8-3-2006

Inferring the Structure of Signal Transduction Networks from Interactions between Cellular Components and Inferring Haplotypes from Informative SNPS

Kelly Anthony Westbrooks

INFERRING THE STRUCTURE OF SIGNAL TRANSDUCTION NETWORKS FROM INTERACTIONS BETWEEN CELLULAR COMPONENTS AND INFERRING HAPLOTYPES FROM INFORMATIVE SNPS

by

KELLY WESTBROOKS

Under the Direction of Alexander Zelikovsky

ABSTRACT

Many problems in bioinformatics are inference problems, that is, the problem objective is to infer something based upon a limited amount of information. In this work we explore two different inference problems in bioinformatics.

The first problem is inferring the structure of signal transduction networks from interactions between pairs of cellular components. We present two contributions towards the solution to this problem: an mixed integer program that produces and exact solution, and an implementation of an approximation algorithm in Java that was originally described by DasGupta et al. An exact solution is obtained for a problem instance consisting of real data.

The second problem this thesis examines is the problem of inferring complete haplotypes from informative SNPs. In this work we describe two variations of the linear algebraic method for haplotype prediction and tag SNP selection: Two different variants of the algorithm are described and implemented, and the results summarized.

INDEX WORDS: Signal transduction networks, Haplotype prediction, Transitive reduction

INFERRING THE STRUCTURE OF SIGNAL TRANSDUCTION NETWORKS FROM INTERACTIONS BETWEEN CELLULAR COMPONENTS AND INFERRING HAPLOTYPES FROM INFORMATIVE SNPS

by

KELLY WESTBROOKS

A Thesis Submitted in Partial Fulfillment of Requirements for the

Degree of Master of Science

in the College of Arts and Sciences

Georgia State University

2006

INFERRING THE STRUCTURE OF SIGNAL TRANSDUCTION NETWORKS FROM INTERACTIONS BETWEEN CELLULAR COMPONENTS AND INFERRING HAPLOTYPES FROM INFORMATIVE SNPS

by

KELLY WESTBROOKS

Major Professor:	Alexander Zelikovsky
Committee:	Rajshekhar Sunderraman
	Saeid Belkasim

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
August 2006

ACKNOWLEDGEMENTS

I would like to begin by offering thanks to my advisor, Dr. Alex Zelikovsky, for his unending patience with me as I haphazardly fumbled my way through this thesis. Thanks to my family, Lavonne, Chuck, and Karen, for teaching me that the only thing preventing me from achieving my objectives in life is my attitude. Thanks to Dumitru Brinza, my roommate and colleague, for being a “constructive competitor” and pushing me to be a smarter computer scientist. Finally, thanks to my partner, Vanessa Bertollini. Without her support and encouragement, this thesis would have never been possible.

Table of Contents

List of Figures

1 Introduction

Solving an inference problem means reckoning about unknowns in the solution based upon only partial information. Often the answer to an inference problem is a “best guess” determined using only the available data. This work deals with two different inference problems in bioinformatics, namely, the problem of inferring the structure of signal transduction networks from observations on the interactions between cellular components, and the problem of inferring unknown haplotypes from informative SNPs. Both problems are worthy of study because fast and efficient solutions to them could potentially bring about other breakthroughs in medicine and biology that could lead to treatments or cures for genetic diseases.

1.1 Inferring the Structure of Signal Transduction Networks

In order to adapt to their environment and cooperate with surrounding tissue, living cells utilize a complex collection of interacting chemicals and molecules to communicate changes and respond to stimuli, a process known as *signal transduction*. Signal transduction is the primary mechanism for maintaining equilibrium between the cell and its surroundings and allows cell to react to events outside the cell membrane. In order to further understand how intracellular signaling processes operate, biologists conduct experiments to record the the interaction strength between the pairs of chemicals involved in the signaling pathways. Once enough observations are recorded, the next task is to refine the information by discarding weak interactions and to infer the underlying structure of the signaling network formed by the interacting components. Informally, the problem of inferring the structure of signal transduction networks (STNs) can be stated in the following manner: given a set of observations on interactions between various

members of the signaling pathways, infer the smallest subset of observations that can explain the entire set of interactions.

The fundamental difficulty when inferring signal transduction network structure is that it is experimentally difficult to distinguish between direct interactions between cellular components and indirect interactions, brought about by cascading sequences of direct interactions. For a given pair of cellular components, it may be possible to explain an interaction-observation in more than one way, but impossible to determine which explanation is the correct one. This work focuses on a parsimonious approach to the problem: we attempt to explain the entire set of interactions using the smallest subset of observations possible.

1.1.1 STNs as Directed Graphs and the Binary Transitive Reduction

One way to mathematically model the collection of interaction data between cellular components is to consider a directed graph whose vertices represent the interacting cellular components under study and whose edges represent regulating interactions. Regulating interactions can be of two different types: promoting interactions, where one cellular component increases the expression of the other component, and inhibiting interactions, where one cellular component decreases the expression of the other component. The resulting directed graph captures the direct interactions between members of the signaling pathways, but it may also capture indirect interactions, resulting from a sequence of direct promoting or inhibiting interactions. From this graph-theoretical perspective, the objective of inferring signal transduction network structure is to detect and remove edges from the interaction graph that represent indirect interactions.

Since using only the data in the interaction graph it is impossible to differentiate between

direct and indirect interactions, in this work we attempt to minimize false positive inferences at the risk of making false negatives inferences when deciding which interactions are direct interactions and which interactions are indirect. With this objective in mind, we can see that in order to solve the STN inferencing problem, we need to compute a special version of the transitive reduction on the interaction graph that takes into account the interaction types. This "special" transitive reduction, known as a binary transitive reduction, maintains reachability relationships across sequences of promotion and inhibition interactions.

1.1.2 Exact Solutions to the STN Inference Problem using Mixed Integer Programming

One of the primary contributions of this work is a formulation of the binary transitive reduction problem as a mixed integer program. Mixed integer programming is one way to A Java program is presented which translates instances of the binary transitive reduction problem into a form suitable for consumption by GLPK, a popular linear and mixed integer program solver from the GNU Software Foundation. The solution given by GLPK is then re-mapped into the original problem space and the solution to the STN inference problem is outputted. The computation time for problems of various sizes, including both real and simulated data is presented.

1.1.3 An Approximation Algorithm for the STN Inference Problem

Another contribution of this work is the implementation of an approximation algorithm for the binary transitive reduction problem by DasGupta et al. This algorithm works by combining solutions for the special cases where the problem instance is a directed acyclic graph

with the case where the problem instance is a strongly connected graph. A Java program which implements this approximation algorithm is presented and its output is compared to the output of the mixed integer program in terms of both accuracy and execution time.

1.2 Inferring Haplotypes using Informative Single Nucleotide Polymorphisms

For a given species, the vast majority of the DNA between individuals is identical. However, some small subset of nucleotides exhibit variation across individuals within a species. Each nucleotide position that exhibits variation greater across a population is known as a single-nucleotide polymorphism (SNP). It is believed that SNPs account for most of the genetic variation between individuals in a given species, and more importantly, can be used as accurate predictors of the susceptibility of a given individual to a particular genetic disease. When genotyping individuals, sampling every individual SNP is expensive, and many of the sampled SNPs don't contribute much information useful for distinguishing individuals, so it is desirable to sample only the most informative subset of SNPs. These informative SNPs are known as *tag* SNPs.

1.2.1 The Haplotype Prediction Problem

There are two problems related to tag SNPs. The first problem is the tag SNP selection problem. Informally, it can be stated as follows: given a set of SNPs taken across a sample population of haplotypes, find k tag SNPs that distinguish the largest number of the haplotypes. In this problem, the focus is the discovery and identification of specific SNP sites in the haplotype population that are useful for distinguishing individuals. The second problem is the

haplotype prediction problem: given a sample population of haplotypes along with the locations of *k* tag SNPs in that population, and the values of the tag SNPs of some unknown haplotype, reconstruct the entire unknown haplotype. In this problem, the tag SNP sites are known in advance, and the values of the tag SNPs of the unknown individual, along with the entire sample population are used to infer the values of all of the non-tag SNPs in the unknown haplotype. The later half of this thesis addresses the haplotype prediction problem.

1.2.2 Linear Algebraic Method for Haplotype Prediction

There are a large number of published solutions to the haplotype prediction problem. In this work, we expand upon previously published results by exploiting natural linear dependency between SNPs. A subset of SNPs from the sample population is selected which forms a basis over the space spanned by the SNPs in the sample population, and non-tag SNPs are represented as linear combinations between the tag SNPs. The net result is a reduction in the amount of data needed to accurately represent the sample population. A Java program is presented which takes as input the sample population, the positions of k tag sites, and the values of the tag SNPs of some unknown haplotype and produces the reconstructed haplotype as output. Leave-one-out validation is used to measure the quality of the algorithm.

1.3 Roadmap

This work is organized in the following manner: Chapter 2 presents the signal transduction network inferencing problem. First, the biological motivation for the problem is introduced. In section 2.1, we show how the problem can be mathematically modeled as a binary transitive reduction problem on a graph. Section 2.2 summarizes previous work on the binary transitive reduction problem. Section 2.3 describes a mixed integer program that successfully

solves the binary transitive reduction problem. In Section 2.4, we present an approximation algorithm to compliment the exact algorithm. Section 2.5 concludes the chapter with a discussion of the maximum confidence transitive reduction problem, which may prove to be a more appropriate mathematical abstraction in future work in signaling networks.

Chapter 3 focuses on the haplotype prediction and tag SNP selection problems. The chapter begins by providing the reader with the sufficient biological background necessary to motivate the haplotype prediction problem. In section 3.1, the notations and mathematical formalism that are to be used throughout the rest of the chapter are introduced. Section 3.2 summarizes previous work on the problem. Section 3.3 describe a framework for building a greedy tag selection algorithm from any haplotype prediction algorithm. Section 3.4 introduces the linear algebraic SNP prediction algorithm (LASPA) in Euclidean space. Section 3.5 concludes the chapter with a variant of LASPA that confines computations to a finite field.

Chapter 4 provides implementation details for two Java programs that were developed for solving the BTR and haplotype prediction problems. Chapter 5 contains a discussion of the performance and results of the methods of this thesis. Finally, chapter 6 concludes the thesis.

2 Inferring the Structure of Signal Transduction Networks

Living cells use a complex network of interacting cellular components to communicate changes in their environment, respond to stimuli, and maintain equilibrium with their surroundings. These cellular components consist of proteins, DNA, RNA, and other small molecules which participate in chemical reactions with other cellular components [4]. Sequences of these regulatory interactions effectively behave like signals propagating across a network, carrying information about events and state changes to other parts of the cell. The process of transmitting information inside a cell though a series of chemical reactions is called *signal transduction* and the structure formed by the various signaling pathways between the cellular components is known as a *signal transduction network*.

The complex structure of signal transduction networks gives cells their biological characteristics and distinguishes them from simple, non-living chemical systems. Therefore, understanding the structure of the signaling pathways inside the network is critical to understanding how biological processes operate at a cellular level. However, there are a number of difficult experimental problems to overcome in the laboratory when mapping out signaling networks.

The primary problem lies in the way the data must be collected. It is difficult to determine if there is any interaction between two cellular components in isolation; the data must be collected inside the cell where it is possible that other unknown cellular components participate in the interaction. As a consequence, the data collected consists of both direct and indirect interactions. *Direct interactions* occur when one cellular component directly influences another via chemical reactions, while *indirect interactions* occur when one cellular component influences

another through a series of intermediate direct interactions. The fundamental objective of the signal transduction network inferencing problem is to identify which interactions are direct and which are indirect.

Other problems exists as well. Since signaling pathways are used inside the cell for self-regulation and equilibrium maintenance, they are typically self-intersecting. Experimental evidence suggests that cellular signaling networks may consist of thousands of cellular components, however these experiments may not cover all possible regulatory interactions [10,11,20,21]. Further, interactions between different components can occur with differing strengths and timings, yet be equally important members of the signaling pathway.

Conversely, not every cellular component has equal influence over the entire signaling pathway. One key to solving some of the problems stated above is to identify which cellular components affect the entire process the most, and determine which interactions these key components are directly involved in [9]. Once this is done, biologists will have a partial idea of what the signaling network looks like, which they can use to make educated guesses about which types of experiments would be most useful in completing the overall picture.

There are three types of evidence that are useful in determining if a pair of cellular components participate in a regulatory interaction. The first type is biochemical evidence obtained by direct observation of protein-protein interactions. Biochemical evidence of a regulatory interaction requires some knowledge about the specific chemical reactions involved between the two cellular components and often leads to inferences about direct interactions. The second type is genetic evidence based upon observations of how wild-type and mutant organisms differ in their responses to a specific stimulus. With genetic evidence, the specific chemical reactions related to the regulatory interaction may be unknown, but it is still possible to infer the

presence of an indirect interaction because the mutant organism lacks the cellular component that completes the signaling pathway and, consequently, fails to express the traits related to that pathway. Finally, pharmacological evidence in which the experimenter artificially eliminates or substitutes cellular components in the pathway, is helpful for inferring the presence of indirect interactions. Although biochemical evidence produces inferences on direct interactions, genetic and pharmacological evidence is more common and easier to obtain. However, since biological evidence for specific interactions exists, it is important that algorithmic methods for differentiating between direct and indirect interactions accept as an input the assumed initial set of direct interactions.

In addition to regulatory interactions being classified as either direct or indirect, they can also be classified in another manner as well: interactions can either be of the *promoting* sense, where one cellular component increases the expression of the other component, or of the *inhibiting* sense, where one cellular component decreases the expression of the other component [22]. Since an indirect interaction is composed of both promoting and inhibiting direct interactions, it is possible to determine the sense of an indirect interaction by using the following rules:

- An indirect interaction is of the promoting sense if it contains an even number of inhibiting direct interactions.
- An indirect interaction is of the inhibiting sense if it contains an odd number of inhibiting direct interactions.

In biological terms, the signal transduction network inferencing problem can be formulated in the following manner: Given a set of observations of promoting and inhibiting interactions between cellular components and the subset of the observations that are assumed to

be direct interactions, from the remaining set of unknown interactions infer which are direct and which are indirect. Any feasible solution to the STN inferencing problem should have the following two characteristics: The promoting/inhibiting sense of each indirect interaction in the solution should be explainable by the senses of a sequence of the inferred direct interactions, and the assumed set of direct interactions should be a subset of the inferred set of direct interactions. As stated previously, in this work we seek solutions to the STN inferencing problem which minimize the size of the inferred set of direct interactions.

2.1 Signal Transduction Networks as Weighted Directed Graphs

Any algorithmic method for automatically inferring signal transduction network structure needs to recast the problem as a computational problem by mapping biological concepts onto mathematical concepts. For the STN inferencing problem, the mathematical framework which is most useful for modeling the problem is computational graph theory. In this section, we describe how the STN inferencing problem is mapped onto a special type of transitive reduction problem in graph theory called the *binary transitive reduction*.

Consider a directed graph whose vertices represent cellular components and whose edges represent interactions. To capture promoting/inhibiting sense information about each interaction, we label edges in the graph with 0 if they represent promoting interactions and with 1 if they represent inhibiting interactions. The choice of using 0 and 1 for indicating the interaction sense is not arbitrary. According to the aforementioned rules for determining the sense of an indirect interaction, the modulo-2 sum of the edge labels of each direct interaction constituting the indirect interaction determines the sense of the indirect interaction. Finally, the subset of edges

representing the interactions which are assumed to be direct will be further labeled as “critical” edges. This simple model is sufficient for capturing all the information available to any algorithmic method for separating direct interactions from indirect interactions.

In order to completely formalize the STN inferencing problem mathematically, we introduce the following notations:

- $G=(V,E)$ is a directed graph whose vertex set *V* consists of cellular components and whose edge set *E* consists of both direct and indirect interactions.
- Let $u \in V$ be any vertex. $outgoing(u) \subseteq E$ Denotes the set of directed edges with *u* as the tail, while $incoming(u) \subseteq E$ denotes the set of directed edges with *u* as the head.
- $w : E \to \{0,1\}$ is the edge-labeling function where *0* denotes a promoting interaction and *1* denotes an inhibiting interaction.
- $E_{critical} \subseteq E$ is the initial set of interactions that are assumed to be direct.
- The *parity* of a path *P* from vertex *u* to vertex *v* is $\sum_{e \in P} w(e) (mod\ 2)$. A path of parity *0* has *even* parity while a path of parity *1* has *odd* parity.
- $u \to_x v$ denotes an edge from vertex *u* to vertex *v* with edge labeling *x* and $u \Rightarrow_x v$ denotes a path from vertex *u* to vertex *v* with parity *x*.
- For a subset of edges $E' \subseteq E$, $reachable(E')$ is the set of all ordered triples (u,v,x) such that $u \Rightarrow_x v$ is a path in the restricted subgraph (V,E')

The binary transitive reduction problem is defined as follows:

Problem name: Binary Transitive Reduction (BTR)
Instance: A directed graph $G=(V,E)$ with an edge-labeling function

$w : E \rightarrow \{0,1\}$ and a set of critical edges $E_{critical} \subseteq E$

Valid solutions: A subgraph $G' = (V, E')$ where $E_{critical} \subseteq E' \subseteq E$ and $reachable(E') = reachable(E)$

Objective: Minimize $|E'|$

The binary transitive reduction problem is a generalization of the standard transitive reduction problem familiar to any student of graph theory. In the standard transitive reduction problem, we seek the smallest subgraph that maintains reachability relationships in the original graph. The binary transitive reduction problem is similar, but with two primary differences: first, standard transitive reduction doesn't contain any notion of path parity. Path parity in binary transitive reduction is a crucial part of the problem and generalizes the notion of reachability. Second, in the binary transitive reduction we impose that the solution contain some subgraph given as input to the problem, whereas the standard transitive reduction problem imposes no such requirement.

Consider the example in fig 2.1. In this case, the problem instance consists of 3 vertices and 3 edges. No edges are marked as critical edges. The reachability set of the problem instance is *{ (1,2,1), (1,3,0), (3,2,1) }*. The edge *(1,2)* is not present in the binary transitive reduction vertex *2* is *1*-reachable from vertex *1* by traversing edge *(1,3)*, then edge *(3, 2)*.

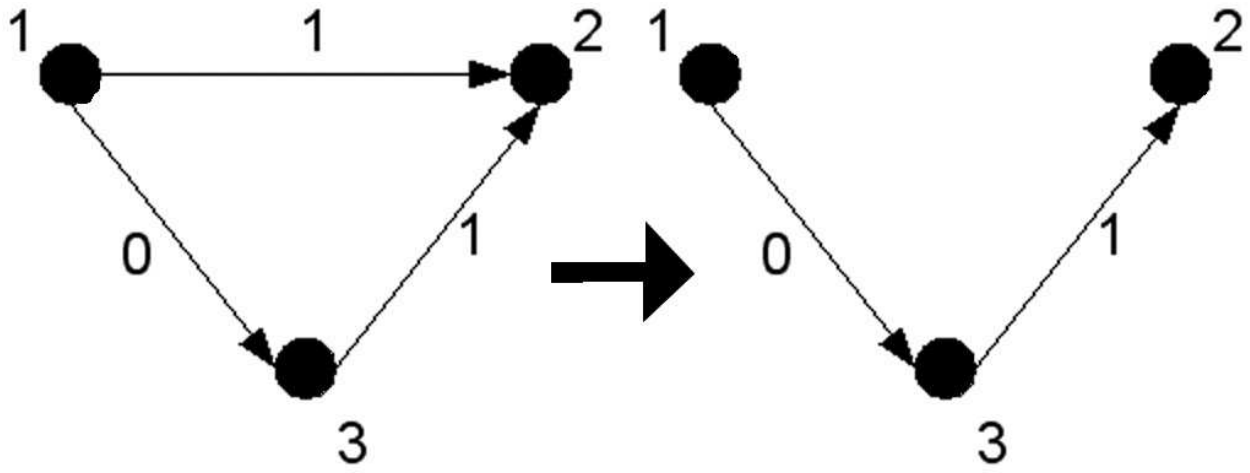

Figure 2.1. Minimum non-trivial instance. The problem instance is on the left, and the solution is on the right. Edge (1,2) can be eliminated since it doesn't contribute any additional reachability relationships to the graph.

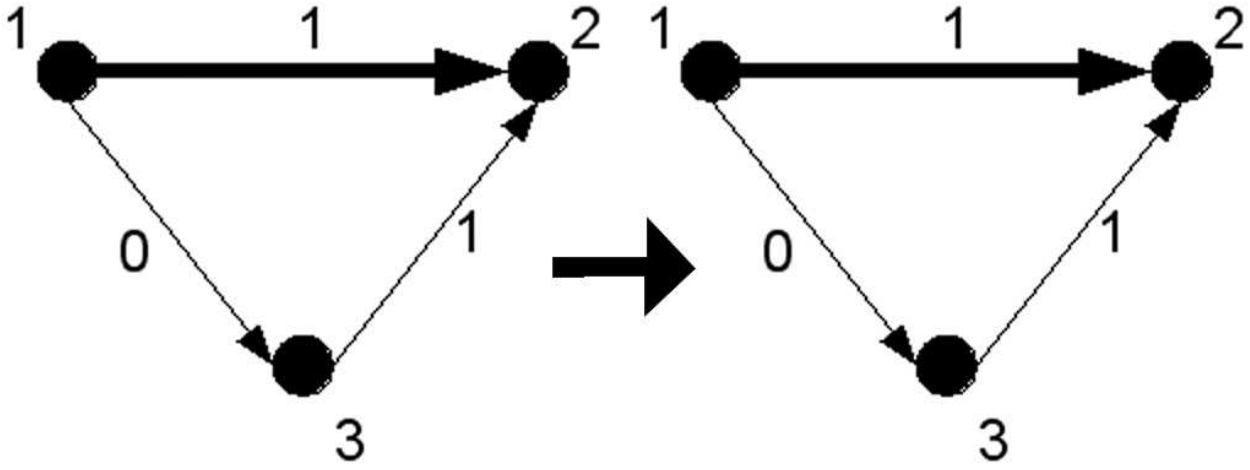

Figure 2.2. Instance with a critical edge. The problem instance is on the left, and the solution is on the right. Edge *(1,2)* is a critical edge. Even though edges *(1,3)* and *(3,2)* together satisfy the reachability relationship between vertices *1* and *2*, edge *(1,2)* is critical and therefore must be in binary transitive reduction.

This is the smallest example of a problem instance where the binary transitive reduction differs from the original problem instance. Figure 2.2 is similar to figure 2.1 with one exception: edge (1,2) is a critical edge. In this case, no edges can be eliminated from the graph without either breaking a reachability relationship or removing a critical edge.

2.2 Previous Work

Problems related to the transitive reduction and transitive closure of a graph are some of the most well-studied problems in computational graph theory. The paper of Aho et al. [3] presents a comprehensive treatment of the transitive reduction problem without considering edge parities. One of the primary contributions of this paper was showing the equivalence of the transitive reduction and transitive closure problems. Another important contribution from this work is a polynomial time algorithm for computing the transitive reduction of a directed acyclic graph. In section 2.4.1, the ideas of the Aho et al. are extended to give a polynomial time algorithm for finding the binary transitive reduction in the case where the problem instance is acyclic.

If all edges in the problem instance are labeled with 0 and there are no critical edges, then the binary transitive reduction problem reduces to the *minimum equivalent digraph* (MED) problem. The MED problem is known to be MAX-SNP Hard. The papers of Khuller et al. [16,17,18] presents an approximation algorithm for the MED problem with an approximation ration of $1.617+\varepsilon$ for any $\varepsilon > 0$. The algorithm of the Khuller et al. paper uses cycles to solve the MED problem for the strongly connected components of the problem instance, contracts the cycles to single vertices, and recurses. A similar idea is employed in section 2.4.2 for the binary transitive reduction problem in the case where the problem instance is strongly connected.

2.3 An Exact Solution to the BTR Problem using Mixed Integer Programming

An optimization problem is one where we seek to maximize or minimize a given objective function subject to a given set of constraints. If the objective is a linear function and all of the constraints are linear inequalities, then the problem becomes a *linear programming* problem. Linear programming is well-known in the field of operations research and there are a number of general techniques for solving linear programming problems, the simplex method being the most well-known among them. If we impose the further restriction that all of the unknown variables in the solution be integers, then the problem is known as an *integer linear programming* problem. If some variables in the solution must be integers, while others are free from the restriction, then the problem is called a *mixed integer programming* problem. Integer and mixed integer programming is useful in computer science and discrete mathematics because it allows for the reformulation of any NP-complete problem into an equivalent integer linear program.

It is interesting to note that, in general, a given NP-complete problem can be encoded into many different integer linear programs. For any given method for solving general mixed integer programs (such as the simplex method), different mixed integer programs for the same problem can have vastly different performance characteristics. In this section, we describe one way to encode the binary transitive reduction problems as a mixed integer program that makes use of the idea of a *flow network* between vertices in the problem instance. Although there are more efficient methods for encoding the binary transitive reduction into a mixed integer program, network-flow based mixed integer programs are the easiest to understand and implement.

The first difficulty encountered when encoding the binary transitive reduction problem as a mixed integer program is that some edges are labeled with *0*, while others are labeled with *1*. It

would simply things significantly if we could, via some transformation, remove the edge labellings from the problem. The following procedure constructs a new graph $G_1=(V_{1,}E_1)$ from the original graph $G=(V,E)$ which preserves the reachability relationships in the original graph *G* while simultaneously eliminating the need for edge labellings.

(1) Start with G_1 empty
(2) Add every vertex in *G* to G_1
(3) For each edge $e=u\rightarrow_1 v$ in *G*, add *e* to G_1
  (3a) If *e* is a critical edge, then mark *e* as a critical edge in G_1
(4) For each edge $e=u\rightarrow_0 v$ in *G*:
  (4a) Add a new vertex *w* to G_1
  (4b) Add edges $e_1=u\rightarrow_1 w$ and $e_2=w\rightarrow_1 v$ to G_1
  (4c) If *e* is a critical edge, then mark e_1 and e_2 as critical edges in G_1

After performing this procedure, every edge in G_1 has the same labeling, so we may simply disregard the edge labellings in G_1. To find the binary transitive reduction of *G*, we will simply calculate the binary transitive reduction of G_1 and map the results back onto *G*.

The first step when formulating any linear, integer, or mixed integer program is to specify the space in which our problem is embedded. Typically, this means to decide which variables the objective function and constraint inequalities are linear combinations of. For the binary transitive reduction problem, the problem space can be factored into two different sets of variables:

- For every $e\in E_1$ we introduce the *edge variable* $x_e\in\{0,1\}$ where $x_e=0$ indicates that edge *e* is a not member of the transitive reduction of G_1, while $x_e=1$ indicates that edge *e* is a member of the transitive reduction of G_1.
- For every $u,v\in V, e\in E_1$ we introduce variables, f_{uve}^{even} and f_{uve}^{odd}, both taking

values in the nonnegative real numbers, called the *even* and *odd flow variables*, respectively. Later, we will show how the flow variables are used to guarantee that feasible solutions of the mixed integer program have the same reachability as the original graph.

Note that, for the problem instance $G=(V,E)$, the solution space has $|V|^2\times|E_1|+|E_1|$ dimensions, of which $|E_1|$ dimensions are discrete, taking values in $\{0,1\}$ while the remaining $|V|^2\times|E_1|$ dimensions are continuous, taking any nonnegative real value. The mixed integer program which correctly solves the binary transitive reduction program is given below:

Objective: minimize $\sum_{e\in E_1} x_e$

Subject to: (1) $\forall\, e\in E_1$ and e is a critical edge, then $x_e=1$.

(2) $\forall\, e=u\to v\in E$, the following constraints hold:

(2a) $\sum_{x\in outgoing(u)} f_{uvx}^{even}=1$

(2b) if $w(e)=0$ then $\sum_{x\in incoming(v)} f_{uvx}^{even}-f_{uvx}^{odd}=-1$

otherwise if $w(e)=1$ then $\sum_{x\in incoming(v)} f_{uvx}^{even}-f_{uvx}^{odd}=1$

(2c) $\forall\, x\in V_1-\{u,v\}$, the following constraints hold:

$$\sum_{y\in incoming(x)} f_{uvy}^{even}-\sum_{y\in outgoing(x)} f_{uvy}^{odd}=0 \quad \text{and}$$

$$\sum_{y\in incoming(x)} f_{uvy}^{odd}-\sum_{y\in outgoing(x)} f_{uvy}^{even}=0$$

(3) $\forall\, e\in E_1, u,v\in V, \sum f_{uve}^{even}+f_{uve}^{odd}\le x_e$

The objective ensures that the solution will be the minimum subgraph that satisfies the constraints, and the first constraint in this mixed integer program ensures that the solution will contain every critical edge. The second set of constraints are used to ensure that the solution has the same reachability properties as the original graph and are derived from the concept of flow

networks from graph theory. In order to further explain how these “flow constraints” guarantee connectivity of the solution, it is useful to invoke the following analogy.

Imagine if each edge in the problem instance was a pipe through which water flows. Water is continually produced at a fixed rate at a single *source* vertex in the graph. Another vertex, known as the *sink*, continually consumes water at the same rate it is produced at the source. Every other vertex in the graph is a junction where pipes converge together or split off to other junctions. In order for such a system to sustain itself, the source vertex must be connected to the sink, otherwise water would accumulate in some part of the network, unable to drain down the sink. In particular, the existence of a flow between the source and the sink implies that the sink is reachable from the source.

In the binary transitive reduction problem, there are two different types of path parities: even and odd. For every edge in the problem instance, the flow constraints in the mixed integer program assume the existence of a flow network in the solution (an even or odd flow network, depending upon if the edge in the original problem instance was labeled with 0 or 1). Each flow variable represents a specific edge relative to some flow and its parity in the original problem instance. The first of the flow constraints, (2a), simply states that there is some positive flow coming from the source in the flow network. The second constraint, (2b), states that the flow is consumed at the sink in the flow network. The final constraint, (2c), states that the flow is conserved across every vertex that is not a source or a sink. Constraint (3) exists so that if a flow variable for some edge is used in some flow network, then that edge is in the solution. Although these constraints are sufficient for obtaining a solution for the BTR problem, in practice an implementation of the mixed integer program will likely use additional constraints and/or a modified objective function to assist with mapping the solution of the BTR problem on G_1 back

to the original problem instance *G*. The implementation details in chapter 4 describe how this can be done.

In a nutshell, each edge in the problem instance represents a reachability relationship that should be present in the solution, so for each edge in the problem instance, we assume the existence of a flow network in the solution and assign values to the flow variables for each edge, which in turn selects edges that must be present in the solution. There are many different ways we could do this, but the objective of the mixed integer program guarantees that we pick the solution with the minimum number of edges chosen. Finally, if we assume that the reachability relationship in question represents a direct interaction (i.e. The edge is a critical edge), we automatically include that edge in the solution.

2.4 An Approximation Algorithm for the Binary Transitive Reduction Problem

Since the binary transitive reduction problem is NP-complete, there isn't much hope of finding an exact method that scales well to large problem instances. While the mixed integer program presented in the previous section is performant for many types of problem instances, it suffers on highly connected instances where each vertex is reachable from most other vertices. For many NP-complete problems like the binary transitive reduction problem, approximation algorithms can be employed that trade exactness of the solution for speed. In this section, we describe an approximation algorithm for the binary transitive reduction problem. This method is a hybrid of two different methods for two different subclasses of problem instances. First, we describe an exact method for solving the BTR problem where the problem instance is acyclic. Next, we examine an approximation algorithm where the problem instance is a single, strongly

connected component. Finally, we combine the two methods to create an approximation algorithm for use in the general case.

2.4.1 An Exact Solution for Directed Acyclic Graphs

For the case where the problem instance is a directed acyclic graph (DAG), an optimal polynomial-time algorithm for computing the binary transitive reduction exists:

(1) **Topologically sort** the vertices into a sequence $v_1, v_2, ..., v_n$.
(2) **For** $i = n$ to 1
  (3) **For** $j = i$ to 1
    (4) **If** there is an edge $e = v_j \rightarrow_x v_i$ with edge labeling x
      (5) **If** there is a path $v_j \Rightarrow_x v_i$ that doesn't contain e
        (6) **Delete** edge e

This algorithm moves backwards topologically through the graph. For every pair of vertices in the graph, if there exists a reachability relationship (an edge) between the vertices that can be satisfied without using that edge itself, the edge is deleted from the graph. When the algorithm terminates, the edges that remain constitute the binary transitive reduction for the graph. This algorithm is a simple extension of the algorithm for finding the standard transitive reduction for a directed acyclic graph.

It is obvious that this algorithm only produces feasible solutions since an edge is deleted if and only if there is already a path between the two vertices of that edge. This characteristic of the algorithm guarantees that the reachability set of the reduced graph must be identical to that of the original graph. Second, the solution is optimal since every path in the graph corresponds to an element of the reachability set; the deletion of any additional edge will necessarily break paths

and delete elements from the reachability set of the reduced graph.

2.4.2 Approximating the Binary Transitive Reduction for Strongly Connected Components

A strongly connected component of a directed graph is a subgraph with the following property: for any pair of vertices in the subgraph, there is a path in the subgraph connecting them. It is well known that every directed graph can be partitioned into a set of strongly connected components. Another important result from the study of strongly connected components shows us that if one replaces each strongly connected component of a directed graph *G* with a single vertex, the result is a directed acyclic graph known as the *component graph* of *G*.

As was mentioned previously, if the BTR problem instance has all edge labellings 0 and consists of a single strongly connected component, then the problem reduces to finding a Hamiltonian cycle in the graph, a problem for which approximation algorithms already exist. In this section we show how to build an approximation algorithm for the BTR problem by combining an algorithm that uses cycles to approximate the transitive reduction for strongly connected components with the exact algorithm of the previous section that is invoked on the component graph in order to unify the sub-solutions for each strongly connected component.

2.4.3 Single and Multiple Parity Components

Each strongly connected component of a BTR problem instance is either a single parity component or a multiple parity component. A *single parity component* has the property that for

any pair of vertices in the component, there is either a even parity or and odd parity path between them, while a *multiple parity component* has the property that for any pair of vertices in the component, there is both an even and an odd parity path that connect them. There is a simple way to determine if a strongly connected component is single or multiple parity: if the component contains any simple cycle of odd parity, then the entire component is multiple parity, since one may produce paths of even or odd parity between any two vertices simply by traversing the odd cycle the appropriate number of times.

Determining if a strongly connected component contains an odd cycle is possible by invoking a variation of the popular Floyd-Warshall algorithm for computing all-pairs shortest paths on a graph. The algorithm is identical to Floyd-Warshall, except all addition should be performed modulo-2. Using this method, it is possible to classify each strongly connected component in the graph as either single or multiple parity. This is necessary because later, as we shall see, single parity components and multiple parity components and multiple parity components are handled separately in the final approximation algorithm.

Figure 2.3 illustrates the difference between a single and a multiple parity component. The graph on the left is a single parity component. All paths between any two vertices in the graph have exactly the same parity. The graph on the right is an instance of a multiple parity component. Between any two vertices in that graph, it is possible to generate both a path of even parity and a path of odd parity, simply by traversing the cycle one additional time.

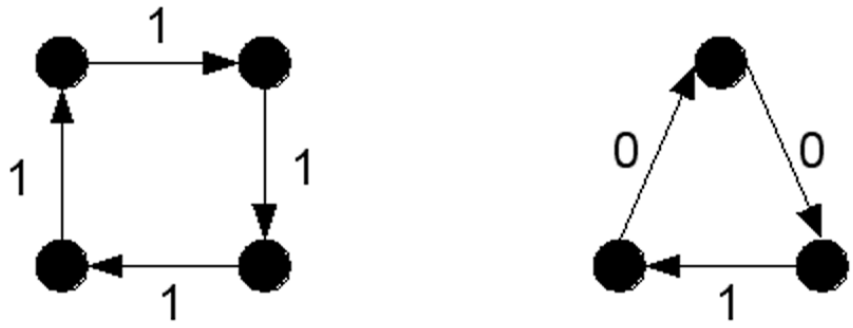

Figure 2.3. Single and multiple parity components. The graph on the left is a single parity component. The graph on the right has multiple parity. Note that in the graph on the right it is possible to generate paths of any parity between any two vertices simply by traversing the cycle an appropriate number of times.

2.4.4 The Cycle-Contraction Algorithm [16,17]

The cycle-contraction algorithm described in Khuller et al. is an approximation algorithm for the *minimum strongly connected spanning subgraph* (minimum SCSS) problem. The minimum SCSS is the MED problem restricted to the case where the problem instance is a single strongly connected component. Observe that the solution to the MED problem for a strongly connected component is a Hamiltonian cycle through the vertices in the problem instance. The Khuller et al. algorithm works by recursively searching for cycles in the graph and contracting the cycle to a single vertex. Each edge removed from the graph during the contraction phase is reported to be present in the solution.

When searching for cycles in the graph to contract, in order to avoid exponential search time, a maximum cycle length k is chosen beforehand. The algorithm searches for cycles of length k, then of length $k-1$, then $k-2$, etc. until the algorithm is finally searching for cycles of length 3. Another result of Khuller et al. is an exact algorithm when the problem instance contains no cycles of length greater than 3. When the cycle contraction algorithm finally descends to searching for cycles of length 3, it switches to the exact algorithm. The algorithm is

as follows:

CONTRACT-CYCLES$_k$(G):
for *i = k, k-1, k-2, ..., 3*
    while the graph contains a cycle with at least i edges
        Contract the edges on such a cycle
Invoke the exact algorithm CONTRACT-CYCLES$_3$ on the remaining graph
return the contracted edges

The details of the exact algorithm for CONTRACT-CYCLES$_3$ can be found in the Khuller et al. paper. Essentially, it selects a vertex at random from the graph and runs a depth-first search from that vertex. Every time it traverses an edge that completes a cycle in the explored subgraph longer than length 2, it adds that edge to a set of edges *S* known to be in the solution and deletes that edge from the graph. At the end of the procedure, the the set *S* taken together with every cycle of length 2 from the graph constitute the solution.

DasGupta et al. describes a way to generalize the cycle contraction algorithm of Khuller et al. to solve the BTR problem when the problem instance is a single strongly connected component. In order to use the cycle-contraction algorithm for the BTR problem, two obstacles need to be overcome. First, how should the cycle-contraction algorithm be extended to handle the case then the set of critical edges is non-empty? Second, how should handle multiple parity components be handled?

Handling critical edges is simple: for each critical edge $e = u \rightarrow_x v$ of parity *x*, delete the edge from the problem instance and replace it with a new vertex *w* and two new edges: $e_1 = u \rightarrow_0 w$ and $e_2 = w \rightarrow_x v$. Edges e_1 and e_2 will always be selected by the cycle-contraction algorithm, since by not selecting e_1 and e_2, either reachability to or from vertex *w* will violated, if not during the recursive phase of the algorithm then during the invocation of CONTRACT-

$CYCLES_3$.

Handling components of multiple parity is also easy. From our earlier discussion of single and multiple parity components, if a strongly connected component has at least 1 simple cycle of non-zero parity, then it is a multiple parity component. Let *C* be such a cycle. Note that if each edge in *C* is included with the output from the cycle-contraction algorithm, then its inclusion would yield paths of both even and odd parity between any pair of vertices in the component. Any odd cycle is sufficient for inclusion. The modified Floyd-Warshall algorithm discussed earlier identifies such cycles.

2.4.5 Combining the Algorithms

Suppose that we identify each of the strongly connected components in the graph. We can use the modified version of the cycle-contraction algorithm from the previous section to obtain approximate solutions for each component. The final question remains: how can we connect each of the approximate solutions together to obtain an approximate solution for the entire graph? One simple way of doing this would be to simply run the exact algorithm for directed acyclic graphs detailed in section 2.4.1 on the component graph and add those edges to the union of all of the edges obtained by solving each strongly connected component separately. However, a more sophisticated method exists.

The idea, first proposed by DasGupta et al, is to replace each strongly connected component with a directed acyclic graph in a manner that doesn't alter the reachability relationships between components. The replacements DAGs are known as *gadgets*. After replacement, the entire graph is acyclic, and the procedure of section 2.4.1 is invoked. This

approach requires that we have two more procedures: a method for constructing a gadget to replace a single parity component, and a method for constructing a gadget to replace a multiple parity component. Ironically, gadgets for replacing multiple parity components are actually easier to build than those for single parity components.

To replace a multiple parity component, simply collapse the entire component to a single vertex. For each incoming edge $e = u \rightarrow_0 v$ into the component, add a new edge $e_1 = u \rightarrow_1 v$ and for each incoming edge $e = u \rightarrow_1 v$ into the component, add a new edge $e_1 = u \rightarrow_0 v$ so that each incoming edge has a corresponding "brother" of opposite parity. Finally, we apply the same transformation to each outgoing edge in the component. Figure 2.4 illustrates the procedure.

Replacing single parity components with gadgets is more complicated. The gadget contains four vertices: one vertex v_{i0} for incoming paths of even parity, one vertex v_{i1} for incoming paths of odd parity, one vertex v_{o0} for outgoing paths of even parity, and one vertex v_{o1} for outgoing paths of odd parity. For every incoming edge into the component from vertex u with label 0, introduce edges $u \rightarrow_0 v_{i0}$ and $u \rightarrow_1 v_{i1}$. For every incoming edge into the component from vertex u with label 1, introduce edges $u \rightarrow_1 v_{i0}$ and $u \rightarrow_1 v_{i0}$. Outgoing edges are constructed in a similar manner: each outgoing edge from the original component becomes two outgoing edges in the gadget, one for each parity. The tails of each outgoing edge in the gadget are v_{o0} and v_{o1}. The set of internal edges in the gadget is simply $\{v_{i0} \rightarrow_0 v_{o0}, v_{i0} \rightarrow_1 v_{o1}, v_{i1} \rightarrow_0 v_{o1}, v_{i1} \rightarrow_1 v_{o0}\}$. This transformation is illustrated in figure 2.5.

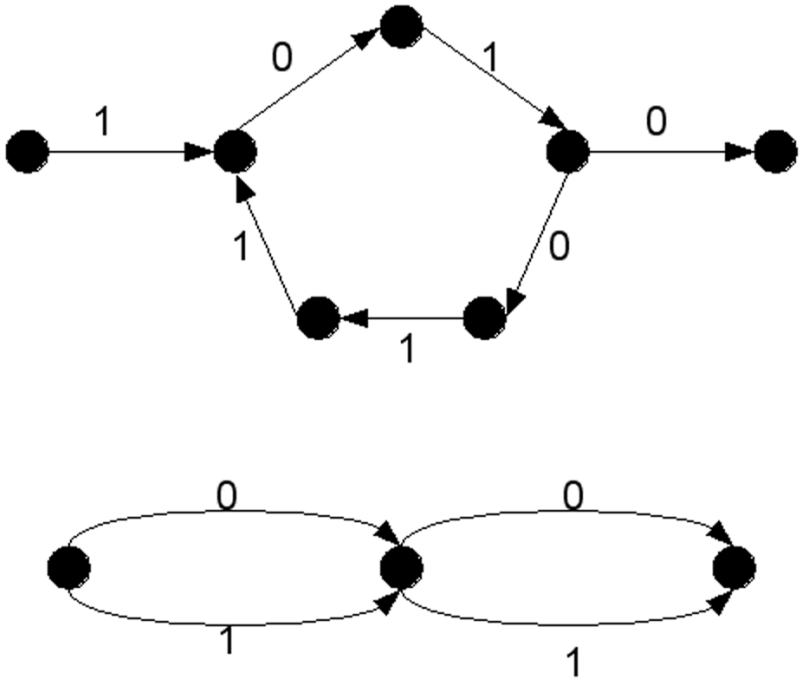

Figure 2.4. A gadget for multiple parity components. The multiple parity component on the top is replaced by the gadget on the bottom.

We now have all of the pieces needed to build the approximation algorithm. First, identify each strongly connected component in the graph and classify as either single parity or multiple parity. Then, solve each component separately using the modification of the cycle-contraction algorithm detailed in section 2.4.4. Finally, replace each component with an appropriate gadget and use the algorithm from section 2.4.1 to unify the solutions.

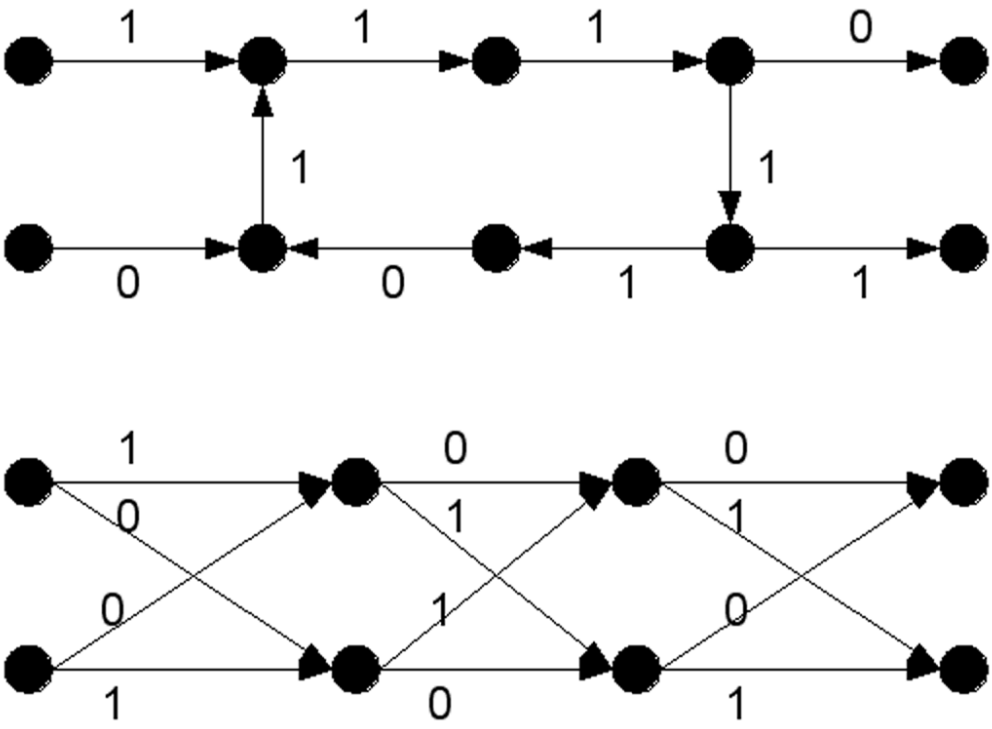

Figure 2.5. A gadget for single parity components. The single parity component on the top is replaced by the gadget on the bottom.

2.5 A Formulation of the Maximum Confidence Binary Transitive Reduction Problem

In the original STN inferencing problem, the idea of a critical edge was introduced to represent an interaction that the experimenter assumes to be direct based upon biochemical evidence. In reality, biochemical evidence doesn't imply with 100% certainty that a given interaction is definitely direct. The experimenter may have various levels of confidence that certain interactions are direct or indirect. An algorithmic method for inferring the structure of STNs would be more useful if it could take into account the experimenters level of confidence in the directness of each interaction. In this section, we present a generalization of the original problem formulation that takes levels of confidence into account.

In the maximum confidence transitive reduction problem, vertices still continue to

represent cellular components and edges continue to represent interactions. Interactions are still either of the promoting or inhibiting sense. Instead of marking some subset of edges as critical edges, to each edge we will assign a confidence measure that reflects the experimenters confidence that the given interaction is a direct interaction. Edges that previously were marked as critical are now assigned a high confidence measure, while other edges that were not previously marked as critical will have lower confidence measures.

Before we present the formal problem statement for the maximum confidence binary transitive reduction problem, it is necessary to introduce the following notations:

- A problem instance for the maximum confidence binary transitive reduction problem consists of a directed graph G=(V,E), an edge labeling $w : E \to \{0,1\}$ and a confidence measure.
- The *confidence measure* is a function $c : E \to [0,1]$ that assigns a confidence level to every edge of the problem instance. *c(e) = 1* indicates 100% confidence that the given edge represents a direct interaction.
- For any subgraph *G' = (V, E')* where $E' \subseteq E$, the *total confidence* c_t of *G'* is

$$c_t = \prod_{e \in E'} c(e)$$

The objective of the maximum confidence binary transitive reduction problem is the following: from all subgraphs *G'* of *G* whose binary transitive closure is equal to the binary transitive closure of *G*, identify the one that has maximum total confidence. This problem differs from the previous one in two respects: First, the objective doesn't explicitly attempt to minimize the number of edges in the solution. Second, we do not require that any specific edges be inside the solution, whereas in the previous problem, the critical edges were always present in the

solution. In the future, we will attempt to use the existing ideas for solving the standard binary transitive reduction problem to devise exact and approximate methods for solving the maximum confidence binary transitive reduction problem.

3 Inferring Haplotypes from Informative SNPs

The DNA of a diploid organism is organized into two sets of genetic structures called chromosomes. During sexual reproduction, one set of chromosomes is donated from the male gamete, while the other set comes from the female gamete. Each chromosome consists of a long chain of nucleotides, each nucleotide being composed of two linked nucleobases. The sequence of the nucleotides encode all of the information needed for manufacturing the proteins necessary for sustaining a living cell. Together, the information contained in the two sets of chromosomes are known as the organism's *genotype*, while the information in a single set of chromosomes is known as the organism's *haplotype*.

The vast majority of the DNA between two individuals in the same species is identical, that is, the same nucleobase-pairs appear in corresponding positions of the nucleotide sequence. However, a small number of positions in the nucleotide sequence exhibit variation across individuals in a species. These nucleotide positions are known as *single nucleotide polymorphisms*, or SNPs. SNPs are responsible for genetic differences between individuals in a given species. Since all non-SNP nucleotide positions are identical in all organisms in a species, a unique genetic identifier for a particular organism can be constructed by using only SNPs alone. Further, it has been observed by scientists that virtually every known SNP occurs in exactly two different forms. The most frequently observed form of a SNP is known as the *wild type* of the SNP, while the less frequently observed form is known as the *mutant type* of that SNP.

Obtaining a genetic identification of an organism by sampling each of the SNPs for that organism's species is a process known as *genotyping*. After genotyping, a SNP whose specific value is known is called a *typed* SNP. Genotyping an organism involves inspecting SNP sites and

determining if the nucleotide at the site is of the wild or mutant type. In disease association studies, the DNA of individuals from a healthy population and a sick population is sampled and the contrasting haplotype structure between the two populations is served as evidence for the genetic basis of the disease. The statistical significance of the study is a function of the size of the sample population, but the total cost of the study is a function of the number of SNPs typed. Often, many of the SNPs in an organism are highly correlated with each other. For example, simultaneous mutations at two different SNPs may be fatal for the organism, and since we never observe that combination of mutations in living organisms, the frequency of non-fatal combinations is higher.

In practice, it is experimentally difficult to determine the exact values of specific nucleotides on the individual haplotypes that comprise a single genotype. More often, only genotype data is available, that is, the experimenter can determine if a specific SNP site is homozygous-wild, heterozygous, or homozygous-mutant for an organism, but if the organism is heterozygous, it is unknown which particular haplotype contains the mutant nucleotide. Determining haplotypes from genotypes is an interesting problem unto itself, but it is not the subject of this thesis. In this work, we assume that reliable haplotype data is available, either through direct experimental observation, or via an algorithmic inferencing process on genotype data.

The objective of the haplotype prediction problem is to infer the values of all other SNPs in an organism based upon the values of an informative subset of SNPs. The informative SNPs are known as *tag* SNPs. Once tag SNPs are identified for a given species, genotyping a member of that species will be possible by merely inspecting the values of the tag SNPs and predicting the values of the remaining SNPs based upon the values of the tag SNPs. Inferring haplotypes

actually involves two related, but distinct problems. The first problem is determining which set of SNPs to use as tag SNPs. Ideally, the experimenter will want to choose the set of SNPs that yields the highest accuracy when predicting the values of the other SNPs. The second problem is how to predict the values of the non-tag SNPs once the tag SNPs have been identified. Although we will briefly discuss methods for solving th former problem, it is the latter problem which is the main focus of this chapter.

3.1 Mathematically Modeling Haplotype Populations as Binary Matrices

Since SNPs occur in a sequence and the nucleotides only occur in one of two different forms, it is possible to model a haplotype as a vector of binary digits. Each position in the bit vector corresponds to a particular SNP on a chromosome in the genome. If the bit at a given position in the vector is 0, then the SNP at the corresponding location on the chromosome is of the wild type, while a 1 in the same position in the vector corresponds to the mutant type. A population of organisms can then be represented as a binary matrix; each haplotype in the population occupies a single row in the matrix, while corresponding SNPs share the same column in the matrix.

In order to give a mathematically rigorous formulation to the haplotype prediction problem, we must first introduce the following conventions and notations:

- The haplotype population sample *S* consists of *m* haplotypes sampled across *n* SNPs.
- $S_{ij} \in \{0,1\}$ is the value of the jth SNP on the ith haplotype in the population. $S_{ij} = 0$ means that the SNP in question is of the wild type, while $S_{ij} = 1$ mean that the SNP is of the mutant type.

- u represents the unknown haplotype to be predicted. $u_j \in \{0,1\}$ is the value of the SNP at position j in the unknown haplotype.
- $k \leq n$ of the SNPs are designated as tag SNPs. Tag SNPs are denoted by t_1, t_2, ..., t_k.

Figure 3.1 shows an example of 5 haplotypes typed across 11 nucleotide positions. Note that only 3 of the 11 nucleotide positions exhibit variation, these are the SNPs. Further, notice that each SNP comes in two forms: the more frequent form (the wild type) and the less frequent form (the mutant type). Each nucleotide sequence is re-encoded into a 0-1 SNP sequence in the following manner: first eliminate all of the nucleotide positions that don't exhibit variation (i.e. the ones that are not SNPs). Then for a given SNP, if the haplotype in question exhibits the wild type, we assign 0. Otherwise, we assign 1.

Figure 3.2 shows a sample of 8 haplotypes typed across 13 SNPs. SNPs in positions 3, 6, and 8 are designated as tag SNPs. In this example, no better choice for tag SNPs can be made, since the values of tag SNPs can completely distinguish any two haplotypes. In practice, however, it may not always be possible to choose tag SNPs in a way that completely distinguishes any pair of haplotypes in the sample, either due to the haplotypic structure inherent to the sample, or because the cost of typing the sample for so many tag SNPs exceeds the available resources.

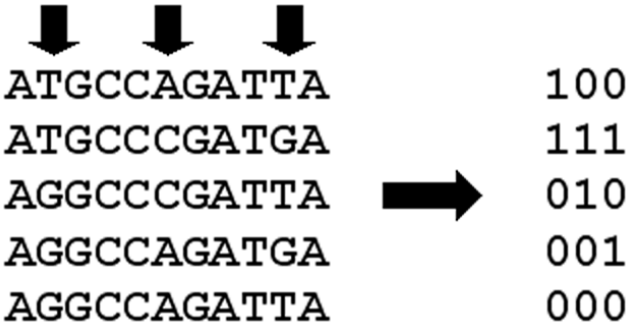

Figure 3.1. Mapping nucleotide sequences onto SNP sequences. 5 nucleotide sequences of length 11 are encoded into bit vectors of length 3. The black arrows on top of the nucleotide sequences indicate the positions of the SNPs.

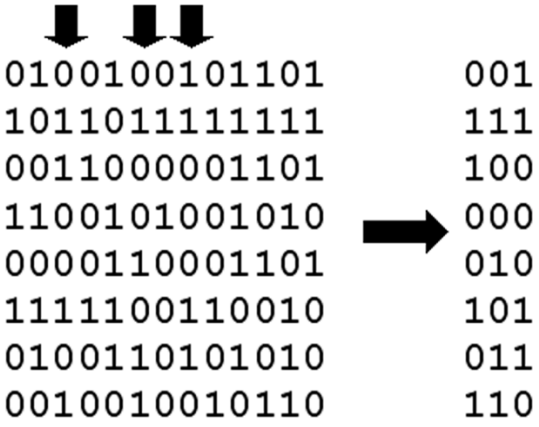

Figure 3.2. Selecting tag SNPs from a sample population. A sample of 8 haplotypes consisting of 13 SNPs. 3 of the SNPs are designated as tag SNPs. In this case, these 3 tags make a particularly good choice since they completely distinguish any two haplotypes, as evidenced by the projection of the sample onto the tags on the right.

The Haplotype Prediction Problem can be formulated as follows: Given a sample of a population of haplotypes S over n SNPs, as well as the positions of k tag SNPs, reconstruct an entire unknown haplotype using only the values of the unknown haplotype's tag SNPs. In this work, we introduce two variants of a linear algebra-based method for tag SNP selection and corresponding methods for inferring non-typed SNPs from tag SNPs. Both methods infer haplotypes by embedding the population sample in a metric space. One variant uses the mathematics of continuous Euclidean space, while the other uses the mathematics of finite fields.

3.2 Previous Work in Haplotype Prediction and tag SNP Selection

For the problem of tag SNP selection, methods for solving the problem fall into two different camps: *block-based* methods and methods that ignore block structure [5,6,15]. The term *linkage disequilibrium* refers to the phenomenon that SNPs that are closer together tend to be highly correlated; A sequence of SNPs can effectively be partitioned into blocks of low diversity. Then, tag SNPs are selected for the each block to predict the other non-typed SNPs inside the block. Clark et al. [7] contains a discussion of the benefits and limitations of block-based methods. On the other hand, *block-free* methods that ignore block structure have the capability of allowing tag SNPs to infer the values of non-typed SNPs that are located far away in the genome [2].

The problem of predicting haplotypes based on upon only tag SNPs has received less attention. The paper of Zhang et al. [23] reconstructs haplotypes using a variation of the partition-ligation-expectation-maximization algorithm. The block-free method of Halldorsson et al. [12] considers a graph whose vertices represent SNPs; the presence of edge indicates that one SNP can be used to reliably predict the other and the SNPs whose representative vertices with high degree are chosen as tags. To infer a haplotype, the neighbor's of each vertex representing non-typed SNPs are inspected and a majority vote is taken. The Halldorsson et al. method is tested with leave-one-out cross-validation and can recover 90% of the haplotype data using only 20% of SNPs as tags. Halperin et al. [13] describes a method for haplotype inference and then uses it to develop a method for tag SNP selection. Non-typed SNPs are predicted by inspecting the two closest tag SNPs on both sides; the value of the unknown SNP is given by majority vote. Dynamic programming is used to select tags that yield the best prediction score. The Halperin et al. method is able to infer haplotypes with 80% accuracy on the SNPs in the Daly et al. [8] data

set using only two SNPs as tags.

3.3 A Greedy Framework for Prediction-Based Tag Selection

As stated earlier, any haplotype prediction algorithm A_k accepts as its input the values of k tags $t_1, t_2, ..., t_k$ of an unknown haplotype u along with the known sample S in which every SNP in the sample is typed. The output of A_k is the reconstruction of x, that is, A_k infers the values of each of the non-tag SNPs in u. In this section, we show how any haplotype prediction algorithm can give rise to a corresponding tag SNP selection algorithm.

A fundamental assumption of haplotype prediction is that any method that can reliably infer unknown haplotypes will also reliably infer the known haplotypes in the sample population S. Assuming the truth of this "self-similarity" postulate, we can derive a tag SNP selection algorithm from any haplotype prediction algorithm A_k by adopting the following procedure: For every k-tuple of tag SNPs, choose the k-tuple with the minimal number of errors when predicting the non-tag SNPs in the sampled population. Although the SNPs in the sample population are completely typed, a haplotype prediction algorithm can still make errors because the number of SNPs may be not sufficient to distinguish any two haplotypes in the sample population. The procedure described above is the brute-force solution to the following problem:

> *Tag SNP Selection Problem*: Given a prediction algorithm A_k and a sample S, find k tags such that the prediction error e of A_k averaged over all of the SNPs in S (including the tag SNPs) is minimized.

Obviously, testing each and every subset of candidate tags of size *k* becomes prohibitively expensive as *k* becomes large. Instead of testing each subset, we propose the following Greedy Tag Selection Algorithm (GTSA) for tag selection using any arbitrary prediction algorithm:

GTSA(A_k, S)
Let $T = \emptyset$
for *i = 1, 2, ..., k*
    Let $R_{max} = \infty$
    for *j = 1, 2, ..., n* and $j \notin T$
        R_j = the error rate from using A_k to predict each haplotype *h* in *S* using tags in *T*.
        if $R_j < R_{max}$
            $R_{max} = R_j$
            $t = j$
    $T = T \cup \{t\}$
return *T* as the set of tag positions

This algorithm works by first finding the best tag for predicting the values of the other SNPs in the sample S. Then, it finds the next best tag, along with the first that together predict the remaining SNPs in the sample. It continues greedily selecting tags in this manner until k tags have been chosen. It is possible that the set of tags chosen by this algorithm is not optimal; there could exist some other set of tags of size k that achieves higher prediction accuracy, but whose tags taken individually are poorly suited for prediction. However, in practice, the above algorithm is both computationally feasible as well as sufficiently accurate.

3.4 LASPA: The Linear-Algebraic SNP Prediction Algorithm

The matrix representation of the sample population strongly suggests that is might be fruitful to embed the sample population in a high-dimensional vector space. Each haplotype in

the sample population corresponds to a specific vector in this vector space; each SNP corresponds to a particular dimension. The set of tag SNPs becomes a subspace of this vector space, and the task of predicting a haplotype based solely upon its SNPs reduces to the task of predicting the coordinates of a vector based upon its projection onto the subspace defined by the tags SNPs.

The *Linear-Algebraic SNP Prediction Algorithm* (LASPA) [14] iteratively predicts the value of each non-typed SNP of the unknown haplotype separately. The following is a high-level overview of how the method works: On each iteration, it constructs two candidate vectors, one vector assumes the unknown SNP is of the wild type, the other assumes it is of the mutant type. Each vector is then projected onto the space spanned by the unknown haplotype restricted to the tags SNPS together with the sample population restricted to the tag SNPs. The distance between the candidate vector and its projection is computed and the candidate vector that is closest to its projection determines the predicted value for that SNP.

More formally, suppose we are attempting to predict the value of the *i*th non-tag SNP. Let U_T be the unknown haplotype restricted to the tag SNPs and let S_T represent the sample population restricted to the tag SNPs. S_i is the sample population restricted to the non-tag SNP we are predicting. Consider the following matrix:

$$\begin{bmatrix} U_T & x_i \\ S_T & S_i \end{bmatrix} = \begin{bmatrix} u_{t_1} & u_{t_2} & \cdots & u_{t_k} & x_i \\ S_{1t_1} & S_{1t_2} & \cdots & S_{1t_k} & S_{1i} \\ S_{2t_1} & S_{2t_2} & \cdots & S_{2t_k} & S_{2i} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ S_{mt_1} & S_{mt_2} & \cdots & S_{mt_k} & S_{mi} \end{bmatrix}$$

The vector $c_0 = \begin{bmatrix} 0 \\ S_i \end{bmatrix}$ and $c_1 = \begin{bmatrix} 1 \\ S_i \end{bmatrix}$ are the candidate vectors. We project each candidate

vector onto the space spanned by the columns of $M = \begin{bmatrix} U_T \\ S_T \end{bmatrix}$ by forming the projection matrix $P = M \cdot (M^T \dot{M})^{-1} \cdot M^T$ and then calculating $d_0 = |c_0 - P \cdot c_0|$ and $d_1 = |c_1 - P \cdot c_1|$ and if $d_0 \leq d_1$ then we predict x_i to be 0, otherwise we predict x_i to be 1. Note that the matrix P needs to be calculated only once for each haplotype that we predict.

The method just described has been implemented; the implementation details are in Chapter 4 and experimental results are in Chapter 5. In this method, all addition, subtraction, and multiplication is done using vectors in ordinary Euclidean space, and it is the variant for which we report results. In the next section, we discuss another (unimplemented) variant of the method where addition and multiplication are done in a *k*-dimensional vector space over the finite field GF(2).

3.5 Vector Spaces over GF(2)

One drawback of the algorithm in section 3.4 is that the projection of the candidate vectors onto the space spanned by the tag SNPs often falls outside of the set $\{0,1\}^m$. An alternative variation of the algorithm which keeps project computations inside $\{0,1\}^m$ is to perform all operations inside the finite field GF(2). Suppose that we take the algorithm of section 3.4 and we make the following substitutions:

- Replace ordinary addition with bitwise exclusive-or (XOR)
- Replace ordinary multiplication with bitwise AND

- Replace Euclidean distance with Hamming distance

The first two modifications ensure that the projection operation is closed inside of *m*-dimensional binary vector space. The third modification gives us an appropriate way to measure the distance between two binary vectors.

This variant of the linear algebraic method for haplotype prediction has not yet been implemented. Hopefully future research into linear algebraic methods for haplotype inference will allow for the comparison of this variant with the method described in the previous section.

4 Implementation Details

This chapter contains the implementation details of all of the software that was developed for exploring the two inference problems discussed in this thesis. Unless noted otherwise, all software was developed using version 1.5 of the Java programming language and tested on a PC running Microsoft Windows XP.

4.1 BTR – A Java Program that Solves the Binary Transitive Reduction Problem

BTR (pronounced as it is spelled: B-T-R) is a Java Program that solves the Binary Transitive Reduction Problem using the mixed integer program presented in section 2.3. The program is packaged as a JAR file and is invoked from the command line with two parameters. The first parameter is the name of a XML file that contains the problem instance. The second parameter is the name of a file to output the resulting transitive reduction to. Both the input and output BTR take the same format: an XML file that concisely describes a directed graph along with 0-1 edge labellings as well as a flag for each edge to indicate if that edge is critical or not. The following command invokes BTR with input coming from `input.xml` and output being written to `output.xml`:

```
% java -jar btr.jar input.xml output.xml
```

The schema for the XML file in which the problem instances are encoded is informally described as follows. The `<graph>` tag is the root tag of the XML that contains the problem instance. The `<graph>` tag contains zero or more `<edge>` tags. Each `<edge>` tag corresponds to a specific edge in the problem instance. Each `<edge>` tag has four mandatory attributes

- `source` – the id of the tail vertex for the edge

- `target` – the id of the head vertex for the edge
- `label` – the edge label/parity for the edge. Either `0` or `1`.
- `critical` – Either `true` or `false` depending on if the edge is a critical edge or not

Vertex Ids are positive integers and must be unique for each vertex in the problem instance. Ids were chosen to be positive integers for a variety of reasons. First, since both the mixed integer program as well as the approximation algorithm demand the creation of new vertices at runtime, the code should contain a procedure for creating new vertices with predictable ids. Also, having positive integers as vertex ids simplifies the implementation of the Floyd-Warshall algorithm, which is used for detecting and classifying strongly connected components.

Figure 4.1 is graphical representation of a sample instance of the BTR problem. Vertex ids are adjacent to their vertices and critical edges are marked in bold. The instance contains 21 vertices and 25 edges. Figure 4.2 shows this example translated into the XML format that the BTR program accepts.

The output format of the BTR program is the same format as the input. However, only edges that remain inside the binary transitive closure have <edge> tags present in the output. Choosing the input and output formats for the BTR code to be identical has the benefit that, as a smoke test, the output can be re-inputted into the program, and if the output from this “sanity check run” differs from the input, then we can infer the presence of a bug in the code.

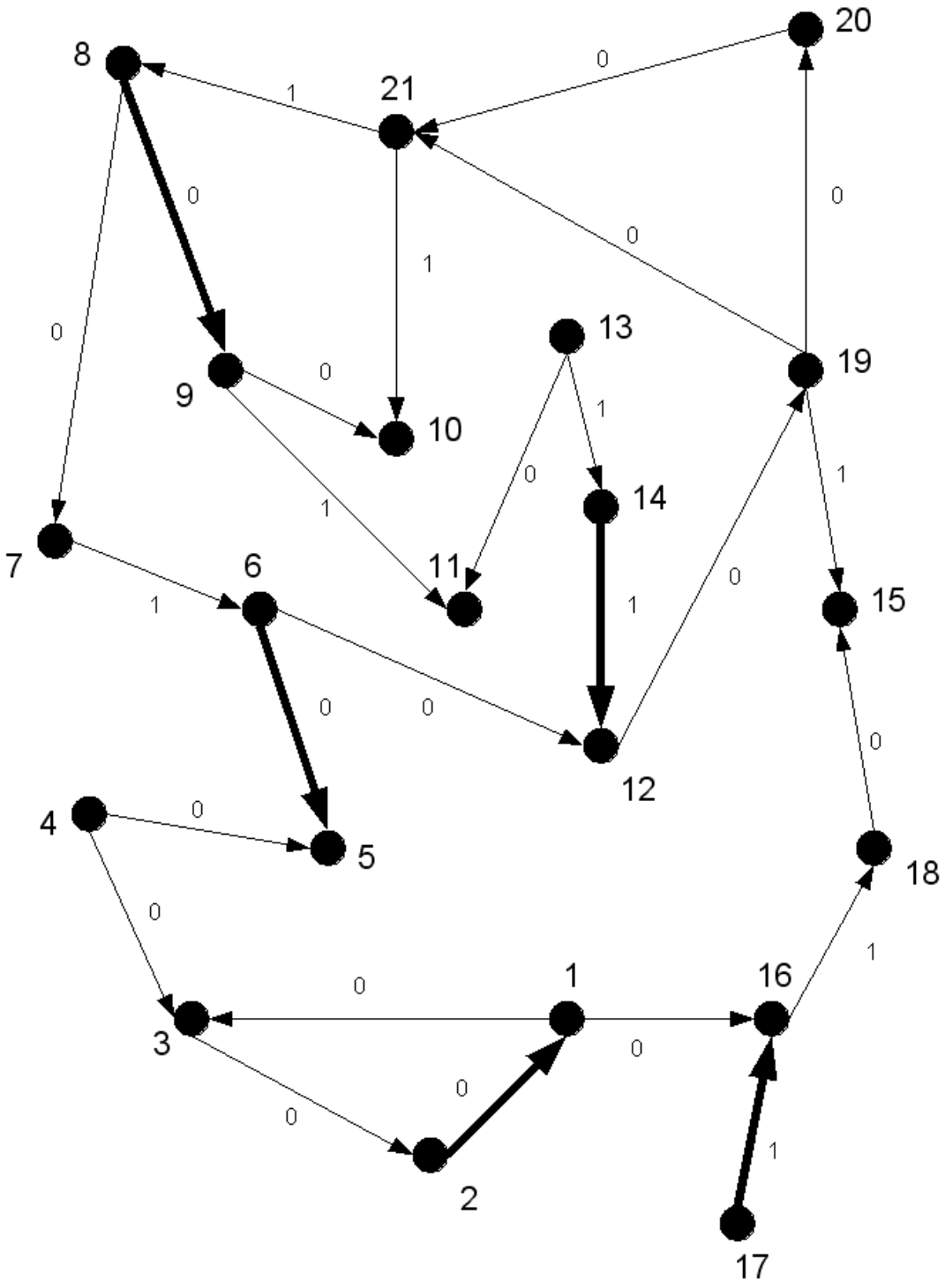

Figure 4.1 A sample BTR problem instance

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<graph>
  <edge source="1"  target="3"  label="0" critical="false"/>
  <edge source="1"  target="16" label="0" critical="false"/>
  <edge source="2"  target="1"  label="0" critical="true" />
  <edge source="3"  target="2"  label="0" critical="false"/>
  <edge source="4"  target="3"  label="0" critical="false"/>
  <edge source="4"  target="5"  label="0" critical="false"/>
  <edge source="6"  target="5"  label="0" critical="true" />
  <edge source="6"  target="12" label="0" critical="false"/>
  <edge source="7"  target="6"  label="1" critical="false"/>
  <edge source="8"  target="7"  label="0" critical="false"/>
  <edge source="8"  target="9"  label="0" critical="true" />
  <edge source="8"  target="10" label="0" critical="false"/>
  <edge source="9"  target="11" label="1" critical="false"/>
  <edge source="13" target="11" label="0" critical="false"/>
  <edge source="13" target="14" label="1" critical="false"/>
  <edge source="14" target="12" label="1" critical="true" />
  <edge source="16" target="18" label="1" critical="false"/>
  <edge source="17" target="16" label="1" critical="true" />
  <edge source="18" target="15" label="0" critical="false"/>
  <edge source="19" target="15" label="1" critical="false"/>
  <edge source="19" target="20" label="0" critical="false"/>
  <edge source="19" target="21" label="0" critical="false"/>
  <edge source="20" target="21" label="0" critical="false"/>
  <edge source="21" target="8"  label="1" critical="false"/>
  <edge source="21" target="10" label="1" critical="false"/>
</graph>
```

Figure 4.2. XML encoding of the problem instance

The BTR code doesn't actually solve the mixed integer program itself. Instead, it merely re-encodes the problem instance as a mixed integer program and then passes control to GLPK. GLPK, the GNU Linear Programming Kit, is a free, high-performance general-purpose linear program solver. After GLPK solves the mixed integer program instance, it hands control back over to BTR, which maps the solution of the mixed integer program back into the problem

domain and outputs the solution.

BTR binds to GLPK via a Java-to-native code adapter library. During the testing of BTR, this adapter library was found to be rather buggy. Unfortunately, once the problem instance scales beyond several tens of vertices or edges, the adapter library fails in unpredictable ways, so it is difficult to judge the true performance of the mixed integer program. Hopefully future research will have a more robust GLPK Java binding available for testing larger problem instances. Another way to overcome this limitation is to use a different linear program solver, such as ILOG CPLEX.

The Java code for BTR is organized into three packages:

- `btr.model` contains classes to model a problem instance
- `btr.io` contains classes to load and save problem instances as XML
- `btr.impl` contains the classes which build the GLPK mixed integer program from the problem instance.
- The main entry point into the program is in a class `btr.Main`

Figure 4.3 shows the interfaces in the `btr.model` package. The problem instance model in the BTR program is designed so that graph-mutation operations, such as adding or removing vertices and edges, would be as computationally cheap as possible. Another objective of the implementation was to allow for reuse of the object model for the problem instance later when implementing the approximation algorithm for the problem. Each interface has a corresponding implementation class.

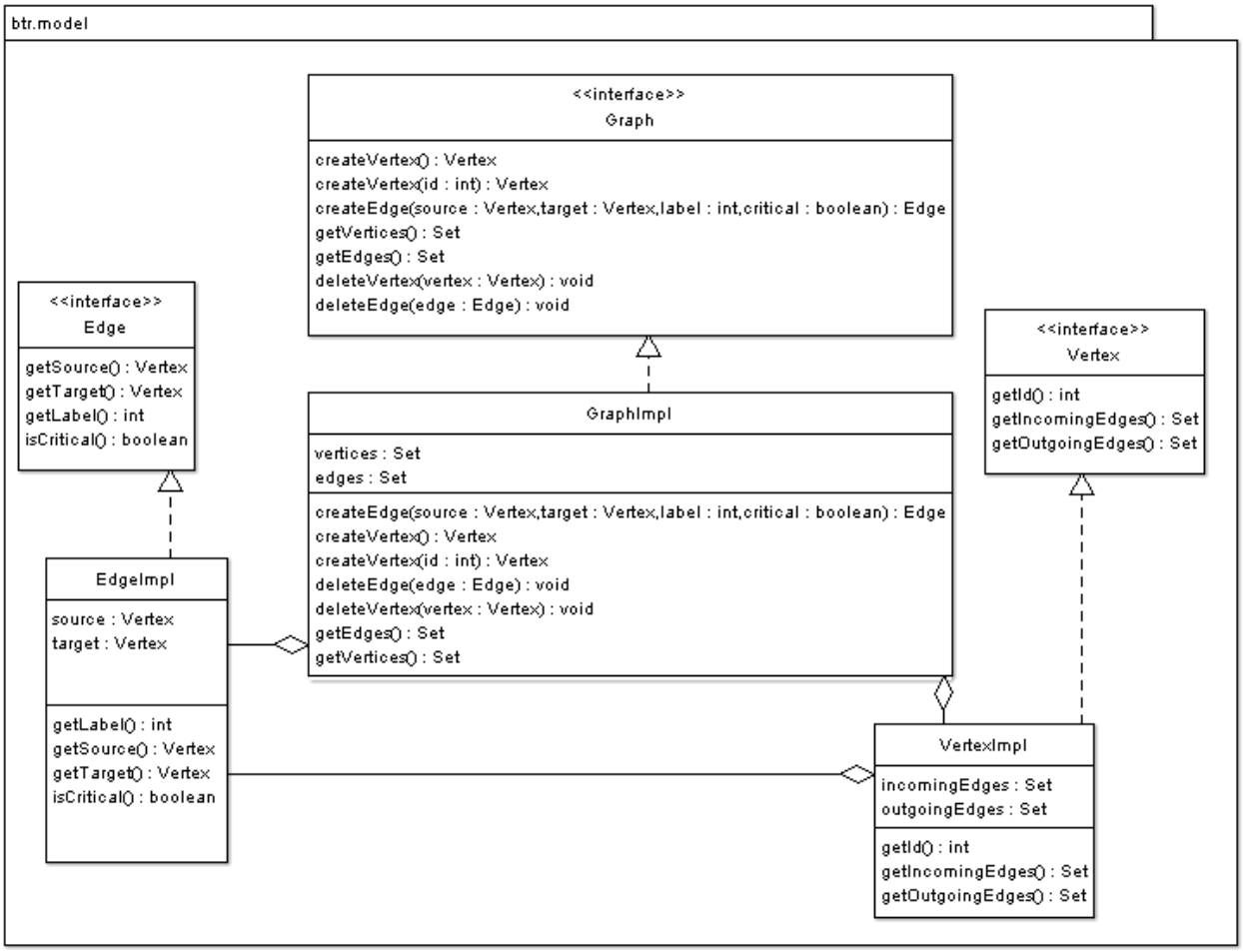

Figure 4.3 The classes and interfaces in the `btr.model` package

The `btr.io` package, shown in Figure 4.4, contains two classes: `GraphInputStream` and `GraphOutputStream`. Both classes employ the standard subclassing pattern encouraged by the design of Java IO. The classes are responsible for marshaling and unmarshaling the object model of the problem instance to and from XML. XML processing is done using XOM, an open-source, easy-to-use XML library for Java.

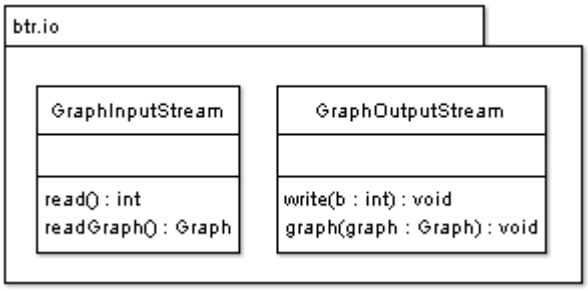

Figure 4.4 The classes and interfaces in the `btr.io` package

Figure 4.5 shows the btr.impl package. This package contains the code necessary to solve the problem by translating the problem instance into the mixed integer program and invoking the linear program solver. It is organized into an interface and an implementation class. The implementation class is the only place in the code that has a strong dependency on GLPK; switching to a different linear program solver such as ILOG CPLEX is as simple as writing a new implementation of the interface.

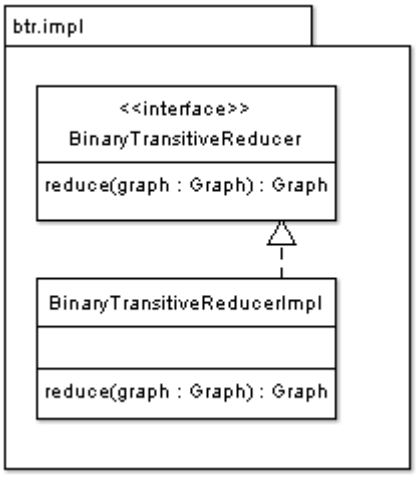

Figure 4.5 The classes and interfaces in the `btr.impl` package

4.2 LAHP – A Java Program that implements the LASPA Algorithm

LAHP, or Linear Algebraic Haplotype Predictor, is a Java Program that implements the LASPA algorithm of section 3.4. The program is packaged as a JAR file and accepts four command line parameters. The first parameter is the name of a file containing the sample population. The second parameter is the name of a file containing the indices of the tag SNPs. The third parameter is the name of a file containing the only the values of the tag SNPs of the set of haplotypes to be predicted. The final parameter is the name of the output file; at the end of execution it will contain the fully predicted haplotypes from the third file. The following command invokes LAHP using the sample population contained in `sample.txt`, the tags in `tags.txt`, the unknown haplotypes in `unknowns.txt` and writes the predicted haplotypes to `predicted.txt`:

```
% java -jar lahp.jar sample.txt tags.txt unknowns.txt predicted.txt
```

The format of the first, third, and forth files are the same, only the second file differs in the structure of its contents. The first, third, and fourth files all contain newline-delimited strings consisting of 0s and 1s. The second file contains a single line of comma-delimited positive integers, each the index of some tag SNP. The need not occur in any specific order, and counting begins with 1, not 0. Figure 4.6 contains three sample input files. In this example, the sample consists of 10 haplotypes typed across 10 SNPs. The SNPs in positions 2, 9, and 10 are designated as tag SNPs and we are predicting 3 haplotypes. The projection matrix needs to be recomputed 3 times, once for each haplotype we are predicting. For each haplotype, we are predicting each of the 10 – 3 = 7 non-typed SNPs separately.

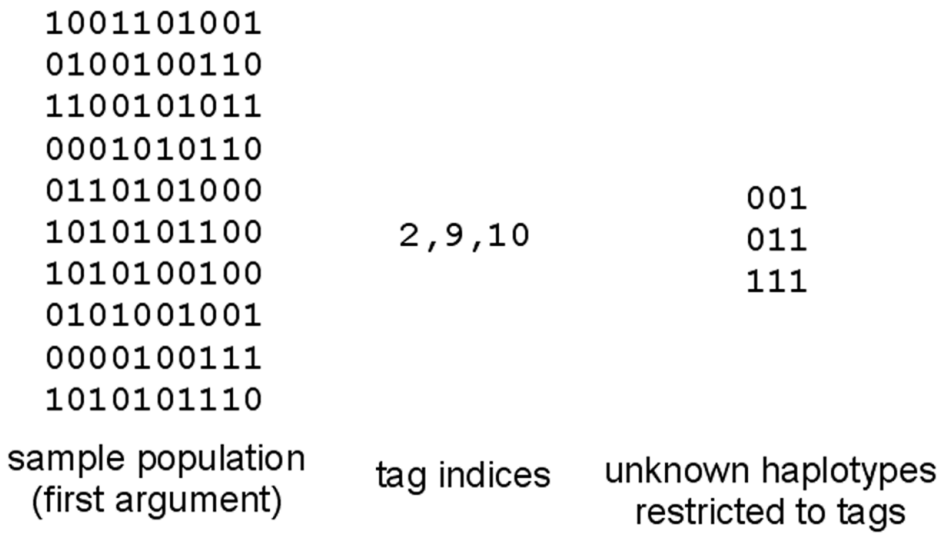

Figure 4.6 Sample input for LAHP

The LAHP code is organized very similar to the way BTR was organized. There are three packages:

- `lahp.model` contains the interfaces and classes necessary to represent large boolean matrices and lists of tags.
- `lahp.io` contains code to read and write binary matrices and tag lists to and from disk.
- `lahp.impl` contains the implementation of the linear algebraic method for haplotype prediction.
- The entry point into the program is in the class `lahp.Main`

Figure 4.7 is a package diagram of the `lahp.model` package. The package consists of a single `Matrix` interface, and a class `MatrixImpl` that implements the interface. The class contains a number of common linear-algebraic operations, such as methods for elementary row operations, which are composed together to create the final algorithm. Many of the methods, including the `concatenate*()` and `submatrix()` methods, are implemented using anonymous

inner proxy classes to avoid the overhead of having to make copies of the original matrix in memory. This optimization increased the performance rather dramatically.

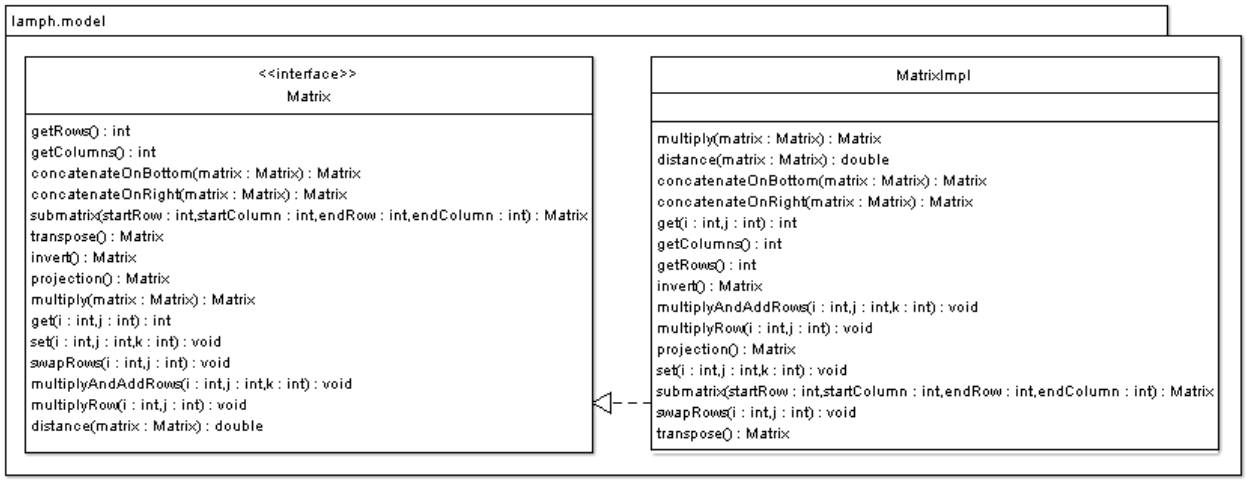

Figure 4.7 The classes and interfaces in the `lamph.model` package

The `lahp.io` package contains three classes: `MatrixReader`, `MatrixWriter`, and `TagReader`. `MatrixReader` is used to read the first and third parameters into `Matrix` data structures. `TagReader` reads the file in the second parameter into a `List` of `Integers`. `MatrixWriter` is responsible for creating the file specified by the forth parameter. Like the IO classes in the BTR program, `lahp.io` classes conform to the standard Java subclassing paradigm of Java IO. Figure 4.8 is diagram containing the classes in the `lahp.io` package.

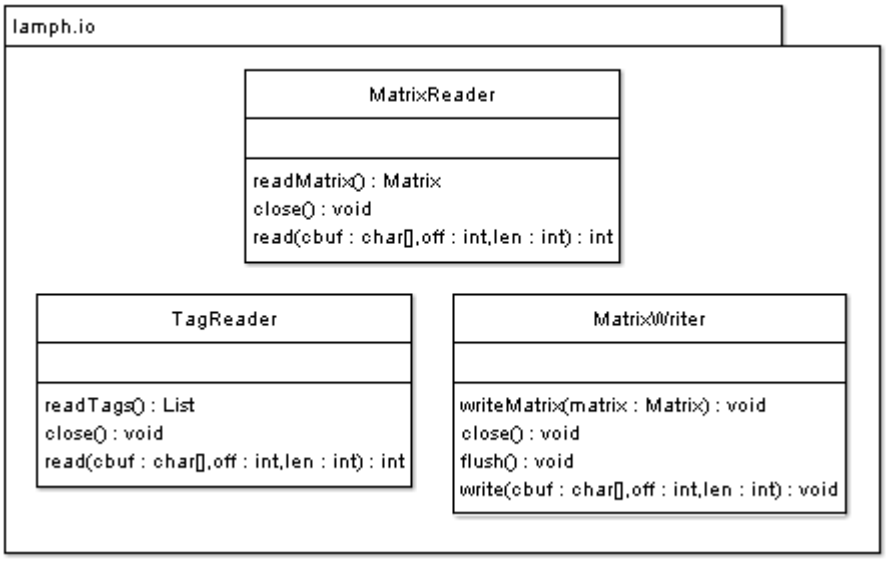

Figure 4.8 The classes and interfaces in the `lahp.io` package

The `lahp.impl` package, illustrated in figure 4.9, contains the actual algorithm implementations for both tag selection and haplotype prediction. The interface `TagSelectionAlgorithm` and its corresponding implementation class selects tag SNPs from the sample according to the procedure outlined in section 3.3. The interface `PredictionAlgorithm` is a generic interface that any type of prediction algorithm implementation can conform to. In this case, there is only 1 implementation class: `LinearAlgebraicMethod`.

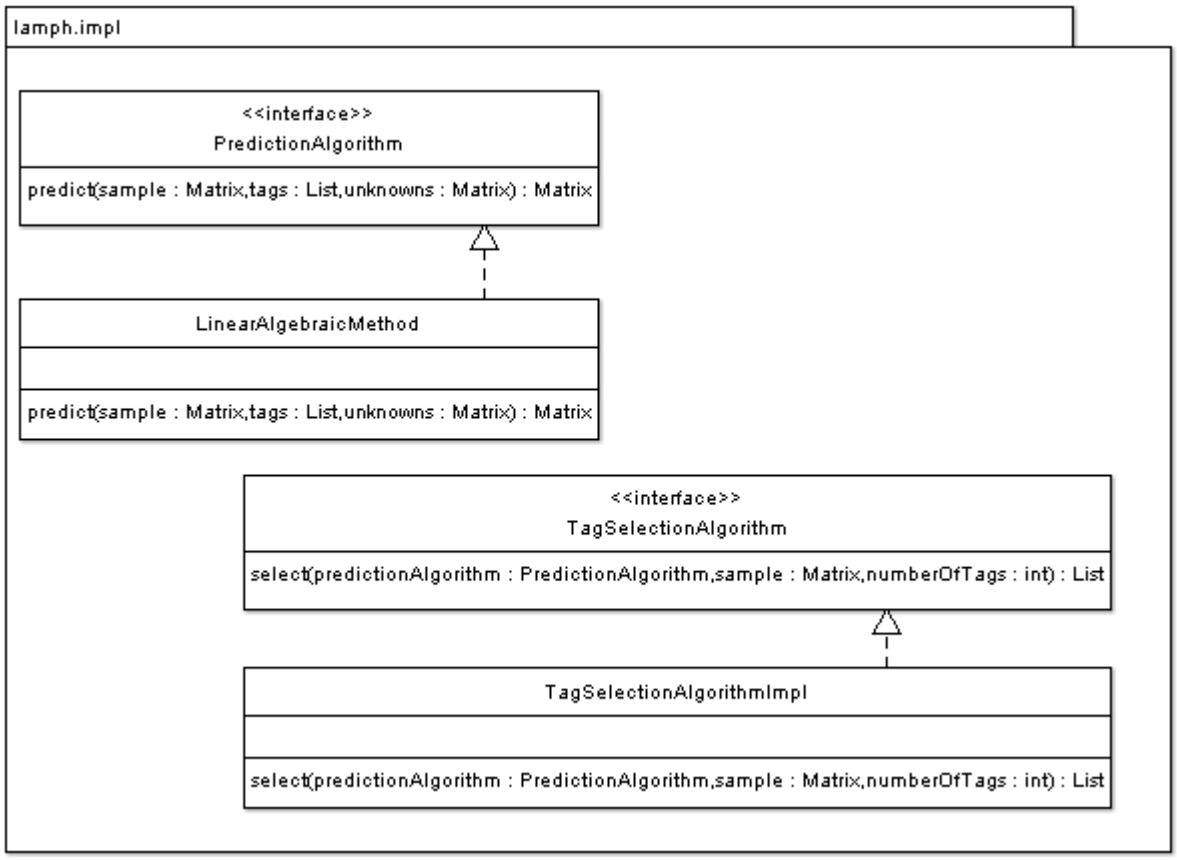

Figure 4.9 The classes and interfaces in the `lamph.impl` package

5 Results

5.1 Results for the Binary Transitive Reduction Problem using BTR

The BTR program was tested using several interesting and representative instances. The test instances were manually constructed as to verify that the program output matched the expected output. Figure 5.1 shows 8 example instances with which BTR was tested.

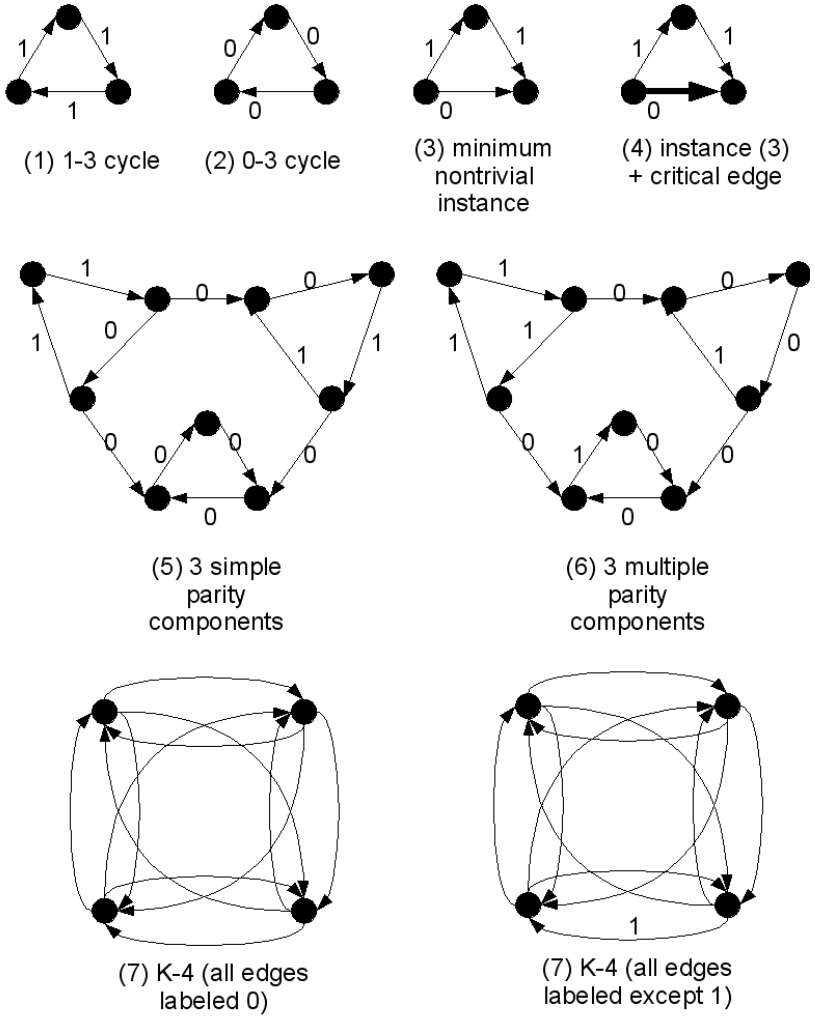

Figure 5.1 BTR test instances

Each example test instance from the figure is described below:

- Test instance 1 consists of three vertices connected in a cycle, each edge having odd parity and no critical edges. In this case, the solution is identical to the instance.
- Test instance 2 is identical to test instance 1, except in this case, all edge parities are even instead of odd. In this case, BTR needs to split each edge into two edges of odd parity before solving; the solution is identical to the instance.
- Test instance 3 is an example of a minimal non-trivial instance. In this case, the bottom edge is dropped from the graph; only the top two edges remain in the solution.
- Test instance 4 is identical to test instance 3, except this time the edge that would normally be dropped is marked as critical. The solution is identical to the instance.
- Test instance 5 consists of three strongly connected components, each with single parity. In this case, the bottom, leftmost edge can be dropped from the graph.
- Test instance 6 is similar to instance 5, except this time each strongly connected component has multiple parity. Like the previous instance, the bottom, leftmost edge can be dropped from the graph.
- Test instance 7 is a maximally-connected graph with 4 vertices; each edge is labeled with 0, making the graph a single parity component. In this case, BTR should identify a Hamiltonian cycle in the graph.
- Test instance 8 is similar to test instance 7, except 1 edge is of odd parity, making the entire graph a multiple parity component. Like the previous test case, BTR should identify a Hamiltonian cycle in the graph.

In addition to these test instances, a instance consisting or real data was available. The author would like to thank Bhaskar DasGupta of the University of Illinois at Chicago for

contributing the data to this project. This instance consists of 49 vertices connected by 68 edges (54 even edges, 14 odd edges, 16 critical edges). After processing the data through BTR, it was found that 55 edges remained inside the binary transitive closure; 13 edges were redundant. Appendix A contains the data for this instance, along with the solution.

5.2 Results for the Haplotype Prediction problem using LAHP

In contrast to the binary transitive reduction problem, significantly more real data is available for testing algorithms for the haplotype prediction problem. Each of these following datasets were used in previous research by Halperin et al. and Halldorsson et al., so we have also used the same datasets in order to conduct a meaningful comparison study. As stated previously, our algorithm accepts haplotype data as input, but the available data is genotype data, so following Halperin et al, we use GERBIL [19] to phase the genotypes into haplotypes and use the resulting haplotypes as test data sets. Each data set consisted of family trios. A family trio consists of a mother, father, and offspring. The following data sets were used:

Three ENCODE Regions from HapMap. Three Regions (ENm013, ENr112, ENr113) from 30 CEPH family trios are obtained from HapMap ENCODE Project [1]. These regions are derived from 500 KB regions of chromosomes 7q:21:13, 2p16:3 and 4q26. The number of typed SNPs in each region is 361, 412 and 515 respectively.

Two gene Regions form HapMap. Two gene regions STEAP and TRPM8 from 30 CEPH family trios are obtained from HapMap. We took the HapMap SNPs that are spanned by the gene plus an additional 10,000 kilobases upstream and downsteam from the site. The number of SNPs

genotyped in each gene region is 23 and 102 SNPs.

Chromosome 5q31. The data set collected by Daly et al. is derived from the 616 kilobase region of human Chromosome 5q31 that may contain a genetic variant responsible for Crohn's disease. This data set contains 103 SNPs for 129 trios.

LPL & Chromosome 21. The Clark et al.5 data set consists of the haplotypes of 71 individuals typed over 88 SNPs in the human lipoprotein lipase (LPL) gene. The Chromosome 21 data set consists the first 1,000 of 24,047 SNPs typed over 20 haploid copies of human Chromosome 21.

In order to simulate unknown haplotypes, we apply leave-one-out cross-validation. Leave-one-out cross validation works in the following manner: One by one, each haplotype in the data set is removed and acts as the unknown haplotype we wish to predict; the remaining haplotypes act as the sample population. The haplotype that was “left out" is reconstructed based only on its tag SNPs and the haplotypes in the sample. The average number of errors in the reconstruction of every left-out haplotype is used as a measure of the prediction accuracy of the method. Table 1 presents the results of leave-one-out experiments on the 6 datasets.

Datasets (Number of SNPs)	Prediction Accuracy %											
	80	85	90	91	92	93	94	95	96	97	98	99
ENm013 (360)	2	3	6	6	7	8	9	9	11	15	22	254
ENr112 (410)	6	9	14	16	18	20	24	33	63	95	126	187
ENr113 (514)	4	5	10	11	13	15	18	40	55	80	104	200

STEAP (22)	1	1	1	2	2	2	2	2	3	3	4	4
TRPM8 (101)	1	2	4	5	5	6	7	8	10	15	15	24
5q31 (103)	1	2	5	7	7	9	13	16	21	31	41	55

Table 1. Performance of GTSA/LASPA and LMTSA/LASPA.

Table 2 compares GTSA/LASPA with STAMPA and IdSelect. For each method, we present the minimum number of tag SNPs needed to reach 80% and 90% prediction accuracy in the leave-one-out tests. According to Halperin et al. IdSelect obtains the number of tag SNPs based on Pearson Correlation. Our experiments shows that GTSA/LASPA needs fewer tags than other methods. For example, for the ENr113 dataset our methods use half as many tags as STAMPA to obtain the same 80% prediction accuracy.

Data Set	80% Accuracy			90% Accuracy		
	GTSA/LASPA	STAMPA	IdSelect	GTSA/LASPA	STAMPA	IdSelect
ENm013	2	5	84	6	12	189
ENr112	6	9	97	14	17	169
ENr113	4	11	83	10	18	325
STEAP	1	2	20	1	2	22
TRPM8	1	3	38	4	6	35
5q31	1	2	64	5	6	91

Table 2. Comparison GTSA/LASPA with STAMPA and IdSelect.

In Table 3 we compare our GTSA/LASPA with STAMPA and HapBlock. The number of tag SNPs is determined according to Halperin et al. by using HapBlock with default parameters. In all of the data sets, GTSA/LASPA obtains better prediction accuracy. For small data sets, the GTSA/LASPA method is significantly faster than STAMPA; for larger data sets, the difference in running time is less significant.

Data Set	Number of tag SNPs	Prediction Accuracy			Running Times (seconds)		
		LASPA	STAMPA	HapBlock	LASPA	STAMPA	HapBlock
ENm013	15	0.971	0.929	0.759	100	78	8,710
ENr112	33	0.951	0.939	0.822	132	87	3,810
STEAP	3	0.985	0.951	0.763	0	3	5
TRPM8	12	0.966	0.942	0.811	6	34	140
5q31	17	0.954	0.949	0.889	22	179	17,311

Table 3. Comparison LASPA with STAMPA and HapBlock on Prediction and speed performance.

6 Conclusion

This thesis discussed two different inference problems in bioinformatics. The first problem was inferring the structure of signal transduction networks from observations on the interactions between pairs of cellular components. The problem was reduced to a type of generalized transitive reduction problem known as the binary transitive reduction. An exact algorithm using mixed integer programming was presented that successfully solved the problem and tested on real and simulated instances and an approximation algorithm for the problem originally suggested by DasGupta et al. was discussed. Finally, a formulation of the maximum-confidence binary transitive reduction problem was offered as a generalized version of the binary transitive reduction problem. Future research in this area should result in a successful working implementation of the approximation algorithm, with which we can compare with the exact results provided by the mixed integer program.

The second problem, inferring haplotypes from informative SNPs, was introduced along with a related problem, tag SNP selection. A framework for generating greedy tag SNP Selection algorithms based on any haplotype prediction algorithm was presented. The linear algebraic method for haplotype prediction was explained, and an implementation was developed and tested against real datasets. A variant of the method utilizing vector spaces over finites fields was introduced, but no results are available yet. Future research will hopefully develop an implementation of the GF(2) variant of the linear-algebraic method for haplotype prediction and a comparison with the Euclidean variant.

Bibliography

[1] http://www.hapmap.org

[2] H. Ackerman, S. Usen, R. Mott, A. Richardson, F. Sisay-Joof, P. Katundu, T. Taylor, R. Ward, M. Molyneux, M. Pinder, D. P. Kwiatkowski. (2003) `Haplotypic analysis of the TNF locus by association efficiency and entropy', *Genome Biology*, Vol.4, pp. 24.

[3] A. Aho, M. R. Garey and J. D. Ullman. The transitive reduction of a directed graph, SIAM Journal of Computing, 1 (2), pp. 131-137, 1972.

[4] B. Alberts. Molecular biology of the cell, New York: Garland Pub., 1994.

[5] H. I. Avi-Itzhak, X. Su, and F.M. de la Vega. (2003) `Selection of minimum subsets of single nucleotide polymorphism to capture haplotype block diversity', *Proceedings of Pacific Symposium on Biocomputing*, Vol. 8, pp. 466-477.

[6] C. S. Carlson, M. A. Eberle, M. J. Rieder, Q. Yi, L. Kruglyak, and D. A. Nickerson. (2004) `Selecting a maximally informative set of single-nucleotide polymorphisms for association analyses using linkage disequilibrium', *American Journal of Human Genetics*, Vol. 74, No. 1, pp. 106-120.

[7] A. Clark. (2003) `Finding genes underlying risk of complex disease by linkage disequilibrium mapping', *Current Opinion in Genetics & Development*, Vol. 13, No. 3, pp. 296-302.

[8] M. Daly, J. Rioux, S. Schaffner, T. Hudson, and E. Lander. (2001) `High resolution haplotype structure in the human genome', *Nature Genetics*, Vol. 29, pp. 229-232.

[9] C. P. Fall, E. S. Marland, J. M. Wagner and J. J. Tyson. Computational Cell Biology, New York: Springer, 2002.

[10] L. Giot, J. S. Bader et al. A protein interaction map of Drosophila melanogaster, Science 302, 1727-1736, 2003.

[11] J. D. Han, N. Bertin et al. Evidence for dynamically organized modularity in the yeast protein-protein interaction network, Nature 430, 88-93, 2004.

[12] B. V. Halldorsson, V. Bafna, R. Lippert, R. Schwartz, F. M. de la Vega, A. G. Clark, and S. Istrail. (2004) `Optimal haplotype block-free selection of tagging SNPs for genome wide association studies', *Genome Research* Vol. 14, pp. 1633-1640.

[13] E. Halperin, G. Kimmel, and R. Shamir. (2005) `Tag SNP Selection in Genotype Data for Maximizing SNP Prediction Accuracy', *Bioinformatics* 21:i195-i203;.

[14] J. He and A. Zelikovsky. (2005) `Linear Reduction Method for Predictive and Informative Tag SNP Selection', , *International Journal Bioinformatics Research and*

Applications, to appear.

[15] R. Judson, B. Salisbury, J. Schneider, A. Windemuth, and J. C. Stephens. (2002) `How many SNPs does a genome-wide haplotype map require?', *Pharmacogenomics*, Vol. 3, pp. 379-391.

[16] S. Khuller, B. Raghavachari and N. Young. Approximating the minimum equivalent digraph, SIAM Journal of Computing, 24(4), pp. 859-872, 1995.

[17] S. Khuller, B. Raghavachari and N. Young. On strongly connected digraphs with bounded cycle length, UMIACS-TR-94-10/CS-TR-3212, January 1994.

[18] S. Khuller, B. Raghavachari and A. Zhu. A uniform framework for approximating weighted connectivity problems, 19th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 937-938, 1999.

[19] G. Kimmel and R. Shamir. (2004) `GERBIL: Genotype resolution and block identification using likelihood', *PNAS* vol. 102, pp 158-162.

[20] T. I. Lee, N. J. Rinaldi et al. Transcriptional regulatory networks in Saccharomyces cerevisiae, Science 298, 799-804, 2002.

[21] S. Li, C. M. Armstrong et al. A map of the interactome network of the metazoan C. elegans, Science 303, 540-543, 2004.

[22] S. Li, S. M. Assmann and R. Albert. Predicting essential components of signal transduction networks: a Boolean model of guard cell signaling, preprint, 2005.

[23] K. Zhang, Z. Qin, J. Liu, T. Chen, M. Waterman, and F. Sun. (2004) `Haplotype block partitioning and tag SNP selection using genotype data and their applications to association studies', *Genome Research*, Vol. 14, pp. 908-916.

Appendix

Real Instance of BTR Problem Data

```
<?xml version="1.0" encoding="UTF-8"?>
<graph>
  <edge source="1"  target="2"  label="0" critical="false"/>
  <edge source="1"  target="3"  label="0" critical="false"/>
  <edge source="1"  target="4"  label="0" critical="false"/>
  <edge source="1"  target="5"  label="0" critical="false"/>
  <edge source="1"  target="6"  label="0" critical="false"/>
  <edge source="1"  target="7"  label="0" critical="false"/>
  <edge source="1"  target="8"  label="0" critical="false"/>
  <edge source="1"  target="9"  label="0" critical="false"/>
  <edge source="1"  target="10" label="0" critical="false"/>
  <edge source="1"  target="13" label="0" critical="false"/>
  <edge source="1"  target="14" label="0" critical="false"/>
  <edge source="1"  target="19" label="0" critical="false"/>
  <edge source="1"  target="21" label="0" critical="false"/>
  <edge source="1"  target="23" label="1" critical="false"/>
  <edge source="1"  target="24" label="0" critical="false"/>
  <edge source="1"  target="25" label="1" critical="false"/>
  <edge source="1"  target="31" label="1" critical="false"/>
  <edge source="1"  target="41" label="0" critical="false"/>
  <edge source="1"  target="47" label="1" critical="false"/>
  <edge source="3"  target="47" label="1" critical="false"/>
  <edge source="5"  target="2"  label="0" critical="false"/>
  <edge source="5"  target="7"  label="0" critical="false"/>
  <edge source="5"  target="8"  label="0" critical="false"/>
  <edge source="5"  target="14" label="1" critical="false"/>
  <edge source="5"  target="42" label="1" critical="false"/>
  <edge source="8"  target="3"  label="0" critical="false"/>
  <edge source="8"  target="7"  label="0" critical="false"/>
  <edge source="8"  target="11" label="0" critical="false"/>
  <edge source="8"  target="12" label="0" critical="false"/>
  <edge source="9"  target="2"  label="0" critical="false"/>
  <edge source="9"  target="3"  label="0" critical="false"/>
  <edge source="10" target="8"  label="0" critical="false"/>
  <edge source="10" target="12" label="0" critical="false"/>
  <edge source="13" target="2"  label="0" critical="false"/>
  <edge source="13" target="3"  label="0" critical="false"/>
  <edge source="13" target="8"  label="0" critical="false"/>
  <edge source="13" target="14" label="1" critical="false"/>
  <edge source="15" target="13" label="0" critical="true" />
  <edge source="16" target="13" label="0" critical="true" />
  <edge source="17" target="8"  label="1" critical="false"/>
  <edge source="17" target="14" label="0" critical="false"/>
  <edge source="17" target="29" label="0" critical="false"/>
  <edge source="18" target="20" label="0" critical="false"/>
  <edge source="20" target="2"  label="0" critical="false"/>
  <edge source="20" target="5"  label="0" critical="false"/>
  <edge source="20" target="42" label="1" critical="false"/>
  <edge source="21" target="20" label="0" critical="false"/>
  <edge source="22" target="21" label="0" critical="true" />
  <edge source="25" target="2"  label="1" critical="false"/>
  <edge source="25" target="24" label="1" critical="false"/>
```

```
  <edge source="27" target="26" label="0" critical="false"/>
  <edge source="30" target="5"  label="0" critical="false"/>
  <edge source="30" target="42" label="0" critical="false"/>
  <edge source="30" target="48" label="0" critical="false"/>
  <edge source="32" target="22" label="1" critical="true" />
  <edge source="33" target="5"  label="0" critical="true" />
  <edge source="33" target="13" label="0" critical="true" />
  <edge source="34" target="13" label="0" critical="true" />
  <edge source="36" target="3"  label="1" critical="false"/>
  <edge source="37" target="28" label="0" critical="true" />
  <edge source="37" target="41" label="0" critical="true" />
  <edge source="38" target="39" label="0" critical="true" />
  <edge source="43" target="40" label="0" critical="true" />
  <edge source="44" target="39" label="0" critical="true" />
  <edge source="45" target="10" label="0" critical="true" />
  <edge source="46" target="28" label="0" critical="true" />
  <edge source="46" target="41" label="0" critical="true" />
  <edge source="49" target="40" label="0" critical="true" />
</graph>
```

Solution

```
<?xml version="1.0" encoding="UTF-8"?>
<graph>
  <edge source="1"  target="4"  label="0" critical="false"/>
  <edge source="1"  target="6"  label="0" critical="false"/>
  <edge source="1"  target="9"  label="0" critical="false"/>
  <edge source="1"  target="10" label="0" critical="false"/>
  <edge source="1"  target="13" label="0" critical="false"/>
  <edge source="1"  target="19" label="0" critical="false"/>
  <edge source="1"  target="21" label="0" critical="false"/>
  <edge source="1"  target="23" label="1" critical="false"/>
  <edge source="1"  target="25" label="1" critical="false"/>
  <edge source="1"  target="31" label="1" critical="false"/>
  <edge source="1"  target="41" label="0" critical="false"/>
  <edge source="3"  target="47" label="1" critical="false"/>
  <edge source="5"  target="2"  label="0" critical="false"/>
  <edge source="5"  target="8"  label="0" critical="false"/>
  <edge source="5"  target="14" label="1" critical="false"/>
  <edge source="5"  target="42" label="1" critical="false"/>
  <edge source="8"  target="3"  label="0" critical="false"/>
  <edge source="8"  target="7"  label="0" critical="false"/>
  <edge source="8"  target="11" label="0" critical="false"/>
  <edge source="8"  target="12" label="0" critical="false"/>
  <edge source="9"  target="2"  label="0" critical="false"/>
  <edge source="9"  target="3"  label="0" critical="false"/>
  <edge source="10" target="8"  label="0" critical="false"/>
  <edge source="13" target="2"  label="0" critical="false"/>
  <edge source="13" target="8"  label="0" critical="false"/>
  <edge source="13" target="14" label="1" critical="false"/>
  <edge source="15" target="13" label="0" critical="true" />
  <edge source="16" target="13" label="0" critical="true" />
  <edge source="17" target="8"  label="1" critical="false"/>
  <edge source="17" target="14" label="0" critical="false"/>
  <edge source="17" target="29" label="0" critical="false"/>
```

```
  <edge source="18" target="20" label="0" critical="false"/>
  <edge source="20" target="5"  label="0" critical="false"/>
  <edge source="21" target="20" label="0" critical="false"/>
  <edge source="22" target="21" label="0" critical="true" />
  <edge source="25" target="2"  label="1" critical="false"/>
  <edge source="25" target="24" label="1" critical="false"/>
  <edge source="27" target="26" label="0" critical="false"/>
  <edge source="30" target="5"  label="0" critical="false"/>
  <edge source="30" target="42" label="0" critical="false"/>
  <edge source="30" target="48" label="0" critical="false"/>
  <edge source="32" target="22" label="1" critical="true" />
  <edge source="33" target="5"  label="0" critical="true" />
  <edge source="33" target="13" label="0" critical="true" />
  <edge source="34" target="13" label="0" critical="true" />
  <edge source="36" target="3"  label="1" critical="false"/>
  <edge source="37" target="28" label="0" critical="true" />
  <edge source="37" target="41" label="0" critical="true" />
  <edge source="38" target="39" label="0" critical="true" />
  <edge source="43" target="40" label="0" critical="true" />
  <edge source="44" target="39" label="0" critical="true" />
  <edge source="45" target="10" label="0" critical="true" />
  <edge source="46" target="28" label="0" critical="true" />
  <edge source="46" target="41" label="0" critical="true" />
  <edge source="49" target="40" label="0" critical="true" />
</graph>
```