A Near-Linear Time Algorithm for Haplotype Determination on General Pedigrees

Duong D. Doan*, Patricia A. Evans, Joseph D. Horton
Faculty of Computer Science, University of New Brunswick
Fredericton, N.B., Canada, E3B 5A3
{b89ct, pevans, jdh}@unb.ca

Abstract

An $O(nm\alpha(m))$ time algorithm is given for inferring haplotypes from genotypes of non-recombinant pedigree data, where $n$ is the number of members, $m$ is the number of sites, and $\alpha(m)$ is the inverse of the Ackermann function. The algorithm works on both tree and general pedigree structures with cycles. Constraints between pairs of heterozygous sites are used to resolve unresolved sites for the pedigree, enabling the algorithm to avoid problems previously experienced for non-tree pedigrees.
1 Introduction

Haplotypes indicate which gene variations are on which chromosome copy, while genotypes indicate only which gene variations are present at each site of the genome. In diploid organisms such as humans, usually only genotypes are collected since the cells of these organisms carry two copies of each chromosome, and a biochemical method to extract single chromosomes directly is expensive (Gusfield 2002). Therefore, a computational methodology to infer haplotypes from genotypes is required. Data from a multigenerational family pedigree or from a population group can be used to deduce the haplotypes for all group members. Haplotype inference is complicated by recombinant data, where complementary parts of both of a parent’s haplotypes can be inherited as a single combined haplotype of a child. Also complicating the problem are pedigree structures that are not trees, where there are multiple inheritance paths between some family members.

The haplotyping problem has been studied extensively in the last few years, both for pedigree and population data. If recombinations are allowed, the problem of inferring haplotypes for pedigrees with the minimum number of recombinations is NP-hard (Li and Jiang 2003b). For reconstructing haplotype configurations for pedigree data, Qian and Beckmann (Qian and Beckmann 2002) proposed a rule-based algorithm with a time complexity $O(2^{d^2}n^2m^3)$, where $d$ is the largest number of children in a family, $n$ is the number of members and $m$ is the number of sites. The main princi-
ple of their algorithm is that the best haplotype configuration for pedigree data is the one that minimizes the number of recombination events (the Minimum-Recombinant Haplotype Configuration (MRHC) problem). In (Li and Jiang 2003b) (Li and Jiang 2003a) Li and Jiang proposed an $O(dmn)$ block-extension algorithm for the MRHC problem using a greedy heuristic to resolve adjacent sites. However, as discussed in (Li and Jiang 2004), this algorithm did not always find the haplotypes that minimized the number of recombinations, and worked under some restrictions. In order to improve the performance and handle missing data, an integer linear programming (ILP) formulation (Li and Jiang 2004) was proposed, in which a branch-and-bound algorithm was used to narrow the search space. In fact, with missing data, the haplotyping problem is NP-hard even if there is no recombinant event in the genomic data (Liu et al. 2005). While inheritance in practice normally allows recombination, analysis of populations is discovering haplotype blocks (Gabriel et al. 2002), series of consecutive sites that are not known to be recombined. These discoveries enable us to limit data to blocks without recombination, especially for the close relationships found in pedigrees.

Various results exist for non-recombinant pedigree data, though the fastest algorithm deals specifically with tree pedigrees and cannot handle multiple inheritance paths. Haplotypes can be inferred for tree pedigrees in linear time (Chan et al. 2006), by capturing parity constraints between members and using them to resolve genotype sites. An $O(nm\alpha(n))$ algorithm (Li and Li 2008) is also proposed for tree pedigrees, with extensions to more com-
plex pedigrees and missing data. However, the complexity of this algorithm does not hold for cycle pedigrees or pedigrees with missing data. For general pedigrees, an $O(m^3 n^3)$ algorithm (Li and Jiang 2003b) represents and solves pedigree constraints using linear equations over the cyclic group $Z_2$; this algorithm has also been improved to take $O(mn^2 + n^3 \log^2 n \log \log n)$ time (Xiao et al. 2007) by eliminating redundant equations in the system. Several deduction techniques have also been combined into the program HAPLORE (Zhang et al. 2005), including a haplotype deduction algorithm (Qian and Beckmann 2002), an inconsistent haplotype elimination algorithm (O’Connell and Weeks 1999), and a haplotype frequency estimation algorithm (Qin et al. 2002). Though quick in practice, HAPLORE is largely described as a set of logic rules, and its computational complexity is not concretely specified.

In this paper, we present an $O(nm^\alpha(m))$ time algorithm for inferring haplotypes from genotypes of non-recombinant pedigree data, where $\alpha(m)$ is the extremely slowly growing inverse of the Ackermann function. Basic concepts and motivation are given in Section 2. Sections 3-7 step through the stages of the algorithm, with preliminary processing in Sections 3-4 and more complex computation in Sections 5-7. Section 8 reports experimental results for randomly generated general pedigrees, which verifies correctness on more than 10,000 pedigrees and uses 640 of them to analyze the efficiency and need for the stages of the algorithm. The implemented algorithm’s performance is measured using a variety of different criteria. This testing confirms the necessity and the efficiency of the stages and shows that the algorithm ran
very quickly and handled a variety of different pedigrees.

2 Basic Concepts

A member is an individual. A set of members is called a family if it includes only two parents and their children; it is a parent-offspring trio (hereafter a trio) if only two parents and one child are considered. A set of families connected through known family relationships is a pedigree. A parent is an internal parent if it is a child of another family; it is an external parent otherwise.

In diploid organisms, a cell contains two copies of each chromosome. The description data of the two copies are called a genotype while those of a single copy are called a haplotype. A specific location in a chromosome is called a site and its state is called an allele. There are two main types of sites, microsatellites and single nucleotide polymorphisms. A microsatellite site has several different states while a single nucleotide polymorphism (SNP) site has exactly two possible states, denoted by 0 and 1. Only SNPs are considered in this paper, like other works on haplotype inference.

If the states at a specific site in two haplotypes are the same, then this site is a homozygous site (0-0 or 1-1); if they differ, it is heterozygous (0-1 or 1-0). Two haplotypes combine together to form one genotype. Each member \( u \) has two haplotypes, denoted by \( h_{1u} \) and \( h_{2u} \), which are vectors of 0 and 1’s of length \( m \), where \( m \) is the number of sites. The genotype
of $u$, $g_u$, is a vector of 0’s, 1’s and 2’s of length $m$, where $g_u[i] = 0$ means $h_{1u}[i] = 0 = h_{2u}[i]$, $g_u[i] = 1$ means $h_{1u}[i] = 1 = h_{2u}[i]$, and where $g_u[i] = 2$ means $\{h_{1u}[i], h_{2u}[i]\} = \{0, 1\}$. We say $h_{1u}$ and $h_{2u}$ are consistent with $g_u$. The complement haplotype of a haplotype $h$ at a heterozygous site is denoted by $\bar{h}$, where $\bar{h} = 1 - h$ so, $\bar{0} = 1$ and $\bar{1} = 0$.

The problem in this paper is to find the haplotypes $h_{1u}$ and $h_{2u}$ for all members $u$, given their genotypes $g_u$. When $g_u[i] = 0$ or 1, then $h_{1u}[i]$ and $h_{2u}[i]$ are known, but if $g_u[i] = 2$, we may not yet know the value of $h_{1u}[i]$ and $h_{2u}[i]$, in which case we give the value "?". In this case the site is unresolved.

In the absence of recombinant events, haplotypes of members in a pedigree follow the Mendelian law of inheritance, where the two haplotypes of a child are transferred from its parents, one haplotype from its father and the other from its mother. In this work, we address the problem of inferring haplotypes for pedigree members based on their genotype data and the relationships between members. Our problem can be formally defined as follows.

**Pedigree Haplotype Inference (PHI):** Given a pedigree consisting of $n$ members, and given a genotype $g_u$ of length $m$ for each member $u$, find haplotypes $h_{1u}$ and $h_{2u}$ for each member such that:

1. vector $h_{1u}$ and $h_{2u}$ are consistent with $g_u$;

2. of a individual’s two haplotypes, one is identical to one of the haplotypes of one parent, and the other is identical to one of the haplotypes of the other parent.
In the PHI problem, all genotype values are from the same set of sites. If there are multiple sets of haplotypes that are consistent with the data, then all of them are determined. Each set of haplotypes is called a haplotype configuration.

Chan et al. (Chan et al. 2006) propose a linear time algorithm for inferring haplotypes from genotypes. This algorithm works well with tree pedigree structures but it does not work with a pedigree structure that has cycles. The problem with this algorithm is that it captures parity constraints between members for each site and thus cannot maintain parity constraints along paths once a cycle exists. We overcome this problem by capturing parity constraints among sites within each member. Our algorithm works on both tree and general pedigree structures.

In this paper, we use a running example (Figure 1.a) adapted from (Chan et al. 2006) to illustrate the steps of our algorithm. We have added six more members to create a cycle in the pedigree that cannot be solved by the existing linear algorithm for tree pedigrees (Chan et al. 2006) due to the existence of multiple inheritance paths between members. In Figure 1.a, the genotype of a member is displayed below the member’s name. Each child in the pedigree has an arrow from each of its two parents.

The algorithm as implemented assumes every member has two parents or no parents. If a pedigree were to contain a member who has only one parent in the pedigree, we could add a second dummy external parent whose genotypes are all heterozygous, and thus generate any possible haplotype.
We use \( u \) and \( v \) to represent members, from 1 to \( n \); and \( i \), \( j \), and \( r \) to represent sites, from 1 to \( m \).

3 Initial Data Validation

The algorithm starts by validating the input genotypes along with the pedigree structure to ensure that all members satisfy the Mendelian law of inheritance, where the haplotypes of a child should come from its parents. In every family, a pattern is created from the genotypes of two parents \( u \) and \( v \), and is used to validate against the genotypes of all children. The pattern \( p[i] \) for each site \( i \) (\( i = 1 .. m \)) is determined as follows:

\[
g_u[i] = g_v[i] \neq 2 \rightarrow p[i] = g_u[i];
\]

\[
g_u[i] \neq g_v[i], g_u[i] \neq 2, \text{ and } g_v[i] \neq 2 \rightarrow p[i] = 2;
\]

\[
g_u[i] = 0 \text{ or } g_v[i] = 0 \rightarrow p[i] \neq 1 \text{ denoted by } p[i] = \sim 1;
\]

\[
g_u[i] = 1 \text{ or } g_v[i] = 1 \rightarrow p[i] \neq 0 \text{ denoted by } p[i] = \sim 0;
\]

otherwise \( p[i] = 3 \) (can be 0, 1 or 2).

These rules are listed in order of their priorities when creating a pattern. If the genotype of any child does not fit its pattern, the input data are invalid. For example, if a pattern is 1 2 3 \( \sim 1 \sim 0 \) and the genotypes of two children are 1 2 2 0 1 and 1 2 0 2 1, then the input data are potentially valid. However, if the genotype of a child is 1 2 2 1 1, then the input data are invalid in
position 4, and the algorithm stops. Validating input data using patterns takes $O(nm)$ time since each child’s genotype only needs to be validated against one pattern and this requires comparisons in $m$ sites.

In addition to validating the pedigree using patterns, every family is checked to ensure that there should not be any two sites $i$ and $j$, which are heterozygous in two parents and both heterozygous or both homozygous in a child, but one homozygous and one heterozygous in another child; otherwise there is a family problem (Chan et al. 2006). Checking for a family problem can be done in $O(nm)$ time (Chan et al. 2006). If there is no such violated family in the pedigree, we continue our algorithm; otherwise, the algorithm stops.

4 Haplotype Inferences

This section focuses on inferring haplotypes for each member and determining the origins for the inferred haplotypes.

4.1 Initial Haplotype Inferences

We infer haplotypes for each member by comparing genotypes of each trio in families, in a top-down traverse on the families of the pedigree. In each family, for every trio consisting of two parents $u_1$, $u_2$, and a child $v$, the haplotype pair $(h_{1v}, h_{2v})$ of $v$, where $h_{1v}$ comes from $u_1$ and $h_{2v}$ comes from $u_2$, is inferred as follows. For each site $i$: 
if $g_v[i] \neq 2$ then $h_{1v}[i] = h_{2v}[i] = g_v[i]$

if $g_v[i] = 2$ then

if $g_{u1}[i] \neq 2$ then $h_{1v}[i] = g_{u1}[i]; h_{2v}[i] = \overline{g_{u1}[i]}$

if $g_{u1}[i] = 2$

if $g_{u2}[i] \neq 2$ then $h_{1v}[i] = \overline{g_{u2}[i]}; h_{2v}[i] = g_{u2}[i]$

if $g_{u2}[i] = 2$ then $h_{1v}[i]$ and $h_{2v}[i]$ are not resolved, denoted by $h_{1v}[i] = ?; h_{2v}[i] = ?$

The haplotype pair $\langle h_{1u1}, h_{2u1} \rangle$ of $u_1$ is inferred as follows. If the haplotypes of $u_1$ have not been previously inferred by any trio genotype comparison, let $v$ be the child of $u_1$ that has the maximum number of resolved sites, $h_{1u1}$ is set to $h_{1v}$. For each site $i$:

if $g_{u1}[i] \neq 2$ then $h_{2u1}[i] = g_{u1}[i]$

if $g_{u1}[i] = 2$

if $h_{1u1}[i] \neq ?$ then $h_{2u1}[i] = \overline{h_{1u1}[i]}$ else $h_{2u1}[i] = ?$

The haplotype pair $\langle h_{1u2}, h_{2u2} \rangle$ for $u_2$ is similarly resolved. Figure 1.b shows partially inferred haplotypes for all members in the pedigree. The name of a member displayed below each inferred haplotype means the haplotype comes from that member. If all sites of members in the pedigree are resolved at the end of this phase, then one haplotype configuration is achieved, and the algorithm stops. Otherwise, we proceed to the next phase.
of our algorithm. The partial haplotype determination phase takes $O(nm)$ computational steps.

4.2 Initial Parental Haplotype Determination

One thing that is vital for any haplotype algorithm is to determine which haplotype of a parent is given to a child. In the following, $u_1$, $u_2$ are the parents of child $v$, haplotype $h_{1v}$ comes from $u_1$, and haplotype $h_{2v}$ comes from $u_2$. We want to determine whether $h_{1v}$ comes from $h_{1u_1}$ or $h_{2u_1}$, and
also whether \( h_2_v \) comes from \( h_{1_u_2} \) or \( h_{2_u_2} \).

**Incompatible Haplotype:** Let \( h_{1_v} \) be the haplotype of a child \( v \) that comes from a parent \( u_1 \). If there is a resolved site \( i \) in \( h_{1_v} \) and \( h_{1_u_1} \), where \( h_{1_v}[i] \neq h_{1_u_1}[i] \), then \( h_{1_u_1} \) and \( h_{1_v} \) are incompatible. Otherwise, \( h_{1_u_1} \) and \( h_{1_v} \) are compatible.

Haplotype \( h_{1_v} \) comes from \( h_{1_u_1} \) iff \( h_{1_u_1} \) and \( h_{1_v} \) are compatible and \( h_{2_u_1} \) and \( h_{1_v} \) are incompatible. Haplotype \( h_{1_v} \) comes from \( h_{2_u_1} \) iff \( h_{2_u_1} \) and \( h_{1_v} \) are compatible and \( h_{1_u_1} \) and \( h_{1_v} \) are incompatible. If \( g_{u_1}[j] \neq 2 \) at every site \( j \), we can arbitrary set the origin of \( h_{1_v} \) to either \( h_{1_u_1} \) or \( h_{2_u_1} \).

If there is a haplotype of a child that comes from a parent but is incompatible with both haplotypes of that parent, the input data is invalid and the algorithm stops.

When the number of sites in each member is large, the possibility for determining the origins for all haplotypes in the pedigree using the above criteria is very high. However, for the illustration in the running example, we can determine the exact origin for only one haplotype in the pedigree. In Figure 1.b, the parental haplotype of \( h_{2_Q} \) is determined, denoted by a double line arrow coming from its parental haplotype \( h_{1_P} \). This initial determination of parental haplotypes for all haplotypes in a pedigree takes \( O(nm) \) time.
5 Graphs

There is more data that can be inferred about the haplotype data than is immediately obvious. First, let us introduce important properties of a pedigree.

**Lemma 1.** If $u$ and $v$ are a parent and a child in a family and there are two sites $i$ and $j$ heterozygous in both $u$ and $v$, then the haplotype pairs at $i$ and $j$ in $u$ and $v$ are the same, \{⟨$h_1[u][i], h_1[u][j]$⟩, ⟨$h_2[u][i], h_2[u][j]$⟩\} = \{⟨$h_1[v][i], h_1[v][j]$⟩, ⟨$h_2[v][i], h_2[v][j]$⟩\}.

**Proof.** Because ⟨$h_1[u][i], h_1[u][j]$⟩ and ⟨$h_2[u][i], h_2[u][j]$⟩ are the alleles at $i$ and $j$ in the two haplotypes of $u$, and $u$ is heterozygous at $i$ and $j$, $h_1[u][i] = h_2[u][i]$ and $h_1[u][j] = h_2[u][j]$.

If $h_1[u]$ is transferred to, without loss of generality, $h_1[v]$, then ⟨$h_1[v][i], h_1[v][j]$⟩ is ⟨$h_1[u][i], h_1[u][j]$⟩. Because $i$ and $j$ are heterozygous in $v$, ⟨$h_2[v][i], h_2[v][j]$⟩ is ⟨$h_1[v][i], h_1[v][j]$⟩=⟨$h_1[u][i], h_1[u][j]$⟩=⟨$h_2[u][i], h_2[u][j]$⟩. Thus, the haplotype pairs at $i$ and $j$ in $u$ and $v$ are the same.

If $h_2[u]$ is transferred to, without loss of generality, $h_1[v]$, then ⟨$h_1[v][i], h_1[v][j]$⟩ is ⟨$h_2[u][i], h_2[u][j]$⟩. Because $i$ and $j$ are heterozygous in $v$, ⟨$h_2[v][i], h_2[v][j]$⟩ is ⟨$h_1[v][i], h_1[v][j]$⟩=⟨$h_2[u][i], h_2[u][j]$⟩=⟨$h_1[u][i], h_1[u][j]$⟩. Thus, the haplotype pairs at $i$ and $j$ in $u$ and $v$ are the same even though the two haplotypes of $v$ are exchanged.

**Corollary 1.** If the genotypes of a parent and a child are identical, their haplotypes are the same.
Proof. Let \( \langle h_{1u}, h_{2u} \rangle \) be the haplotype pair of a parent \( u \). In every heterozygous site \( i \), we have \( h_{2u}[i] = \overline{h_{1u}[i]} \).

Without loss of generality, we consider that \( h_{1u} \) is transferred to \( h_{1v} \) of a child \( v \). If genotypes of \( u \) and \( v \) are the same, then the haplotype pair of \( v \) at every heterozygous site \( i \) is \( \langle h_{1v}[i], h_{2v}[i] \rangle = \langle h_{1u}[i], \overline{h_{1u}[i]} \rangle = \langle h_{1u}[i], h_{2u}[i] \rangle \). Moreover, for every homozygous site \( j \), if \( g_{v}[j] = g_{u}[j] \), then \( h_{1u}[j] = h_{2u}[j] = h_{1v}[j] = h_{2v}[j] \). Therefore, if genotypes of \( u \) and \( v \) are the same, then \( \langle h_{1v}, h_{2v} \rangle = \langle h_{1u}, h_{2u} \rangle \). \( \square \)

We extend the property in Lemma 1 to capture constraints of pairs of sites between members as follows.

**Lemma 2.** If \( u \) and \( v \) are two members and there are two sites \( i \) and \( j \) heterozygous in all members on a path from \( u \) to \( v \), then the haplotype pairs at \( i \) and \( j \) in \( u \) and \( v \) are the same.

**Corollary 2.** If there is a path of members, where genotypes of all members on the path are identical, then haplotype pairs of the members are the same.

Since all members have the same genotypes, by Lemma 2 they have the same haplotype pairs.

To give an intuitive idea of our approach, consider Figure 1.c where a green edge is inserted between two heterozygous sites 2 and 3 of \( P \), reflecting that they have the same allele value (0 or 1). Since \( L \) is also heterozygous at sites 2 and 3, the haplotype pairs at these two sites of \( L \) are the same as those of \( P \) (Lemma 1). We transfer the green edge between sites 2 and 3 from
$P$ to $L$ and similarly transfer this green edge to $H$ and to $D$. Because the allele of site 3 of $H$ is 0 and there is a green edge between sites 2 and 3, the allele of site 2 of $H$ is 0. Similarly, the allele of site 3 of $D$ is 0. Propagating these edges thus enables information from $P$ to be used to resolve $H$ and $D$ without needing to resolve the intermediary $L$ first. With these newly resolved sites, we can determine the original haplotypes for the haplotypes of $D$ and $H$.

Other graphs are introduced to help in transferring data from one member to another.

### 5.1 Haplotype Graph

We create a graph $G_H = (V_H, E_H)$ of haplotypes. A node $u$ in $V_H$ is a haplotype, with two for each member. A **black edge** is created between two nodes if one node is the parental haplotype of the other node. Haplotypes connected by a path of black edges must be identical.

Data can be inserted into a haplotype when the other haplotype of its member is found. To facilitate these inferences, we insert **blue edges** into the haplotype graph joining the two haplotypes of each member of the pedigree. All the blue edges are inserted at initialization, while the black edges are determined later, if they can be determined.
5.2 Site Graphs

The haplotype graph sometimes allows us to resolve all the heterozygous sites. However, in the experiments that we ran, about 26% required further processing, as indicated in Table 1 (Section 8). In such a case, a site graph is constructed for each member. The nodes are member-site pairs \((u, i)\), where the genomic data for a member \(u\) is heterozygous at site \(i\). These are the sites that can determine black edges, because the data in the two haplotypes must be different. If two heterozygous sites are resolved and their alleles are the same, we insert a green edge between them. Otherwise, if their alleles are different, we insert a red edge between them.

When an edge is inserted into a site graph, we check if they are in the same component, and if they are, we verify that the sites are resolved consistently. If they are in different components, then we replace the two components by their union, pointing the representative of the smaller component at the representative of the larger. The color of the edge joining the representatives is determined by the parity of the number of red edges on the paths joining the nodes to their representatives and the color of the inserted edge. We can do path compression on these forward edges, whenever we have to find out the component that a node is in, by pointing each node on this path at the component representative.

We cannot construct the whole site graph in linear time in general, but we do need to be able to resolve a site if it is connected by a path in the site graph to some other resolved site. To reduce the time complexity, we do not
record all the edges of each site graph, but record only enough information so that the edge color can be determined quickly if we ever need it. To this end, we use the union-find data structure (Tarjan 1975). We keep track of the connected components, and let one node represent each component. Each node points at another node which is closer to its representative, except that the representatives point at themselves. The color of the edges corresponding to these pointers are sufficient to work out the color of the edge between each node and its representative. The time complexity to build each site graph is $O(m\alpha(m))$ (Tarjan 1975).

One of the components of each site graph consists of the sites that have already been resolved. The other components are constructed from edge propagation, explained in the next section.

We also want to be able to find out all the elements in a given component in time proportional to the number of nodes in the component. To do this, we keep back pointers whenever an edge joins two formerly disconnected nodes. By following the back pointers, we can get to all the nodes in the connected component. The back pointers form a spanning tree of the component.

Figure 1.c shows member $P$ having two site components. The first component is site 1. The second component is sites 2 and 3, and there is a green edge between these two sites. The representative of the second component can be site 2.
6 Subroutines

There are four subroutines to insert and move data around these graphs.

1. Black edge insertion

2. Resolution of a heterozygous site of a member

3. Inserting an edge into a site graph

4. Edge propagation between site graphs

These operations are done in this order of priority. Operations 3 and 4 are only performed after site graphs are constructed, as operations 1 and 2 often completely solve the problem. Black edge insertion is performed as soon as a black edge is discovered. The next two operations, resolution and edge insertion into a site graph, both require more processing than just inserting data into the haplotypes. In both cases, we use queues to control the order of priority of operations, with the resolution taking precedence over the edge insertion. Edge propagation is done only after the queues are empty, when there is nothing else to do.

6.1 Black edge insertion

Immediately upon discovering that one haplotype is passed from a parent to a child, we insert a black edge joining these haplotypes into the haplotype graph, and resolve all sites that we can. To do this efficiently, we can find
the sites that are newly resolved on each side of the new black edge. (Time complexity is \(O(m)\) per black edge: total \(O(nm)\).) The newly resolved sites are copied along all the black edges and inverted along blue edges, using depth first search. The time complexity is proportional to the amount of data to be updated, so is \(O(nm)\). If a discrepancy is found at this time because of a cycle in the pedigree graph, the algorithm stops.

Note that the initial parental haplotype determination (Section 4.2) finds black edges.

### 6.2 Resolution of the site

When for whatever reason a haplotype site \(i\) is resolved in a member \(u\), it is put into a queue for later processing. This later processing consists of:

a. Detecting and checking black edges

   For each member adjacent to \(u\), if there already is a black edge between \(u\) and this member, and the member is already resolved at site \(i\), we check to ensure that the two haplotypes joined by the black edge agree at site \(i\). If there is a black edge and the other end is not resolved, then it can be resolved now, and the result added to the queue. If there is no black edge, and the site \(i\) is already resolved at the adjacent member, then a new black edge is determined and is inserted. If the neighbor is not resolved at site \(i\), then there is nothing to do.

   The total time complexity is \(O(nm)\) as each pair of adjacent members
are considered at most $m$ times from each end.

b. Resolving the other sites to which site $i$ is connected in $u$’s site graph

This substep is done only if site graphs have been built. We follow the original back pointers from the current representative of the component and resolve every site found. We resolve a node the same as the representative if the path between these nodes contains an even number of red edges; otherwise, we resolve it the other way.

The time complexity of all such operations on a site graph is proportional to the number of edges of the spanning tree as each edge is looked at twice, once from each end.

c. Uniting resolved site components

This substep is done only if site graphs have been built. The edge of $u$’s site graph connecting the component of site $i$ to the component of resolved sites is inserted between the components’ representatives. It is colored green if these nodes are resolved the same and red if they are different.

6.3 Initial site graph construction

Before edge propagation starts, some preprocessing steps are done.

a. Setting up transitive links
We traverse each edge of the pedigree graph; for each site such that the site is heterozygous in both members, we insert a transitive link between the site in each of the two site graphs at the equivalent site in the other site graph. This transitive link is colored green. This needs to be done only once for each (member, site) pair, so the combined time is $O(nm)$.

b. Storing edges between resolved sites in site graphs

In each site graph, as we insert an edge between two resolved sites $i$ and $j$, we copy transitive links from one site to the other. Specifically, if $i$ has a transitive link to site $i'$ in member $v$, but $j$ does not have a transitive link to $v$, then we add a transitive link from $j$ to $i'$ in $v$. Thus a site edge in a member has either 0 or 2 transitive links to each adjacent member. The color of the new transitive link is determined by the parity of the number of red edges on the path from $j$ to $i$ to $i'$. The total complexity of copying and coloring transitive links is $O(nm)$ as each operation takes a constant time and the number of transitive links is bounded by the number of sites. We put all red and green edges in the component of resolved sites into the edge queue for later processing. Using the union-find data structure, only the spanning tree of site edges is formed and stored, so the time to store all edges is $O(nma(m))$ (Tarjan 1975).

c. Creating edges from homozygous sites
If \( i \) and \( j \) are both homozygous sites in a member \( u \), and both are heterozygous sites in a parental member \( v \), then we create an edge between sites \( i \) and \( j \) in the site graph at \( v \). It is colored green if \( g_u[i] = g_u[j] \) and colored red otherwise. This edge is put into the edge queue for later processing. The equivalent edges for children are already examined. Only a spanning tree of the heterozygous sites of the neighboring homozygous sites of \( u \) need to be created. The time to create these edges from all homozygous sites in all members to site graphs of their parental members is \( O(nm) \).

### 6.4 Inserting an edge into a site graph

Assume that we are to insert an edge between site \( i \) and site \( j \) in the site graph of a member \( u \), and the color of the edge is known (red or green). From \( i \) we follow the forward pointers up the union-find tree, noting whether each node is resolved the same way or the opposite way with respect to \( i \), until we come to the representative \( r \) of the component, which points at itself. Similarly we follow up from \( j \) to find the representative \( r' \) for the other component.

If it turns out that \( r = r' \), then the edge we were to insert is within one component of the site graph, and does not need to be added. We must check that the color to be inserted agrees. As we are using path compression we point every node that we passed on the way at their representative \( r = r' \), and record the color of the edge from those nodes to \( r \).

If \( r \) and \( r' \) are different, then the edge joins two previously separate com-
ponents. We assume that the component of $r$ is at least as large as that of $r'$. We change the pointer at $r'$ to point at $r$, and add a backpointer at $r'$ onto the list of backpointers from $r$. We record the size of the component of $r$ as the sum of the sizes of the two components. The color of these two pointers is green or red, depending on the parity of the number of red edges on the path from $r$ to $i$ to $j$ to $r'$: green if even, red if odd. Again we point every node that we passed on the way at their representative $r$, and record the color of the edge from those nodes to $r$. The backpointers are not changed.

If one of the components involved is the resolved component, then all the sites of the other component can also be resolved. We can find the sites to be resolved by following the backpointers, and how to resolve them by the parity of the number of red edges on the path from them to $r$. The time complexity of this edge insertion is included in the analysis for edge propagation.

6.5 Edge propagation between site graphs

Assume that we have two sites $i$ and $j$ that are heterogenous in two adjacent members $u$ and $v$. If the two sites are in the same component for member $u$ but not in member $v$, then we can insert the edge between $i$ and $j$ in the site graph for $v$. The color of the new edge is the same as the color of the original, red or green. Propagating edges between site graphs is performed as follows.

We process the edges in the queue one at a time, to check if it can be propagated to the adjacent members. Let the dequeued edge join sites $i$ and
$j$ in member $u$. Two operations are performed.

First, for child member $v$ of $u$, if $i$ is homozygous and $j$ is heterozygous and unresolved in $v$, we resolve $j$ using the color of the edge between $i$ and $j$ in the site graph of $u$.

Second, for member $v$ adjacent to $u$, check if sites $i$ and $j$ have transitive links. If neither do, then nothing needs to be done. If both have transitive links, say $i$ to $i'$ and $j$ to $j'$, then we might be able to do an edge propagation, and, if we cannot, we still have to do a consistency check. Note that transitive links always point at sites which are heterozygous in the original site graph. Suppose $i'$ and $j'$ are in different components in the site graph for $v$, so that the edge from $i'$ to $j'$ can be propagated from $u$ to $v$. As color must remain the same, we add a new edge in the site graph of $v$. Otherwise, if $i'$ and $j'$ are in the same component in the site graph for $v$, the edge does not need to be propagated, but we do a consistency check on the color of the existing edge and the color of the edge we would insert. If there is a color mismatch, the algorithm stops.

Note that a heterozygous site in a member can be a homozygous site in an adjacent member. Therefore, we cannot always transfer a site edge from a member to an identical location in another member like the situation in Figure 2.a. There are some situations where an edge is transferred from a member to another member through an implied edge. Figure 2.b shows such a situation, where sites 1, 2, 3, and 4 are all heterozygous in $u$ while only sites 2 and 4 are heterozygous in $v$. Without the use of transitive links, any
Figure 2: Transferring edges between members using transitive links.

effect of $u$ cannot be used directly to unite sites 2 and 4 in $v$. That is why transitive links are used, as they serve as anchors to transfer implied edges. When we unite sites 1, 2, 3 and 4 of $u$ into one site component, we create transitive links $trans(u_1, v_2)$ from site 1 of $u$ to site 2 of $v$, and $trans(u_3, v_2)$ from site 3 of $u$ to site 2 of $v$. Therefore, edge $\circ$ between sites 3 and 4 of $u$ can be used to unite sites 2 and 4 in $v$. Here, the order of edges to be processed is not important and does not affect our algorithm due to using transitive links. We can process edges $\bigcirc$, $\bullet$, and $\circ$ in any order, but we still can propagate the implied edge $e_{24}$ to $v$.

As the number of edges in the site graphs is $O(nm)$, the amount of work to be done is constant per operation except for finding the component representatives. Finding representatives is done using the union-find data structure, so the total work done is $O(nma(m))$. 
Completing the haplotype graph

It is possible that at some points none of the operations can be performed, yet there still are edges in the pedigree graph for which the haplotype inheritance is not yet determined. There are two cases: the end points are connected by a path in the haplotype graph, or they are not connected.

Suppose that the parent and child nodes of such an edge are connected by a path in the haplotype graph for which all the black edges have been determined. Then the edges and the path combine to form a cycle. In this case, we can determine which black edge must be chosen, depending on the following three sub-cases.

If there is a site that is heterozygous everywhere on the cycle, we determine the black edge to join endpoints of the missing edge such that the number of blue edges is even. If there is a site which is homozygous somewhere on the cycle but heterozygous at the parent node, then the cycle resolves this site at every member on the cycle, including at each endpoint of the missing edge. The black edge to be inserted is determined just by this single site. Otherwise, the parent node is homozygous at all sites so either black edge can be chosen as mentioned in Section 4.2.

After all cases of edges that form a cycle have been handled, every undetermined edge connects two different components of the haplotype graph. One cannot always choose any potential black edge, as shown in Figure 3. Each edge in the pedigree is labeled 0 if the left (right) haplotype of a parent
is transferred to the left (right) haplotype of a child, and it is labeled 1 otherwise. If $AB$, $AC$ and $DE$ are labeled 0, no other black edge is forced by the algorithm. Now choosing $CE$ as either 0 or 1 causes a conflict on edge $BD$.

However, choosing the black edges in the order $AB$, $AC$, $BD$ leads to a solution (Figure 3.c).

Nevertheless one can in practice choose almost any potential black edge. Note that such a problem will not happen in tree pedigrees or pedigrees with one site heterozygous for all members as mentioned earlier in this section. It is unlikely to happen in practice as only one heterozygous and resolved site is required to distinguish which haplotype of a parent is transferred to a child. The problem is more likely to occur in cases with only two or three sites.

We conjecture that as long as the arbitrarily chosen black edges always
occur in the same component of the haplotype graph, the algorithm will always find a solution. To satisfy this condition, the algorithm chooses the next black edge from among the edges joining this special component to other components.

8 Experimental Results

To verify the efficiency and the correctness of the algorithm, we have implemented our algorithm in C++ and tested it on more than 10,000 cases. The results for 640 cases were analyzed in detail. The remaining test cases were used to check the conjecture that the algorithm always works. The program and test data is available at http://www.cs.unb.ca/profs/pevans/research/phi.

Two parameters for a test case are specified, number $n$ of members and number $m$ of sites. The number of external parents is randomly generated ranging from 2 to $\lfloor \frac{n}{2} \rfloor$ members. A list of members is generated, initially containing only the external parents. We assign parents for each remaining member in the pedigree by randomly picking two members from the list of parents. If neither of the two parents has been assigned genders, their genders are assigned arbitrarily with opposite values. If only one of the parents has been assigned a gender, the gender of the other parent is assigned to be its opposite. Otherwise, if the genders of two parents are the same, two other parents are picked. The member with newly assigned parents is then added to the list of members and can become an internal parent. Any possible pedigree
structure can be generated by this method, including highly complex pedigree structures with many cycles. We discard the resulting pedigree if it is not connected.

To generate genotypes for the pedigree, we first randomly generate haplotypes for external parents. Each child member randomly receives one haplotype from each parent based on the Mendelian law of inheritance. The genotype of a member is generated by combining its two generated haplotypes.

The following tables report experimental results from the first dataset, which contains 640 pedigrees. The values of $n$ vary from 10 to 1280, and $m$ from 5 to 640. For each $(m, n)$ pair, 10 cases were generated. Seven features are reported in Tables 1 to 7: number of times the site edge propagation step is used in the cases, average number of free choices, maximum number of free choices, average time to generate the first solution, maximum time to generate the first solution, average time to generate all solutions, maximum time to generate all solutions. A computer with a Dual-Core AMD 2.8GHz CPU and 16 GB of RAM was used.

Since the number of solutions generated from a pedigree can be very large, the number of solutions is counted but only the first 256 solutions are generated. Moreover, whenever a choice of black edge is made, the current data structures are stored to generate all possible solutions later. However, storing data structures is time consuming and does affect the time to generate one solution. Therefore, another computation on the same dataset determines
the time needed to generate one solution without storing data structures, as reported in Tables 8 and 9.

Table 1 shows the number of time the site edge propagation step is used to resolve new sites. The results show that the site edge propagation is used frequently, in 166 of the 640 cases. Site edge propagation is used more often when there are many members but few sites. The chance to determine the origins of haplotypes is higher when the number of sites is larger, which in turn can help to resolve more sites.

Similarly, the more sites a case has, the more chance that the origins of all haplotypes are determined just by the initial parental haplotype determination step. Therefore, the algorithm made many choices when the number of sites is small and the number of members is large, as shown in Tables 2 and 3. In the extreme case with 1280 members and 5 sites, there is an average of more than 21 choices, leading to more than two million solutions. When there were more than 20 sites, few cases had more than 1 choice.

Table 4 displays the average time, in seconds, to generate the first solution. It shows that the running time of the algorithm increased with the increase of the number of members, but appears to be independent of the number of sites. Actually, the running time of the program is not independent of the number of sites as shown in Tables 8 and 9. The time to generate one solution is much reduced when the number of sites increases, as shown in the last three columns of Table 4. Although smaller haplotypes are computed when the number of sites is small, the chance to completely determine
the origins for all haplotypes is less. As a result, the program has to perform
the site edge propagation step and spends time for storing data structures
when free choices of black edges are made. When the number of sites is large,
larger haplotypes are computed but are easier to distinguish, and fewer free
choices are made. The program can solve any pedigree within a minute, as
shown in Table 5.

Tables 6 and 7 display the average and the maximum times to generate
multiple solutions of pedigrees. Since the number of solution can be very
large, $2^{33}$, it is not possible to generate all of them. Instead, only the first
256 solutions will be generated. The program took only about 18 minutes to
generate 256 solutions in the worst case.

The average and maximum times to generate multiple solutions depend
on the number of free choices (Tables 2 and 3). In Tables 6 and 7, the pair
$(n=80, m=80)$ has an unexpectedly long time compared to other results in
the same column. This is the result of a single unusual case with 9 free
choices, taking 83.85 seconds to run.

Whenever a choice of black edge is made, the current data structures
are stored to generate all possible solutions later. Storing data structures
substantially increases the time to generate the first solution. Moreover, it
is very difficult to conclude if the running time of the algorithm depends on
the number of members and sites, especially when the number of choices is
large. Aside from these issues, some users are only interested in finding one
solution for a given pedigree. Therefore, another experiment is performed
without storing the current data structures. The timing results appear in Tables 8 and 9.

According to these results, the running time of the program usually increases with the number of sites. This increase is more evident in Table 8 for the cases with 20, 40 and 160 members, where the number of times site edge propagation is used relatively the same (as shown in Table 1).

When $n=640$ and $m=320$, and $n=1280$ and $m=230$, the running times to generate one solution without storing data structures (Tables 8 and 9) are longer than storing data structures (Tables 4 and 5). The reason is that there are some cases among 10 datasets that have only one solution, which do not need to store data structures, and these cases unexpectedly ran faster in the previous testing as shown in Tables 4 and 5.

The program was also tested on a dataset containing more than 10,000 randomly generated pedigrees. The number of members increased from 10 to 640 and the number of sites from 5 to 635, with 10 values added for each increase. The experimentation always finds a solution, and thus supports the conjecture stated at the end of the previous section.

9 Conclusion

In this paper, we present an $O(nma(m))$ time algorithm for inferring haplotypes from genotypes of non-recombinant pedigree data, where $\alpha(m)$ is the extremely slowly growing inverse of the Ackermann function. This nearly-
linear time algorithm works on both tree and general pedigree structures, as confirmed by extensive experimentation, and comes from implementing the union-find data structure on graphs of related sites. Our result improves on the work of (Xiao et al. 2007) by a factor of $n$ and additionally handles cycle pedigree situations when compared to the work of (Chan et al. 2006) and (Li and Li 2008). Future work will determine how our algorithm can be extended to handle recombinant and missing data.

References


Table 1: Number of times the site edge propagation is used over 10 cases.

<table>
<thead>
<tr>
<th>♯ sites</th>
<th>♯ members</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>3</td>
</tr>
<tr>
<td>160</td>
<td>1</td>
</tr>
<tr>
<td>320</td>
<td>0</td>
</tr>
<tr>
<td>640</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 2: Average number of free choices.

<table>
<thead>
<tr>
<th># sites</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.20</td>
<td>0.20</td>
<td>0.80</td>
<td>1.20</td>
<td>2.90</td>
<td>4.70</td>
<td>13.60</td>
<td>21.40</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.20</td>
<td>0.30</td>
<td>0.80</td>
<td>0.20</td>
<td>0.60</td>
<td>1.20</td>
<td>4.40</td>
</tr>
<tr>
<td>20</td>
<td>0.00</td>
<td>0.50</td>
<td>0.50</td>
<td>0.30</td>
<td>0.60</td>
<td>0.10</td>
<td>0.00</td>
<td>0.10</td>
</tr>
<tr>
<td>40</td>
<td>0.10</td>
<td>0.40</td>
<td>0.50</td>
<td>0.40</td>
<td>0.70</td>
<td>0.80</td>
<td>1.50</td>
<td>0.00</td>
</tr>
<tr>
<td>80</td>
<td>0.40</td>
<td>0.20</td>
<td>0.40</td>
<td>1.60</td>
<td>0.30</td>
<td>0.10</td>
<td>0.80</td>
<td>0.70</td>
</tr>
<tr>
<td>160</td>
<td>0.10</td>
<td>0.20</td>
<td>0.30</td>
<td>0.00</td>
<td>0.00</td>
<td>0.40</td>
<td>0.00</td>
<td>0.70</td>
</tr>
<tr>
<td>320</td>
<td>0.00</td>
<td>0.20</td>
<td>0.50</td>
<td>0.00</td>
<td>0.20</td>
<td>0.10</td>
<td>0.10</td>
<td>0.40</td>
</tr>
<tr>
<td>460</td>
<td>0.00</td>
<td>0.40</td>
<td>0.50</td>
<td>0.40</td>
<td>0.90</td>
<td>0.50</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Table 3: Maximum number of free choices.

<table>
<thead>
<tr>
<th># sites</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>45</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>160</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>320</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>640</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 4: Average time (seconds) to generate the first solution.

<table>
<thead>
<tr>
<th>♯ sites</th>
<th>♯ members</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.11</td>
<td>0.14</td>
<td>0.42</td>
<td>0.75</td>
<td>2.03</td>
<td>4.04</td>
<td>10.56</td>
<td>23.39</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.04</td>
<td>0.14</td>
<td>0.24</td>
<td>0.60</td>
<td>0.75</td>
<td>1.73</td>
<td>4.07</td>
<td>10.78</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.04</td>
<td>0.21</td>
<td>0.30</td>
<td>0.40</td>
<td>0.94</td>
<td>1.45</td>
<td>2.51</td>
<td>5.44</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0.07</td>
<td>0.19</td>
<td>0.30</td>
<td>0.42</td>
<td>1.02</td>
<td>1.57</td>
<td>3.36</td>
<td>4.94</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>0.18</td>
<td>0.14</td>
<td>0.31</td>
<td>0.97</td>
<td>0.93</td>
<td>1.40</td>
<td>4.84</td>
<td>9.85</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>0.06</td>
<td>0.16</td>
<td>0.32</td>
<td>0.31</td>
<td>0.63</td>
<td>2.18</td>
<td>2.53</td>
<td>7.56</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>0.04</td>
<td>0.19</td>
<td>0.38</td>
<td>0.31</td>
<td>1.68</td>
<td>2.44</td>
<td>5.01</td>
<td>14.62</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>0.04</td>
<td>0.46</td>
<td>1.07</td>
<td>2.47</td>
<td>4.92</td>
<td>4.06</td>
<td>2.64</td>
<td>5.48</td>
<td></td>
</tr>
</tbody>
</table>
Table 5: Maximum time (seconds) to generate the first solution.

<table>
<thead>
<tr>
<th># sites</th>
<th># members</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>0.42</td>
<td>0.48</td>
<td>0.96</td>
<td>1.18</td>
<td>3.07</td>
<td>6.63</td>
<td>24.83</td>
<td>30.29</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>0.05</td>
<td>0.44</td>
<td>0.48</td>
<td>1.30</td>
<td>1.15</td>
<td>2.83</td>
<td>5.01</td>
<td>22.73</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0.05</td>
<td>0.61</td>
<td>1.08</td>
<td>1.35</td>
<td>2.11</td>
<td>2.93</td>
<td>3.00</td>
<td>10.84</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>0.40</td>
<td>0.67</td>
<td>1.39</td>
<td>1.57</td>
<td>2.72</td>
<td>3.94</td>
<td>11.00</td>
<td>6.26</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>0.71</td>
<td>0.42</td>
<td>0.91</td>
<td>3.48</td>
<td>2.34</td>
<td>4.00</td>
<td>8.61</td>
<td>18.63</td>
</tr>
<tr>
<td>160</td>
<td></td>
<td>0.33</td>
<td>0.52</td>
<td>0.98</td>
<td>0.37</td>
<td>0.79</td>
<td>6.35</td>
<td>3.46</td>
<td>29.78</td>
</tr>
<tr>
<td>320</td>
<td></td>
<td>0.05</td>
<td>1.27</td>
<td>2.44</td>
<td>0.38</td>
<td>6.32</td>
<td>12.18</td>
<td>25.58</td>
<td>52.66</td>
</tr>
<tr>
<td>640</td>
<td></td>
<td>0.05</td>
<td>2.00</td>
<td>3.61</td>
<td>7.83</td>
<td>16.65</td>
<td>29.87</td>
<td>3.25</td>
<td>7.34</td>
</tr>
</tbody>
</table>
Table 6: Average time (seconds) to generate first 256 solutions.

<table>
<thead>
<tr>
<th># sites</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.13</td>
<td>0.16</td>
<td>0.66</td>
<td>1.13</td>
<td>9.31</td>
<td>61.51</td>
<td>205.40</td>
<td>423.34</td>
</tr>
<tr>
<td>10</td>
<td>0.04</td>
<td>0.20</td>
<td>0.27</td>
<td>0.85</td>
<td>0.77</td>
<td>1.99</td>
<td>4.88</td>
<td>38.42</td>
</tr>
<tr>
<td>20</td>
<td>0.04</td>
<td>0.29</td>
<td>0.50</td>
<td>0.55</td>
<td>1.18</td>
<td>1.47</td>
<td>2.51</td>
<td>5.47</td>
</tr>
<tr>
<td>40</td>
<td>0.08</td>
<td>0.28</td>
<td>0.70</td>
<td>0.79</td>
<td>1.65</td>
<td>1.76</td>
<td>16.16</td>
<td>4.94</td>
</tr>
<tr>
<td>80</td>
<td>0.26</td>
<td>0.17</td>
<td>0.39</td>
<td>9.28</td>
<td>1.00</td>
<td>1.42</td>
<td>5.41</td>
<td>11.17</td>
</tr>
<tr>
<td>160</td>
<td>0.07</td>
<td>0.19</td>
<td>0.38</td>
<td>0.31</td>
<td>0.63</td>
<td>2.29</td>
<td>2.53</td>
<td>20.28</td>
</tr>
<tr>
<td>320</td>
<td>0.04</td>
<td>0.25</td>
<td>1.25</td>
<td>0.30</td>
<td>1.72</td>
<td>2.47</td>
<td>5.06</td>
<td>15.42</td>
</tr>
<tr>
<td>640</td>
<td>0.04</td>
<td>0.58</td>
<td>1.25</td>
<td>2.56</td>
<td>7.58</td>
<td>5.72</td>
<td>2.64</td>
<td>5.48</td>
</tr>
</tbody>
</table>
Table 7: Maximum time (seconds) to generate first 256 solutions.

<table>
<thead>
<tr>
<th>η sites</th>
<th>η members</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10  20  40  80  160  320  640  1280</td>
</tr>
<tr>
<td>5</td>
<td>0.52  0.57  2.35  2.58  52.79  282.42  456.41  1084.51</td>
</tr>
<tr>
<td>10</td>
<td>0.05  0.53  0.60  2.78  1.27  3.54  8.70  238.61</td>
</tr>
<tr>
<td>20</td>
<td>0.05  1.07  2.60  2.83  3.78  3.10  3.00  11.20</td>
</tr>
<tr>
<td>40</td>
<td>0.49  1.13  5.09  5.27  8.22  5.91  139.05  6.26</td>
</tr>
<tr>
<td>80</td>
<td>1.33  0.57  1.37  83.85  2.94  4.16  11.83  30.59</td>
</tr>
<tr>
<td>160</td>
<td>0.42  0.67  1.48  0.37  0.79  7.28  3.46  157.01</td>
</tr>
<tr>
<td>320</td>
<td>0.05  1.79  11.15  0.38  6.48  12.46  26.05  58.65</td>
</tr>
<tr>
<td>640</td>
<td>0.05  2.65  3.94  8.10  40.59  46.50  3.25  7.34</td>
</tr>
</tbody>
</table>
Table 8: Average time (seconds) to generate one solution (no storing data structures).

<table>
<thead>
<tr>
<th>♯ sites</th>
<th>♯ members</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>0.04</td>
<td>0.08</td>
<td>0.19</td>
<td>0.43</td>
<td>0.83</td>
<td>1.78</td>
<td>3.76</td>
<td>7.22</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>0.04</td>
<td>0.09</td>
<td>0.17</td>
<td>0.40</td>
<td>0.69</td>
<td>1.52</td>
<td>3.47</td>
<td>8.16</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0.04</td>
<td>0.10</td>
<td>0.18</td>
<td>0.32</td>
<td>0.78</td>
<td>1.39</td>
<td>2.45</td>
<td>5.30</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>0.04</td>
<td>0.09</td>
<td>0.18</td>
<td>0.33</td>
<td>0.81</td>
<td>1.47</td>
<td>2.80</td>
<td>4.95</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>0.05</td>
<td>0.10</td>
<td>0.23</td>
<td>0.51</td>
<td>0.83</td>
<td>1.38</td>
<td>4.49</td>
<td>9.31</td>
</tr>
<tr>
<td>160</td>
<td></td>
<td>0.04</td>
<td>0.12</td>
<td>0.27</td>
<td>0.30</td>
<td>0.61</td>
<td>2.07</td>
<td>2.48</td>
<td>7.04</td>
</tr>
<tr>
<td>320</td>
<td></td>
<td>0.04</td>
<td>0.14</td>
<td>0.24</td>
<td>0.29</td>
<td>1.68</td>
<td>2.41</td>
<td>5.05</td>
<td>14.86</td>
</tr>
<tr>
<td>640</td>
<td></td>
<td>0.04</td>
<td>0.35</td>
<td>1.00</td>
<td>2.47</td>
<td>4.93</td>
<td>3.89</td>
<td>2.62</td>
<td>5.34</td>
</tr>
</tbody>
</table>
Table 9: Maximum time (seconds) to generate one solution (no storing data structures).

<table>
<thead>
<tr>
<th>♯ sites</th>
<th>♯ members</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.06</td>
<td>0.12</td>
<td>0.27</td>
<td>0.67</td>
<td>1.16</td>
<td>2.24</td>
<td>4.73</td>
<td>9.55</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.05</td>
<td>0.13</td>
<td>0.27</td>
<td>0.59</td>
<td>0.97</td>
<td>2.45</td>
<td>4.46</td>
<td>9.74</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.05</td>
<td>0.14</td>
<td>0.31</td>
<td>0.60</td>
<td>1.27</td>
<td>2.72</td>
<td>2.81</td>
<td>9.74</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0.08</td>
<td>0.17</td>
<td>0.35</td>
<td>0.72</td>
<td>1.56</td>
<td>3.02</td>
<td>6.20</td>
<td>6.54</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>0.10</td>
<td>0.22</td>
<td>0.44</td>
<td>0.93</td>
<td>1.78</td>
<td>3.95</td>
<td>7.97</td>
<td>16.11</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>0.15</td>
<td>0.31</td>
<td>0.70</td>
<td>0.40</td>
<td>0.76</td>
<td>6.10</td>
<td>3.11</td>
<td>25.12</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>0.04</td>
<td>0.70</td>
<td>1.13</td>
<td>0.36</td>
<td>6.51</td>
<td>11.87</td>
<td>26.27</td>
<td>53.69</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>0.05</td>
<td>1.44</td>
<td>3.51</td>
<td>8.14</td>
<td>16.87</td>
<td>28.51</td>
<td>3.07</td>
<td>6.79</td>
<td></td>
</tr>
</tbody>
</table>