Optimizing Exact Genetic Linkage Computations

Ma'ayan Fishelson
Computer Science Department
Technion,Haifa 32000, Israel
fmaayan@cs.technion.ac.il

Dan Geiger
Computer Science Department
Technion,Haifa 32000, Israel
dang@cs.technion.ac.il

ABSTRACT

Genetic linkage analysis is a challenging application which requires Bayesian networks consisting of thousands of vertices. Consequently, computing the likelihood of data, which is needed for learning linkage parameters, using exact inference procedures calls for an extremely efficient implementation that carefully optimizes the order of conditioning and summation operations. In this paper we present the use of stochastic greedy algorithms for optimizing this order. Our algorithm has been incorporated into the newest version of SUPERLINK, which is currently the fastest genetic linkage program for exact likelihood computations in general pedigrees. We demonstrate an order of magnitude improvement in run times of likelihood computations using our new optimization algorithm, and hence enlarge the class of problems that can be handled effectively by exact computations.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; G.3 [**Probability and Statistics**]: Probabilistic algorithms

General Terms

Algorithms, Performance, Experimentation

Keywords

Bayesian networks, Combinatorial Optimization, DAG Models, Genetic Linkage Analysis, Greedy Algorithms, Probabilistic Algorithms, SUPERLINK, Treewidth

1. INTRODUCTION

Genetic linkage analysis is a useful statistical tool for mapping disease genes and for associating functionality of genes with their location on the chromosome. Two main approaches to computing pedigree likelihood exactly are Elston-Stewart [10] and Lander-Green [16, 17, 18]. Both algorithms are variants of variable elimination methods that use different strategies to find an elimination order. The Elston-Stewart algorithm "peels" one nuclear family after another, whereas the Lander-Green algorithm "peels" one locus after another.

In [11], we propose a new algorithm to computing pedigree likelihood exactly, and implement it in a computer program called SUPERLINK. This algorithm combines and generalizes the previous two approaches by using the framework of Bayesian networks as the internal representation of linkage analysis problems. Using this representation enables efficient handling of a wide variety of linkage problems, by choosing the elimination order automatically according to the linkage problem at hand. It has been shown in [11] that SUPERLINK V1 outperforms leading linkage software with regards to speed and memory requirements. The main computational step is computing sums of products of high dimensional probability tables. The order in which the sums and products are performed has a large impact on both the time and memory requirements of the computations. In SUPERLINK V1, a simple greedy algorithm was used to determine this order.

In this paper we define a refined optimization problem behind the main computational step and present a stochastic greedy algorithm as a heuristic to solving this problem. The stochastic greedy algorithm has been incorporated into the newest version of SUPERLINK (V1.1) reported herein. We present the experimental results of this optimized version of SUPERLINK and compare it to the previous version (V1). As can be seen from the experimental results, we often attain orders of magnitude speedup in run times on the tested data sets comprising of 100-400 individuals and 10-20 loci. No other linkage program can perform these exact likelihood computations.

The rest of the paper is organized as follows. Section 2 elaborates on definitions and methods for Bayesian networks, which are the basis for efficient likelihood computations in linkage analysis. Section 3 defines the optimization problem and describes two greedy algorithms for approximately solving it. Section 4 gives a brief description of the program SUPERLINK. Experimental results are reported in Section 5. Comparison to other algorithms is discussed in Section 6. Finally, Section 7 summarizes our conclusions.

2. BAYESIAN NETWORKS

Consider a directed acyclic graph G, namely, a directed graph with no directed cycles, such that each vertex v corresponds to a variable X_v and is associated with a probability distribution $P(X_v = x_v \mid \mathbf{Pa_v} = \mathbf{pa_v})$ where $\mathbf{Pa_v}$ are the variables corresponding to vertices that have edges leading into v. Further, define the joint probability distribution for $X_1, \ldots, X_n$ via

$$P(x_1, ..., x_n) = \prod_{v=1}^{n} P(x_v \mid \mathbf{pa_v}) \quad (1)$$

The directed acyclic graph together with the joint probability distribution is called a *Bayesian network* and is denoted by (G, P) [20, 23].

Note that in the above definition, and throughout this paper, we use capital letters for variable names and lowercase letters to denote specific values taken by those variables. Sets of variables are denoted by boldface capital letters, and assignments of values to the variables in these sets are denoted by boldface lower case letters. We also use $P(x)$ as a short hand notation for $P(X = x)$.

We define the *inference problem* as follows. The input is a Bayesian network along with a subset of vertices E. The output is the probability table $P(\mathbf{E} = \mathbf{e})$ for a given disjoint subset of variables $\mathbf{E} \subseteq \{X_1, ..., X_n\}$.

Suppose that $X_1, ..., X_k$ are the variables not in E, then using Eq. (1),

$$\begin{aligned} P(\mathbf{e}) &= \sum_{x_1} ... \sum_{x_k} P(x_1, ..., x_k, \mathbf{e}) \\ &= \sum_{x_1} ... \sum_{x_k} \prod_{i} P(x_i \mid \mathbf{pa_i}). \end{aligned}$$

The problem at hand is to perform the summation and multiplication in an efficient order.

This computation can be abstracted into the problem of evaluating expressions of the form,

$$E = \sum_{x_1} ... \sum_{x_k} \prod_{l} f_l(\mathbf{Y_l}) \quad (2)$$

which consist of sums and products of high dimensional matrices (called factors). More specifically, each f_l is a *factor* (or a table) that contains an entry for each value assignment $\mathbf{y_l}$ of $\mathbf{Y_l} \subseteq \{X_1, ..., X_k\}$.

The product of two factors $f_m(\mathbf{Y_m}) f_n(\mathbf{Y_n})$ is defined as the factor $f_l(\mathbf{Y_l})$, where $\mathbf{Y_l} = \mathbf{Y_m} \bigcup \mathbf{Y_n}$. An entry of $f_l(\mathbf{Y_l})$ which corresponds to the value assignment $\mathbf{y_l}$ is obtained through the product of the entries of the factors $f_m(\mathbf{Y_m})$ and $f_n(\mathbf{Y_n})$ which correspond to the value assignments $\mathbf{y_m}$ and $\mathbf{y_n}$ consistent with $\mathbf{y_l}$.

The process of summing a variable $X_i \in \mathbf{Y_l}$ from a factor $f_l(\mathbf{Y_l})$ produces a new factor $f_l^*(\mathbf{Y_l^*})$, where $\mathbf{Y_l^*} = \mathbf{Y_l} \backslash \{X_i\}$. An entry of $f_l^*(\mathbf{Y_l^*})$ which corresponds to the value assignment $\mathbf{y_l^*}$ is obtained through the sum of the entries in the factor $f_l(\mathbf{Y_l})$ corresponding to the value assignment $\mathbf{y_l}$ for every possible value x_i of X_i.

Two extreme ways to compute expression (2) are: *variable elimination* and *conditioning* (e.g., [9, 27]). In variable elimination we eliminate one variable at a time, by multiplying all the factors that include the variable and then summing over all possible values for the variable. In conditioning we instantiate some of the variables, perform the computation for each possible instantiation of the variables and then sum the results. Factors including instantiated variables are reduced to include only those entries corresponding to assignments consistent with the instantiation.

In variable elimination we eliminate one variable at a time, by summing over all possible values for the variable, until the expression does not contain any summations. For example, assume that we want to eliminate X_k from the expression. This is done in several steps. First, we rearrange the order of summation so that the sum over X_k is the innermost. Then, we move all the terms $f_l(\mathbf{Y_l})$ where $X_k \notin \mathbf{Y_l}$ outside the summation over X_k. Suppose that the factors $f_1, ...f_m$ remain in the scope of the summation over X_k. A new factor $f(\mathbf{Y})$ which is a product of these m factors, defined over $\mathbf{Y} = \bigcup_{j=1}^{m} \mathbf{Y}_j$, is constructed:

$$f(\mathbf{Y}) = \prod_{j=1}^{m} f_j(\mathbf{Y}_j). \quad (3)$$

In the last step, we marginalize X_k out of $f(\mathbf{Y})$ by summing over all possible values of X_k. We obtain a new factor $f'(\mathbf{Y'})$, where $\mathbf{Y'} = \mathbf{Y} - \{X_k\}$:

$$f'(\mathbf{Y'}) = \sum_{x_k} f(\mathbf{Y}). \quad (4)$$

Note that with these steps we have rewritten E of Eq. (2) as

$$E = \sum_{x_1} ... \sum_{x_{k-1}} f'(\mathbf{Y'}) \prod_{l>m} f_l(\mathbf{Y_l}).$$

The resulting expression has the same general form of Eq. (2). Therefore, other variables can now be eliminated by repeating the same sequence of steps.

The time complexity of variable elimination is proportional to the total size of the factors created during the computation, and it depends on the order of elimination.

The second approach to compute E in Eq. (2) is to perform the calculation using *conditioning*. We compute E by considering expressions of the form

$$E_{x_1} = \sum_{x_2} ... \sum_{x_k} \prod_{l} f_l^*(\mathbf{Y}_l^*).$$

where f_l^* is f_l restricted to the case where $X_1 = x_1$ and $\mathbf{Y_l^*} = \mathbf{Y} - \{X_1\}$. Note that

$$E = \sum_{x_1} E_{x_1}.$$

The motivation for this approach is that computing E_{x_1} is easier than computing E, since it involves one less summation and some of the factors are smaller. The complexity of this procedure depends on the number of possible joint assignments to the variables that we condition on, since the same computation is repeated for each joint assignment. This approach is called global conditioning in [27].

The advantage of conditioning over variable elimination is the lower memory overhead. The main disadvantage of this approach is that in different evaluations of E_{x_1}, identical subexpressions are often recomputed several times. This results in higher time requirements. Therefore, hybrid algorithms, such as the one suggested in [8], combining variable elimination with conditioning to achieve a space-time tradeoff have been proposed. These algorithms first choose a set of variables to condition on, then the probability of the data is computed for each joint assignment of values to these variables, using variable elimination. Finally, the results are summed to obtain the desired likelihood.

3. CONSTRAINED ELIMINATION

In this section we define the *constrained elimination problem*, the solution of which optimizes variable elimination procedures with or without memory constraints. Section 3.1 provides a formal definition of the problem. Sections 3.2 and 3.3 describe two greedy algorithms for approximately solving the constrained elimination problem. Section 3.4 highlights some implementation choices we made.

3.1 Terminology

The following sequence of definitions are needed in order to formulate the constrained elimination problem.

Definition (graph concepts). A *weighted undirected graph* $G(V, E, w)$, $w : V \to \mathcal{N}$, is an undirected graph where each vertex $v \in V$ is assigned a weight $w(v)$. Two vertices $u, v \in V$ are said to be *adjacent* (or *neighbors*) if $(u, v) \in E$. We denote by $N_G(v) = \{u \in V \mid (u, v) \in E\}$ the set of vertices which are adjacent to v in G. We denote by $\bar{N}_G(v) = N_G(v) \bigcup \{v\}$ the set of vertices which are neighbors of v, including v itself. A *complete subgraph* (or *clique*) $G[S]$ of G is a graph with vertices S such that all vertices are neighbors. A *maximal clique* in a graph G is a clique $G[S]$ such that there exists no vertex set $S' \supset S$ for which $G[S']$ is a clique.

Definition (elimination sequence). The process of making $N_G(v)$ a complete subgraph of $G(V, E)$ and removing v and its incident edges from G is called *eliminating* vertex $v \in V$. Let α denote a permutation on $\{1, \ldots, n\}$. Let $G_i = (V_i, E_i)$, $i = 2, \ldots, n$, denote the graphs obtained from $G_1 = G$ by eliminating in order the vertices $X_{\alpha(1)}, \ldots, X_{\alpha(i-1)} \in V$. Note that $G_n = (X_{\alpha(n)}, \emptyset)$. The sequence $X_\alpha = (X_{\alpha(1)}, \ldots, X_{\alpha(n)})$ is called an *elimination sequence* of G. The *cost* of eliminating a vertex $v \in V_i$ from graph G_i is $C_{G_i}(v) = \prod_{u \in \bar{N}_{G_i}(v)} w(u)$. The *elimination cost* of an elimination sequence $X_\alpha = (X_{\alpha(1)}, \ldots, X_{\alpha(n)})$ is

$$C(X_\alpha) = \sum_{i=1}^{n} C_{G_i}(X_{\alpha(i)}) \tag{5}$$

Note that $C_{G_i}(X_{\alpha(i)})$ depends on the permutation α, albeit this dependence is suppressed in our notation. The cost function $C(X_\alpha)$, which is often referred to as the *total state space* (e.g., [14]), is a good approximated measure of the time and space complexity of computations in Bayesian networks, provided that the heaviest clique created, having the weight $max_i C_{G_i}(X_{\alpha(i)})$, fits into the RAM size of the working environment. An *optimal elimination sequence* $\hat{X}_\alpha$ is a sequence which satisfies $\hat{X}_\alpha = \arg\min_\alpha C(X_\alpha)$. When the heaviest clique does not fit into memory, as is often the case in practice, a more elaborated definition is needed.

Definition (constrained elimination sequence). The process of removing vertex $v \in V$ and its incident edges from G is called *conditioning on* v. Let $\beta = (\beta_1, \ldots, \beta_n)$ be a vector where $\beta_i \in \{0, 1\}$. A *constrained elimination sequence* $X_{\alpha,\beta} = \big((X_{\alpha(1)}, \ldots, X_{\alpha(n)}), \beta\big)$ is a sequence of vertices along with a binary vector β such that vertex $X_{\alpha(i)}$ is eliminated if $\beta_i = 0$ and conditioned on if $\beta_i = 1$. Graph G_{i+1} is obtained from graph G_i by eliminating or conditioning on vertex $X_{\alpha(i)}$, depending on the value of β_i.
The elimination cost $C(X_{\alpha,\beta})$ of a constrained elimination sequence $X_{\alpha,\beta}$ is given by

$$\sum_{i=1}^{n} \left[\prod_{k=1}^{i-1} w(X_{\alpha(k)})^{\beta_k} \right] (1 - \beta_i) C_{G_i}(X_{\alpha(i)}) \tag{6}$$

Note that when $\beta_i = 0$ for each i, this cost function reduces to the cost function (5).

Definition. The *constrained elimination problem* is to find a constrained elimination sequence $\hat{X}_{\alpha,\beta}$ which satisfies $\hat{X}_{\alpha,\beta} = \arg\min_{\alpha,\beta} C(X_{\alpha,\beta})$, such that for every i, $\max_i C_{G_i}(X_{\alpha(i)}) \le T$.

If for a given weighted undirected graph G and a threshold T, it is the case that $T < \min_{X \in V_i} C_{G_i}(X)$ for some i for each constrained elimination sequence $X_{\alpha,\beta}$, then the constrained elimination problem cannot be solved for this input. On the other hand, if $T \ge \prod_{i=1}^{n} w(X_i)$, then the constrained elimination problem reduces to the (unconstrained) elimination problem. This claim holds because the elimination cost of a vertex is never larger than $\prod_{i=1}^{n} w(X_i)$, in which case all vertices are eliminated, namely, $\beta_i = 0$ for all i.

The solution of the constrained elimination problem provides us with a solution to the problem presented in Section 2 of combining variable elimination with conditioning to achieve a desired time-space tradeoff. When we condition on a variable, we fix its value. If we set T to the memory limitations of the working environment, then a solution to the constrained elimination problem will provide us with the best choice, with regards to time complexity, that meets memory requirements.

We now present two algorithms that attempt to find close to optimal constrained elimination sequences.

3.2 A Greedy Algorithm

Our deterministic greedy algorithm receives as input a weighted undirected graph, $G(V, E, w)$ and a threshold T, and outputs a constrained elimination sequence $X_{\alpha,\beta}$ such that the elimination cost of each vertex is at most T. In iteration i, the greedy algorithm chooses a vertex X_i which satisfies $X_i = \arg\min_{X \in V_i} C_{G_i}(X)$. The algorithm terminates when every vertex has been removed from the graph. If in iteration i the elimination cost $C_{G_i}(X)$ of every vertex is above the given threshold T, then a vertex X_i is chosen to be conditioned on (rather than eliminated). The vertex X_i to be conditioned on can be chosen in various ways, one of which is via

$$X_i = \arg\max_{X \in V_i} \sqrt{|N_{G_i}(X)|} C_{G_i}(X),$$

where $N_{G_i}(X)$ is the set of vertices which are neighbors of X in G_i. The algorithm is shown in Figure 1. See Section 3.4 for an alternative way of choosing the vertex to be conditioned on.

3.3 Stochastic-Greedy Algorithm

The deterministic greedy algorithm can be vastly improved on large problems by introducing randomized selection of vertices to be eliminated, and running the algorithm sufficiently many times.

Algorithm Deterministic-Greedy(G,T)

Input: A weighted undirected graph $G(V, E, w)$, and a threshold T.

Output: A constrained elimination sequence $X_{\alpha,\beta}$ such that the elimination cost of each vertex $\leq$ T.

1. Initialize vector β of size n with zeroes.
2. $i \leftarrow 1$
3. $G \leftarrow G_i$
4. While G_i is not the empty graph **do**
- **forall** $X \in V_i$ compute $C_{G_i}(X)$
- Pick $X_k = \arg\min_{X \in V_i} C_{G_i}(X)$
- **If** $C_{G_i}(X_k) > T$ then {**conditioning on** X_k}
  forall $X \in V_i$ compute $\sqrt{|N_{G_i}(X)|} C_{G_i}(X)$
  Pick $X_k = \arg\max_{X \in V_i} \sqrt{|N_{G_i}(X)|} C_{G_i}(X)$
  $\beta_i \leftarrow 1$
  Else {**eliminating** X_k}
  $E_i \leftarrow E_i \bigcup \{(u, v) | u, v \in N_{G_i}(X_k)\}$
- Remove X_k and its incident edges from G_i
- $\alpha(i) \leftarrow k$
- $i \leftarrow i + 1$

5. Return $X_{\alpha,\beta} = ((X_{\alpha(1)}, \ldots, X_{\alpha(n)}), \beta)$

Figure 1: Algorithm Deterministic-Greedy

Our stochastic greedy algorithm first applies the deterministic greedy algorithm to determine the complexity of the problem at hand, which is estimated according to the cost of the elimination sequence found. Only if this cost is above a predetermined cost C_{min}, stochastic greedy iterations are applied. The time spent trying to find a better elimination sequence is also determined according to this cost. The algorithm is shown in Figure 2.

The stochastic greedy algorithm uses a procedure $sg(G, T)$ which is called in each iteration. This procedure finds a constrained elimination sequence using randomized selection of vertices. If the cost of the elimination sequence found is smaller than the cost of the best previously found sequence, the new sequence and cost are recorded. The only difference between procedure $sg(G, T)$ and algorithm Deterministic-Greedy(G,T), described in the previous section, is in the way the vertex to be eliminated is chosen. Procedure $sg(G, T)$ finds in each iteration m vertices with the smallest elimination cost and flips a coin to choose between them. The coin is biased according to the elimination costs of the m vertices considered for elimination. If we denote the m vertices chosen in iteration i by $X_{i1}, \ldots, X_{im}$, then the probability of vertex X_{ij} to be chosen is:

$$Pr(X_{ij}) = \frac{[C_{G_i}(X_{ij})]^{\gamma}}{\sum_{k=1}^{m} [C_{G_i}(X_{ik})]^{\gamma}},$$

where $\gamma = \frac{1}{2}$. Note that $\gamma \to 0$ implies a random choice among m candidates and $\gamma \to \infty$ implies that with probability 1 the stochastic run chooses a vertex of lowest cost like the deterministic run, regardless of m. The choice of the parameters $\gamma = \frac{1}{2}$ and $m = 3$ has been tested experimentally.

Setting I, the limit on the number of iterations, in line 3 of Algorithm Stochastic-Greedy is determined so that the optimization time would not be larger than a fraction of the total running time of the subsequent likelihood computation which can be approximated roughly from $C(X_{\alpha,\beta})$. Details are given in the next section.

Algorithm Stochastic-Greedy(G,T,C_{min})

Input: A weighted undirected graph $G(V, E, w)$, a threshold T, and a minimum cost C_{min}.

Output: A constrained elimination sequence $X_{\alpha,\beta}$ such that the elimination cost of each vertex $\leq$ T.

1. $X_{\alpha,\beta} \leftarrow$ **deterministic-greedy(G,T)**
2. If $C(X_{\alpha,\beta}) < C_{min}$ then **return** $X_{\alpha,\beta}$
3. Set I according to $C(X_{\alpha,\beta})$
  For $i \leftarrow 1$ to I **do**
  - $X^{temp}_{\alpha,\beta} \leftarrow$ **sg(G,T)**
  - **If** $C(X^{temp}_{\alpha,\beta}) < C(X_{\alpha,\beta})$ then
    $X_{\alpha,\beta} \leftarrow X^{temp}_{\alpha,\beta}$

4. Return $X_{\alpha,\beta}$

Figure 2: Algorithm Stochastic-Greedy

3.4 Implementation

We now present several implementation choices we made in order to speed up the computations and improve memory utilization.

The first choice concerns the elimination cost of a vertex. If the set of neighbors of some vertex v already form a clique (such a vertex is often referred to as *simplicial*, e.g. [14]), then we set the cost of the variable to 0 and hence make it a preferred vertex to be eliminated.

The second choice affects the process of searching for a vertex (or 3 vertices) with the minimum elimination cost. Since this search can be time consuming, if the algorithm encounters a vertex having a cost below a threshold T_{min}, then the search for the minimum is discontinued and this vertex is chosen. In the current implementation we use $T_{min} = 15$, which is a very small cost.

The third choice determines the amount of time spent on trying to improve the elimination sequence. Based on extensive experiments, we constructed a table with 10 complexity classes determined by the cost of the elimination sequence found. The first class is for problems with *cost* $< 10^7$, and the last class is for problems with *cost* $\geq 10^{15}$. The intervals $10^i <$ *cost* $< 10^{i+1}, i = 7, \ldots, 14$ determine the eight intermediate complexity classes. This cost function serves as a rough estimate of the run time needed for the subsequent likelihood computation, and according to it the maximum amount of optimization time that is allocated for each class is set to a fraction of the overall run time of the likelihood computation. Problems in the first complexity class, namely with a cost lower than 10^7, are not allocated any time for optimization.

The complexity class and the maximum number of iterations are set initially according to the cost of the elimination sequence found by the deterministic greedy algorithm. These numbers are then updated whenever an elimination sequence with lower cost is found. If an elimination sequence with a cost below 10^7 is found, then the optimization is stopped and this sequence is used. Every 200 iterations the improvement rate, per iteration, in the elimination cost of sequences is checked, and if it is below a prescribed threshold of 0.0001, then optimization is terminated. As mentioned above, the cost of the elimination sequence is a rough estimate of the run time needed for the likelihood computation, with some variance. Improving this time estimate and adapting it to a specific computer at hand could lead to further improvement in total run time.

The fourth choice relates to the way we choose a vertex to condition on. The input to the problem presented in Section 2 is in fact a Bayesian network. Each variable of the Bayesian network corresponds to a vertex in the graph and appears in several factors, each of which is a probability function. During the computation, some of the original factors are deleted and new factors are created. The vertex X_i that SUPERLINK chooses to condition on satisfies

$$X_i = \arg\max_{X \in V_i} \sqrt{|f_{G_i}(X)|} C_{G_i}(X), \tag{7}$$

where $f_{G_i}(X)$ are the factors in G_i that include X. The objective is to condition on a vertex whose elimination reduces memory requirements as much as possible in as many factors as possible. The idea behind the condition that appears in Section 3.2 is that if we condition on a vertex with many neighbors, we reduce the number of neighbors for many vertices. Experimentally, the condition expressed by Eq. (7) has been found to be slightly better.

4. APPLICATION IN GENETICS

Genetic linkage analysis is a challenging application of Bayesian networks since it requires the use of networks consisting of thousands of vertices. Consequently, likelihood computations in such networks calls for extremely efficient order of summation and conditioning operations. To validate our algorithms with realistic data we incorporated them in the newest version of the genetic linkage program SUPERLINK.

In SUPERLINK, pedigrees are represented in a detailed manner using Bayesian networks. A pedigree, which is the input to a genetic linkage problem, defines a joint distribution, parameterized by the recombination fraction θ, of the *genotypes* and *phenotypes* of the individuals in the pedigree. The goal of genetic linkage analysis is to estimate the recombination fraction θ between a disease gene and known loci on the chromosome. The approximate physical location of the disease gene on the chromosome can be inferred from the estimated recombination fraction. For more details on the exact structure of the Bayesian networks constructed by SUPERLINK, we refer the reader to [11, 12].

Using Bayesian networks as the representation of linkage analysis problems allows us to give a unified treatment to the entire spectrum of linkage problems and combine the two extreme approaches for computing pedigree likelihood exactly, the Elston-Stewart approach and the Lander-Green approach. These two approaches are both variants of variable elimination methods that use different elimination orders.

Whenever feasible, SUPERLINK uses variable elimination alone to compute the likelihood of pedigree data. Otherwise, variable elimination is combined with conditioning to achieve the best time-space tradeoff given the memory available for the linkage analysis problem. The choice of variable elimination order, including the choice of variables for conditioning, is determined by the greedy algorithms presented in Section 3.

5. EXPERIMENTAL RESULTS

We performed four experiments. The first experiment compared the performance of the stochastic greedy algorithm versus the deterministic greedy algorithm on a variety of large realistic simulated pedigree data sets. We compared both run times and costs of the elimination sequences found by each algorithm. The experiment shows the superiority of the stochastic greedy algorithm over the deterministic greedy algorithm in almost all cases. We can see an order of magnitude improvement in run times of likelihood computations.

The second experiment evaluated the performance of the stochastic greedy algorithm for finding an (unconstrained) elimination sequence such that the *size* of the largest clique created during the elimination process is as small as possible and assuming an unweighted graph as input, namely, the weights of all vertices are fixed to the same value, say $c \geq 2$. This quantity (minus one) is called the *treewidth* of the graph. It is an important quantity for many applications, such as constraint satisfaction, independent set, dominating set, graph K-colorability and Hamiltonian circuit [2].

The third and fourth experiments were performed in order to determine the optimal values for the parameters γ and m used in the stochastic greedy algorithm. The running environment for experiments A,C and D was a Sun OS version 5.8 (sun4u) with 2624 MB RAM. Experiment B was performed by Arie Koster on a 4 processor Pentium, 1700 MHz, with 1GB RAM.

Experiment A. The first experiment compared between the performance of the deterministic greedy algorithm and the stochastic greedy algorithm (Table 1). We can see that using the stochastic greedy algorithm of SUPERLINK V1.1 enables us to process data sets which could not be processed using the deterministic greedy algorithm of SUPERLINK V1, and that the run time improvement is often over an order of magnitude. Note that other leading linkage programs, such as FASTLINK V4.1 [5, 7, 26] and GENEHUNTER V2.1 [17] cannot run any of these data sets. VITESSE V2.0 [21, 22] can only run the first data set.

All the data sets used in this experiment are based on a single growing pedigree, in terms of the number of loci and the number of people. For each data set we show its parameters in terms of the number of people in the pedigree and the number of loci. The number of vertices in the Bayesian network constructed for the specific problem is also shown; This number does not include variables that have been determined to be constant due to phenotypic data.

Note that the times reported include the optimization time and the subsequent likelihood computation time. For the deterministic algorithm, we report the cost of the elimination sequence found and the run time. We also show the improvement ratios in cost and time of the stochastic algorithm versus the deterministic algorithm. For the stochastic algorithm, the results are reported over 10 runs. The differences between the improvement ratio in run time and the improvement ratio in cost are due to the optimization time included in the run time and the fact that the cost does not fully predict the run time. There is a significant variance in the results.

♯People/♯Loci/♯Nodes	Run Time - sec				Improvement Ratio (Run Time)	Cost (DG)	Improvement Ratio (Cost)
	deterministic greedy (DG)	stochastic greedy (SG)					
		min	avg	max			
100 / 10 / 1676	5.8	5.8	5.8	5.8	**1**	2.0E6	1
100 / 15 / 2449	17	12.3	16	17.9	**0.95 - 1.4**	1.3E7	1 - 1.9
100 / 20 / 3224	84	29.1	52	84.9	**0.99 - 2.89**	6.1E7	1 - 4.7
200 / 10 / 3254	38.8	8.2	16.2	23.8	**1.63 - 4.73**	1.7E8	12.1 - 34
200 / 15 / 4720	219.4	89.2	140.4	182.7	**1.2 - 2.5**	3.8E9	17.3 - 38.4
200 / 20 / 6242	*	358.8	1703	4446.3	∞	1.3E14	7411.8 -114545.5
300 / 10 / 4859	554.7	12.1	27.2	46.8	**11.9 - 45.9**	4.6E10	478.9 - 9100
300 / 15 / 7040	2273.8	139.5	367.2	966.5	**2.4 - 16.3**	2.6E11	71.4 - 419
300 / 20 / 9276	*	6352.7	19940	36932.3	∞	4.4E15	6753.8 - 51647.1
400 / 10 / 6372	283.6	20.6	50.7	94	**3 - 13.8**	3.7E10	442.2 - 3336.4
400 / 15 / 9292	9701.8	385.5	1163.9	3517	**2.8 - 25.2**	1.1E12	144.3 - 475
400 / 20 / 12278	*	32336.9	69914.5	164708.7	∞	1.6E17	14909.1 -298181.8

Table 1: *Stochastic Greedy* versus *Deterministic Greedy*. The bold column shows the ratio of run time of SG, over 10 runs, versus DG. The last 2 columns show the cost of the elimination sequences found by DG and the ratio of cost found by SG versus DG. The symbol * means over 100 hrs.

Experiment B. The second experiment compared the performance of our algorithms with three known heuristic algorithms for finding treewidth: MCS^+, $\mathrm{LEX\_P}^+$ and $\mathrm{LEX\_M}^+$ [15]. Our stochastic algorithm (SG) is found to be superior as can be seen from Table 2.

The three algorithms consist of two steps. First a preprocessing step in which the graph is contracted by a series of graph reductions is applied [6]. In the MCS^+ algorithm a Maximum Cardinality Search [29] is executed at the second step. The other two algorithms are variants of a lexicographic breadth first search algorithm [25] with graph reductions as a preprocessing step.

Note that the run time of MCS^+ is about 10 fold the run time of SG with 100 iterations, as can be seen from Table 2. This is because MCS^+ is run $|V|$ iterations; in each iteration a different vertex serves as a starting point. Also note that occasionally there is a vast reduction in the size of the largest clique produced by MCS^+, but usually SG gives smaller largest clique. The other two algorithms ($\mathrm{LEX\_P}^+$, $\mathrm{LEX\_M}^+$) are inferior to MCS^+ on these data sets, as can be seen from Table 2.

In all graphs, the stochastic greedy algorithm (using 100 iterations) reduced the size of the largest clique created compared to the deterministic greedy algorithm; On the average, the size of the largest clique created in these examples reduced by 29%. Recall that an improvement of Δ in the largest clique size suggests an improvement of c^{Δ} in run time. Further note that although the larger graphs have clique sizes up to 40 vertices, they can still be used as inputs to SUPERLINK due to the conditioning operation which sufficiently reduces the larger cliques.

Experiment C. The third experiment compared the class of functions $[C_{G_i}(X)]^{\gamma}$ for the biased coin for the optimal value of γ. We found that on the average, if we set γ to a fixed value shared by the entire run of our algorithm, then the best results are achieved by setting $\gamma = \frac{1}{2}$. Coincidentally, this value was the natural choice made before any experimentation took place. We ran 127 random graphs of sizes ranging from 4000 vertices up to 20000 with various treewidths. The graphs were obtained from simulated pedigree data. Each graph was run on the following values of γ: $0.25, 0.33, 0.5, 0.75, 1$, and the results were compared with regards to the cost of the elimination sequence found. The variance of performance was very high. The distribution of the most successful runs with regards to the cost of the elimination sequence found, as a function of γ, is given below showing the superiority of $\gamma = \frac{1}{2}$:

γ	**0.25**	**0.33**	**0.5**	**0.75**	**1**
#runs	18	21	51	21	16

Experiment D. The fourth experiment tested the optimal value for m ranging from 3 to 15 using similar graphs to those used in Experiment 3. There is no clear indication that a specific fixed choice of m is significantly better than others. The value $m = 3$ for SUPERLINK has been chosen quite arbitrarily.

6. RELATED WORK

In [14], a survey and evaluation of several triangulation algorithms, which are analogous to algorithms for finding unconstrained elimination sequences, is reported. Also presented are several simple heuristic methods. The experimental results in [14] show that the *minimum weight* heuristic, which corresponds to the *deterministic-greedy* algorithm described herein, gives overall the best results in most cases; results which we improve with our stochastic algorithm. In some cases, *simulated annealing* produces considerably better results than this heuristic, and in other cases, there is only a slight improvement. Kjærulff concludes with the following:

♯Nodes	♯Edges	Treewidth				Time (sec)			
		SG	**MCS$^+$**	**LEX_P$^+$**	**LEX_M$^+$**	**SG**	**MCS$^+$**	**LEX_P$^+$**	**LEX_M$^+$**
3915	6813	40	**26**	33	33	15.03	210.63	342.09	467.05
3979	6941	**21**	24	32	31	14.15	224.43	246.51	355.89
4637	8184	18	**15**	**15**	16	21.78	173.89	195.5	353.58
4720	8211	**22**	26	31	31	22.71	233.73	299.42	476.3
4733	8246	**23**	32	32	31	23.34	254.89	366.57	554.46
6158	10822	**17**	19	18	18	41	375.43	377.41	669.73
6166	10695	37	**22**	33	33	47.4	359.98	537.6	886.51
6242	10814	**27**	30	42	42	45.61	495.24	607.17	928.6
6253	10755	**19**	**19**	26	27	47.04	416.32	457.38	813.79
6309	10829	**19**	22	29	27	38.63	323.52	438.42	712.15
6372	11198	**21**	24	31	31	45.06	426.14	500.13	834.41
6393	10891	**22**	23	26	26	43.27	403.67	402.29	724.6
9135	15550	**23**	**23**	32	32	99.52	802	1098.87	1755.69
9223	15579	**22**	26	33	33	92.56	781.33	1056.06	1799.59
9276	15926	**30**	34	45	46	104.74	949.2	1642.84	2464.48
9292	16096	**24**	33	43	42	100.07	1131.35	1330.88	2191.66
11800	20073	26	**25**	33	33	162.41	1450.43	2022.12	3606.19
12058	20270	**23**	26	34	36	166.4	1382.61	2055.31	3630.71
12278	21215	**32**	40	51	62	183.93	2211.95	3249.85	5325.91
12606	21440	**24**	32	39	39	171.55	1764.04	2038.08	3697.2

Table 2: *Treewidth Experiments*. The bold letters show which algorithm(s) achieved the best result for a given instance. Note that SG achieves the best result on 16/20 instances.

> The lesson to be learnt from this comparison is: If time is not a crucial parameter it might be recommendable to use annealing.
> ...
> On the other hand, if triangulation has to be fast we can by no means recommend the annealing algorithm which is orders of magnitude slower than the heuristic algorithms.

Our *stochastic greedy* algorithm minimizes the total run time of likelihood computations which includes the time for finding a good elimination order and the time for performing the elimination. Simulated annealing may occasionally produce better elimination orders but we expect that the time for finding an improvement would usually be larger than the time gained due to the improved elimination order.

Several hybrid algorithms combining variable elimination with conditioning to achieve a space-time tradeoff have been proposed in the context of constraint satisfaction. The algorithms proposed in [19, 24] interleave variable elimination with conditioning. Both algorithms are controlled by a parameter which determines in each step whether variable elimination will be performed or a variable to condition on will be chosen. Consequently, our *stochastic greedy* algorithm can also optimize these algorithms for constraint satisfaction.

7. DISCUSSION

In this paper we defined the *constrained elimination problem* which is to find an optimal elimination sequence of a graph under the constraint that the weight of the heaviest clique created is below a certain threshold. We presented a stochastic greedy algorithm for approximately solving this problem and implemented it in the genetic linkage program SUPERLINK.

We have demonstrated the significance of using the stochastic greedy algorithm in genetic linkage analysis. Our optimization algorithm can also be utilized in a variety of different applications, such as: constraint satisfaction, independent set, dominating set, graph K-colorability and Hamiltonian circuit [2, 9] as well as in other applications of Bayesian networks.

The heuristic algorithm we presented is indeed simple and yet is new and the most effective algorithm to date for minimizing the total time of optimization and likelihood computation. The success of this algorithm stems from the fact that it is an any time algorithm; after each iteration, which is very fast, there is a valid elimination order and a good estimate of the total run time. The history of improvements and the estimated total run time provide a mechanism to determine when to stop the optimization phase and move on to computing the likelihood. Currently, we use the cost of the elimination sequence as a rough estimate of the total run time. We plan to develop a cost function which would serve as a more accurate estimate of the total run time.

Recall that if summation in the cost function (5) is replaced with maximization and the weight of every vertex in the graph is constant and $T = \infty$, then the problem is reduced to finding the *treewidth* of a graph, which is known to be NP-complete [3]. Table 2 shows that our stochastic greedy algorithm is closer to finding treewidth than several known algorithms.

Due to the small amount of time it took for the reduction rules due to [6] to run (times not shown) we conclude that reduction rules have a potential for further optimization as a preprocessing step before SG is applied. We plan to adapt a weighted version of these reduction rules and test it thoroughly within SUPERLINK to determine whether the additional running time needed for optimization still yields a gain on the total run time. We also plan to compare our algorithm to other alternatives for computing treewidth such as those in [4, 28] and the algorithm implemented in the HUGIN 6.1 system [1, 13].

Acknowledgments

We thank Arie Koster for performing Experiment B and to Uffe Kjærulff for pointing us to HUGIN 6.1. We also thank Natalia Graiz and Marina Shteinberg for implementing an earlier version of our stochastic algorithm and running initial experiments. This research was supported by the Israel Science Foundation.

8. REFERENCES

[1] S. K. Andersen, K. G. Olesen, F. V. Jensen, and F. Jensen. HUGIN - a shell for building Bayesian belief universes for expert systems. In *Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1080–1085, 1989.

[2] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposibility. *BIT*, 25:2–23, 1985.

[3] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.

[4] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal clique trees. *Artificial Intelligence*, 125(1-2):3–17, 2001.

[5] A. Becker, D. Geiger, and A. A. Schäffer. Automatic selection of loop breakers for genetic linkage analysis. *Hum. Hered.*, 48(1):4960, 1998.

[6] H. Bodlaender, A. Koster, F. Eijkhof, and L. Gaag. Pre-processing for triangulation of probabilistic networks. In *Proc. 17th Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 32–39, 2001.

[7] R. W. J. Cottingham, R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *Am. J. Hum. Genet.*, 53:252–263, 1993.

[8] R. Dechter. Topological parameters for time-space tradeoff. In *Proc. Twelfth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 220–227, 1996.

[9] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In J. M. I. (Ed.), editor, *Learning in Graphical Models*, pages 75–104. Kluwer Academic Press., 1998.

[10] R. C. Elston and J. Stewart. A general model for the analysis of pedigree data. *Hum. Hered.*, 21:523–542, 1971.

[11] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(Suppl. 1):S189–S198, 2002.

[12] N. Friedman, D. Geiger, and N. Lotner. Likelihood computation with value abstraction. In *Proc. Sixteenth Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 192–200, 2000.

[13] Hugin. The API reference manual version 5.4. February 2002.

[14] U. Kjærulff. Triangulation of graph - algorithms giving small total state space. Technical Report R90-09, Department of Computer Science, Aalborg University, Denmark, March 1990.

[15] A. Koster, H. Bodlaender, and S. Hoesel. Treewidth: Computational experiments. Technical Report ZIB 01-38, December 2001.

[16] L. Kruglyak, M. J. Daly, and E. S. Lander. Rapid multipoint linkage analysis of recessive traits in nuclear families including homozygosity mapping. *Am. J. Hum. Genet.*, 56:519–527, 1995.

[17] L. Kruglyak, M. J. Daly, M. P. Reeve-Daly, and E. S. Lander. Parametric and nonparametric linkage analysis: A unified multipoint approach. *Am. J. Hum. Genet.*, 58(6):1347–1363, 1996.

[18] E. S. Lander and P. Green. Construction of multilocus genetic maps in humans. In *Proc. Natl. Acad. Sci.*, volume 84, pages 2363–2367, 1987.

[19] J. Larrosa. Boosting search with variable elimination. In *Proc. 6th International Conf. on Principles and Practice of Constraint Programming (CP2000)*, pages 291–305, September 2000.

[20] S. L. Lauritzen. *Graphical Models.* Oxford University Press., 1996.

[21] J. R. O'Connell. Rapid multipoint linkage analysis via inheritance vectors in the Elston-Stewart algorithm. *Hum. Hered.*, 51(4):226–240, 2001.

[22] J. R. O'connell and D. E. Weeks. The VITESSE algorithm for rapid exact multilocus linkage analysis via genotype set-recoding and fuzzy inheritance. *Nat. Genet.*, 11:402–408, 1995.

[23] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, San Francisco, CA, 1988.

[24] I. Rish and R. Dechter. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning*, 24(1-2):225–275, January 2000.

[25] D. Rose, E. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

[26] A. A. Schäffer, S. K. Gupta, K. Shriram, and R. W. J. Cottingham. Avoiding recomputation in linkage analysis. *Hum. Hered.*, 44:225–237, 1994.

[27] R. D. Shachter, S. K. Andersen, and P. Szolovits. Global conditioning for probabilistic inference in belief networks. In *Proc. Tenth Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 514–522, 1994.

[28] K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proc. 14th National Conf. on Artificial Intelligence (AAAI)*, pages 185–190, July 1997.

[29] R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.