Journal of Bioinformatics and Computational Biology
Vol. 7, No. 3 (2009) 521–545
© Imperial College Press

Imperial College Press
www.icpress.co.uk

AN ALMOST LINEAR TIME ALGORITHM FOR A GENERAL HAPLOTYPE SOLUTION ON TREE PEDIGREES WITH NO RECOMBINATION AND ITS EXTENSIONS

XIN LI* and JING LI†

Department of Electrical Engineering and Computer Science
Case Western Reserve University
10900 Euclid Ave, Cleveland
Ohio 44106, USA
**xin.li2@case.edu*
†jingli@case.edu

Received 11 July 2008
Revised 21 November 2008
Accepted 26 November 2008

We study the haplotype inference problem from pedigree data under the zero recombination assumption, which is well supported by real data for tightly linked markers (i.e. *single nucleotide polymorphisms* (SNPs)) over a relatively large chromosome segment. We solve the problem in a rigorous mathematical manner by formulating genotype constraints as a linear system of inheritance variables. We then utilize disjoint-set structures to encode connectivity information among individuals, to detect constraints from genotypes, and to check consistency of constraints. On a tree pedigree without missing data, our algorithm can output a general solution as well as the number of total specific solutions in a nearly linear time $O(mn \cdot \alpha(n))$, where m is the number of loci, n is the number of individuals and α is the inverse Ackermann function, which is a further improvement over existing ones. We also extend the idea to looped pedigrees and pedigrees with missing data by considering existing (partial) constraints on inheritance variables. The algorithm has been implemented in C++ and will be incorporated into our PedPhase package. Experimental results show that it can correctly identify all 0-recombinant solutions with great efficiency. Comparisons with other two popular algorithms show that the proposed algorithm achieves 10 to 10^5-fold improvements over a variety of parameter settings. The experimental study also provides empirical evidences on the complexity bounds suggested by theoretical analysis.

Keywords: Haplotype inference; linear system; disjoint-set.

1. Introduction

Experimental data have shown that genetic variation is structured in haplotypes rather than isolated SNPs[1] and haplotypes may provide substantially increased power in detecting gene-disease association. However, the human genome is a diploid

†Corresponding author.

and, in practice, haplotype data are not collected directly, especially in large scale sequencing projects mainly due to cost considerations. Hence, efficient and accurate computational methods and computer programs for the inference of haplotypes from genotypes are highly needed.

Recent years have witnessed intensive research on haplotyping methods (see review Refs. 2–6), mainly driven by the HapMap project.[1] For family data, there exist two types of haplotyping methods, statistical methods and combinatorial (or rule-based) methods. There is a tendency to merge these two types of approaches.[7,8] In general, the goal of statistical approaches[9] is to find a haplotype assignment for each individual with the maximum likelihood or to output all consistent solutions with their corresponding probabilities. Recently, population haplotype frequencies have been taken into considerations[9] to account for correlations among tightly linked markers (known as *linkage disequilibrium*). A key step in most statistical approaches is to enumerate all possible inheritance patterns and to check the genotype consistency for each of them.[9] Due to the large degrees of freedom, this step usually leads to high time complexity (usually exponential hence computational intractable for large data sets). On the other hand, rule-based algorithms first partially infer haplotypes or inheritance vectors based on genotype constraints, and then search final solutions from the reduced space. Therefore, rule-based algorithms[7,10–13] can potentially gain advantage over statistical methods in efficiency. The zero recombinant assumption states that recombination is nonexistent within a pedigree for a sufficiently large number of tightly linked markers. As a realistic assumption, it has been used in both statistical approaches as well as rule-based approaches. Furthermore, a solution to the problem with no recombinant can be served as a subroutine of a general procedure to solve the general haplotype inference problem. Therefore, investigation of efficient algorithms to obtain all 0-recombinant solutions from a pedigree is of high interests.

For a given pedigree, the goal of the *zero recombinant haplotype configuration* (ZRHC) problem is to identify all possible haplotype assignments with no recombination. An important advance in the development of rule-based algorithms for haplotype inference in pedigrees in general and the ZRHC problem in particular is the introduction of variables to represent uncertainties. The problem can then be discussed and solved with mathematical rigor. Li and Jiang[11] first formulate the problem as a linear system on "*ps*" (a binary indicator of parental source) variables and solve it using Gaussian elimination with a complexity of $O(m^3n^3)$, where m is the number of markers and n is the number of individuals. More recently, Xiao *et al.*[13] formulate another linear system on "*h*" (a binary indicator of inheritance relationship) variables, and lower the complexity to $O(mn^2 + n^3 \log^2 n \log\log n)$. For loop-free (tree) pedigrees, Xiao's method can produce a general solution in $O(mn^2 + n^3)$ and a particular solution in $O(mn + n^3)$ time. Here, a particular solution means a specific assignment for each variable which satisfies the constraints, while a general solution is a description of all solutions in a general form where some variables are designated as free (meaning that they are allowed to take any value), and the remaining variables are represented by a linear combination of these free variables. For tree pedigrees, Chan *et al.*[10] further reduce the complexity of finding a particular solution to a linear time $O(mn)$ by manipulating the constraints on a graph structure. Liu and Jiang[12] also propose an algorithm to produce a particular solution in $O(mn)$ and a general solution in $O(mn^2)$ by further exploring features of their h-variable system on a tree pedigree. However, with missing data, it has been shown that ZRHC is NP-hard.[14] Therefore, it seems impossible to incorporate missing data into a pure linear constraint system without enumerations. Li and Jiang[7] propose an integer linear programming algorithm for the minimum recombinant haplotype inference problem, and it can solve ZRHC with missing data as a special case. However, because it does not use zero recombinant constraints explicitly, it may need to enumerate almost all possible haplotype assignments.

In this paper, we propose an elegant and more efficient algorithm for detecting, recording and consistency checking of constraints on h-variables. Notice that it is not necessary to solve the h-variable system explicitly, as it was in Ref. 13. Instead, we encode constraints on h variables using disjoint-set forests. By applying an adapted disjoint-set union-find procedure, we can update the disjoint-set structures incrementally upon new constraints, and determine the consistency of the encoded linear system simultaneously. Based on the disjoint-set union-find procedure, the proposed algorithm can produce a general solution in almost linear time ($O(mn \cdot \alpha(n))$ for a tree pedigree, where α is the inverse Ackermann function,[15] improved from the best known algorithm with $O(mn^2)$ time complexity.[12] We further extend the algorithm to looped pedigrees and pedigrees with missing data, by utilizing the constraints imposed from existing data. Experimental results show that the algorithm can output all solutions with zero recombinant and it is much more efficient than two popular existing algorithms because of the significant reduction of the enumeration space.

The rest of the paper is organized as follows. In Sec. 2, we introduce the linear system on h variables together with some basic concepts and notations concerning the ZRHC problem. By representing constraints using a linear system, one can formally investigate different strategies to solve the problem in a rigorous manner. Different strategies of manipulating and integrating the constraints will result in different complexities. Our algorithm of detecting and processing constraints from pedigree data is presented in Sec. 3. In both sections, we assume that input genotype data have no missing alleles and the ZRHC problem under this case is polynomially solvable. Our algorithm is almost optimal by achieving a nearly linear time complexity on tree pedigrees with complete data. In Sec. 4, we show how to extend the algorithm to cope with missing data and looped pedigrees by effectively reducing the search space before enumerations. The performance of our algorithm and comparisons with other two programs are examined in Sec. 5. We discuss future directions and make concluding remarks in Sec. 6.

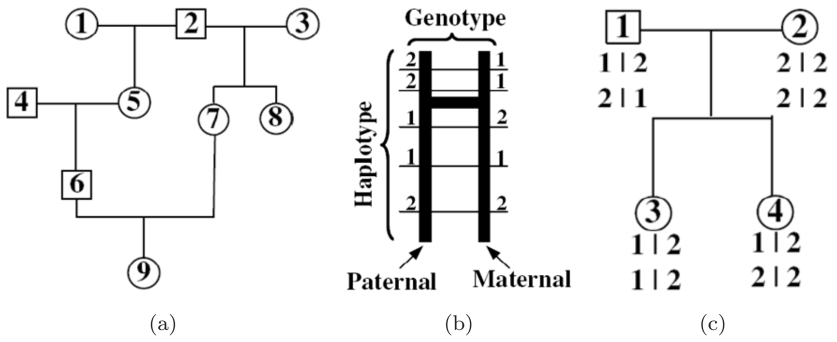


Fig. 1. (a) A pedigree graph. We use a circle to represent a female, and a square to represent a male in a pedigree. (b) A haplotype is composed of all alleles on one chromosome segment. Each allele is an integer value representing the status of a marker at a chromosome locus. (c) A recombination event occurs when a child does not inherit a complete haplotype from its parents. Individual 3 has a paternal haplotype 11 which is not seen in his father. So there must be a crossover between two chromosomes of his father in meiosis, which results in a recombinant haplotype.

2. Preliminaries

A *pedigree graph* indicates the parent–child relationships among an extended family. Figures 1(a) and 2(a) present pedigrees in a conventional manner. The pedigree in Fig. 1(a) has a *mating loop*, where an offspring (node 9) is produced by the mating of two relatives (nodes 6 and 7). A pedigree without mating loops is called a *tree pedigree*. A *nuclear family* only consists of both parents and their children. For any pair of homologous chromosomes from a diploid organism such as human, exactly one is from its father and the other one is from its mother, as illustrated in Fig. 1(b). A physical position on a chromosome is called a *locus* and the status of a locus is called an *allele*, represented using an integer ID. We focus on single nucleotide poly-morphism (SNP) data in this study thus assume that there are only two alternative alleles (i.e. bi-allelic data), which turns out to be the hardest case for ZRHC.[11,13]

At each locus i, a child may inherit either of the paternal or maternal allele. We use a binary variable to indicate the parental sources (ps) of the two alleles in a child.

Definition 1. ps variable $p_i^x \in \{0, 1\}$ is defined for each locus i of each individual x. $p_i^x = 0$ if the smaller allele of locus i is of paternal source, $p_i^x = 1$ if it is of maternal source. We technically let $p_i^x = 0$ if locus i is homozygous (two alleles being the same).

Loosely speaking, a *haplotype* consists of all alleles on a chromosome. *Recombination* events or *crossovers* occur when a child inherits a shuffled version of its parent's two haplotypes (see Fig. 1(c) for an example). However, for a sufficiently large segment of chromosome with m SNPs, the likelihood of recombination between a parent–child pair is extremely small. For example, a rough estimation of the relationship of genetic distances and physical distances is about 1 Mbps/cM. The average marker interval distance of a 500K SNP chip is about 6 Kbps. Therefore, the probability of seeing a single recombination event from a parent–child pair of 170 SNP markers ($\sim$1 Mbps) is only $\sim 1\%$. One can assume that a child inherits an entire haplotype segment from a parent for a sufficiently large number of SNPs (i.e. zero recombinant assumption). In such a case, the inheritance behavior between a parent–child pair is unique throughout all m loci, and it is convenient and practically appealing to use a single binary variable (h) to indicate the inheritance behavior between a parent–child pair.

Definition 2. Inheritance variable $h^{x_1x_2} \in \{0, 1\}$ is defined between a parent x_1 and a child x_2. $h^{x_1x_2} = 0$ if x_2 inherits the paternal haplotype of x_1, $h^{x_1x_2} = 1$ if x_2 inherits the maternal haplotype of x_1.

2.1. *Mendelian constraints as a linear system*

Mendelian laws of inheritance impose constraints on *ps* and h variables for each parent–child pair at each locus. These constraints can be represented by a linear relationship of *ps* and h variables over the group $(Z_2, +)$ (where $0 + 0 = 0$, $0+1 = 1$, $1+1 = 0$). Table 1 summarizes all cases of constraints at a certain locus i for a parent–child pair. When an individual is homozygous at a certain locus, its *ps* variable at this locus is determined by definition. When one or both of the parents of an individual are homozygous at a certain locus, this individual's *ps* variable at this locus is also determined. In both cases, the *ps* variable is pre-determined. In all the other cases, there is a constraint for each parent–child pair between *ps* variables and the h variable, as shown in the last three cases in Table 1. The constraints introduced by the zero recombinant assumption is enforced by the single h variable between each parent–child pair. Therefore, the system formed by the sets of constraints collected based on Table 1 consists of all the constraints from data. The satisfiability (or consistency) of this system is equivalent to whether there is a zero recombinant solution.

2.2. *Locus graphs*

To process constraints, Xiao *et al.*[13] introduced the concept of locus graphs. We give a brief introduction here for the sake of completeness. A locus graph $L_i(V, E_i)$

Table 1. Constraints for a parent–child pair x, y.

Genotype		Constraint		
Parent x	Child y		If x is father	If x is mother
1/1	1/2	$p_i^x = 0$	$p_i^y = 0$	$p_i^y = 1$
2/2	1/2	$p_i^x = 0$	$p_i^y = 1$	$p_i^y = 0$
1/2	1/2		$p_i^y = p_i^x + h^{xy}$	$p_i^y = p_i^x + h^{xy} + 1$
1/2	1/1	$p_i^y = 0$	$p_i^y = p_i^x + h^{xy}$	$p_i^y = p_i^x + h^{xy}$
1/2	2/2	$p_i^y = 0$	$p_i^y = p_i^x + h^{xy} + 1$	$p_i^y = p_i^x + h^{xy} + 1$

is constructed for each locus i to record the constraints on h variables. V consists of all individuals as nodes. There exists an edge in E_i between a parent–child pair only if the *ps* variables of this pair are constrained on the correspondent h variable, i.e. when the parent is heterozygous at locus i (the last three cases in Table 1). Each edge is also labeled by the h variable and the constant associated with the constraint. We refer to this kind of constraints (a linear equation consists of *ps* variables and an h variable) as *edge constraints*. Figure 2(b) shows an example of a locus graph.

The original idea of Ref. 13 was to integrate edge constraints to construct a new subsystem that only consists of h variables. Their algorithm then solved the subsystem and used h variable solutions to solve *ps* variables. We also record edge constraints on locus graphs. However, instead of explicitly listing and solving the constraints on h variables, we use disjoint-set structures to collect, encode and thus examine the consistency of these constraints, which help us achieve a better time complexity result to obtain a general solution.

2.3. *Linear constraints on h variables*

There are essentially two types of constraints on h variables in a locus graph L_i: *path constraints* and *cycle constraints*. Notice that the classification of constraints here is more succinct than those in previous work[12,13] because our method of handling

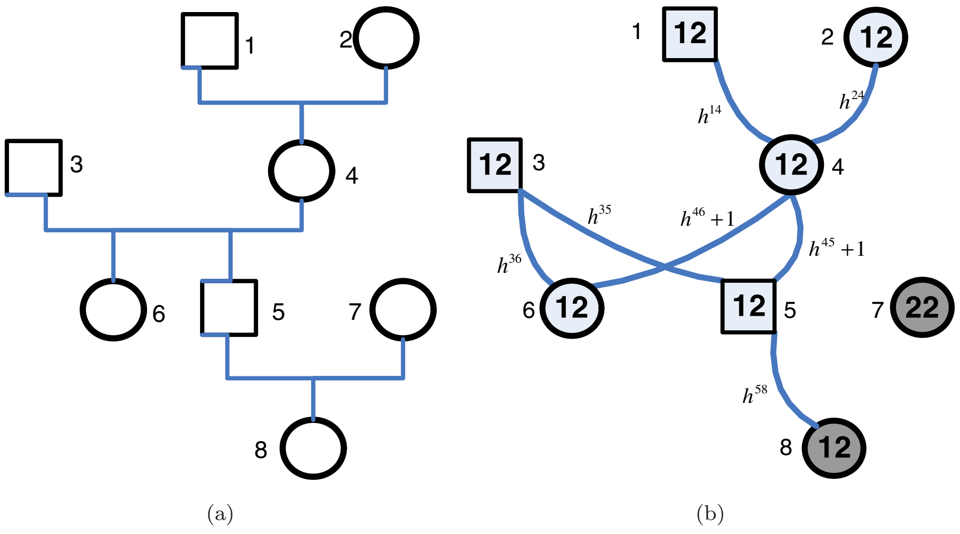


Fig. 2. (a) A pedigree with eight members. (b) Given the genotype at a certain locus i, the correspondent locus graph L_i and h variable constraints. *ps* variables of shaded members (2, 4, 7, 8) are pre-determined. From this locus graph, we can generate two non-redundant h variable constraints, one is a cycle constraint, $h^{35} + h^{36} + h^{45} + h^{46} = 0$ (formed by individual 3, 4, 5, 6), the other is a path constraint, $h^{45} + h^{58} = 0$ (from individual 4 to 8 via 5).

constraints does not require further discrimination of them. According to Table 1, each edge e_{xy} in a locus graph represents an edge constraint in the form $p_i^x + p_i^y = h^{xy} + c_i^{xy}$, where c_i^{xy} is a constant $\in \{0,1\}$. We use a subscript i for c_i^{xy} because for different loci, the constant between a parent–child pair may be different, which depends on the genotype at that locus as specified in Table 1. For a path $P_{\widetilde{s,t}}$ from individual s to individual t in locus graph L_i, if we sum up all edge constraints on this path, we have

$$\sum_{e_{xy}\in P_{\widetilde{s,t}}} (p_i^x + p_i^y) = p_i^s + p_i^t = \sum_{e_{xy}\in P_{\widetilde{s,t}}} (h^{xy} + c_i^{xy}).$$

If p_i^s and p_i^t are pre-determined constants, we end up with a path constraint on h variables, which is

$$\sum_{e_{xy}\in P_{\widetilde{s,t}}} h^{xy} = p_i^s + p_i^t + \sum_{e_{xy}\in P_{\widetilde{s,t}}} c_i^{xy}, \tag{1}$$

where the right-hand side is a constant. Similarly, for a cycle C in locus graph L_i, which may exist even on a tree pedigree (e.g. when a nuclear family has more than one heterozygous children), we sum up all edge constraints on C,

$$\sum_{e_{xy}\in C} (p_i^x + p_i^y) = 0 = \sum_{e_{xy}\in C} (h^{xy} + c_i^{xy}),$$

and finally have a cycle constraint on h variables

$$\sum_{e_{xy}\in C} h^{xy} = \sum_{e_{xy}\in C} c_i^{xy}.$$

3. Methods

By exploiting special features of the constraints on h variables, it is not necessary to explicitly list every path and cycle constraint to check their consistency. We employ disjoint-set structures to detect and to check the consistency of constraints on h variables. For each locus graph L_i, we build a disjoint-set structure D_i to encode its connectivity information. We update the disjoint-set structure incrementally upon processing each edge constraint on a locus graph. Path constraints on a locus graph are detected during this process and will be stored in another disjoint-set structure D. The whole algorithm works on $m+1$ such disjoint-set structures, one D_i for each locus graph L_i and one D for encoding all path constraints.

In this section, we assume that the inputs are tree pedigrees with complete data. Cycles on a locus graph from a tree pedigree can only be generated within a nuclear family when it has multiple children. We first discuss a node splitting strategy in Sec. 3.1 to break all such short cycles, to obtain only path constraints for further processing. Construction of D_i from each locus graph L_i to detect path constraints will be discussed in Sec. 3.2. Processing of constraints and consistency check will be discussed in Sec. 3.3 and a general solution of h variables will be decoded from the disjoint-set structure D. Solutions of *ps* variables will then be obtained. The analysis of time complexity and correctness of the algorithm on tree pedigrees will be discussed in Sec. 3.4. One of the advantages of the proposed algorithm is that it can be easily extended to the general cases of looped pedigrees and pedigrees with missing data; we show these extensions in Sec. 4.

3.1. *Split nodes to break cycles*

In order to simplify the constraint detection, we first transform cycle constraints to path constraints by breaking cycles in locus graphs. There are essentially two kinds of cycles in a locus graph: *global* cycles that are introduced by marriages between relatives and *local* cycles that are introduced by multiple children within one nuclear family [e.g. Fig. 2(b)]. Only local cycles will exist in a tree pedigree and will be dealt with in this subsection. The treatment of global cycles will be deferred to Sec. 4.1 when we discuss the extension to looped pedigrees. We break local cycles for each nuclear family with multiple children by splitting some child nodes and by remounting their edges on each locus graph. More specifically, when a nuclear family has multiple children, any child node v (except an arbitrarily fixed one v_0) and its genotypes will be duplicated to create a new node v' in the same manner across all locus graphs. New *ps* variables will be introduced for these duplicated nodes. For each splitting node v, the edge from its mother (if there is) will be reconnected to node v'. All other edges regarding node v remain untouched. Figure 3 shows an example on how node splitting is performed.

By doing so, we technically avoid the treatment of cycle constraints. After the duplication, all new locus graphs (actually *locus trees* now) still have the same set of nodes. Notice that one has to record all local cycle constraints on h variables and constraints that the *ps* variables of duplicated nodes must have the same assignments as those in their original copies. Their constraints can be easily dealt with for local cycles because they only involve local structures (nuclear families). This will be further discussed in the next subsection.

3.2. *Detect path constraints from locus graphs*

We develop an incremental procedure to detect all path constraints from a locus graph by utilizing a disjoint-set structure. As we can see from the constraints on h variables in Eq. (1), a path constraint is specified by the *ps* variables of its end nodes and summation of the constant parity value c_i^{xy} associated with the edge constraint on each of its edges. Our goal is to detect each non-redundant path on a locus graph with pre-determined end nodes and meanwhile obtain the constant parity summation associated with that path.

To do so, we maintain a disjoint-set structure D_i for each locus graph L_i and update it incrementally. The disjoint-set structure is defined by a pair of values $rep_i[v]$, $offset_i[v]$ for each node v in $V(L_i)$. We use subscript i here to emphasize that the disjoint-set structure D_i is specific for each locus graph. $rep_i[v]$ indicates

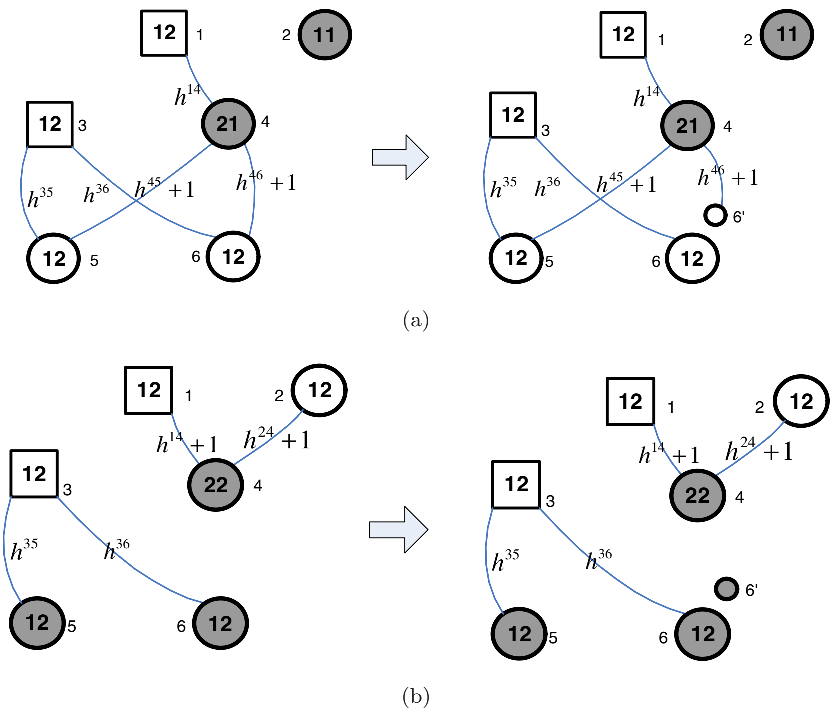


Fig. 3. Node splitting applied to a nuclear family at two loci to remove local cycles. (a) The original locus graph (left), and the locus graph (right) with edges remounted after node 6 was duplicated. (b) A locus graph at another locus before (left) and after (right) node 6 was split. Though no local cycle exists in the locus graph in (b), node 6 was also duplicated so that all locus graphs will still have the same number of nodes after splitting.

the node which acts as the representative of the set containing v. And the offset of a node $offset_i[v]$ indicates the summation of the constants associated with the edge constraints on the path from v to its rep_i. Namely, if $rep_i[v] = v_0$, then $offset_i[v] = \sum_{e_{xy} \in P_{\widetilde{v,v_0}}} c_i^{xy}$, where $P_{\widetilde{v,v_0}}$ is the path with end nodes v and v_0, c_i^{xy} is the constant associated with the edge constraint on edge e_{xy} (as specified in the last three cases of Table 1).

Initially, for every node in V: $rep_i[v] \leftarrow v$, $offset_i[v] \leftarrow 0$. We examine each $e^{xy} \in L_i$ and update D_i by considering the edge constraint $p_i^x + p_i^y = h^{xy} + c_i^{xy}$ represented by e^{xy}. If both $p_i^{rep_i[x]}$ and $p_i^{rep_i[y]}$ are pre-determined, we report a path constraint and record it in D for consistency check (see Sec. 3.3). The two sets represented by $rep_i[x]$ and $rep_i[y]$ will always be merged into one because they are connected by an edge e^{xy} and we always let one pre-determined representative be the representative of the new set if there is such one. At the end, any two nodes connected by a path in L_i will be merged into one set and a set in D_i only consists of connected nodes in L_i. By doing so, we can safely detect all path constraints on L_i. Furthermore, the constant associated with a path constraint between two nodes

s and t in the same set can be reconstructed as

$$\sum_{e_{xy} \in P_{\widetilde{s,t}} \in L_i} c_i^{xy} = \mathit{offset}_i[s] + \mathit{offset}_i[t].$$

The procedure is illustrated in Algorithm 1.

Algorithm 1 Union$_i(x,\, y,\, c_i^{xy})$

```
if both $p_i^{rep_i[x]}$ and $p_i^{rep_i[y]}$ are pre-determined then
    Report a path constraint $P$ from node $rep_i[x]$ to $rep_i[y]$: $\sum_{e_{xy} \in P} h_i^{xy} = c$, where $c = p_i^{rep_i[x]} + p_i^{rep_i[y]} + \mathit{offset}_i[x] + \mathit{offset}_i[y] + c_i^{xy}$. Encode the constraint in $D$ by applying $Union(rep_i[x], rep_i[y], c)$.
end if
if $p_i^{rep_i[y]}$ is not pre-determined then
    $\mathit{offset}_i[rep_i[y]] \leftarrow \mathit{offset}_i[y] + \mathit{offset}_i[x] + c_i^{xy}$
    $rep_i[rep_i[y]] \leftarrow rep_i[x]$
else
    $\mathit{offset}_i[rep_i[x]] \leftarrow \mathit{offset}_i[x] + \mathit{offset}_i[y] + c_i^{xy}$
    $rep_i[rep_i[x]] \leftarrow rep_i[y]$
end if
```

We also need to capture all constraints that may not have been processed yet in the above procedure due to node splitting. This is easy for a tree pedigree which possibly only has local cycles to split. There are *three* possible types of constraints that need special attention due to node splitting, i.e. local cycle constraints themselves, *ps* variables between duplicated nodes and their corresponding splitting nodes, and some path constraints originally existing in the locus graph before splitting, but broken by splitting. We examine each of these constraints by case analysis. First of all, no node splitting is needed if a nuclear family has only one child. Secondly, *a local cycle constraint exists in an original locus graph before splitting if and only if both parents of a nuclear family with multiple children are heterozygous.* Therefore, we only have two cases for nuclear families (with multiple children): (i) both parents are heterozygous (local cycles exist); (ii) at least one parent is homozygous (no local cycles).

We first focus on **case one**. To collect such a local cycle constraint after node splitting, we can examine every splitting node v and its duplicate v'. Based on the splitting strategy, it is easy to see that a cycle constraint exists in the original locus graph if and only if there exists a path between the two nodes v and v' in the new locus graph after node splitting. Notice that when processing edge constraints, any nodes that are connected have been grouped into one set in D_i. Therefore, the existence of a path between v and v' can be verified by checking whether their representatives are the same. That is, for each pair (v, v'), a local cycle constraint exists in the original locus graph before splitting if and only if $rep_i[v] = rep_i[v']$. This local cycle constraint now can be represented by a path constraint P that consists of v', m, v_0, f, v, where m and f are the parent nodes of v, and v_0 is the anchor child node in this nuclear family. The path constraint should have the form of $\sum_{e_{xy} \in P} h_i^{xy} = \mathit{offset}_i[v] + \mathit{offset}_i[v'] + ps_i^{v} + ps_i^{v'}$. However, one should notice that

v' is the duplicate of v and their ps variables should always be the same. Therefore, we will add a path constraint in the form of $\sum_{e_{xy} \in P} h_i^{xy} = \mathit{offset}_i[v] + \mathit{offset}_i[v']$ to D instead. In this way, both the local cycle constraint and the ps variable constraint introduced by node splitting have been enforced. It turns out that the third type of constraints (path constraints originally existing before node splitting that go through the edge e_{mv}) have also been taken care of. Because for each of such a path P with $e_{mv} \in P$, there exists an alternative path P' that goes through the edges $m \to v_0 \to f \to v$. As long as the local cycle constraint has been enforced, the two alternative paths will be equivalent. Because our normal procedure will collect constraint P' from a locus graph after node splitting, P is redundant and can be safely dismissed.

When at least one parent is homozygous (**case two**), no local cycle constraints exist. The ps variables of all children, including the duplicated nodes, are predetermined because at least one parent is homozygous. Therefore ps assignments of v and v' will always be the same. If a path constraint P consists of edge e_{mv} before splitting, it must end at node v because v is predetermined. It is easy to see that it is now being replaced by a path constraint P' consisting of edge $e_{mv'}$ and ending at v'. The only difference between the two paths P and P' is that edge $e_{mv} \in P$ is replaced by $e_{mv'}$. But the constrains on these two paths are the same and only one (i.e. constraint on P', which has been processed) is needed.

Thus all three types of constraints have been correctly processed. We illustrate the cases using an example in Fig. 4.

Figure 5 gives an example on how to detect constraints on a locus graph L_i. In the actual implementation of a disjoint-set forest, a node may not directly point to its set representative. We omit the details (see Refs. 15 and 17) here for clear demonstration purpose.

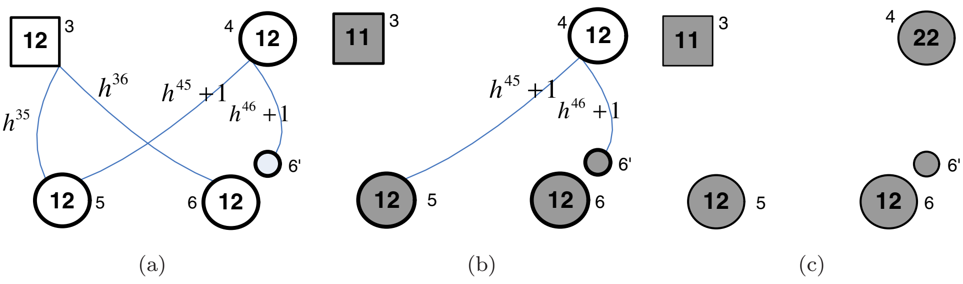


Fig. 4. This example illustrates all possible patterns of locus graphs of a nuclear family on a tree pedigree. (a) If neither of the parents is homozygous at this locus, then there should be a loop constraint, $h^{36} + h^{35} + h^{45} + h^{46} = c$. Since we split node 6, it is expressed as a path constraint on path $P_{\widetilde{6,6'}}$. Since the locus graph is still connected, no path via this nuclear family will be broken up due to the split of node 6. (b) and (c) If one or both of the parents are homozygous at this locus, then both of the children are pre-determined. In this situation, path constraints such as $P_{\widetilde{5,6'}}$ will only take the children as end nodes such that they remain on a consecutive path, unaffected by the split of node 6.

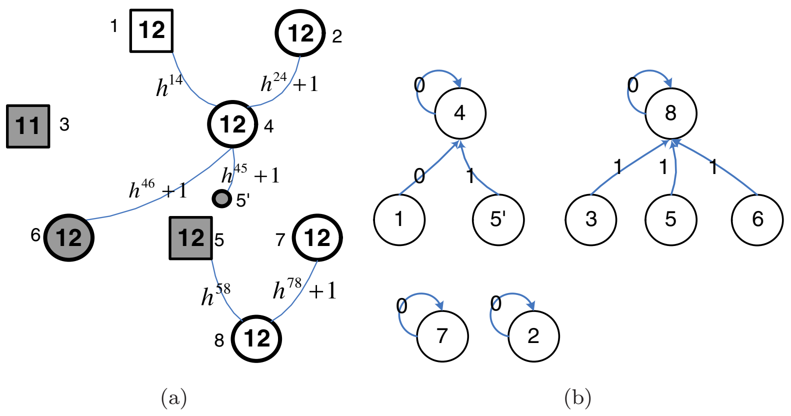


Fig. 5. An example shows the detection of all constraints from a locus graph after node splitting. (a) Locus graph L_i of a pedigree with eight nodes at a certain locus i. Shaded nodes are pre-determined. (b) The disjoint-set forest formed by adding edges 1–4, 4–5′, 3–5, 3–6 and 5–8 of the locus graph L_i in (a). No path constraint has been detected so far. We simply merge the sets containing each pair of nodes. A pointer is annotated with the offset of a node to its representative. If we further process edge 4–6 of L_i, because both 4 and 6 have a representative with pre-determined *ps* variable, a path between the two representatives (nodes 4 and 8) will induce a path constraint, which is $\sum_{e_{xy} \in P_{\widetilde{4,8}}} h^{xy} = 0$. This is a 3rd type of constraint defined in the text. When dealing with splitting node pair 5 and 5′, the local cycle constraint (1st type) has been replaced by a path constraint $\sum_{e_{xy} \in P_{\widetilde{5,5'}}} h^{xy} = 0$. By doing so, the *ps* variables of nodes 5 and 5′ (2nd type) have been forced to be the same.

3.3. *Encode path constraints in disjoint-set structure D*

Once we detect a path constraint, we also encode this constraint in a disjoint-set structure D. As usual, D is defined by a pair of values $rep[v]$ and $\mathit{offset}[v]$ for each node $v \in V$. $rep[v]$ is a pointer to a node and $\mathit{offset}[v] \in \{0, 1\}$ is a constant. We maintain this disjoint-set structure D such that any two nodes k and l in the same set encode a path constraint in the form of $\sum_{e_{xy} \in P_{\widetilde{k,l}}} h^{xy} = \mathit{offset}[k] + \mathit{offset}[l]$.

Initially, $rep[v] \leftarrow v$, $\mathit{offset}[v] \leftarrow 0$, for any $v \in V$. When processing a path constraint $\sum_{e_{xy} \in P_{\widetilde{i,j}}} h^{xy} = c$, we check whether the representatives of the two end nodes i and j are the same. *If they are not the same*, which means no constraints on h variables between these two nodes have been discovered so far, we merge the two sets represented by $rep[i]$ and $rep[j]$ as illustrated in Algorithm 2. *When* $rep[i]$ *and* $rep[j]$ *are the same* (a constraint already exists before seeing the current constraint), we must check the consistency and redundancy between the current constraint and the previous constraint. This can be easily done by comparing the constant c associated with the new constraint and the constant associated with the existing constraint $\mathit{offset}[i] + \mathit{offset}[j]$. If the two constants are the same, the new constraint is redundant and will be dropped; otherwise, inconsistency exists and the program reports no solutions with zero recombination and terminates. The procedure is summarized in Algorithm 2.

Algorithm 2 Union(i, j, c)

```
if $rep[i] = rep[j]$ then
  if $offset[i] + offset[j] != c$ then
    Report inconsistency
  end if
else
  $offset[rep[j]] \leftarrow offset[j] + offset[i] + c$
  $rep[rep[j]] \leftarrow rep[i]$
end if
```

After all path constraints have been processed, the nodes will form several independent sets. A general solution of h variables can be easily decoded from D. More specifically, for each set representative v of D, we define a free binary variable α_v (notice α_v is not the same as *ps* variables). A general solution of h variables can be represented by a linear system of α variables (which are all free) in the form of

$$h^{xy} = \alpha_{rep[x]} + \mathit{offset}[x] + \alpha_{rep[y]} + \mathit{offset}[y], \tag{2}$$

where $\alpha_{rep[x]}$ and $\alpha_{rep[y]}$ are free variables, and $\mathit{offset}[x] + \mathit{offset}[y]$ is a constant. The complete solution of all h variables (the inheritance vector) can be written in a matrix form,

$$\mathbf{h} = \mathbf{A}\alpha + \mathbf{b}. \tag{3}$$

Suppose the number of h variables is n_h and the number of independent sets in D after adding all constraints is n_D, then $\mathbf{A}$ is a $n_h \times n_D$ matrix, where each row either has exactly two "1"s or is all "0"s (in Eq. (2), α variables are canceled out if x, y are in the same set). Also notice that due to this special structure, the rank of $\mathbf{A}$ is $n_D - 1$. We can prove that the described solution space holds all consistent configurations of inheritance variables.

Lemma 1. *The general solution as provided in Eq.* (3) *satisfies all path constraints and there are no other h variable assignments that satisfy all path constraints.*

Proof. We can verify that such a solution satisfies all path constraints. For each path constraint $\sum_{e_{xy} \in P_{\widetilde{i,j}}} h^{xy} = c$, we plug in the above solution of h variables to its left-hand side: $\sum_{e_{xy} \in P_{\widetilde{i,j}}} (\alpha_{rep[x]} + \mathit{offset}[x] + \alpha_{rep[y]} + \mathit{offset}[y]) = \alpha_{rep[i]} + \mathit{offset}[i] + \alpha_{rep[j]} + \mathit{offset}[j]$, with all intermediate nodes canceled out. Notice that every path constraint is encoded in D, which means $rep[i] = rep[j]$ (so the α variables are also canceled out). Based on the construction of D, the left-hand side $\mathit{offset}[i] + \mathit{offset}[j]$ is the same as the right hand side c, and the constraint is satisfied. We can further argue that there are no other h variable assignments that satisfy all path constraints. This can be shown by examining the relationship of the number of non-redundant path constraints on h variables and the number of freedom defined by Eq. (3). The degrees of freedom and the number of exact solutions of h variables depend on the number of independent sets in D. If there are n_D sets in D formed after adding all constraints, there will be 2^{n_D} different ways to assign all α variables.

But due to symmetry (flipping the values of all α variable assignments will yield the same h variable solution in Eq. (2)), there are only 2^{n_D-1} different h variable solutions instead. This can also be shown by noticing that the rank of matrix $\mathbf{A}$ in Eq. (3) is actually $n_D - 1$. Assume there are n' nodes in locus graphs after node splitting, the number of h variables is $n' - 1$ because no cycle exists any more. The number of non-redundant constraints encoded in D is $n' - n_D$ because the number of constraints in each set $S \in D$ is $|S| - 1$. Therefore, the possible degree of freedom in h variables is $(n'-1)-(n'-n_D) = n_D - 1$ and our general solution has captured all freedom. □

Next, let us consider how to compute ps variable solutions from h variable solutions. For each node v in D_i, v is connected to its set representative $rep_i[v]$ through a path P on L_i. We have $p_i^v + p_i^{rep_i[v]} = \sum_{e_{xy} \in P \in L_i} (h^{xy} + c_i^{xy}) = \sum_{e_{xy} \in P} h^{xy} + \sum_{e_{xy} \in P} c_i^{xy} = \sum_{e_{xy} \in P} h^{xy} + \mathit{offset}_i[v]$. By plugging in the solution of h variables in Eq. (2), we will finally get a general solution for the ZRHC problem,

$$p_i^v = p_i^{rep_i[v]} + \alpha_{rep[rep_i[v]]} + \mathit{offset}[rep_i[v]] + \alpha_{rep[v]} + \mathit{offset}[v] + \mathit{offset}_i[v]. \quad (4)$$

If $p_i^{rep_i[v]}$ is not pre-determined, we have one more degree of freedom in Eq. (4).

3.4. *Analysis of the algorithm on tree pedigrees with complete data*

The overall algorithm is summarized in Algorithm 3. We omit the preprocessing steps (such as node splitting, construction of locus graphs) because all those operations can be done in linear time. Here we also state our main result of the algorithm as a theorem.

Algorithm 3 Process_All_Constraints

```
for $i = 1$ to $m$ do
  for all edge $e_{xy} \in L_i$ do
    $Union_i(x, y, c_i^{xy})$
  end for
  for all splitting node $v$ do
    if $rep_i[v] = rep_i[v']$ then
      $Union(v, v', \mathit{offset}_i[v] + \mathit{offset}_i[v'])$
    end if
  end for
end for
```

Theorem 1. *For a tree pedigree with complete data, Algorithm* 3 *correctly outputs a general solution* [*Eqs.* (2) *and* (4)] *and the number of specific solutions* (*degrees of freedom*) *for the ZRHC problem if it has a solution, and reports inconsistency otherwise. Its running time is bounded from above by* $O(mn\alpha(n))$, *where* m *is the number of loci,* n *is the number of individual and* $\alpha()$ *is the inverse Ackermann function.*[15]

Proof. We first need to show that the proposed algorithm can detect all necessary constraints if the pedigree is a tree pedigree without missing data. The algorithm processes every edge constraint from each locus graph L_i and every constraint resulting from node splitting using the $Union_i$ function, and stores connectivity information using disjoint-set structures D_i. During this procedure, path constraints (including local cycle constraints) are detected and consistency is checked by applying $Union()$ on D. It is easy to understand that all non-redundant path constraints in L_i have been detected since each D_i keeps the connectivity information of all pairs of nodes from each locus graph. For a tree pedigree, all cycle constraints are local cycle constraints. By introducing the splitting nodes, such local cycle constraints have been expressed as a path constraint ending in a pair of splitting nodes (e.g. path $P_{\widetilde{5,5'}}$ in Fig. 5), and have been correctly processed in Algorithm 3. All other cases (constraints involving a splitting node) have been discussed in Sec. 3.2. Therefore, the proposed algorithm can detect all necessary constraints for a tree pedigree. And any particular h variable solution obtained from Eq. (2) is compatible with the genotype data.

In terms of time complexity, the outside for-loop in Algorithm 3 is over each locus i. For each locus i, the total number of union operations on D_i (function $Union_i$) is bounded by the summation of the number of edges and the number of splitting nodes in locus graph L_i, which is bounded by $O(n)$ even after considering node splitting. There is at most one union operation on D (function $Union$) for each $Union_i$, therefore, the total number of union operation on D is bounded by the total number of union operations on D_i, which is $O(n)$. The number of elements in D_i and D is the same and also bounded by $O(n)$. Both $Union_i()$ and $Union()$ are essentially conventional union-find procedures on disjoint-set structures. The extra cost to maintain the offset value of each node takes only constant time for each operation, therefore it does no change to the time complexity. Despite the simplified presentation in Algorithms 1 and 2, we implement the union-find procedure on a forest structure using Tarjan's algorithm.[15] The worst case time complexity of $O(n)$ disjoint-set operations on $O(n)$ elements is $O(n\alpha(n))$,[17] where $\alpha()$ is the inverse Ackermann function. Therefore, the total running time of the algorithm to output a general solution is $O(mn \cdot \alpha(n))$, where m is the number of loci, and n is the number of individuals of the pedigree. □

4. Extension to General Cases

4.1. *Pedigrees with mating loops*

We can further extend the above algorithm to pedigrees with mating loops and pedigrees with missing data. For a looped pedigree, we apply a similar splitting rule to locus graphs as we did for a tree pedigree, except that for a mating between two relatives, all their children are duplicated in order to break a global cycle. We use the same method described in Secs. 3.2 and 3.3 to detect all path constraints on each locus graph. However, Theorem 3.1 does not hold anymore in this case because the method does not guarantee the detection of all necessary constraints. The difference lies in the detection of path constraints broken by splitting nodes. All such path constraints can be recovered when breaking a local cycle but may not be recovered when breaking a global cycle. Figure 6 gives such an example on a looped pedigree. In the locus graph L_i in Fig. 6(b), we have a path constraint on path $P_{\widetilde{6,6'}}$, which is originally a cycle constraint before splitting of node 6. This type of constraints may still be able to be captured with some extra efforts. However, in another locus graph L_j in Fig. 6(c), there is a path constraint $h^{46} + h^{56} = 0$ in the original locus graph. But this constraint is not on a consecutive path in L_j after node splitting. Thus it is not able to be encoded in the disjoint-set structure D.

Although the set of constraints is not sufficient, we can still obtain all the solutions for a looped pedigree using the following procedure. If there are already inconsistent constraints during consistency check, no solutions with zero recombinant exist. Otherwise, all the h variable solutions obtained based on the general solution [Eq. (2)] will be examined. If a specific h variable assignment is not consistent with the genotype, we simply drop that assignment. Otherwise, it will result in real haplotype solutions. To check the consistency of an h variable assignment with existing genotypes, we use another disjoint-set structure to encode constraints on alleles. This step is the same for pedigrees with loops and pedigrees with missing data, and will be discussed in Sec. 4.2. Essentially for looped pedigrees, we avoid cycle constraints by splitting nodes with the expense that we may miss some constraints. We start to enumerate h variables after processing existing partial constraints. However, as it will be shown in the experiment, the number of all possible h variable assignments from this set of partial constraints is usually very small for a pedigree with complete data, and in most times there is only one solution for pedigrees with 20 or more loci. Therefore, the above extension can efficiently handle looped pedigrees in practice.

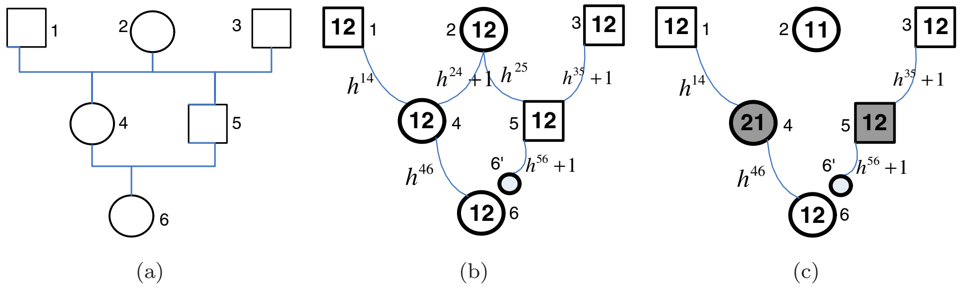


Fig. 6. An example of constraints on a looped pedigree. (a) A pedigree with a mating loop, where node 6 is produced by the mating of two relatives 4 and 5. (b) One locus graph L_i, where there is a path constraint $\sum_{e_{xy} \in P_{\widetilde{6,6'}}} h^{xy} = h^{24} + h^{25} + h^{46} + h^{56} = 0$. (c) Another locus graph L_j, where there is a constraint $h^{46} + h^{56} = 0$. Due to the splitting at node 6, this constraint is not on a consecutive path.

4.2. *Pedigrees with missing data*

For an algorithm to be practically useful, it has to be applicable on real data. Most real data contain missing information. One advantage of the proposed algorithm is that it can be easily extended to deal with missing data. Extension of the constraint-finding framework[10,12,13] to handle missing data is not trivial at all. As mentioned earlier, the ZRHC problem with missing data is NP-hard[14] in general. Therefore, it is unlikely that a linear system will exist to incorporate all uncertainties. We take a similar approach as in Sec. 4.1 to deal with missing data by making use of existing constraints and verifying every compatible inheritance vector. Partial constraints on h variables will be collected based on existing genotype data. Solutions of h variables will be obtained based on the set of partial constraints and will be checked for consistency with existing genotype data. More specifically, for a pedigree with missing data, we construct the locus graph L_i for each locus i as usual with node splitting if necessary. The edges in L_i will only be constructed by examining every parent–child pair whose genotypes are complete at this locus i. We apply Algorithm 3 to process all edge constraints from such locus graphs. And from the partial constraints on h variables, we get a solution in its general form [Eq. (2)]. The degree of freedom, which is $n_D - 1$ where n_D is the number of independent sets, usually is significantly smaller than the degree of freedom of the original h variables without constraints, which is usually close to $2n$. Therefore, our algorithm has the potential to be significantly faster than those algorithms based on the enumeration of all possible h variables (such as Merlin[9]).

For each specific h variable assignment, the compatibility check with the input genotype data is also examined by utilizing another disjoint-set structure on allele variables. Let f_i^x (m_i^x) denote the paternal (maternal) allele of individual x at locus i, which takes the integer value 0 and 1 for the smaller and bigger allele, respectively. For a fixed assignment of h variables, the relationship of alleles between a parent and a child is specified by the definition of h variables. This relationship is also expressed as a linear system on Z_2. For example, for a father–child pair x and y, we have constraint $f_i^y + f_i^x = 0$ if $h^{xy} = 0$, and $f_i^y + m_i^x = 0$ if $h^{xy} = 1$ by Definition 2. Similar constraints can be obtained for each mother–child pair. In addition, constraints between the two allele variables at each locus for an individual exist when the genotype data are available. More precisely, if an individual x is homozygous or pre-determined at locus i, then both f_i^x and m_i^x are fixed. Otherwise we have the constraint $f_i^x + m_i^x = 1$. All these constraints only involve two variables, so we can encode this linear system in a disjoint-set structure and develop the same set manipulating procedure as we did in the integration of constraints on inheritance variables. By doing so, we can efficiently check the consistency between a given h variable assignment and the input genotype data, and generate a set of assignments of alleles that are consistent with the h variable assignment. The total number of h variable assignments is $2^{n_D - 1}$, and for each assignment, the complexity of genotype consistency check is $O(mn \cdot \alpha(n))$.

5. Experimental Results

We study the performance of our program (denoted as DSS) under different settings (pedigree size, number of loci, missing rate, pattern of missing) and compare its performance with two representative programs Merlin[9] and PedPhase (the integer linear programming ILP algorithm in Ref. 7). Merlin is one of the most widely used statistical packages for linkage analysis, and we only use its haplotyping functionality in this comparison. It also uses the zero recombinant assumption. However, it examines all possible configurations of inheritance variables and only outputs those compatible ones. PedPhase.ILP is another widely used rule-based algorithm developed by our own group. It can produce all optimal haplotype solutions with minimal recombinants for any pedigree structures with missing data. It can solve the zero recombinant problem as a special case. But because it does not use the zero recombinant assumption explicitly, its efficiency is expected to be inferior to the current algorithm. Under the zero recombinant assumption, all three methods are **exact** algorithms that output all compatible solutions. Our experiments show that their implementations indeed generate the same set of haplotype assignments on the same inputs. This again shows that the ZRHC formulation is valid for tightly linked markers, and the set of solutions is the same as the set of solution obtained based on likelihood approaches. Therefore, we only present results on the efficiency comparison.

We test all three approaches on different sizes of pedigrees (17, 29, 52, 128), all are real human pedigree structures obtained from literatures. Different number of loci (20, 50, 100, 200), different missing rates (0.05, 0.10, 0.15, 0.20) and different missing patterns are considered. We run Merlin and DSS on a Linux machine with two 3.0-GHz Quad-Core Xeon 5365 processors and 16-G memory. PedPhase. ILP only has a Windows version, and it was tested on a much slower Windows machine with a much lesser memory (Pentium 4 3.2-GHz with 2-G memory). We measure the time needed for each of the algorithms to output all possible haplotyping solutions of a pedigree. Due to hardware limitations, the result of PedPhase.ILP on pedigree size 128 is not acquired. To generate genotype data that closely resemble real data, we use the Simulated Rheumatoid Arthritis (RA) Data from Genetic Analysis Workshop (GAW) 15. Chromosome 6 of GAW data mimics a 300-K SNP chip with an average inter-marker spacing of 9586 bp. The beginning 20, 50, 100 and 200 loci are truncated to test the three algorithms. Population haplotype frequencies are calculated based on the true haplotype assignments in the simulated data, and are then fed to *SimPed*,[16] together with each pedigree structure. *SimPed* will then sample founder haplotypes based on their population frequencies and generate genotype data for each member in a pedigree, assuming no recombination. The three pedigree structures are shown in Fig. 7, among which the pedigree with size 17 [Fig. 7(a)] is a looped one. The pedigree with size 128 is too large to fit in one page and will be provided on our website.

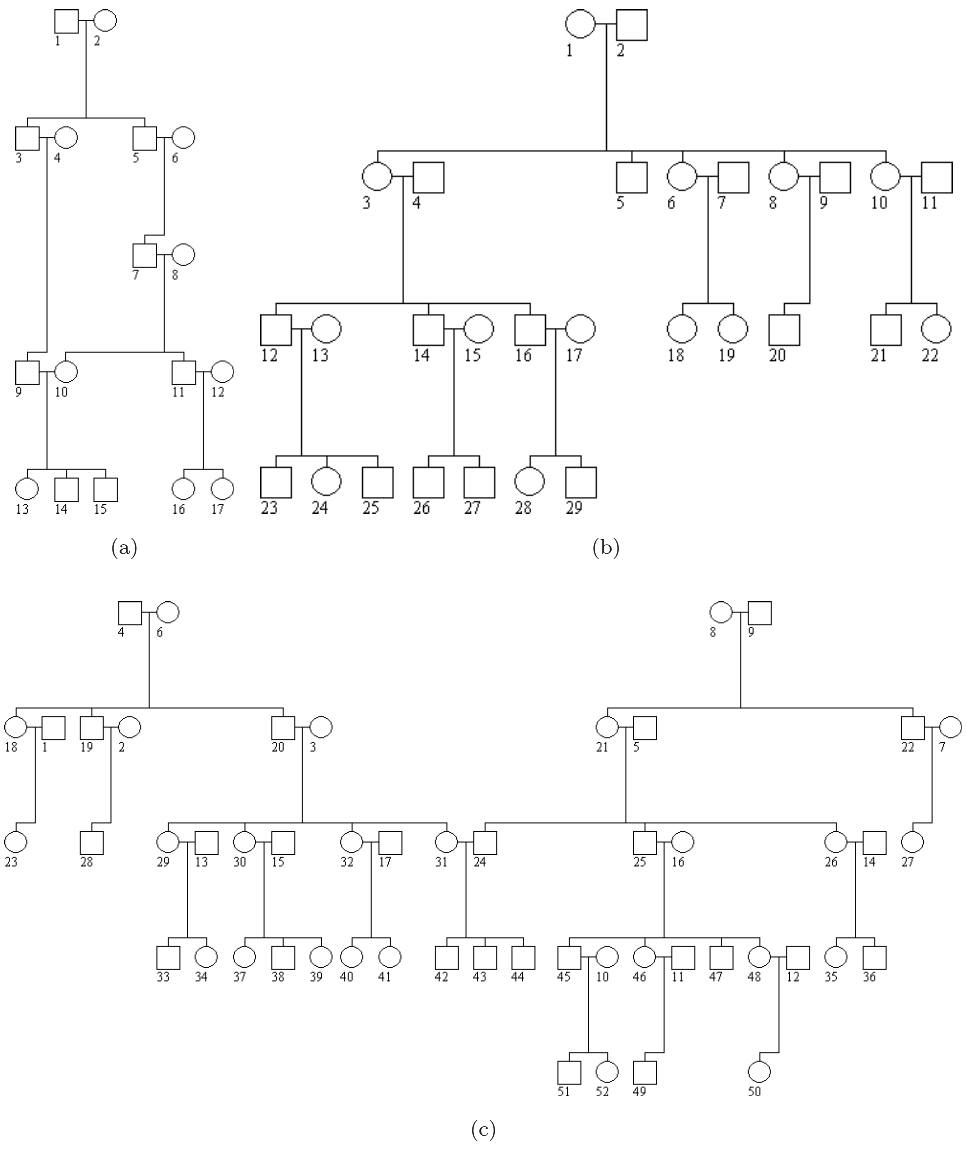
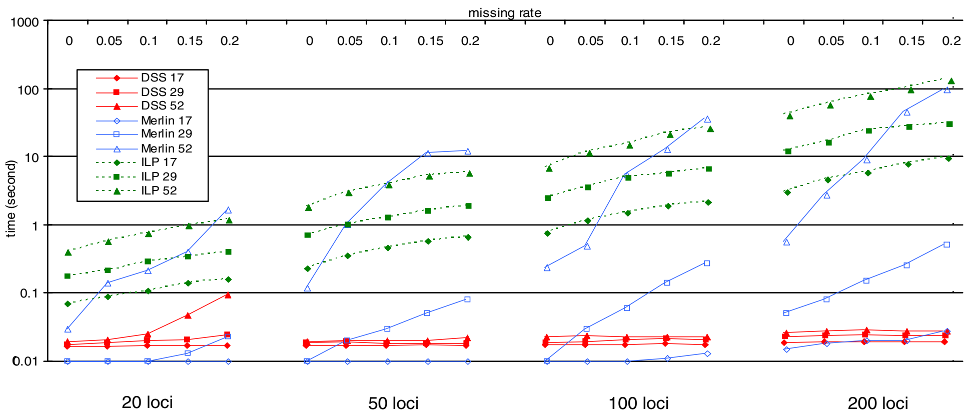
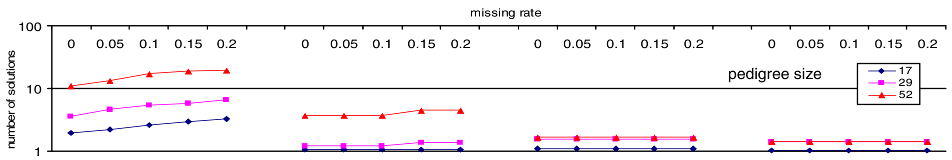


Fig. 7. Pedigree structure used in simulation.

We designate two ways to generate samples with missing data so as to examine the behavior of the methods with respect to both missing rate and missing pattern variations. We generate the first set of samples by randomly assigning a locus to be missing at a specified missing rate. Second, we make all top generation of a pedigree completely missing for all loci, which is common in real data. For each testing category, we simulate 100 independent datasets and report the average running time.



(a)



(b)

Fig. 8. (a) comparison of running time (in seconds). (b) average number of solutions.

For the random missing case, Fig. 8(a) shows the running time of the three programs under different settings, except for the pedigree size 128, for which the running time of Merlin is too large to be juxtaposed with DSS. The result on the pedigree with size 128 is listed in Table 2. The running time of Merlin increases exponentially with the pedigree size, the number of loci and also the missing rate. The running time of PedPhase.ILP (on a slower machine) also has an exponential growth with the increase of the missing rate and the number of loci but with a smaller constant compared to Merlin. It also shows a much smaller growth rate with the pedigree size. In contrast, DSS scales smoothly with all parameters (except for the missing rate when the number of loci is 20), and the improvement over Merlin or PedPhase.ILP is from 10 to 10^5 folds for large pedigrees with large number of loci or high rate of missing data. In fact, neither Merlin nor PedPhase.ILP can successfully infer haplotypes from the pedigree with size 128 when the number of marker is 200. However, DSS can obtain all solutions within 0.05 second, even for data with 20% missing information. This shows that by solving the linear system based on partial constraints from existing data, we significantly reduce the enumeration space of inheritance variables. The experimental results show that when the number of loci is large, the program can still maintain the same linear complexity even for data with 20% missing information. But for small number of loci, the running time of

Table 2. Comparison of running time (in seconds) between DSS and Merlin on pedigree size 128. The running time of Merlin under some data settings exceeds an hour, and is thus omitted from our measurement.

Number of loci			20					50		
Missing rate	0.00	0.05	0.10	0.15	0.20	0.00	0.05	0.10	0.15	0.20
DSS	0.0267	0.1539	0.3517	0.4991	0.6540	0.0259	0.0361	0.0368	0.0378	0.0360
Merlin	70	300	600	800	1100	360	800	1000	>1300	—
Block-extension	0.05	0.05	0.05	0.05	0.05	0.09	0.1	0.1	0.1	0.1
Number of loci			100					200		
Missing rate	0.00	0.05	0.10	0.15	0.20	0.00	0.05	0.10	0.15	0.20
DSS	0.0311	0.0426	0.0373	0.0431	0.0461	0.0433	0.0587	0.0518	0.0575	0.0503
Merlin	800	1200	>2400	—	—	—	—	—	—	—
Block-extension	0.2	0.2	0.19	0.2	0.2	0.53	0.63	0.67	0.64	0.63

DSS increases as missing rate increases (though DSS can finish all the cases within 0.1 second). This is because the number of constraints on h variables is roughly in proportion to the number of loci. So for small number of loci, the remaining degrees of freedom on inheritance variables after solving the linear system could still be high. This number could be partly reflected by the number of all compatible solutions in the end. Figure 8(b) compares the number of h variable solutions in different circumstances. It grows with both the pedigree size and the missing rate, but decreases with the number of loci.

While PedPhase.ILP takes very long time to work on pedigree size 128, we run Block-extension algorithm also from PedPhase package as a substitute in this category just for reference purposes. Different from DSS, Merlin and Pedphase.ILP, Block-extension is a heuristic algorithm which employs some greedy strategy to obtain a particular solution with minimum recombination. Since it is a heuristic, it does not explore every possible configuration and may not reach optimality in all circumstances. As is shown in Table 2, Block-extension does a fast job on this large pedigree and it scales well with both the number of loci and the missing rate. And in most cases (>90%), it can find a solution with no recombination. This high efficiency makes such heuristic approaches useful in certain applications where completeness or optimality of the solution space is not enforced.

Next, we investigate the performance of all three algorithms on special missing patterns. Figure 9 gives some representative result on the pedigree with size 52, for which all individuals at the top generation (members 4, 6, 8, 9) are missing. For this pedigree, such missing data equal a missing rate of ~7.7%. In terms of absolute time, DSS ($0.2 \sim 0.8$ s) is much better than the other two algorithms ($0.2 \sim 100$ s). However, the running time is higher than its own running time with a missing rate of 10%. The running time of Merlin and PedPhase.ILP on this special dataset

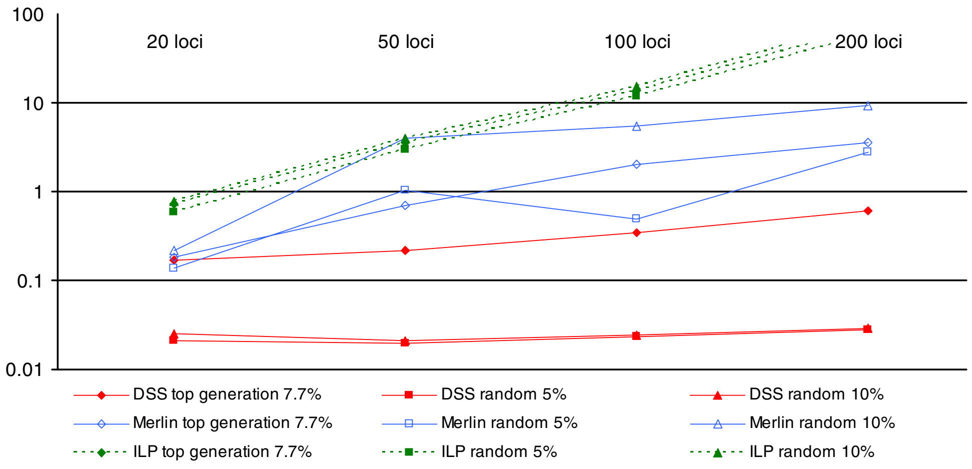


Fig. 9. Comparison of DSS and Merlin on different patterns of missing data.

is between those of missing rate 5% and 10%. DSS is somewhat sensitive to this special missing pattern because when all genotypes of an individual are missing, none of the inheritance variables between her and her parents or children could be determined. A further investigation on this special missing pattern is warranted.

6. Discussion

We propose an algorithm for haplotype inference from pedigree data without recombinant using disjoint-set structures. For each locus, we generate a locus graph based on the genotype information between each parent–child pair. A disjoint-set structure D_i is built to encode the connectivity information of nodes in each locus graph L_i. We update D_i by incrementally adding new edge constraints from locus graph L_i. During this process, we detect all path constraints and encode them in another disjoint-set structure D. D will be used for consistency check for each new detected path constraint. Finally, we can generate a general solution for h variables by taking each set representative in D as a free variable. All the h variable solutions are guaranteed to be compatible with the genotype data on each locus for a tree pedigree. Cycles are broken by splitting nodes. We further extend the algorithm to looped pedigrees and pedigrees with missing data by enumerating all h variables based on the set of partial constraints. The compatibility check of each inheritance variable assignment is performed by collecting constraints on allele variables using another disjoint-set structure. The proposed algorithm can output a general solution for a tree pedigree with complete data in time $O(mn\alpha(n))$, which is a further improvement upon existing results. For a general pedigree, or a pedigree with missing data, by using the same framework, our method can significantly reduce degrees of freedom on inheritance variables and thus narrow down the search scope. Experimental results show that the algorithm is efficient in practice for both complete data and missing data, and outperforms two popular algorithms on large datasets. For data with large number of markers, the performance of the algorithm hardly deteriorates as the missing rate increases.

Though several theoretical results of ZRHC were recently reported,[10,12,13] none of them have been implemented. The empirical examination of the performance of our algorithm offers some evidence for the theoretical bounds on the complexity of such haplotyping approaches based on linear systems.

The performance of our algorithm on pedigrees with missing data depends on the number of constraints the linear system can capture. We observe that the efficiency of this linear system is influenced by variation in missing patterns. So as a possible piece of future work, we can consider a special strategy to handle individuals with all loci missing. Other possible directions are to combine the proposed algorithm with statistical approach to assign a probability likelihood for each of the assignments, and to design algorithms for whole chromosome by calling the current algorithm as a subroutine. Theoretically, it also remains open whether the linear time complexity can be observed for a general pedigree with complete data.

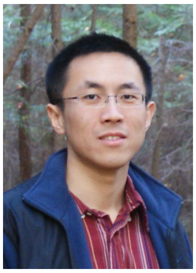
Acknowledgments

This research was supported by National Institutes of Health/National Library of Medicine grant LM008991, and in part by National Institutes of Health/National Center for Research Resources grant RR03655 and National Science Foundation CRI0551603. Support for generation of the GAW15 simulated data was provided from NIH grants 5RO1-HL049609-14, 1R01-AG021917-01A1, the University of Minnesota, and the Minnesota Supercomputing Institute. We would also like to acknowledge GAW grant R01 GM031575.

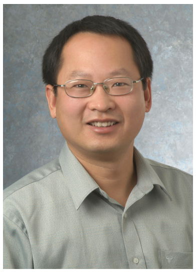
References

1. The international HapMap Consortium, A second generation human haplotype map of over 3.1 million SNPs, *Nature* **449**:851–61, 2007.
2. Bonizzoni P, Vedova GD, Dondi R, Li J, The haplotyping problem: An overview of computational models and solutions, *J Comp Sci Tech* **18**(6):675–88, 2003.
3. Gusfield D, An overview of combinatorial methods for haplotype inference, *Lecture Notes in Computer Science (2983): Computational Methods for SNPs and Haplotype Inference*, pp. 9–25, 2004.
4. Halldórsson BV, Bafna V, Edwards N, Lippert R, Yooseph S, Istrail S, A survey of computational methods for determining haplotypes, *Lecture Notes in Computer Science (2983): Computational Methods for SNPs and Haplotype Inference*, pp. 26–47, 2004.
5. Li J, Jiang T, A survey on haplotyping algorithms for tightly linked markers, *J Bioinform Comput Biol* **6**(1):241–259, 2008.
6. Zhang XS, Wang RS, Wu LY, Chen L, Models and algorithms for haplotyping problem, *Curr Bioinform* **1**(1):105–114, 2006.
7. Li J, Jiang T, Computing the minimum recombinant haplotype configuration from incomplete genotype data on a pedigree by integer linear programming, *J Comput Biol* **12**:719–739, 2005.
8. Zhang K, Zhao H, A comparison of several methods for haplotype frequency estimation and haplotype reconstruction for tightly linked markers from general pedigrees, *Genet Epidemiol* **30**(5):423–437, 2006.
9. Abecasis GR, Wigginton JE, Handling marker–marker linkage disequilibrium pedigree analysis with clustered markers, *Am J Hum Genet* **77**:754–767, 2005.
10. Chan MY, Chan W, Chin F, Fung S, Kao M, Linear-time haplotype inference on pedigrees without recombinations, *Proc 6th Ann Workshop on Algorithms in Bioinformatics (WABI'06)*, pp. 56–67, 2006.
11. Li J, Jiang T, Efficient inference of haplotypes from genotype on a pedigree, *J Bioinform Comput Biol* **1**(1):41–69, 2003.
12. Liu L, Jiang T, Linear-time reconstruction of zero-recombinant mendelian inheritance on pedigrees without mating loops, *Proc Genome Informatics Workshop (GIW'2007)*, pp. 95–106, 2007.
13. Xiao J, Liu L, Xia L, Jiang T, Fast elimination of redundant linear equations and reconstruction of recombination-free mendelian inheritance on a pedigree, *Proc 18th Ann ACM-SIAM Symoposium on Discrete Algorithms (SODA'07)*, pp. 655–664, 2007.
14. Liu L, Chen X, Xiao J, Jiang T, Complexity and approximation of the minimum recombination haplotype configuration problem, *Proc 16th Int Symp Algorithms and Computation (ISAAC'05)*, pp. 370–379, 2005.

15. Cormen TH, Leiserson CE, Rivest RL, Stein C, *Introduction to Algorithms*, 2nd edition, McGraw-Hill Book Company, Boston, pp. 498–517, 2003.
16. Leal SM, Yan K, Müller-Myhsok B, SimPed: A simulation program to generate haplotype and genotype data for pedigree structures, *Hum Hered* **60**:119–122, 2005.
17. Tarjan RE, Leeuwen J, Worst-case analysis of set union algorithms, *J ACM* **31**(2):245–281, 1984.



Xin Li received his Bachelor's degree in Computer Science from Tsinghua University, China in 2005. He is currently working on his Ph.D. degree at the Department of Electrical Engineering and Computer Science at Case Western Reserve University, USA. His research is centered on haplotyping algorithms.



Jing Li is an Assistant Professor in the Department of Electrical Engineering and Computer Science at Case Western Reserve University, USA. He received his B.S. in Statistics from Peking University, Beijing, China, in July 1995, his M.S. in Statistical Genetics from Creighton University, USA, in August 2000, and Ph.D. in Computer Science from University of California-Riverside, USA in June 2004. He was a winner of the ACM Student Research Competition in 2003. Jing Li's recent research interest includes Bioinformatics/computational molecular biology, algorithms and statistical genetics.