The Incremental Haplotyping of Incomplete Genotypes using Perfect Phylogeny Paths

Master's Thesis in Engineering Mathematics

by William H. Garner

Examiner:
Docent Peter Damaschke
Department of Computer Science

Institutionen för Matematiska Vetenskapar

Chalmers Tekniska Högskola

Göteborgs Universitet

Part I
Introduction

1 Genotypes and Haplotypes

The nuclear genetic information of most multicellular plants and animals (including humans) is contained in DNA on sets of paired chromosomes. An individual's genome contains exactly two specimens of each chromosome, one inherited from the father and the other from the mother. Organisms with this genetic arrangement are called *diploid*. This genetic information can be represented as a map of sites and nucleotide base pairs. At each site one of four possible base pairs is present with one base belonging to each of the two paired chromosomes. An individual's *genotype* specifies the pair of bases at each site, but does not specify which base occurs on which chromosome. The sequence of bases on each chromosome separately is called a *haplotype*.

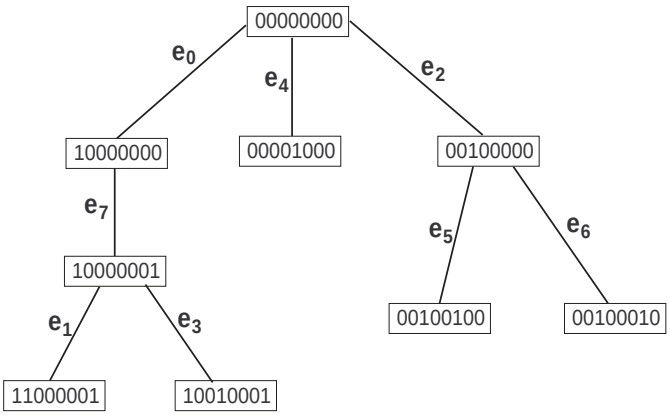
It is the case then that while there is much genotype variation from generation to generation, even in families, there is very little change in haplotypes. A chromosomal haplotype passes from parent to child unchanged unless a mutation occurs. Mutations are generally rare events and usually alter single nucleotides. Such mutations lead to the genetic variations within a species. The base sites which show variability of nucleotides within a population are called *single nucleotide polymorphisms* (SNPs).

The determination of the haplotypes within a population is thought to be essential for understanding genetic variation and the inheritance of complex traits. Especially of interest are inherited disease or disease resistance. The International HapMap Project, a successor to the human genome project, seeks to determine the common haplotypes in the human population. However most of the genetic data thus far gathered on diploid organisms is genotype data from biochemical tests. Biochemical techniques exist for determining haplotype data as well but are more difficult and expensive. Thus algorithms for inferring haplotypes from genotype data are of current practical importance.

2 Perfect Phylogeny

In principle a family tree or *phylogeny* could be drawn for a given haplotype (over some subset of chromosomal base sites) with haplotypes as the nodes

Figure 1: Example of a Perfect Phylogeny of Bit Strings.



and mutation events as the edges. To insure a tree structure, a new edge and node are added with each mutation. It would of course be possible that some sequence of mutations exactly reverses a previous sequence and a haplotype appears more than once in the tree. However mutations are rare events and occur nearly randomly among the base sites, so such repeats must be very rare indeed.

This suggests a simplifying assumption: *suppose that no base site has been mutated more than once.* If this assumption holds then the haplotype phylogeny tree has no repeated nodes. Also since every edge is a mutation event on a unique base site each SNP splits the tree into two components each with a different value at that site. Such a tree is called a *perfect phylogeny.*

3 This project

This paper summarizes the results of a project exploring the application of the perfect phylogeny assumption for haplotype determination from both complete and incomplete genotype data. The primary project goal was to develop a useful haplotyping tool for the incomplete haplotyping problem. It is assumed throughout that the reader is familiar with the basic concepts and terminology of graph theory.

In Part II a basic algorithm is developed. Its procedures and rules are derived from the dependence of the genotype data on the haplotypes and also from the perfect phylogeny assumption. Later in Part III we describe

a cumulative algorithm which repeatedly applies the basic algorithm to a sample of genotypes and progressively determines the haplotypes more completely. Both the basic and cumulative algorithms have been implemented in Java for this project and some program test results are presented.

Part II
An Incremental Haplotyping Algorithm

4 Haplotyping as a Graph Algorithm

In the context of perfect phylogeny then we shall treat the haplotyping problem as a problem of tree construction over a set of binary labelled nodes. Binary labels suffice since only two values may appear at any given site. For the problem at hand then we give a concise definition of a perfect phylogeny of bit vectors.

Definition 1 *A perfect phylogeny* (PP) *is a tree whose nodes are labelled by bit vectors of length* m *and are such that every edge may be labelled* e_i *with* $0 \leq i < m$ *and exclusively represents a change of bit* i.

Under the PP assumption then the haplotyping problem becomes: Given a set of n vectors of m unordered bit pairs, namely $(0,0)$, $(0,1)$, or $(1,1)$, explain each of them by two bit vectors of length m, so that the distinct bit vectors fit in a perfect phylogeny.

To simplify matters, we shall henceforth assume that the genotype data is given as a sequence of values from $\{0,1,2,X\}$ where $0 = (0,0)$, $1 = (1,1)$, $2 = (0,1)$ *unordered*, and X is unreadable data. If the data admits a PP haplotype explanation this must be possible.

4.1 Basic Ideas of the Incremental Algorithm

Given a sample of genotypes of length m which admits a set of explaining haplotypes which fit a PP, we can of course restrict our attention to any subset of $k < m$ bit sites of the original m and it must also admit a set of explaining haplotypes that fit a PP. The algorithm presented here builds up the set of explaining haplotypes one bit at a time. In so doing it uses both the genotype data and the PP tree structure of the preceding step, to infer the new PP tree structure. This incremental strategy follows that of Damaschke [cite paper].

We shall refer to the explaining haplotypes as a **decoding**. The process of advancing a haplotype from k bits to $(k+1)$ bits shall be called the $(k+1)$ **extension step**. A **k-haplotype** is one which has been extended to k bits. Generally when discussing k-haplotypes or $(k+1)$-haplotypes in the context of the $(k+1)$ extension step we shall represent the first k bits by an arbitrary letter. For example $x0$ and $x1$ denote $(k+1)$-haplotypes which differ only in their last bit, while y or z would be distinct k-haplotypes that have not yet been extended in the $(k+1)$ extension step. We shall refer to the perfect phylogeny tree of the k-haplotypes as $\mathbf{P_k}$.

Next we define a graph which exhibits the sample data together with the decodings at a given extension step.

Definition 2 *For any* k*, we define the sample graph* $\mathsf{G_k}$ *as follows. The* k*-haplotypes are the nodes of* $\mathsf{G_k}$*. For every genotype in the sample, the two nodes (*k*-haplotypes) which are the decoding of that genotype's* k *sites are joined by an edge in* $\mathsf{G_k}$*. These edges are labelled by type according to the corresponding data of site* $\mathsf{k+1}$ *of the genotype.*

There is one edge for every genotype and thus G_k may contain loops, parallel edges, and nodes of high degree. The edges maybe of type 00, 11, 01, or X. We shall always refer to the edges of G_k using their type. It is natural to do so and we must take care to be clear whether we are referring to edges in P_k or in G_k since they have the same nodes.

The algorithm proceeds by inferring and assigning either 0 or 1 to each k-haplotype to extend it to a $(k+1)$-haplotype. If the algorithm is not able to extend a k-haplotype then in the next extension step it and all the genotypes for which it is part of the decoding must be dropped. Those genotypes shall be only partially decoded. We shall call the assignment of the new bit, **coloring**. In describing the process it is at times convenient to represent the two values as *blue* and *green* without specifying them since all of our observations and inferences are the same if the roles of the actual bit values are reversed.

Two rules for extending haplotypes are immediate from the very nature of the genotype data.

Extension Rule 1 *A* k*-haplotype* x *which is incident to a* $\mathsf{00}$*-edge is extended to* $\mathsf{x0}$*. If* x *is incident to a* $\mathsf{11}$*-edge it is extended to* $\mathsf{x1}$*.*

Extension Rule 2 *A* k*-haplotype* x *which is incident to a* $\mathsf{01}$*-loop is extended to both* $\mathsf{x0}$ *and* $\mathsf{x1}$*.*

These rules are trivial but either Rule 1 or 2 may lead to a k-haplotype being extended by two different bit values or in other words being colored both blue and green. If this happens we call that k-haplotype a **split node** for extension step $(k+1)$. Finding the split node is important to advancing the decoding as G_0 and P_0 have only one initial node whose label is an empty binary vector. If j different haplotypes are to be found then $j-1$ split nodes must be determined.

If x is a split node for extension step $(k+1)$ then for all genotypes for which it was part of the k bit decoding we need to determine if $x0$ or $x1$ is the correct $(k+1)$-haplotype for its extended decoding. If the genotype has 1 or 0 for its k-site data then the answer is easy. If the k-site data is 2 or X then we must turn to the PP assumption.

4.2 Consequences of Perfect Phylogeny

The following is a well-known characterization of a perfect phylogeny:

Lemma 1 *A population of haplotypes fits a* PP *if and only if, for no pair of bit sites, all four combinations* 00, 01, 10, 11 *appear in the population.*

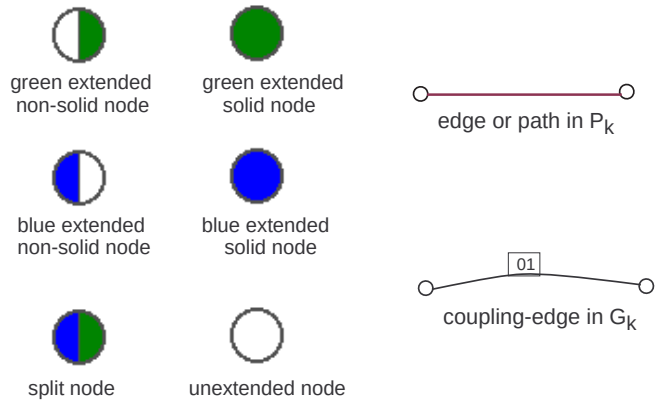
Suppose that in extension step $(k+1)$ we find that two distinct k-haplotypes, x and y are split nodes. Now choose $i < k$, an earlier bit site where x and y differ, say $x_i = 1$ and $y_i = 0$. Then looking at bit sites i and k we find $y0_{(i,k)} = (0,0)$, $y1_{(i,k)} = (0,1)$, $x0_{(i,k)} = (1,0)$, and $x1_{(i,k)} = (1,1)$. And by Lemma 1 the $(k+1)$-haplotypes do not fit a PP. On the other hand given any two x_i and y_i bit values it is impossible to obtain this complete quadruple with a single split at extension step $(k+1)$.

Thus Lemma 1 implies a very important fact relating the PP property and our haplotype extension process:

Lemma 2 *If the* k*-haplotypes fit a* PP*, then the* (k+1)*-haplotypes fit in a* PP *if and only if, at most one of the haplotype nodes of* G_k *is split in extension step* (k+1)*.*

Assuming the PP property we know from Lemma 2 that there is at most one split node at each extension step. Define a **solid node** to be a node which has been extended in the current extension step and is known *not* to be the split node. If the split node has been determined then many of the 01-edges and X-edges in G_k may be resolved owing to the fact that they are incident to a solid node or nodes. By *resolve* we mean unambiguously extend both of the k-haplotypes which the edge couples into a genotype decoding. The notion of solid nodes shall be central to several of the inference methods employed by the algorithm.

Figure 2: Key to Nodes, Edges, and Paths in Diagrams.



5 Applying the PP Assumption

First we consider how to determine solid nodes when no split node has been found. Often it is only possible to determine the split node based on inferences using nodes determined to be solid. Later we explain the set of PP inference rules used by the algorithm.

5.1 Identifying Solid Haplotype-Nodes

Determining solid nodes during an extension step is crucial to the success of the algorithm. How may we find solid nodes when we have not determined a split node? To do so we must make use of the structure of the perfect phylogeny tree. It would of course be possible to store or reconstruct the tree structure but this is not necessary. The algorithm shall make use of a fast method of checking inclusion of paths in the phylogeny tree.

5.1.1 Efficient Phylogeny Path Checking

When no split node has been found the algorithm relies on checking nodes for path inclusion in the current phylogeny tree. Later we shall also introduce several extension rules based on this same test. For the algorithm to

be computationally feasible then it requires a fast operation for checking phylogeny path inclusion.

Consider two k-haplotypes, x and y in P_k. For any bit i for which x and y share the same value they must be on the same side of the edge e_i that marks the unique change of that bit. So must any k-haplotypes on the path between them. Thus all k-haplotypes lying on the x-y path in P_k must agree with x and y on the bits where they agree with each other.

On the other hand let z be a k-haplotype that does not lie on the x-y path. The first edge e_b on the z-x path cannot be part of the x-y path. Further e_b must lie on the z-y path otherwise there would be two distinct paths from z to y, one with e_b and one without it. Thus x and y are on the same side of e_b and z is on the opposite side. So x and y agree on bit b but z has the opposite value. This establishes:

Lemma 3 *Let* x, y, *and* z *be* k*-haplotypes which fit a* PP*. Then* z *lies on the phylogeny path between* x *and* y *if and only if,* z *agrees with* x *and* y *on all the bits where they agree with each other.*

The lemma gives a simple and fast method for checking path inclusion. Say we want to check if z lies on the phylogeny path between x and y. Let $m = \mathsf{Xor}(x, y)$ then z lies on the x-y path if and only if $\mathsf{Or}(z, m) = \mathsf{Or}(x, m)$ where $\mathsf{Xor}()$ and $\mathsf{Or}()$ are the familiar bitwise logical operations. Implemented in a computer language which allows bit level programming, path checking in this way can be done at least as quickly as exact integer multiplication for large values of k.

5.1.2 Using Blue-Green Paths

If during the $(k+1)$ extension step two k-haplotypes have been colored differently, then the path between them in P_k must contain the split node. (If it exists among the k-haplotypes.) This is a consequence of the blue and green subgraphs each being a single connected component. Given a blue-green path in P_k any extended node whose k-haplotype can be determined not to lie on this path may be labelled solid. In practice this may be done in one pass through the list of extended nodes. Say we begin with the end nodes $x0$ and $y1$, if we find $z0$ which lies between $x0$ and $y1$ then $x0$ is labelled as solid and is replaced by $z0$ as an end node. Continuing in this fashion at most two *candidates* for the splitting node are found, one blue and one green. All other extended nodes are solid. If a second pass is made through the node list it may be possible to label one or both of these candidates as solid as well, for the split node must lie on *all* blue-green paths.

5.1.3 Using 01 Genotype-Edges

Sometimes when there is an imbalance among the edge types in G_k only extensions of one color can be made, at least initially. Here no blue-green path is available to determine solids and candidates. Yet there is another way to find such a path. Consider any 01-edge in G_k. If both its nodes have been extended by opposite colors then we simply have a blue-green path as already discussed. But if its ends are not extended with opposite colors then it still gives k-haplotype end nodes of a path from the two components in P_{k+1} which are to be connected by the edge e_{k+1}. This glimpse of the situation in P_{k+1} again allows us to find solids by looking at the paths in P_k. If the k-haplotype end nodes are not oppositely colored, then there are three cases:

Case I Neither end node has been extended. All that may be said is that the candidates lie on the path. Thus any extended node not lying on the path between the two end nodes may be labelled solid.

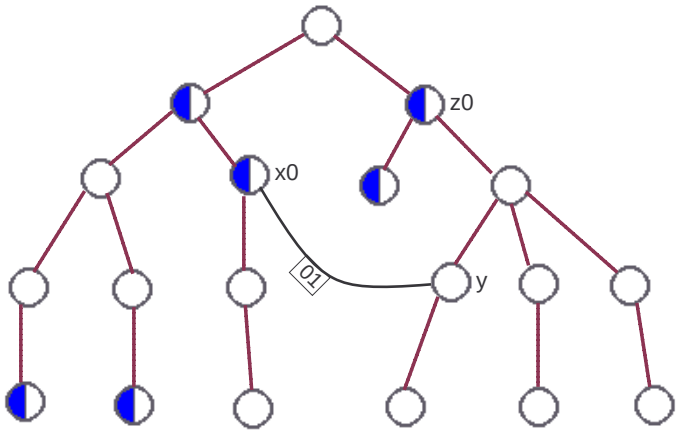
Case II Both of the end nodes have been extended blue. In this case one of the two end nodes must also be extended green. Thus the two end nodes are the only candidates for splitting and all other extended nodes are solid.

Case III One end node, $x\cdot$blue, has been extended blue and the other, y, is unextended. The node $x\cdot$blue might be split and so is a candidate for splitting. If $x\cdot$blue were not to split then y should be colored green and it is as though we have a blue-green path from which we find candidates. From that path the candidates are y (the only green node on the path) and the blue extended node on the path from $x\cdot$blue to y which lies nearest y, call it $z\cdot$blue. Thus all the extended nodes except two, $x\cdot$blue and $z\cdot$blue, may be labelled as solid. (Figure 3 shows an example.)

5.1.4 Additional Comments About Solid Node Determination

Once a node has been labelled as solid this will not change during the extension step. As more nodes are extended it is not necessary to recheck the nodes already labelled solid. A candidate may however later be found to be solid. It is not true that one of the candidates must be the split node. There might be an unextended node between the two candidates which ought to be the split node.

Figure 3: Example of Case III. All the blue extended nodes can be labelled solid except $x0$ and $z0$.



In applying the 01-path method, one pass through the 01-edges suffices to label all the solid nodes that can be determined by the method. And as mentioned before when executing the blue-green path method two passes through the node list labels all the solids which the method can discover. Iterating these solid checking methods is of no use.

However the blue-green path method and 01-path method may be used in combination. After the blue-green path method has found one or two candidates and labelled all other extended nodes solid, the non-solid candidate node(s) may be quickly checked using the 01-path method and possibly found solid.

5.2 PP Inference Rules

Now that we have developed the notion of solid nodes and characterized perfect phylogeny paths we are ready to describe the set of PP extension and resolution rules for the algorithm. The rules are in order of simplest to most complex.

5.2.1 Direct Implications of Solid Nodes

Two simple rules using the solid nodes are evident. These two together address directly the resolution of the 01-edges and the X-edges of G_k for which the first two extension rules did not apply.

Extension Rule 3 *Let* x *and* y *be nodes of* G_k *joined by a* 01*-edge. If* x *is blue extended and solid then* y *shall be extended green.*

This is the simplest but also the most powerful of the PP extension rules.

Recovery Rule for Missing Data *Given an X-edge in* G_k *both of whose end node* k*-haplotypes have been extended and are solid, the genotype data site X value from which this edge's type comes shall be replaced with the value corresponding to the two extension values.*

Being able to resolve the X-edges is important for otherwise the genotype concerned must be dropped from the current run of the incremental algorithm and there are fewer edges (i.e. less data) to work with. Also if recovered data is applied to additional runs it can greatly increase the efficiency of the algorithm.

5.2.2 Path Based Extension Rules

The next two rules aggressively use the fact that the blue and green components of P_{k+1} when regarded as k-haplotypes in P_k are subtrees with only one common node. Together they help complete the coloring of the blue and green subtrees particularly in problems with much missing data or relatively small genotype sample.

Extension Rule 4 *Let* x *and* y *be nodes of* G_k *both colored blue, then any* k*-haplotype on the* P_k *path from* x *to* y *shall also be colored blue.*

Extension Rule 5 *Let* x *and* y *be nodes of* G_k *where* x *is colored blue and is solid and* y *is colored green. If z is a* k*-haplotype such that* x *lies on the* P_k *path from* z *to* y *then* z *shall also be colored blue and labelled as solid.*

Rule 4 is implemented by fixing an extended end node of one color and considering all the possible second end nodes of same color. All unextended nodes are checked for inclusion on each of these paths as are any candidates of the opposite color. This is done for both colors and suffices to cover all paths for which the rule applies. Similarly Rule 5 is implemented by fixing an extended end node of one color and considering all the possible

Figure 4: Example where Rules 4 and 5 may be applied. By Rule 4, nodes b and d may be colored green since they lie between a and e, also nodes e, h, and j may be colored blue since they lie between f and m. By Rule 5, nodes g and k may be colored blue. Notice that if Rule 5 did not require the middle extended node to be solid it would cause node g to be incorrectly colored green since the green node e lies between it and the blue node i.

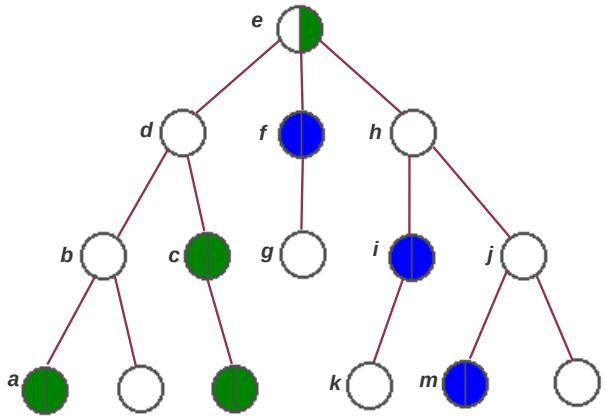
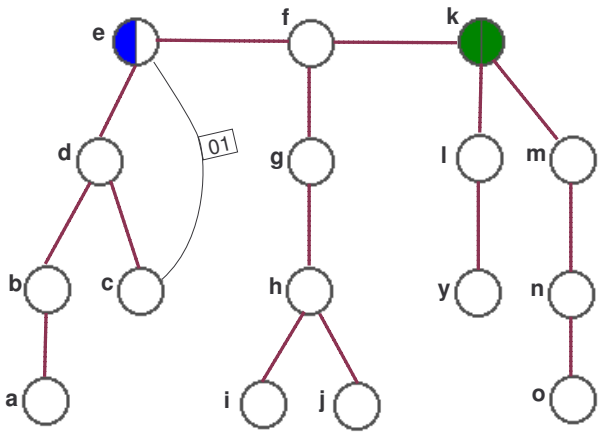


Figure 5: Example where Rule 6 may be applied. By Rule 6, all nodes except a, b, c, and d may be colored green. In this case Rule 6 will find e to be the split node



solid *in-between* nodes of the opposite color. For all unextended nodes, the *in-between* nodes are checked for inclusion on each of these paths from the unextended node to the fixed end node.

Figure 4 gives an example of a situation in which Rules 4 and 5 may be applied. Notice that in this case Rule 4 is able to determine the split node. After doing so all the newly colored nodes would be labelled solid and the two remaining uncolored nodes could be colored by another application of Rule 5. The figure also illustrates the necessity in Rule 5 of the *in-between* extended node being solid.

5.2.3 An Inference from a 01-edge and a PP Path

One additional extension rule was found.

Extension Rule 6 *Let* x *and* y *be nodes of* $\mathsf{G_k}$ *joined by a* 01*-edge. Let* z *be a blue colored node such that* y *lies on the* $\mathsf{P_k}$ *path between* x *and* z*. Then* y *shall be colored blue; also color blue any node* v *for which* y *does not lie on the* $\mathsf{P_k}$ *path between* v *and* z*.*

The derivation of this rule is as follows. The split node s must lie on the P_k path from y to x because of the 01-edge. It is the case then that there is a P_k path from x to s to y to z. Since s and z will both be extended blue so will y. (This is true of course even if y is the split node itself.) Further we may now use y as a root for a blue subtree. Any green node must be on the opposite side of y as z, thus if y does *not* lie on the P_k path between v and z then v is certain to be in the blue subtree.

In some circumstances Rule 6 can be quite powerful. See Figure 5 for such an example. It can fill in the blue and green subtrees and occasionally finds the split node when the other rules cannot. However in typical decoding situations the extensions which it can infer are also inferred by Rule 4 or Rule 5. In fact if the split node has been found then Rule 6 is totally redundant to Rules 4 and 5.

6 Summary of the *IncH* Algorithm

Having described the basic methods and decision rules for incremental haplotyping (*IncH*) , we now summarize the complete algorithm.

The algorithm takes as input a list of genotypes in the previously described format of strings from $\{0, 1, 2, X\}$. For each genotype it returns a haplotype pair composed of symbols from $\{0, 1, X\}$ which together explain

(at least partially) the genotype and such that all the haplotypes fit in a perfect phylogeny tree.

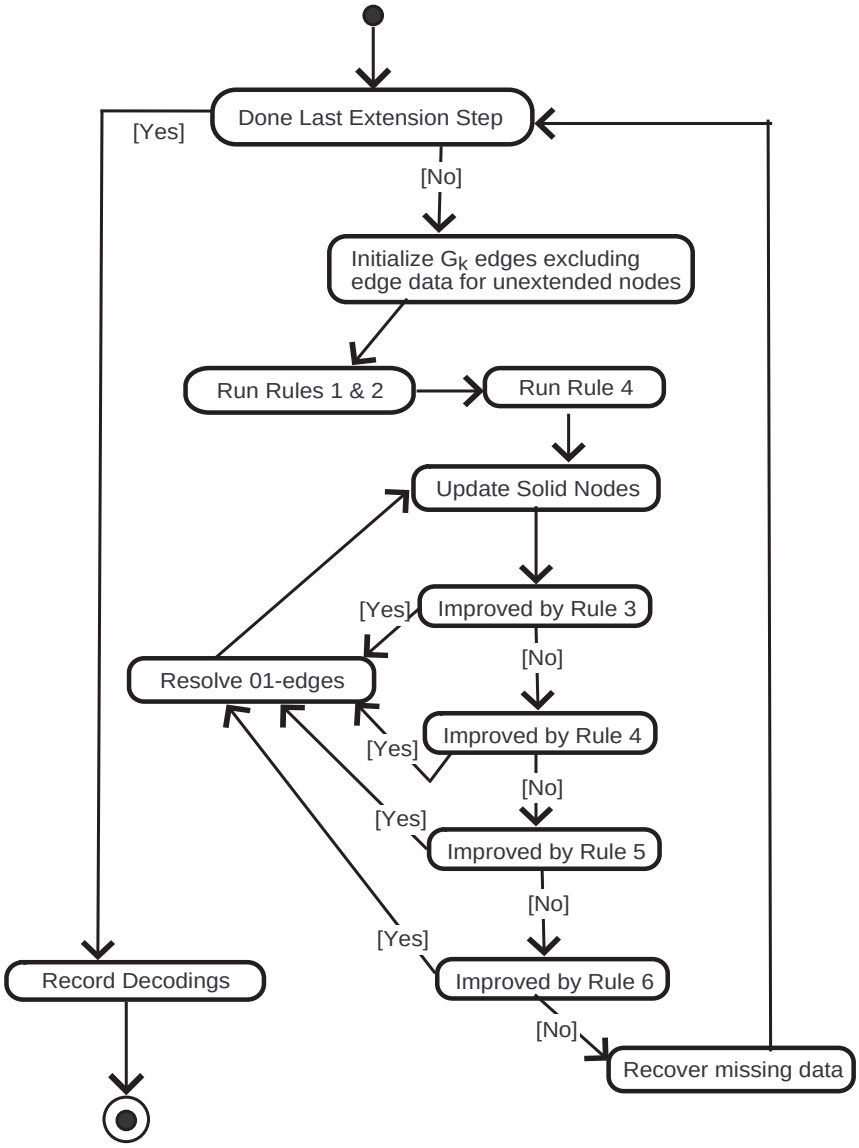
For each genotype in the sample we create a coupling-edge; these are the edges of G_k, the sample graph. The correspondence between these G_k edges and the genotypes is maintained throughout the process. The extension process begins with one node (a zero-haplotype) and for each genotype in the sample a G_0 edge from this node to itself is added. The edge type is set from the first data site value of its genotype. After the node and edges have been initialized in this way, the inference rules are applied.

Extension Rules 1 and 2 are applied to all the edges of the proper type. None of our inferences will increase the scope of these two rules so they need not be repeated during the extension step. Extension Rules 3, 4, 5, and 6 are collectively iterated until no new extension can be found by any rule. These rules are iterated because a node extension found by any of the rules may provide information which will allow that rule or another rule to infer some other extension. (An interesting optimization detail is how these inference rules should best be prioritized. See Figure 6 for our specific scheme.) After a rule has succeeded in extending one or more nodes, unresolved 01-edges incident to a newly extended node are correctly resolved if possible. Next, the solid status of the non-solid extended nodes is checked. Once no more nodes can be extended by the extension rules the Rule for Recovery of Missing Data is applied. This rule actually corrects X data sites in the genotype data sample.

If data sites remain, a new extension step is initialized. All nodes successfully extended in the previous step are used in the new round. For the new extension step the coupling-edges with both end nodes extended and found solid in the previous step may be used. All such edges have their type updated from their genotype. The G_k (or $k+1$ step) type label for each edge is got from its genotype's $k+1$ data site. The unresolved edges which cannot be unambiguously drawn in G_k may not be used in the $(k+1)$-extension step but provide a partial decoding of their genotypes in their $(k-1)$-haplotype end nodes. We now go back and apply the inference rules to the newly updated G_k.

This process is repeated until all the data sites have been processed when the decodings of the remaining edges are recorded and the algorithm ends.

Figure 6: Activity diagram for the Incremental Haplotyping (IncH) algorithm.



Part III
An Implementation of Cumulative Incremental Haplotyping

7 Overview of the Algorithm

The incremental haplotyping (*IncH*) algorithm which we have described can be applied to any sequence of data sites from a sample of genotypes to incrementally decode those genotypes. It is very possible that if the *IncH* algorithm used a different sequence of bit site data it would produce a different set of decodings. If the PP assumption held, these decodings would be compatible with the first set but for any given genotype the set of data sites successfully decoded may not be the same. The cumulative incremental haplotyping (*CIncH*) algorithm repeatedly permutes the data sites of the genotype data sample and applies the *IncH* algorithm then merges the decodings to construct an improved decoding for the genotype sample. For even relatively small genotypes (say ≥ 15 data sites) the number of permutations is so great that for practical purposes the process may be continued indefinitely. This general iterative scheme is outlined in Figure 7.

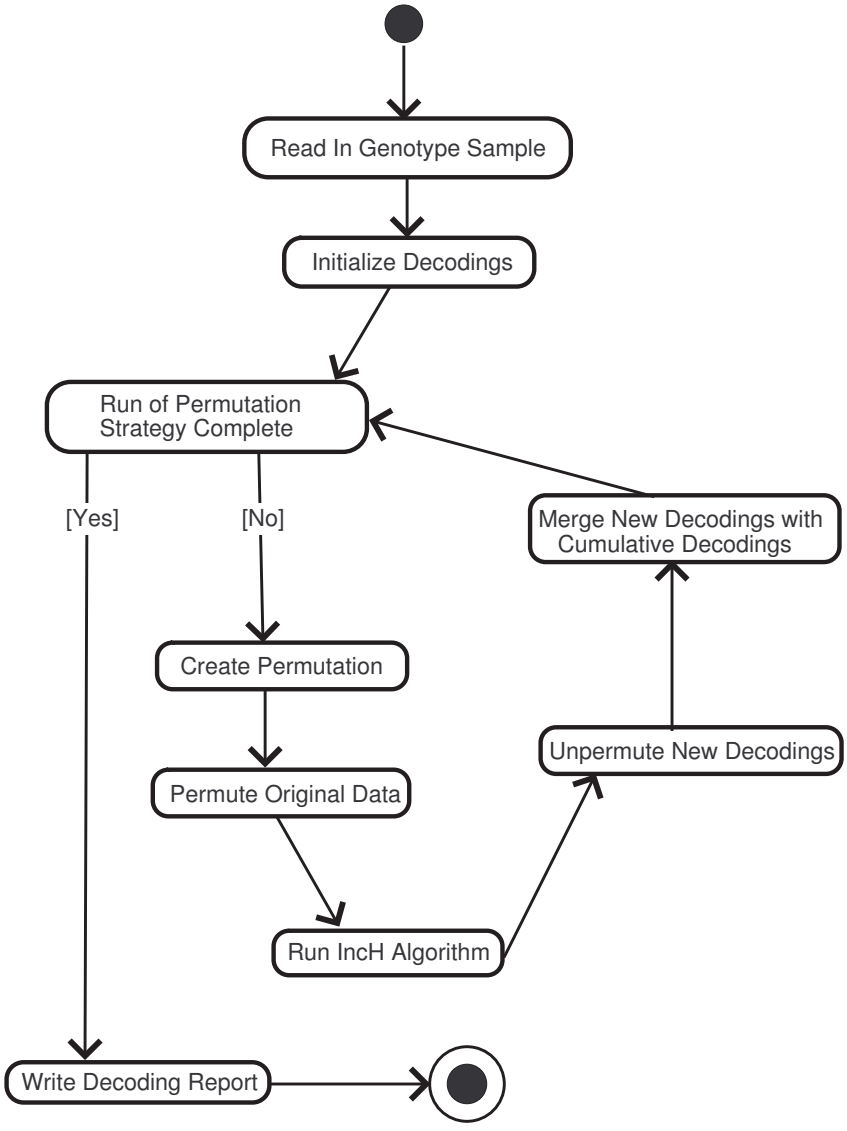
7.1 Permutation Strategies

For haplotyping problems large enough to be of practical interest the number of site permutations is far more than could be processed by the *IncH* algorithm in an acceptable amount of time. Thus it is significant what method is used to generate the permutations in the *CIncH* algorithm. Some permutations may be more fully decoded by the *IncH* algorithm than others. Certain data sites or subsequences of sites extend better and lead to more complete decodings when they appear early in the site sequence processed with *IncH*. The iterative algorithm works effectively when it obtains *IncH* decodings that it may combine to form cumulative decodings which are as complete as possible. But such *IncH* decodings may not necessarily be very complete themselves.

Creating an optimal permutation strategy would be a very complex undertaking so we have implemented two simple strategies which merely try to avoid repetitions of "bad" subsequences which might block progress of the cumulative decoding.

- Natural Order Strategy: The data sites are shifted then processed in

Figure 7: Activity diagram for the cumulative incremental ($CIncH$) algorithm.



their natural (original) order or its reverse. The shift value is incremented for each run.

- Random Permutation Strategy: A starting data site is incrementally chosen and the remaining sites permuted randomly.

Several variations on the Natural Order Strategy have been implemented with the total number of permutations available $2M$ where M is the number of data sites. For the Random Permutation Strategy all $M!$ permutations are possible. No precaution has been taken to avoid repeated random permutations but clearly this is of no practical importance.

7.2 Applying the Inference Rules

The *IncH* algorithm operates as a module which is passed a list of permuted genotypes to which it adds the set of decodings that it infers. As explained earlier, in each extension step Rules 1 and 2 are applied only once for none of our inferences will increase the scope of these rules. Rules 3, 4, 5, and 6 are iterated as a group. The iterated extension rules are executed prioritized by effectiveness, so that Rule 4 is applied if Rule 3 has failed, and so on. (See Figure 6). It should be noted that as an optimization Rule 4 is executed once ahead of the iteration loop. The idea of this is that Rule 4 does not require any determination of solid nodes, so it is done ahead of the first solid node determination.

At the end of each extension step a check on the extended nodes is done to see that they fit a perfect phylogeny. A PP violation has occurred if a phylogeny path from blue to green to blue exists. Checking this is very similar to implementing Rule 4. If such a violation is detected the *IncH* algorithm terminates and the decodings up to the previous step are recorded.

7.3 Merging Decodings

An idea that is central to the *CIncH* algorithm is that decodings of a particular genotype produced by repeated runs of the *IncH* algorithm may be merged into more complete decodings. If two inferred haplotypes have no site at which one has a 0 and the other a 1 then they can be merged. Merging is creating a new haplotype with a 1 or 0 where either of the two originals has a 1 or 0. The process just amounts to using one string to replace some X's in the other.

However deciding whether to merge two haplotypes is more complicated than deciding if they can be merged. To insure correct decodings we must unambiguously determine which inferred haplotypes from each decoding pair should be merged. We will write $h_a \sim h_b$ if the haplotypes h_a and h_b can be merged and $h_a \not\sim h_b$ if they cannot. The haplotype that is produced by merging h_a and h_b we shall represent by $h_a \oplus h_b$

Decoding Merge Rule *Let* h_0, h_1, h_2, *and* h_3 *be strings of equal length from* $\{0, 1, X\}$ *with the unordered pairs* $(\mathsf{h}_0, \mathsf{h}_1)$ *and* $(\mathsf{h}_2, \mathsf{h}_3)$ *partial decodings of a genotype* g.
• *If* $\mathsf{h}_0 = \mathsf{h}_1$ *and* $\mathsf{h}_0 \sim \mathsf{h}_2$ *and* $\mathsf{h}_1 \sim \mathsf{h}_3$, *then* $(\mathsf{h}_0 \oplus \mathsf{h}_2, \mathsf{h}_1 \oplus \mathsf{h}_3)$ *is the correctly merged decoding of* g.
• *If* $\mathsf{h}_0 \sim \mathsf{h}_2$ *and* $\mathsf{h}_1 \sim \mathsf{h}_3$ *and either* $\mathsf{h}_0 \not\sim \mathsf{h}_3$ *or* $\mathsf{h}_1 \not\sim \mathsf{h}_2$, *then* $(\mathsf{h}_0 \oplus \mathsf{h}_2, \mathsf{h}_1 \oplus \mathsf{h}_3)$ *is the correctly merged decoding of* g.
• *If* $\mathsf{h}_1 \sim \mathsf{h}_2$ *and* $\mathsf{h}_0 \sim \mathsf{h}_3$ *and either* $\mathsf{h}_1 \not\sim \mathsf{h}_3$ *or* $\mathsf{h}_0 \not\sim \mathsf{h}_2$, *then* $(\mathsf{h}_1 \oplus \mathsf{h}_2, \mathsf{h}_0 \oplus \mathsf{h}_3)$ *is the correctly merged decoding of* g.

The first part of the rule covers a special case in which one decoding is homozygous and either merge order is correct. The last two parts handle the usual situation in which one merge order is possible but the other is not. This amounts to finding a 2 in the genotype which both decoding pairs have explained. If this can be done the site unambiguously shows which haplotypes correspond to one another.

It may be that no common 2 has been decoded. In such a case either merge order is possible and we cannot merge the decodings with certainty. When this happens the algorithm retains both decodings and attempts to merge them both with any new run decodings that are obtained. Often after a third decoding is merged with one of them, the two can be merged. Storing multiple decodings can lead to eventual complete decoding or capture potentially useful information about the complete decoding.

There is however the worry that unsuccessful merging might cause the number of unmerged decodings to grow in proportion to the number of 2's in the genotype. This would not be desirable for several reasons. An important one is that merging can become computationally complex: each run decoding must be checked for merging with all existing decodings; and if a merge is done its result must be checked for merging with all others. The problem of accumulating unmerged decodings is analogous to the well-known "Birthday Problem" with the 2 sites as the days and a decoding's decoded 2 sites as its birthday(s). In fact the probability of adding an unmerged decoding is lower than adding an individual with an unique birthday in the "Birthday

Problem" solution because the decodings can have more than one birthday (i.e. decoded 2 site) and some birthdays can be common (i.e. some 2 sites are easier to decode). The probabilities weigh heavily against long lists of unmerged decodings just as they do against large sets of people with no two sharing a birthday.

8 Design of the Java Program

Our theoretical development of the *IncH* and *CIncH* algorithms has been accompanied by development of a computer program which implements both. The development and testing of the program has been a large part of this project and occasionally aided development of the algorithms themselves.

We here give a brief description of the program design. The implementation is in Java and in this portion of the paper it is assumed that the reader has at least a casual acquaintance with Java and object-oriented programming. Figure 8 shows the major classes and their use relations. We shall in turn describe each major class and the role it plays in the algorithms. In the text the names of Java classes appear in bolder font.

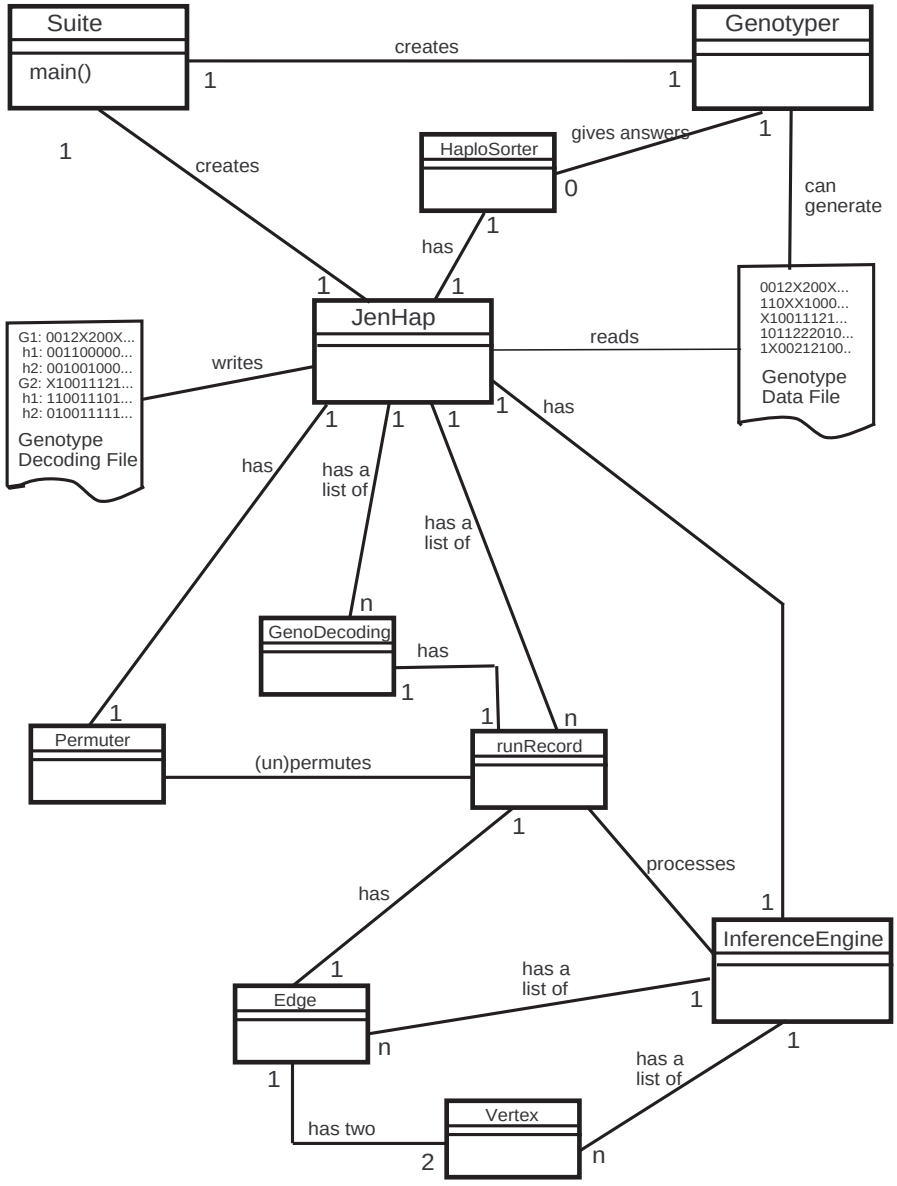
8.1 Suite

The class Suite is the main class. Suite extends the Java class JPanel and provides the basic user interface. That interface is a JTabbedPane with four JFrames. Suite creates an instance of JenHap and an instance of Genotyper, each of which supplies one of the four JFrames. These two classes are described below. The other two JFrames come from two simple classes HowToUse and About. These JFrames display information of the type indicated by their names. The classes Suite, HowToUse, and About are essentially the same as in M. Ewald's implementation of Damaschke's IHI algorithm.

8.2 Genotyper

The class Genotyper extends JPanel to provide its own interface and may be used to generate test data. The user sets the number of data sites for the sample and the sample size. Genotyper builds up a PP set of haplotypes through a process of random exclusive site mutations. The number of haplotypes generated is always one greater than the number of bits. The generated haplotypes are randomly selected (with replacement) and paired to create the genotype data. The user may specify a maximum ratio of the frequency of the most common haplotype in the population to the least.

Figure 8: Class diagram for Java implementation of the cumulative incremental haplotyping ($ChcH$) algorithm.



This to simulate natural haplotype frequency variations in populations and samples. Both the haplotypes and genotypes are written to files specified by the user.

This class is largely the same as in M. Ewald's implementation of Damaschke's IHI algorithm. One important addition is that when Genotyper generates data for a new problem it passes the generated haplotypes to the helper class HaploSorter. This is done so that when testing the program on generated data, HaploSorter which helps in reporting the results of the decoding, may check the correctness of the inferred haplotypes and report it. If a sample file is used which has not been newly generated then no check for haplotype accuracy is done.

8.3 JenHap

JenHap also extends JPanel to display the interface for the *CIncH* algorithm and is the control class for the cumulative algorithm. The user directs the program to the sample file and specifies an output file into which the program writes the accumulated decodings. The user selects a run strategy and JenHap controls the overall execution. The JPanel also displays a large text area in which status and decoding results are available to the user.

At start up, Jenhap creates an instance of the Permuter class, the InferenceEngine class, and the HaploSorter class. It uses the Permuter to permute data and decodings for individual runs of the *IncH* algorithm. The InferenceEngine executes the *IncH* algorithm. The HaploSorter aids in reporting the inferred haplotypes. For each new haplotyping problem JenHap creates and maintains ArrayLists of GenoDecodings and runRecords which contain both the sample data and inferred decodings.

The JenHap class executes a run strategy as a series of single run steps. For each single run step JenHap invokes the *IncH* algorithm but it must also pre-process the genotype data and post-process the run decodings. The single run step process then is as follows:

1. Permute, using Permuter, the sample data in the list of runRecords.
2. Call the InferenceEngine to infer the haplotypes from the list of permuted runRecords.
3. Un-permute the decodings which the InferenceEngine has written into the runRecords.
4. Call each GenoDecoding in the list to try to merge the new run decodings in the runRecords.

This single run process is repeated until the chosen run strategy is complete. The user has the following run strategy options:

- *Even Start Positions* (ESP) - Use the natural ordering of data sites and starting from every even data site execute a single run.
- *All Start Positions* (ASP) - Use the natural ordering of data sites and starting from every data site execute a single run.
- *Reverse* - Add the reverse permutations to either of the above options. With the reverses added these options are designated as RESP and RASP respectively.
- *Random Permutations* (RPM) - Execute a specified number of runs using pseudo-random permutations.
- *RPM forever* - The program continues to run with pseudo-random permutations until stopped by the user.

Upon completion of the run strategy the program writes to file the genotype decodings. These are also displayed in the text area of the JPanel along with a list of inferred haplotypes and a few relevant statistics. If the decoding has been done using a sample file newly generated by Genotyper then the correctness of the inferred haplotypes is reported. When the program is running the RPM forever strategy, the status and decoding results are updated occasionally.

When one strategy option has been completed another may be applied to the same problem so that the decodings are either improved or unchanged.

8.4 GenoDecoding and runRecord

The GenoDecoding class stores the accumulated decoding information and handles the details of merging decodings. GenoDecoding contains one primary decoding but if additional decodings are found which cannot be merged they are stored in a LinkedList. The runRecord is a simpler class with a genotype and a single run decoding.

When a new genotype sample file is read by JenHap, it creates a GenoDecoding for each unique genotype. The GenoDecoding in turn creates a runRecord and an exclusive one-to-one relation holds between the two for the life of the problem. JenHap maintains lists of both classes and clears them when a new genotype input file is read. The list of runRecords is the internal data which is permuted, passed to the InferenceEngine, returned with the inferred decodings, and un-permuted. The list of GenoDecodings accumulates

the decoding information with each merging its own runRecord after every *IncH* algorithm execution. If the new run decoding can be correctly merged with an existing decoding so to improve it then GenoDecoding will attempt to merge all the existing decodings. If however the new run decoding cannot be safely merged, it is added to GenoDecoding's list of decodings.

8.5 Permuter

A single instance of the Permuter is created and used by JenHap. This utility class has functions to process the list of runRecords it is passed when called. It is called first to permute the (original) genotype data for every runRecord and later to un-permute the new run decodings for each runRecord. It handles both the natural order and pseudo-random permutation situations. When the program is running the random permutation strategy the Permuter generates the next permutation to be used by incrementing the start position and then choosing the other sites randomly. The permutation is stored as an integer array and used to un-permute the new decodings.

8.6 InferenceEngine

The *IncH* algorithm is implemented here as shown in Figure 6. A singleton InferenceEngine is created and used by JenHap. When called by JenHap to execute a run of the algorithm, the InferenceEngine is passed an ArrayList of runRecords. One initial Vertex is created, as is an Edge instance for each runRecord. Each Edge has two (not necessarily distinct) Vertexes. The Edges realize the edges of the sample graph G_k and the Vertexes realize the haplotype-nodes. There are member functions for each of the inference rules, for solid checking, edge adjustment, etc.

As the program has been designed to run indefinitely no member instances are created by the InferenceEngine. The lists of Edges and Vertexes that it uses for inference are automatic objects. When control passes back to JenHap from the InferenceEngine after a run of the *IncH* algorithm all Edges and Vertexes are de-referenced and that memory may be freed by the garbage collector.

8.7 Edge and Vertex

When the InferenceEngine is called to infer the haplotypes from the list of runRecords that it is passed, it creates a single initial Vertex and creates one Edge for each runRecord's (permuted) genotype. Each Edge knows its runRecord and two (not necessarily distinct) Vertexes but not vice versa. As

the algorithm proceeds through the extension steps the resolved Edges reset their type from their runRecords. When an Edge cannot be resolved it writes its partial decoding (that is the last certain values of its two Vertexes) to its runRecord, then it is dropped from the list and de-referenced. The Vertex list grows as split nodes are found. Each Vertex stores its inferred bit vector as both a StringBuffer and as a BitSet. The BitSet representation allows for fast phylogeny path checking.

9 Algorithm Complexity

We shall now consider the computational complexity of the major operations of in the *CIncH* and *IncH* algorithms. First we must consider how to reckon the complexity of path checking which is based on bitwise operations on the k-haplotype bit vectors. With or without using the bitwise operations, checking if a sequence of three k-haplotypes lie in a P_k path is $O(k)$. However using the bitwise operations reduces the time by a very large constant factor because of Java's handling of arbitrarily large integers and BitSets. Indeed it is very likely that for the lengths (< 500 bits) of the bit vectors of interest in the haplotyping problem all testing will show path checking to be constant time operation. We shall note this practical complexity along with the theoretical.

Let m be the genotype length and n be the sample size.

9.1 The *IncH* Algorithm

9.1.1 Inference Rules

Rules 1, 2, 3, and the Recovery Rule for Missing Data each require one pass through the Edge list. Their execution then is $O(n)$.

Rules 4 and 5 each fix an end node and allow two other path nodes to range all possibilities. Thus if we use the linear complexity for path checking we get a fairly high worst-case complexity that is $O(m^3)$. However this reduces to quadratic complexity if we use the constant time path checking estimate.

Rule 6 seeks, for each 01-edge, an extended node and then checks all unextended nodes for path inclusion. This gives a complexity that is $O(m^2n)$. If we take path checking to be constant time then this reduces to $O(mn)$.

9.1.2 Solid Checking

The blue-green path method requires at most two passes through the Vertex list with the extended nodes being checked in the first pass and the candidates in the second. Thus the operation is $O(m^2)$ or $O(m)$ if we take path checking as constant time.

The 01 path method if done in the case where only one color extension is made, requires checking path inclusion for each non-solid extended node on every 01-edge path. This gives a complexity that is $O(m^2n)$. If done to the candidates from the blue-green method the complexity is $O(mn)$. These reduce to $O(mn)$ and $O(n)$ respectfully if path checking is taken as a constant time operation. Notice about the one color $O(m^2n)$ case, this will be done at most four times per extension step because if only one extension color exists then Rule 3 has never been successful.

9.1.3 Conclusions

In order to make a final judgement on the computational complexity of the *IncH* algorithm a bound must be set on how many times each of these operations may be executed during an extension step. Clearly Rules 1, 2, and the Recovery Rule for Missing Data are only done once per extension step and so together are only $O(mn)$ for an *IncH* run. So the question of complexity comes down to a question of how many times iterations of the main loop (in which Rules 3, 4, and 6 are called) is possible for a given extension step. The loop is of course iterated so long as nodes are being extended but how long can that process drag out?

The Rules 4, 5, and 6 all by nature fill in subtrees of one color. And if the three are all run twice they have completely filled the subtrees to which they can be applied. Thus if the main loop is to be iterate more than a few times it must be that Rule 3 is extending nodes every few iterations. Is this possible? The P_k tree would have to be elongated and the 01-edges in a very unfavorable arrangement and order on the actual Edge list. In such an extreme case we might see k calls to Rules 3, 4, 5, and 6 in an extension step. It might even be possible to construct a problem sample set where many permutations yield a time complexity similar to the worst imaginable case. That would be k calls to Rule 6 are made at each extension step for $O(m^3n)$ (or $O(m^2n)$ with constant time path checking) for an *IncH* algorithm run. But the odds of such happening are so slight especially for large problems as to be of only theoretical interest.

For practical purposes we expect the average time complexity of the *IncH* algorithm to be dominated by the application of Rules 3, 4, and 5 which will normally complete the extension step in only a few iterations. An upper bound on the iteration of the main loop ought to be the height of the P_k tree but that should normally be a loose bound only approached when the Edge arrangements are very unfavorable. By this average case analysis we would expect to see the time complexity of the *IncH* algorithm to be asymptotic to $(\alpha mn + \beta m^3)$ where α and β are constants. This reduces to $(\alpha mn + \beta m^2)$ if path checking is considered a constant time operation.

9.2 Permuting

Permutation of the genotype data and un-permutation of the run decodings are both $O(mn)$ complexity and done once per *IncH* algorithm run.

9.3 Merging

Merging complexity is $O(mn)$ per *IncH* run if we assume that each GenoDecoding has only one decoding. If the number of decodings were to grow like m for all the genotypes then merging complexity could be as bad as $O(m^3n)$. However as seen above the total number of unmerged decodings, will tend to be low compared to m.

10 Performance of the Program

10.1 Correctness of the Algorithm

The Java program implementing the *CIncH* algorithm is too long and complex to verify its accuracy by any kind of hand execution or step through. We shall instead present test results demonstrating correct results for a very large variety of problem parameters.

The program includes a test data generator, the Genotyper class, and when the program is executed upon generated test data the inferred haplotypes are checked for correctness. An inferred haplotype is considered correct if it is in the perfect phylogeny generated by the Genotyper. In a formal test of the algorithm's correctness under varied parameters a total of 18,029 correct haplotypes and no incorrect haplotypes were inferred. (See Table 1.) In addition to this formal test, no incorrect haplotypes have yet been inferred in a great many informal or indirect tests. The logic of the

Table 1: Test of inferred haplotype correctness. A total of 18,029 haplotypes were inferred in 560 trials with varying problem parameters. All were correct.

genotype sample length X size	missing data	no trials	inferred haplotypes	average found
10 X 10	0%	20	137	6.85
10 X 20	0%	20	200	10.00
10 X 30	0%	20	213	10.65
10 X 20	10%	20	183	9.15
10 X 30	10%	20	196	9.80
10 X 30	25%	20	177	8.85
10 X 40	25%	20	196	9.8
20 X 20	0%	20	268	13.40
20 X 40	0%	20	362	18.10
20 X 60	0%	20	399	19.95
20 X 40	10%	20	342	17.10
20 X 60	10%	20	409	20.45
20 X 60	25%	20	355	17.75
20 X 80	25%	20	370	18.50
40 X 40	0%	20	545	27.25
40 X 80	0%	20	727	36.35
40 X 120	0%	20	785	39.25
40 X 80	10%	20	663	33.15
40 X 120	10%	20	755	37.75
40 X 120	25%	20	625	31.25
40 X 160	25%	20	725	36.25
80 X 80	0%	20	1049	52.45
80 X 160	0%	20	1430	71.50
80 X 240	0%	20	1534	76.70
80 X 160	10%	20	1295	64.75
80 X 240	10%	20	1485	74.25
80 X 240	25%	20	1171	58.55
80 X 320	25%	20	1433	71.65

data generator and the correctness check are very simple and have been independently tested.

Also an internal check that the algorithm is working correctly is done after each extension step. This checks that the extended haplotypes actually form a PP. If a PP violation is detected the *IncH* algorithm is terminated, with the current extension step excluded from the decoding for the run. In practice such a PP violation would give the first indication of a problem with the algorithm or sample data. In the formal trial no PP violations occurred.

We conclude that if any fault exists in the algorithm or its implementation it must be fleetingly rare and/or masked by some bias of the test data generator.

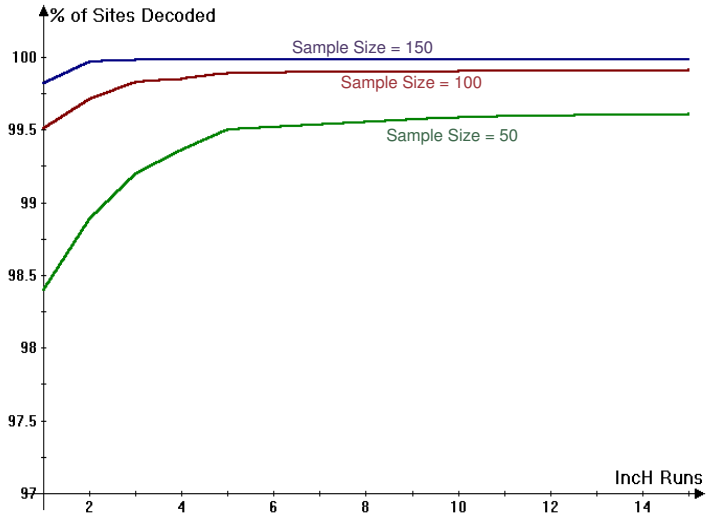
10.2 Learning Rate and Sample Size

The efficiency of the program is also of importance. The trial results in Table 1 show that for complete data one might expect to find all the $m + 1$ haplotypes with a sample size of about $m \log m$ where m is the genotype length. The near total discovery of the PP haplotypes in tests with a sample size of $\geq m \log m$ seems in good accord with Damaschke's estimate of required sample size. These trials used the MAX_WEIGHT of 5 or 2 so there were no extremely common or rare haplotypes. The tests also indicate that an increase in sample size can fully compensate for missing data.

One thing not shown in the Table 1 is the number of iterations of the *IncH* algorithm required to achieve the haplotype decodings. As one might expect the trials with much missing data require many more iterations. To examine the learning rate of the algorithm we have run a series of tests with the genotype length fixed at 50. The missing data rates used were 0%, 10%, and 20%; the sample sizes were 50, 100, and 150. Rather than measure the learning or decoding by the number of haplotypes or genotypes fully decoded, we have used the percentage of all the data sites decoded. This gives a much smoother picture of the progress of the algorithm than the others which tend to shake out in bunches. It should be noted that the percent of decoded data sites always begins fairly high because of the number of 1's and 0's in the sample. (By our calculation that number is about 2/3 of the readable sample data.) These homozygous data sites are always decoded by the program in the first step.

The results of these tests are shown in Figures 9, 10, and 11. Each curve is an average of 5 separate trials. Note the change of the scale on the horizontal axis from Figure 9 (no missing data) to Figure 10 (10% missing

Figure 9: The percentage of decoded sites versus number of Inch runs for complete genotype data. Genotype length was 50.



data) to Figure 11 (20% missing data). Compared to the complete data case, learning with 10% missing data takes about 10 times as many *IncH* runs and learning with 20% missing data takes perhaps 35 times as many. In all cases a larger data sample allowed the program to decode a higher percentage of sites and do so in fewer *IncH* runs.

10.3 Time Performance

Three tests were done to evaluate the effect on the program execution time of varying the size parameters of the genotype data input. These tests have all been done with samples with 10% missing data. All the execution time per run values are based on timing in the JenHap class of the execution of the RPM strategy with 100 permutations. For each parameter set such a time test of 10 different genotype samples was done and the average of the times was used. All tests were executed on an older PentiumII machine with 600mhz processor.

In the first such test the genotype length was fixed at 50 while the sample size was increased from 25 to 500. See Figure 12. Given the test results, it seems reasonable to conclude a linear relationship between sample size and

Figure 10: The percentage of decoded sites versus number of Inch runs for genotype samples with 10% missing data. Genotype length was 50.

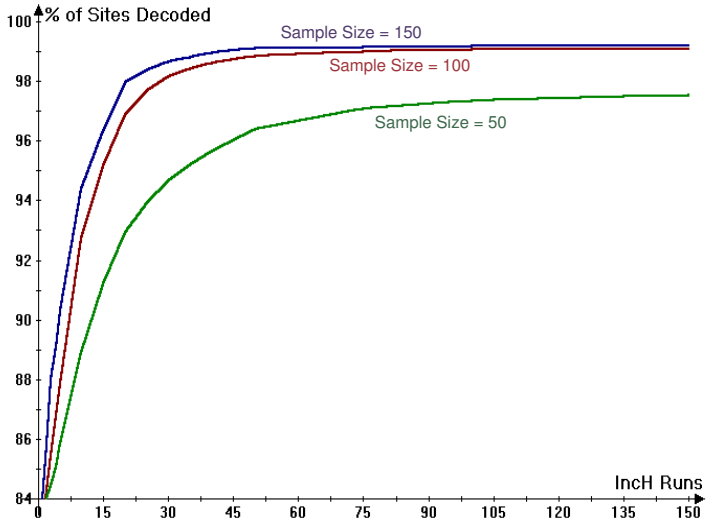


Figure 11: The percentage of decoded sites versus number of Inch runs for genotype samples with 20% missing data. Genotype length was 50.

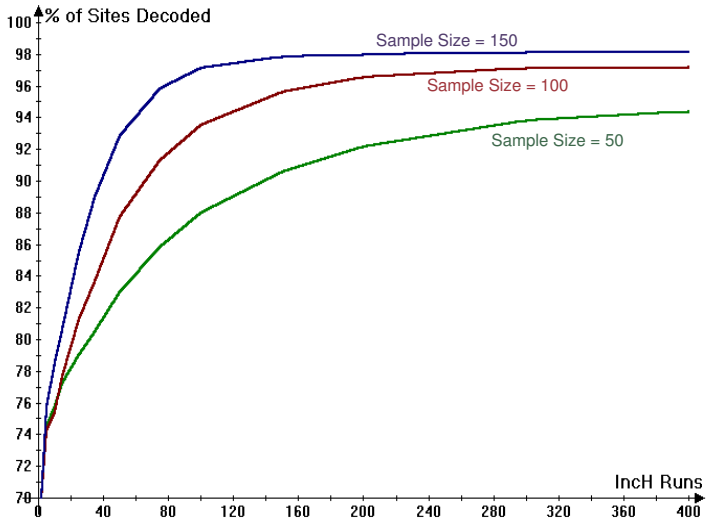
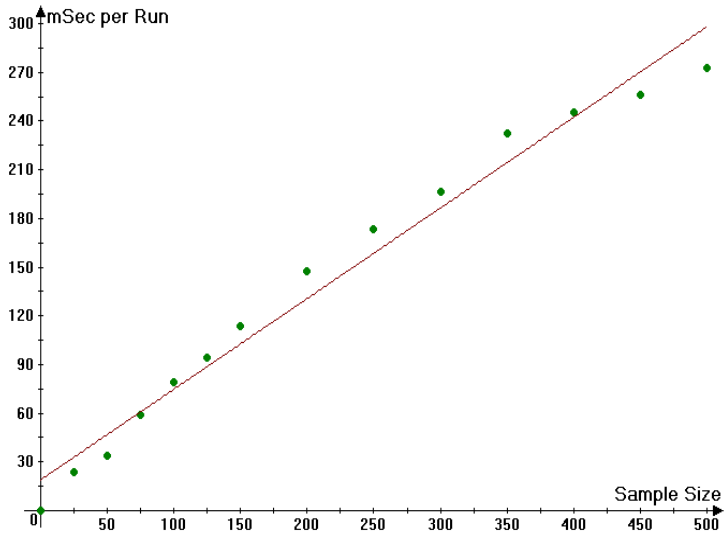


Figure 12: Increasing sample size versus execution time, with genotype length fixed at 50 bits.



IncH execution time. This agrees with our informal average-case complexity estimate of $(\alpha mn + \beta m^2)$ which is (αmn) when the sample size is greater than the genotype length.

For the second such test the sample size was held at 300 while the genotype length was increased from 10 to 200. See Figure 13. Given the test results, it seems a linear relationship exists here between genotype length and *IncH* execution time. This again agrees our average-case complexity estimate of $(\alpha mn + \beta m^2)$ which is (αmn) when the sample size is greater than the genotype length.

In a similar test the sample size was fixed at 100 while the genotype was increased from 80 to 400. The results are shown in Figure 14. The test results show a time complexity that is actually better than our average complexity estimate estimate of $(\alpha mn + \beta m^2)$ which is (βm^2) for $m > n$.

10.4 Tests with Imperfect Phylogeny Data

An important and complex issue for haplotyping algorithms based on perfect phylogeny is how they handle genotype samples that do not support a perfect phylogeny. Perhaps the population contains sites which have been mutated

Figure 13: Increasing genotype length versus execution time, with the sample size fixed at 300.

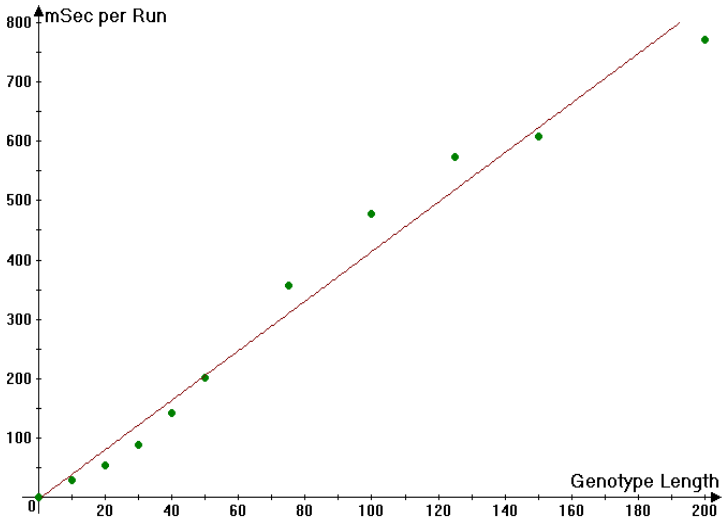
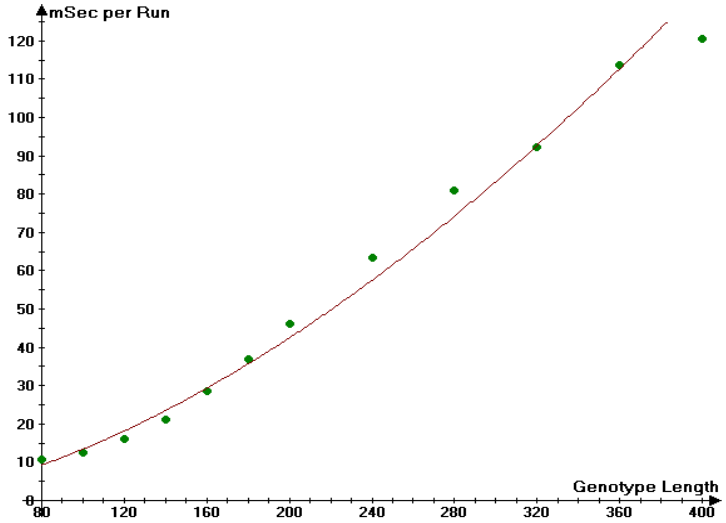


Figure 14: Increasing genotype length versus execution time, with the sample size fixed at 100.



more than once or a few "immigrant" haplotypes that fit the PP not at all or maybe errors in the data mar an otherwise PP supporting sample.

A short series of tests of the current program were run on sample data generated from haplotypes which do not quite form a perfect phylogeny. The main objects of the tests were to learn if the program is stable and how correct its decodings are when given an imperfect phylogeny data sample. The tests were conducted as follows: a set of 41 haplotypes of 40 bits fitting a PP were generated in the usual way by Genotyper; a number of the haplotypes were randomly chosen and mutated; the mutants were assigned a frequency one half that of their original and added to the haplotype population; the genotype sample was generated in the usual way. The mutants are not guaranteed to be outside the PP but are with high probability. For these tests only trials with actual PP violators were used.

The test results are presented in Table 2 but require a bit of explanation. For each sub-test there are two result numbers: the upper is the total inferred haplotypes; the lower is the number of those that were correct. The values are the averages of 10 trials. In the columns headed 1, 2, 4, and 8 that many mutants were added to the population and those mutants had one random bit mutated. In the columns headed 1d, 2d, and 4d that many mutants were added to the population and those mutants had two random bits mutated. The trials with no missing data ran 40 RPMs, the trials with 10% missing data ran 100 RPMs, and the trials with 20% missing data ran 200 RPMs. These numbers of permutations were sufficient for the program to cease finding haplotypes in all cases except some trials with 20% missing data, 4 or 8 mutants, and sample size 160 where more (mostly incorrect) haplotypes were still being inferred.

This imperfect phylogeny test was of course very limited in scope and the test scenario may or may not be realistic, still it suggests that the program is fairly well behaved when the data does not quite admit a PP decoding. For instance with 10% missing data and two (single site) mutants added the program found 90% of the haplotypes correctly with only a few extraneous haplotypes, despite the fact that almost every run of the *IncH* algorithm was terminated due to a PP violation. There is little doubt that a more sophisticated strategy for dealing with these PP violations could allow the *CIncH* algorithm to deal better with imperfect phylogeny data and become a more flexible and reliable algorithm. Development of such strategies is beyond the scoop of the current project but an area of interest for follow up research.

Table 2: Tests with data generated from haplotypes that do not form a perfect phylogeny. For each test a number of non-PP fitting mutant haplotypes were added to the 41 PP fitting haplotypes.

sample	missing	number of added mutants							
size	data	0	1	2	4	8	1d	2d	4d
40x80	0%	36.8	37.0	37.0	44.5	47.5	39.3	38.0	46.3
		36.8	36.4	35.6	38.7	39.1	36.8	34.6	36.8
40x120	0%	38.2	39.8	40.3	45.6	56.0	40.0	44.0	48.1
		38.2	38.7	38.5	40.4	43.4	38.1	39.4	41.0
40x160	0%	39.7	40.5	42.6	51.4	53.9	42.0	44.7	51.4
		39.7	39.7	40.5	43.0	44.4	39.3	40.1	41.8
40x80	10%	32.0	32.0	33.8	40.1	43.5	33.5	34.7	44.0
		32.0	31.4	32.3	34.7	36.2	30.8	31.9	32.2
40x120	10%	36.4	39.0	39.9	44.6	54.3	39.6	44.6	52.2
		36.4	37.9	37.0	39.1	38.1	37.5	37.4	38.5
40x160	10%	37.6	43.1	43.0	48.0	60.9	41.7	45.6	56.2
		37.6	42.2	39.7	42.1	43.3	38.4	39.1	39.9
40x80	20%	27.3	28.2	29.7	34.9	34.9	25.2	28.6	39.2
		27.3	25.8	25.8	27.0	23.5	23.4	21.9	26.6
40x120	20%	34.1	35.1	37.6	46.6	56.2	36.2	46.3	53.6
		34.1	33.0	33.1	35.9	37.4	31.1	34.5	34.9
40x160	20%	36.9	40.3	43.5	48.7	63.1	42.4	49.5	59.6
		36.9	37.9	39.1	38.3	40.7	38.1	37.8	38.5

Part IV
Conclusions

Both the *IncH* and *CIncH* algorithms have been shown under testing to be correct. The learning efficiency suggests that the *IncH* algorithm is making full or near full use of the perfect phylogeny information of the previous step without actually storing or reconstructing it. More direct methods would likely perform faster for complete genotype data admitting a PP solution, however it is the case of incomplete genotype data that is our main interest. Our testing indicates that the *IncH* algorithm is near optimal logically and its time performance is acceptable. In our testing on a relatively slow machine, decoding for 100 site problems could be done in a few minutes and 300 site problems in an hour or so. The average execution time increases approximately linearly with the total number of data sites in the sample and so the algorithm is computationally scalable. The goal of creating and implementing an effective incremental algorithm for haplotyping incomplete genotype data has been achieved.

As a practical tool the *CIncH* algorithm is only as good as the perfect phylogeny assumption upon which it relies. If perfect phylogenies are common at least for certain size blocks of SNP's then likely the *CIncH* algorithm can be a useful tool. In this regard the experiments with genotype data generated from haplotypes that did not quite fit a PP were very suggestive. Those tests showed that for missing data no more than ten percent, the algorithm was stable when confronted with data that did not support a PP. The algorithm still found many mostly correct haplotypes. These tests suggest that some simple blocking scheme or other heuristic might be used to better attack the imperfect phylogeny problem using the *CIncH* algorithm. Development of such imperfect phylogeny methods seem the next step in advancing incremental haplotyping. It is our opinion this next phase ought be developed and tested using real world data and could lead shortly to incremental haplotyping based on perfect phylogeny assuming an important role in the ongoing human haplotyping project.

References

[1] NHGRI staff *Developing a Haplotype Map of the Human Genome for Finding Genes Related to Health and Disease* National Institue of Health. http://www.genome.gov/10001665

[2] HapMap staff *About the International HapMap Project* International HapMap Project. http://www.hapmap.org/index.html.en

[3] P. Damaschke : *Incremental Haplotype Inference, Phylogeny and Almost Bipartite Graphs* 2nd RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes

[4] E. Eskin, E. Halperin, R.M. Karp: *Efficient reconstruction of haplotype structure via perfect phylogeny*, Journal of Bioinformatics and Computational. Biology 1 (2003), 1-20

[5] Eric W. Weisstein. *Birthday Problem.* From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BirthdayProblem.html