_i. (3) All the columns that are '0' must not be in the path to h_i or k_i. The importance of *column sum*s is also noted in [8]. The column sum η_i of a column i in A is the number of '1's in column i in any binary matrix B that is a explanation of A. η_i is given by the following expression:

$$\eta_i = (\text{\# of 1's in} A[*, i] \times 2) + (\text{\# of 2's in} A[*, i]) \quad (1)$$

The column sum gives the exact number of haplotypes that must be in the subtree under the edge labeled with i in any perfect phylogeny for the matrix A. The column sums impose an order on the columns in any perfect phylogeny for A - no column with smaller column sum than column i can label an edge in the path from the root to the edge labeled with the column i. Though the significance of the column sums was noted in [8], the algorithm itself does not make complete use of the ordering imposed by the column sums. The other $O(nm^2)$ solutions for the PPH problem [1, 6, 11] have mainly ignored this property and failed to take advantage of it. In [8], the PPH problem is solved by mapping the problem to a graph realization problem. A simpler, more direct solution was presented in [1]. The approach makes use of the standard four-gamete test:

The four-gamete test: In any binary matrix B, if a sub-matrix formed by a pair of columns consists of all the rows from the set {00, 01, 11, 10}, then the matrix B cannot be realized by a perfect phylogeny.

However, the complexity of the algorithm presented in [1] is still $O(nm^2)$, as the approach is based on explicitly building all pairwise relationships between columns. Wiuf [11] attempted to improve upon the approach taken in [1], and made some interesting observations. Eskin et. al. [6] have taken a completely different approach to the problem. The complexity of the algorithms in [11, 6] is $O(nm^2)$. Two very recent, independent papers [9, 5] have reported $O(nm)$ complexity. However, the algorithm we present in this paper is fundamentally different and independent from the algorithms in [9] or [5]. In addition, the FlexTree data structure presented in this paper provides a simple and intuitive representation of the all possible PPH trees for the given matrix.

2. Motivation for $O(nm)$ algorithm

As mentioned in [8], the column sums induce an order on the columns. This ordering was mostly ignored in all the algorithms cited above. Hence there is some scope for improvement. In order to take advantage of this ordering, we sort the columns in A according to non-increasing column sums, producing the column-sorted genotype matrix A^c. i.e, in the matrix A^c, for any two columns i and j, $i < j$ implies that $\eta_i \geq \eta_j$. The column ordering, along with the fact that root is an all-zero vector, allows us to reduce the 4-gamete test into a 2-gamete test:

2-gamete test: In any column-sorted binary matrix B^c, if any sub-matrix formed by a pair of ordered columns consists of both the rows 01 and 11, then the matrix B^c cannot be realized by a perfect phylogeny..

It was established in [2] that a binary matrix B is realizable by a perfect phylogeny with an all zero root *iff* every sub matrix formed by a pair of columns has two or fewer rows from the set {01,10,11}. Extending this property to the column-sorted matrix B^c implies that the matrix B^c is realizable by a perfect phylogeny *iff* the sub-matrix formed by any ordered pair of columns does not contain more than one row from the set {01,11}. Each column in B^c has at least one '1', and hence the sub-matrix formed by each pair of columns in B^c has at least a 01 row or a 11 row. A pair of columns (x,y), $x < y$ in B^c are said to be *in-phase* if $B^c[*, xy]$ has a 11 row. The columns x and y are said to be *out-of-phase* if $B^c[*, xy]$ has a 01 row. For any binary matrix B^c that has a perfect phylogeny, these phase relationships can be represented by a $m \times m$ phase matrix P_{B^c}, in which $P_{B^c}[x, y]$ gives the phase relationship between the columns x and y. $P_{B^c}[x, y] = 0$ if x and y are in-phase and $P_{B^c}[x, y] = 1$ if x and y are out of phase. If the matrix B^c is not realizable by a perfect phylogeny, $B^c[*, xy]$ can have both rows 01 and 11, in which case the $P_{B^c}[x, y] = \psi$. As the columns x and y have to be ordered, $P_{B^c}[x, y]$ is defined only if $x < y$, and hence only the upper triangle of the matrix B^c is defined.

In order to use the 2-gamete test to determine the realizability of the column-sorted genotype matrix A^c, we need to be able to interpret the '2's in each column. Every row except a 22 row in a sub-matrix $A^c[*, ij]$ of A^c *forces* certain rows in the sub-matrix $B^c[*, ij]$ of any matrix B^c that is an explanation of A^c. A 00, 01, 10 or 11 row in $A^c[*, ij]$ duplicates itself in $B^c[*, ij]$, where as a 02, 20, 12 or 21 row in $A^c[*, ij]$ forces the rows {00,01}, {00,10}, {11,10}, or {01,11} in $B^c[*, ij]$, respectively. If the matrix A^c is to be realizable, both 01 and 11 rows should not be forced in sub-matrix $B^c[*, ij]$. A phase matrix P_{A^c} for A^c can be defined based on these forced rows. For the matrix A^c, $P_{A^c}[i, j] = 0$ if a 11 row is forced in $A^c[*, ij]$, $P_{A^c}[i, j] = 1$ if a 01 row is forced in $A^c[*, ij]$ and $P_{A^c}[i, j] = \psi$ if both 01 and 11 rows are forced in $A^c[*, ij]$. However, if a sub-matrix $A^c[*, ij]$ of A^c has only 00, 22 and 20 rows, the columns i and j are neither forced in-phase nor forced out-of-phase, and $P_{A^c}[i, j]$ is then designated as ϕ. Extending the 2-gamete test to a column-sorted genotype matrix A^c, we can now state the 2-gamete test for a column sorted genotype matrix A^c as follows:

Extended 2-gamete test: The column sorted genotype matrix A^c is not realizable by a perfect phylogeny if there are two columns i and j, $i < j$, such that $P_{A^c}[i, j] = \psi$.

An interesting result from the extended 2-gamete test is that in some situations, we can deduce that the matrix A^c is not realizable just by looking at a single row in A^c. A 21 row in any sub-matrix of A^c forces both 01 and 11 rows in the corresponding sub-matrix in B^c, and hence:

Property 1 *The matrix A^c is not realizable if a '2' occurs to the left of a '1' in any row.*

2.1. Implied relationships

Apart from the phase relationships directly forced by a pair of columns in A^c, additional phase relationships can be forced by rows in the matrix A^c that have at least three '2's. These *implied* relationships are summarized in the following theorem, first presented (using different terminology) in [1]:

Theorem 1 *In any realizable matrix A^c, given three columns x, y and z, if $P_{A^c}[x,y] \varepsilon \{0,1\}$, $P_{A^c}[x,z] \varepsilon \{0,1\}$, and if $A^c[r,x] = A^c[r,y] = A^c[r,z] = 2$ in any row r, then $P_{A^c}[y,z] = P_{A^c}[x,y] \oplus P_{A^c}[x,z]$, where $\oplus$ is the exclusive-or operator.*

Therefore, some of the ϕ entries in the phase matrix P_{A^c} can be assigned a value (of 0 or 1) using these implied relationships. Because of these implied relation ships, the following theorem applies:

Theorem 2 *In any realizable matrix A^c, given three columns x, y and z, $x < y < z$, if there is a row r in which $A^c[r,x] = A^c[r,y] = A^c[r,z] = 2$, then $P_{A^c}[x,y]$ will be in {0,1} if $P_{A^c}[x,z]$ is in {0,1}.*

If we know all the directly forced and implied phase relationships, the following theorem can be used to determine if the matrix A^c is realizable:

Theorem 3 (The Realizability Theorem) *A column-sorted genotype matrix A^c is realizable by a perfect phylogeny* iff *$P_{A^c}[x,y] \neq \psi$ for every pair of columns x and y, $x < y$, in A^c.*

In order to determine if A^c is realizable and to represent all possible PPH trees for the matrix A^c, we need all pairwise relationships between the columns. The FlexTree data structure presented in [10] allows us efficiently manage these pair-wise relationships.

3. The FlexTree Data Structure

3.1. Motivation

The initial motivation for the FlexTree data structure is to maintain the pairwise relationships efficiently, so that most of the pairwise relationships are stored implicitly, rather than explicitly. This can be achieved by taking advantage of the ordering between the columns and the limitations on the pairwise relationships imposed by any realizable matrix A^c. The in-phase and out-of-phase relationships directly translate to relative positions in the PPH tree. If two columns x and y, $x < y$, are forced in-phase, then the edge labeled with column x must be in the path from the root to the edge labeled with column y in any PPH tree for the matrix A^c. Similarly, if the columns x and y are forced out-of-phase, then the edge labeled with column x can not be in the path to the edge labeled with column y in any perfect phylogeny for the matrix A^c. This observation leads to the following theorem:

Theorem 4 *In any realizable matrix A^c, given two sites y and z such that $y < z$ and $P_{A^c}[y,z] = 0$, $P_{A^c}[x,z] = P_{A^c}[x,y]$ for any site $x < y$.*

In any PPH tree, a site j is said to *follow* a site i if the site i is the first site in the path from site j to the root. In the matrix A^c, for each column, we define the following terms:
Parent: The parent of any column j is the column i such that $i < j$, $P_{A^c}[i,j] = 0$ and $P_{A^c}[x,j] = 1$ for every column x such that $i < x < j$. In every PPH tree possible for the matrix A^c, the site j must follow the site i, as every column between i and j is forced out of phase with j. The parent of a column i is not defined (*null*) if there is no such column j for column i.
f-parent0: For a column j for which the parent is not defined, f-parent0 is a column i such that $i < j$, $P_{A^c}[i,j] = \phi$ and $P_{A^c}[x,j] = 1$ for every column x such that $i < x < j$. i.e., f-parent0 of a column j is the column with the highest index that the column j can follow in any PPH tree for A^c.

Introducing a dummy all-1 column with index 0 to the matrix A^c, $P_{A^c}[0,z]$ will be 0 for every column z, $1 \leq z \leq m$. This will ensure that either the parent or f-parent0 are defined for every column $z > 0$. The added column will not violate the column ordering since it has the highest possible column sum.
f-parent1: For a column j for which f-parent0 is defined, f-parent1 is:
(a) The highest column i such that $i <$ f-parent0, $P_{A^c}[i,j] = \phi$ and $P_{A^c}[i, \text{f-parent0}] = 1$, or if there is no such column y:
(b) *null*, if at least one column x such that $x <$ f-parent0 and $P_{A^c}[x, \text{f-parent0}] = \phi$, or, if there is no such column x:
(c) The highest column i such that $P_{A^c}[i,j] = 0$.

For a site that can follow different sites in different PPH trees, f-parent0 and f-parent1 give us the different possibilities for the site. When both f-parent0 and f-parent1 are defined (not *null*) for a site i, it means that the site i has to follow either the site f-parent0 or the site f-parent1 in any

PPH-tree for the matrix M. i.e, there are only two possibilities for the site i. On the other hand, if f-parent1 of a site is *null* it means that there are more than two possibilities for the site i. Out of these, the one with the highest index is f-parent0.

Theorem 4 leads to drastic implications - it tells us that only a part of the phase matrix P_{A^c} needs to be stored explicitly. The rest of P_{A^c} can be inferred by just knowing a small portion of P_{A^c}. Theorem 5 tells us exactly what information in P_{A^c} is necessary in order to deduce the rest of P_{A^c}.

Theorem 5 *In any realizable matrix A^c, the phase matrix P_{A^c} can be constructed if we know the parent, f-parent0 and f-parent1 of each column.*

3.2. The FlexTree

The FlexTree data structure is a special kind of weakly connected directed acyclic graph. For each column, the FlexTree data structure stores the parent, f-parent0 and f-parent1 information. Some additional information is also stored in order to account for the restrictions imposed by the rows in the matrix having more than two '2's. The FlexTree has a tree-like structure. In fact, if the matrix A^c has a unique perfect phylogeny, the underlying undirected graph of the FlexTree for A^c will be a rooted tree. The basic idea behind the FlexTree data structure is to correctly represent all the possibilities for each site j in any perfect phylogeny for the matrix A^c. For any site i such that $i < j$, one of the following three scenarios apply, based on the value of $P_{A^c}[i,j]$: (1) If $P_{A^c}[i,j] = 0$, the the site i must be in the path from the site j to the root in any perfect phylogeny for the matrix A^c. i.e., in the FlexTree, all possible paths from the site j to the root must include the site i. (2) If $P_{A^c}[i,j] = \phi$, then the site i may or may not be in the path from the root to the site j in a given perfect phylogeny for the matrix A^c. For each such site i, there must be at least one path from the site j to the root that includes site i and there must at least one path that does not. (3) If $P_{A^c}[i,j] = 1$, then the site i will not be in the path from the root to the site j in any perfect phylogeny for the matrix A^c. i.e, in the FlexTree, any path form the site j to the root must not include the site i. In the FlexTree, each site is represented by a directed edge labeled with the site (There are many reasons for selecting directed edges - these will be apparent shortly). Figure 1 shows a matrix A^c, the phase matrix P_{A^c} of A^c, and the flex tree T for the matrix A^c.

If a column j has a parent i, the relationship is represented by the edge labeled with site i being adjacent to edge labeled with site j. If a site i is the f-parent0 or f-parent1 of a site is j, the relationship is represented by a directed unlabeled *glue* edge connecting the edge labeled with the site j to the edge labeled with site i. The phase relationships reduce to reachability in the FlexTree. For any site j, if $P_{A^c}[i,j] = 1$ for any site i, $i < j$, then the edge labeled with site i is not reachable from the edge labeled with site j. If $P_{A^c}[i,j] = 0$, every path from edge labeled with site j to the root will include the edge labeled with i. If $P_{A^c}[i,j] = \phi$, then there will at least one glue edge in the path from the edge labeled with site j to the edge labeled with site i. As the FlexTree represents all the phase relationships given by the phase matrix P_{A^c}, any PPH tree for A^c can be built from the FlexTree by removing some glue edges and contracting the others.

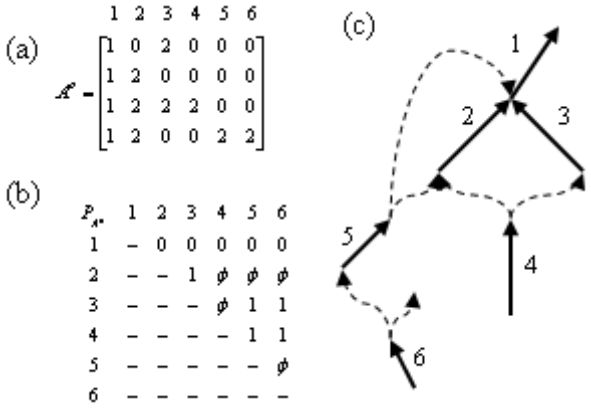

Figure 1. (a) A genotype matrix A^c; (b) The Phase matrix for A^c; (c) The FlexTree T for A^c - the broken edges represent the glue edges.

3.3. Partitions

Though every PPH tree for A^c can be obtained from the FlexTree described above, we need to refer back to A^c to do this. For example, consider the matrix A^c and the phase matrix P_{A^c} shown in Figure 2. In order to build a PPH-tree (or a binary matrix B^c), we need to be able to assign a value of 0 or 1 to every ϕ in the phase matrix P_{A^c}. Let us assume we start doing this by scanning P_{A^c} bottom-to-top and left-to-right, and arbitrarily setting every ϕ to 0 or 1. The first ϕ we will encounter is $P_{A^c}[2,3]$. As $P_{A^c}[3,4] = 1$ and columns 2,3 and 4 are all '2' in the last row of the matrix A^c, Theorem 1 will be applicable on columns 2, 3 and 4 as soon as we set $P_{A^c}[2,3]$ to 0 or 1, and $P_{A^c}[2,3]$ will be equal to $P_{A^c}[2,3] \oplus P_{A^c}[3,4]$. In any binary matrix B^c that is an explanation of A^c, if columns 2 and 3 are in-phase, columns 2 and 4 must be out of phase. Similarly, if columns 2 and 4 are resolved in-phase, then columns 3 and 4 must be out of phase. In other words, if u and v are the vertices in which columns 3 and 4 are incident, the requirement is that u and v must always be distinct vertices. Therefore, whenever we

set a ϕ in P_{A^c} to 0 or 1, we need to refer back to A^c and check every row in A^c to see if there are other ϕ's in P_{A^c} which can be set to 0 or 1 based on Theorem 1. Clearly, there can be $O(m^2)$ such ϕ entries in P_{A^c}, and checking A^c will take $O(nm^2)$ time for each one of them. Because of this, building a PPH tree from the FlexTree might take $O(nm^3)$ time.

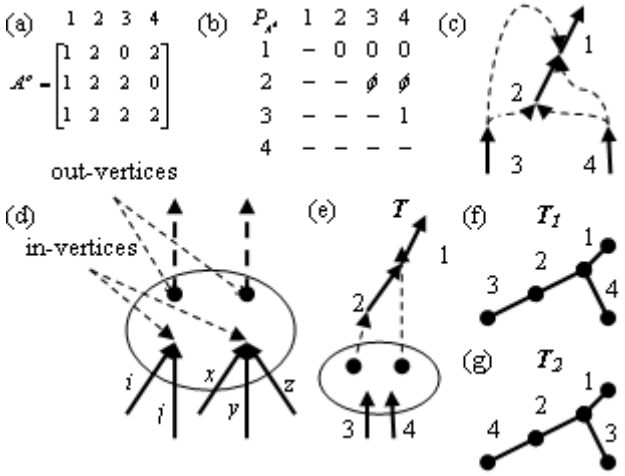

Figure 2. (a) A genotype matrix A^c; (b) The Phase matrix for A^c; (c) The FlexTree T for A^c, without partitions; (d) The general structure of a partition; (e) The FlexTree T for A^c with partitions; (f) and (g) The two PPH trees T_1 and T_2 represented by T

Therefore, we need to store some more information about A^c in the FlexTree in order to be able to construct a PPH tree from the FlexTree without having to refer to A^c, and to enable an $O(nm)$ algorithm. For this, we introduce a special system of vertices called a *partition*. A partition consists of four vertices in total, as shown in Figure 2-(d). Two of these vertices are the in-vertices of the partition - their indegree is at least 1 and outdegree is 0. The other two are out-vertices - their indegree is 0, but outdegree is 1. In other words, two of the vertices are sinks for two sets of labeled edges, and each of the other two act as a source of an unlabeled glue edge. The four vertices represent two vertices in any PPH tree. In any PPH tree, one of the two in-vertices merges with one of the two out-vertices, and the second in-vertex merges with the second out-vertex. As the two in-vertices have to be distinct, they cannot both merge with the same out-vertex. Therefore, there are two possible ways to resolve a partition. However, in the FlexTree, all the edges that are incident on any of the in-vertices are interpreted as being connected to both the glue edges coming out of the partition. This is because of the fact that any edge i incident on one of the in-vertices has two possibilities as given by the two glue edges. It is only in a PPH tree that the edge i has to 'choose' one of these glue edges.

3.4. Representation of the FlexTree

Because of the partitions, the FlexTree is not exactly a DAG. As is evident from the description of a partition, all the columns involved in a partition have the same set of f-parents (f-parent0 and f-parent1). Each partition involves two groups of sites, each group representing the sites that are incident on one of the two in-vertices of the partition. The two groups are arbitrarily numbered as group-0 and group-1. Therefore, for each partition, we need to store the information about the f-parents and the two groups of sites involved in the partition. For each site that is not in a partition, we need to know the *parent*, *f-parent1*, *f-parent0* of the site. If the site is involved in a partition, we need to store a pointer to the partition. In order to optimize the performance of the algorithm, each site involved in a partition also needs to store which group of the partition it is in. The FlexTree is stored as two tables, the column-table and the partition-table, which give information about the sites and partitions, respectively. The *partition* field in the column-table stores a pointer to the partition that the column is involved. The *group* field gives the group number of the column within the partition. For each column, we need a constant amount of space in the column-table. Hence the total space required by the column-table is $O(m)$. The partition table stores the index of each partition, the two f-parents, and the list of sites in each group of the partition. The *size* of a partition is defined as the total number of columns involved in the partition. The size of a partition is equal to the sum of the in-degrees of the two in-vertices. As each column can be involved in only one partition at any given time, the combined size of all the partitions in the FlexTree is $O(m)$. The total number of partitions in the partition table can be up to $O(m/2)$.

4. The OPPH Algorithm

An underlying assumption in the above discussion is that root of the phylogenetic tree is an all-zero vector. If the number of '1's in every column is less than or equal to the number of '0's, then if the matrix has a perfect phylogeny, the root for the phylogeny will be an all-zero vector. Though every column in the input matrix A might not always satisfy this condition, there is a simple transformation that guarantees that the root is an all-zero vector. The transformation is to invert all columns with column sums greater than m - the '1's in the column are changed to '0's, the '0's are changed to '1's, and the '2's are left unchanged. The OPPH algorithm requires the rows in A^c to be sorted using the lexicographic order $1 < 0 < 2$. We denote this row-sorted matrix

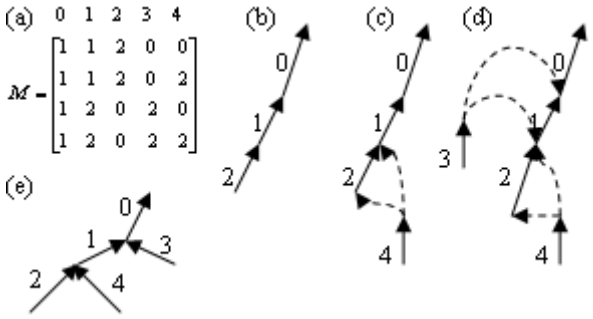

Figure 3. (a) A matrix M (b), (c), (d) and (e): The FlexTree after processing rows 1,2,3 and 4, respectively

with M. As the phase relationships and the column ordering in M are no different from those is A^c, we continue to refer to the phase matrix as P_{A^c} in the rest of the paper. An extra, all-1 column with index 0 is added to M, as explained in the previous section. For the algorithm to be $O(nm)$ it should not take more than $O(m)$ time at each row. To give the reader an idea of how the algorithm works, the FlexTree after processing each row is shown in Figure 3. The OPPH algorithm first builds the FlexTree to accommodate all the pairwise relationships in the first row of the matrix M. It then processes each of the remaining rows, modifying the FlexTree to accommodate the pairwise relationships (both direct and implied) introduced by each additional row.

We define the following terms. A column is said to be *fixed* if it has a (non-null) parent. A column is said to be *flexible* if it has at least one (non-null) f-parent. A column is said to be in the FlexTree if it is either a flexible or a fixed column. The *FlexEnd* of a fixed column i is the first flexible column in the path from the edge labeled with i to the root in the FlexTree. By convention, a flexible column is the *FlexEnd* for itself.

A *partial genotype vector* is a prefix of a row in M, to which a string if 0's have been appended so that the length of resulting vector is exactly $m + 1$. The vector needs to be of length $m + 1$ so that it remains to be a valid genotype vector even when the columns are re-arranged to represent the original order of the sites in matrix A. The ith partial genotype vector of a row r, denoted by $M[r, 0...i]$, is the prefix of row r of length $i + 1$, to which $m - i$ zeros have been appended at the end. The mth partial genotype vector of a row is the row itself. A haplotype vector (or a genotype vector) h is said to *end* in a site j if j is the non-zero column with the highest index in h. We denote the two haplotypes of $M[r, 0...i]$ using h_r^i and k_r^i. By convention, h_r^i is the haplotype vector that ends in the column with the higher index among the two haplotypes h_r^i and k_r^i. For simplicity of notation we denote the site in which h_r^i ends by h_r^i itself, and the site in which k_r^i ends by k_r^i itself. From the context, it will be clear whether it is the haplotype h_r^i or the site h_r^i that is being referred to.

A partial genotype vector is said to be *split* if both the sites h_r^i and k_r^i are defined (not null). Because of the convention, the site h_r^i is always defined. The site k_r^i will be defined if there is only one possible column in which the haplotype k_r^i can end. If there are multiple sites in which the haplotype k_r^i can end, then site k_r^i is not defined. During the construction of the FlexTree, the algorithm maintains two additional arrays $h[]$ and $k[]$, each of size $m + 1$. When the algorithm is processing row r, the fields $h[i]$ and $k[i]$ represent the sites h_{r-1}^i and k_{r-1}^i.

For any row r in the matrix M such that $r \geq 1$, the *EntryPoint* (denoted by e_r) is the column i with the lowest index such that $M[r-1, i] \neq M[r, i]$. i.e, the entry point is the first column from the left in which the rows $r-1$ and r differ. The *SplitPoint* (denoted by s_r) of a row r is the column with the highest index before e_r at which the row $r-1$ is split. i.e., s_r is the highest column i such that $i < e_r$ and the site $k[i]$ (i.e, the site k_{r-1}^i) is defined.

Algorithm 1: The OPPH algorithm

inputs: A^c, n, m
Result: The FlexTree T for A^c
1 Sort the rows in A^c and add an all-1 column with index 0 to produce the matrix M
2 Initialize every entry in the column table and partition table to *null*
3 $h[0] \leftarrow 0$, $k[0] \leftarrow 0$, FlexEnd[0] $\leftarrow 0$
4 ProcessNewRow(1, 1)
5 **for** $r_i = 2$ **to** n **do**
6 $(e_r, s_r) \leftarrow$ ScanForward(M, T, r_i)
7 **if** $M[r_i - 1, e_r] = 1$ *AND* $M[r_i, e_r] = 0$ **then**
8 ProcessNewRow(r_i,e_r)
9 **else**
10 TraceUpRow(r_i, e_r, s_r)
11 TraceDown(r_i)

As none of the pairwise relationships are known before we start with the first row, the row will have a FlexTree as long as it does not violate Property 1. i.e., if there are no '2's to the left of a '1'. The column with index 0 is the dummy all-1 column, hence Parent[0] is initialized to 0, by convention. All other values in the column table are set to *null*, except for $h[0]$ and $k[0]$, which are set to 0. The ProcessNewRow procedure for building the FlexTree T for the first row directly follows from the observations made in Section 1.1.

The algorithm processes the rows in M in lexicographic order and makes modifications to the FlexTree to accommo-

date the pairwise relationships induced by the rows. Hence, when the algorithm is at a row r, all the pairwise relationships induced by the first $r-1$ rows are correctly represented in T. At each row, the algorithm consists of three steps - ScanForward, TraceUp and TraceDown. We describe each one of the steps in detail in the following sections. A high-level description of the OPPH algorithm is presented in Algorithm 1.

4.1. The Scan Forward procedure

In the scan forward step, the algorithm mainly finds e_r and s_r, the EntryPoint and SplitPoint for the row r. The partial genotype vector $M[r, 0...(e_r - 1)]$ is exactly identical to the partial genotype vector $M[r-1, 0...(e_r - 1)]$, from the definition of e_r. Hence, there can be no new pairwise relation ships induced by the partial genotype vector $M[r, 0...(e_r - 1)]$, as all the pairwise relationships in $M[r-1, 0...(e_r - 1)]$ are already represented in T. The scan forward procedure also finds s_r. As both $h[s_r]$ and $k[s_r]$ are defined, one of them must be in the path to h_r^m and the other must be in the path to k_r^m.

4.2. Trace Up

The actual modifications to the FlexTree are done in the TraceUp step. In this step, the algorithm first tries to find the site p_0 in T with the highest index such that $M[r, p_0] = 2$ and $p_0 \geq e_r$. Then it tries to find a site p_1 such that $M[r, p_1] = 2$, $k[s_r] \leq p_1 \leq p_0$, and p_1 is not reachable from p_0. The pairwise relationships of all other columns are inferred w.r.t. the sites p_0 and p_1.

The significance of the sites p_0 and p_1 is that they provide 'anchor's for the row in the FlexTree T. The trace up procedure traces the ancestry of these two sites, defining many phase relationships in the process. Since we know that $P_{A^c}[p_0, p_1] = 1$, for any site $i < p_1$, if the site i is '2' in row r and if we know that any one of the phase relationships $P_{A^c}[i, p_0]$ or $P_{A^c}[i, p_1]$ are '1' or '0', we can infer the other by applying Theorem 1.

The significance of e_r is that it indicates some change in the pairwise relationships involving p_0 or p_1. Because of the column sorting, if $e_r \leq m$, it means that the partial genotype vector $M[r, 0...e_r]$ is being encountered for the first time. As p_0 and p_1 are already in T, for each of them, there will at least be one row before r in which they were non-zero. Let $r_0 < r$ be a row in which p_0 was non-zero. Since the row r_0 precedes the row r, there must be at least one column $x_0 \leq e_r$ so that the pair $(M[r_0, x_0], M[r, x_0])$ is (1,0), (1,2) or (0,2). In all three cases, it will be possible either to declare the matrix M unrealizable, or to assign a 0 or 1 to $P_{A^c}[y, p_0]$ for every column y in the range $x_0 < y < p_0$.

The trace up procedure starts by scanning the row from right to left, and tries to find p_0. If the procedure reaches e_r without finding p_0, then the row r does not involve any non-zero columns after e_r that are already in the tree, and the algorithm moves to the TraceDown procedure directly. In fact, if $M[r-1, e_r] = 1$ and $M[r, e_r] = 0$, then there should be no non-zero column with higher index that e_r that is already in T if matrix M is to be realizable, and the algorithm invokes the ProcessNewRow procedure instead of the TraceUp procedure.

The site $k[s_r]$ is the site with the lowest index that the TraceUp procedure can reach. Since both $k[s_r]$ and $h[s_r]$ are defined, one of the two haplotypes for the partial genotype vector $M[r, 0...i]$ must end in $h[s_r]$ and the other must end $k[s_r]$, in any PPH tree for the matrix M. Therefore, either one of the two sites ($k[s_r]$, $h[s_r]$) must be reachable from p_0, and the other must be reachable from p_1, or both must be reachable from both p_0 and p_1. The TraceUp procedure can terminate as soon as it can ensure this reachability criteria.

Assigning a parent to a flexible a site effectively sets a phi entry in P_{A^c} to 0 and hence may effect other sites in the FlexTree, as per Theorem 1. The following things have to be taken care of when assigning a parent to a site:

***Orphan* sites**: Fixing a flexible site i might result in some *orphan* sites, sites for which only one f-parent is defined, but have only one path to the root. This situation can be handled by using a separate array $L[]$ that stores in $L[i]$ the f-parent f of site i to which i is not fixed. For any site for which i is the only f-parent, $L[i]$ must be the other f-parent after processing the current row, and the trace down procedure carries out this assignment, thus correcting the orphan sites.

Dealing with partitions: When a flexible site i that is involved in a partition needs to be fixed, all the other sites involved in the partition also get effected. All the sites that are on the same side of the partition as i must get fixed to the same site as i. If both the f-parents of the partition are defined, then the sites on the opposite side must get fixed to the other f-parent of the partition. If only one f-parent of the partition is defined, then the sites on the opposite side of the partition must enter into a partition with the FlexEnd of the site i.

4.3. Trace Down

Trace down procedure mainly does four things: (1) Update the FlexEnd of every site (2) Correct *orphan* sites (3) Update $h[]$ and $k[]$ arrays (4) Add the non-zero columns with index greater than p_0 to the FlexTree. The trace down procedure is simple and very straight forward. At each flexible site i at which f-parent1[i] is not defined, f-parent1[i] is set to $L[i]$. At each fixed site, the FlexEnd is copied onto

itself.

4.4. Obtaining a PPH Tree from the FlexTree

The total number of PPH trees represented by the FlexTree is given by the following expression:

$$\gamma = 2^{\{\text{[no. of partitions]}+\text{[no. of flexible sites not in a partition]}\}} \quad (2)$$

Any one of these γ solutions can be computed in $O(m)$ time from the FlexTree. The procedure for obtaining the PPH tree fixes each flexible site, starting from the site with the lowest index and processing the sites in M from left to right. There will be only two possibilities at any flexible site, as all the sites with higher indices are already fixed. Different criteria can be applied to choose between the two choices, in order to obtain the *deepest* or the *broadest* tree.

5. Complexity

It takes $O(nm)$ time to compute the column sums. Once the column sums are computed, it takes $m\log(m)$ time to sort the columns (using quick sort) according to the column sums. The lexicographic ordering of the rows takes $O(nm)$ time and space, using radix sort. The total time required for the preprocessing step is $O(nm)$. The ScanForward step involves a simple scan of the row, and takes $O(m)$ time. As long as partitions are not involved, the Trace Up procedure takes constant time at each site. However, the Trace Up procedure might spend up to $O(m)$ time at sites that are involved in partitions. Merging two partitions into one, or fixing one side or both sides of the partition, takes time in the order of the size of the partition(s) involved. However, the total amortized cost for all the mergers and fixings while processing any single row is $O(m)$. The detailed proof for this complexity is not presented here due to space constraints. The trace down procedure involves a constant number of operations at each site, and therefore takes $O(m)$ time for each row.

6. Results

A OPPH algorithm has been implemented in C++. The results indicate that the performance is as expected, indicating that there are no hidden constraints. Table 1 shows how the OPPH algorithm performs in comparison to algorithms gpph[8] and dpph[1]. The times for opph are averages over one thousand test cases. The times for gpph and dpph are averages over five cases. It is clear that the OPPH algorithm outperforms both gpph and dpph algorithms. The tests were carried on simulated data. A random PPH tree was generated, and the genotypes were obtained by selecting two random haplotypes from the tree and combining them together. The binaries for the implementation are available for download from http://www.cs.ucf.edu/∼rvijaya/opph/.

Test case ($n \times m$)	gpph	dpph	opph
50×50	0.11	0.01	0.007
100×100	0.71	0.07	0.017
200×200	4.49	0.53	0.06
500×500	83.2	7.99	0.28
1000×1000	662	66.5	0.43
1000×2000	did not complete	302.78	0.97

Table 1. Performance results - all times are in seconds on a P4 3GHz machine

.

References

[1] V. Bafna, D. Gusfield, G. Lancia, and S. Yooseph. Haplotyping as perfect phylogeny: A direct approach. Technical Report CSE-2002-21, Department of Computer Science, The University of California at Davis, july 2002.

[2] P. Bonizzoni, G. D. Vedova, R. Dondi, and J. Li. The haplotyping problem: And overview of computational models and solutions. *Journal of Computer Scienc and Technology*, 18(6):675–688, July 2003.

[3] A. G. Clark. Inference of haplotypes from pcr-amplified samples of diploid populations. *Mol. Biol. Evol.*, 7:111–122, 1990.

[4] M. J. Daly, J. D. Rioux, S. F. Schaffner, T. J. Hudson1, and E. S. Lander. High-resolution haplotype structure in the human genome. *Nature Genetics*, 29(2):229–32, Oct 2001.

[5] Z. Ding, V. Filkov, and D. Gusfield. A linear time algorithm for the perfect phylogeny haplotyping (pph) problem. In *Proceedings of RECOMB*, MIT, Cambridge, MA, 2005.

[6] E. Eskin, E. Halperin, and R. M. Karp. Large scale reconstruction of haplotypes from genotype data. In *Proceedings of RECOMB*, 2003.

[7] D. Gusfield. Inference of haplotypes from samples of diploid populations: Complexity and algorithms. *J Comput Biol.*, 8(3):305–323, 2001.

[8] D. Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *Proceedings of RECOMB*, 2002.

[9] Y. Liu and C.-Q. Zhang. A linear solution for haplotype perfect phylogeny problem. In *International Conference on Bioinformatics and its Applications (ICBA)*, Nova Southeastern University, Fort Lauderdale, USA, 2004.

[10] R. VijayaSatya and A. Mukherjee. An optimal algorithm for perfect phylogeny haplotyping. Technical Report CS-TR-05-01, School of Computer Science, University of Central Florida, Orlando, January 14, 2005.

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