Compilation for Fast Calculation over Pedigrees

Peter Szolovits, Ph.D.*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, 02139, USA

This paper was published in *Cytogenetics and Cell Genetics*,
Vol. 59, pages 136–138, 1992.

Efficient computation of probabilistic relationships over family pedigrees is an important tool for a variety of problems in genetics, including genetic counseling and linkage analysis. The development of faster and more comprehensive algorithms has preoccupied geneticists for decades [**?**], and recent general-purpose algorithms for probabilistic inference over arbitrary networks (e.g., [**?**]) have also been proposed as useful tools for such problems [**?**].

We present a set of engineering speed-up methods developed as part of the implementation of a prototype program for assisting genetic counselors, GENINFER-II. The design of the underlying computational algorithm is due to Cooper [**?**], and treats the problem of finding an efficient way to evaluate a Bayes network as a problem of factoring algebraic formulæ. It is thus an extension of the method of [**?**] (though I believe it was independently formulated) and handles consanguinity by extending the factoring method rather than by cutset conditioning (as does, for example, [**?**]). The main arguments presented in this paper are:

1. It is profitable to treat the formulæ that describe a pedigree and its associated information as a computer program that can compute the probability of the given information as a function of various parameters of interest (e.g., recombination fractions).

2. A number of typical compiler optimization techniques can be applied to the resulting program to speed up computations, typically at the expense of (much) more space and more compile-time analysis. We examine specifically the roles of *constant folding* and *loop unrolling*.

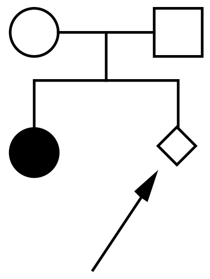
1 Pedigrees as Formulæ

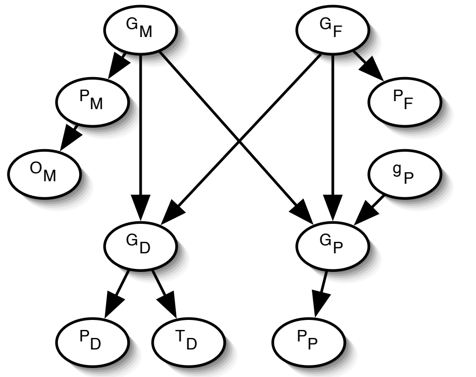
To illustrate pedigree computations, consider the very simple family pedigree shown in Figure **??**a, whose structure as a Bayes network appears in Figure **??**b.[1] An actual problem of clinical or research interest would have many more features, but this simple example should serve to illustrate the issues. For a Bayes network, the joint probability of any complete assignment of values to all the variables is given by the formula:

$$P(x_1, x_2, \ldots, x_n) = \prod_{1\le i\le n} P(x_i|\pi_{x_i}). \tag{1}$$

*The work reported here has been supported (in part) by grants R01 LM 04493 from the National Library of Medicine and R01 HL 33041 from the National Heart, Lung and Blood Institute. The work was performed while the author was a Visiting Professor at the Section on Medical Informatics, Stanford Medical School. Dr. Gregory Cooper developed the computational algorithms used in the work reported here, and Dr. Susan P. Pauker and Ms. Nomi Harris contributed significantly to the project of which this work forms a part.

[1]Although it may seem more natural to treat the gender of the propositus as dependent on his or her genotype, it is convenient instead to make the propositus' genotype jointly dependent on the parents' genotypes and the gender. This allows consideration of different modes of inheritance—e.g., autosomal recessive vs. x-linked—simply by altering the conditional probability matrix at this node.


(*a*)



(*b*)

Figure 1: (a) A very simple pedigree, as it might arise in genetic counseling. The couple have had a daughter affected by a suspected Mendelian disorder whose mode of inheritance is not certainly known, and are concerned about the risk to the propositus, who is indicated by the arrow. (b) The pedigree of (a) corresponds to a network of variables interrelated by probabilistic dependencies. M, F, D and P subscripts on nodes correspond to the mother, father, daughter and propositus, respectively. Nodes labeled G stand for the individual's genotype, P for phenotype, g for gender, T for a test whose outcome depends on the genotype, and O for an observation whose outcome depends on the phenotype. In a Bayes network such as this, it is the *absence* of connections that is significant—indicating probabilistic independence.

where π_x is the set of predecessor nodes for x (those on which it depends probabilistically). For x without predecessors, $P(x|\pi_x) = P(x)$ is simply the *a priori* probability.

When (as is typical) only some subset of variables $X' \subseteq X$ have known values, then the joint probability of the set of known values is simply the sum of terms of the form (**??**), one for each combination of possible values of the unknown variables $X \setminus X'$. If we number the variables so that the first $k = |X \setminus X'|$ are unknown, and if V_i is the set of values that the variable x_i may take on, then

$$P(X') = P(x_{k+1}, \dots, x_n) = \sum_{x_1 \in V_1, x_2 \in V_2, \dots, x_k \in V_k} P(x_1, x_2, \dots, x_n). \tag{2}$$

Note that if X' is empty, i.e., if no variable has been assigned a value, then the joint probability will be 1.0, summing over all possible states of the world. To calculate the conditional probability of some set of variable assignments X_J given X_I (where I and J represent sets of variables), we use the definition of conditional probability: $P(X_J|X_I) = P(X_J, X_I)/P(X_I)$.

Although conceptually simple, this is not an advantageous computational approach. For example, if in the pedigree of Figure **??**a just the four genotypes and the phenotype of the propositus are unknown, (**??**) will sum over 162 terms, and in general the number of terms will be exponential in the number of unknowns. For families without consanguinity, [**?**] shows that such formulæ may be factored along the lines of parental descent to form a more efficient, but mathematically equivalent expression. Recently-developed algorithms for general Bayes networks consider a more varied set of factorizations that can also handle consanguineous pedigrees. The basic idea remains the same: The structure of the network shows that many of the variables can be made independent once "in-between" nodes have been instantiated (given definite values); those instantiated nodes are not necessarily those corresponding to the genotypes of the parents, however. For example, our implementation of Cooper's algorithm [?] will derive the formula

$$\begin{aligned} P \;=\; & \widetilde{\sum_{P_M,G_D}} \Bigg\{ \widetilde{\sum_{G_F}} \Bigg[P(G_F) \widetilde{\sum_{g_P}} \Bigg(P(g_P) \widetilde{\sum_{P_P,G_P}} \Big\{ P(P_P|G_P) \widetilde{\sum_{G_M}} [P(G_P|g_P,G_M,G_F) \\ & \qquad \times P(G_D|G_M,G_F) \\ & \qquad \times P(P_M|G_M)P(G_M)] \Big\} \Bigg) \\ & \qquad \times \widetilde{\sum_{P_F}} P(P_F|G_F) \Bigg] \\ & \qquad \times \Big(\widetilde{\sum_{T_D}} P(T_D|G_D) \Big) \Big(\widetilde{\sum_{P_D}} P(P_D|G_D) \Big) \Big(\widetilde{\sum_{O_M}} P(O_M|P_M) \Big) \Bigg\} \end{aligned} \qquad (3)$$

for the joint probability of the variables in the network of Figure ??b, in case no variables are instantiated ($X' = \emptyset$, when of course the expression simply sums to 1.0). Observe that we may in fact use the same expression to evaluate probabilities when some variables are instantiated, by using the following definition of our variant of the summation operator: When a variable x is instantiated, say to the value v, the meaning of $\widetilde{\sum}_x f(x)$ is just $f(v)$, but if it is not instantiated, then $\widetilde{\sum}_x f(x) = \sum_{x \in V(x)} f(x)$. Then, (??) subsumes all formulæ of the form of (??), but requires substantially less computation. Under the assumption that P_M, P_F, P_D, O_M, T_D, and g_P are known, for example, (??) requires roughly one third as many multiplications for its evaluation. Additional savings are realized by a caching strategy which exploits the fact that in evaluating conditional probability formulæ, many components of the calculation of the numerator may be saved and used again in computing the denominator [?]. This point is not further discussed here, but makes a large practical difference in our implementation. Although networks with many undirected cycles (such as G_M, G_D, G_F, G_P, G_M in Figure ??b and those introduced by consanguinity in pedigrees) in general require exponentially much computation, several factors often help make the computations tractable. Small cycles such as the example here, which result simply from the dependence of all sibling genotypes on the genotypes of their parents, may be eliminated by introducing an extra node for the joint parental genotype [?]; the actual degree of inbreeding in human families is often small; and heuristic factoring methods appear to be practically quite effective, despite the impossibility of theoretical guarantees.

However, there are many situations in which many repeated evaluations of a pedigree are necessary, and methods that are convenient for a single evaluation become unacceptably slow when repeated often. In our application, where information about a family pedigree is presented item-by-item, the pedigree must be repeatedly evaluated to indicate to the counselor the impact of each additional fact. In linkage analyses, inference over a pedigree may have to be done many times to try to estimate the probability of a genetic model being the correct one or attempting to fit a maximum-likelihood formulation. In such cases, additional methods for speeding up the calculation are required. We now turn to consideration of several such methods.

2 Pedigree Formulæ as Computer Programs

Often the most successful speed-up methods require some significant insight into the nature of the problem being solved or the solution method. The factorization scheme already exploited here falls into this category. A second general principle of optimization is that any *constant* feature of a problem may be exploited to advantage. This observation underlies most of the methods to be presented here. The third principle is that often one valuable commodity may be traded off for another. In computing, typical tradeoffs are between space and time. We turn to a few specific techniques for exploiting constant features of a pedigree and some possible time/space tradeoffs.

In a typical probabilistic inference program, the expression that describes a probability to be calculated is represented as some data structure in the memory of the computer. The program that performs the computation then traverses that structure, *interpreting* it to determine what detailed computations must actually be done. The work done by the program thus consists of two parts:

(a) figuring out the values to be operated on and the exact arithmetic operations to be performed, and (b) actually doing them. When a pedigree with fixed structure is to be evaluated repeatedly, part (a) of this work is still done once for each evaluation—a wasted duplication of effort that we eliminate.

Formula (**??**) may be translated directly into a computer program that, when given an instantiation of some of the variables in the network, computes the joint probability of those instantiations. For example, our translation into LISP[2] of the last line of (**??**) is

```
(* (sum (td) (p td gd))
   (* (sum (pd) (p pd gd)) (sum (om) (p om pm))))
```

where an expression such as `(p td gd)` stands for $P(T_D|G_D)$ and is implemented as an array retrieval, and `*` is a conditional k-ary multiplication that returns zero as soon as any of its arguments is zero (to avoid unnecessary multiplications by zero, which arise often). `Sum` implements $\widetilde{\sum}$: it checks to see if its variable is already instantiated, in which case it simply evaluates its expression; otherwise it sets up a loop to iterate over the valid values of the variable, summing the expression over all instantiations. In fact, each of these forms is further macro-expanded to the primitive LISP expressions that implement these operations. Then, the LISP compiler is invoked to further translate the given program into the machine's primitive instructions set.

In our limited experiments with this technique, it speeds up the calculation of probabilities over pedigrees by a factor of between three and eight, depending on which LISP implementation is being used and specifics of that compiler's own optimizations. Correction of known deficiencies in some of these compilers or the use of different target languages should lead to even greater speed improvements. The only costs of this method are the additional time required for the two compilation steps (formulæ to program, and program to machine language) and the additional space required by the compiled program. Unless these steps are unduly slow, this seems eminently worth doing.

In some series of calculations, we may also know that certain parameters will be held constant. For example, we considered the case above where all variables except the four genotypes and the propositus' phenotype are known. In this case, we may compile a special-purpose version of the general expression (**??**):

```
(sum (gd)
  (* (sum (gf)
       (* (p gf)
          (p GENP)
          (sum (pp)
            (sum (gp)
              (* (p pp gp)
                 (sum (gm)
                   (* (p gp GENP gm gf) (p gd gm gf) (p PM gm) (p gm))))))
          (p PF gf)))
     (p TD gd) (p PD gd) (p OM PM)))
```

Here, upper-cased variables have known constant values and may be replaced by those values, making possible further simplifications of the function. `(p GENP)`, for example, is simply the *a priori* probability of that known value and may be replaced by that constant; it may also be factored out of the loops over values of G_D and G_F. In the factor `(p gp GENP gm gf)`, which corresponds to looking up a value in a rank-four array, the fact that one index is a known constant means that the retrieval can be done over a reduced (rank-three) array, with correspondingly fewer index

[2]Our experiments have thus far been performed in LISP, primarily because it is easy to invoke the LISP compiler from within a running program and then to call the just-compiled function—something that is very difficult or impossible in most programming languages. There is no reason, however, that the methods we describe here should be limited to LISP. FORTRAN, C, or other languages could also be targets, using some sophisticated features of the operating system shell to coordinate the resulting executable code segments. In fact, because commercial compilers for such languages typically generate much better optimized machine code than LISP compilers, there are practical advantages to moving to such a language, at least as the *target* of the system.

calculations. If more parameters are known constant, further corresponding simplifications are possible. An almost equivalent way to view this optimization is that the original Bayes net of Figure **??**b may be simplified by graph-reduction methods developed by Shachter [**?**] when some of the probabilistic nodes are known constant. The expression to compute probabilities for that simpler network may then be factored and translated to an expression similar to that shown above.

Once we know which parameters of a model are actually to be treated as constant, the process described here corresponds to the well-known compiler optimization technique called *constant folding* and its offshoots. The principal difficulty in applying it is to know what can be treated as a constant for enough evaluations of the pedigree formula that it is worth going to the extra effort of compilation to speed up the special case. In our genetic counseling application, this optimization has not yet been used because frequent "what if" calculations have made it difficult to predict which instantiated variables will truly remain fixed with their current values. It should be more useful in applications such as linkage analysis, where known data typically do remain constant.

A final speed-up method exploits a possible tradeoff between the size of the compiled program and its running time. For example, because the variable `gm`, the mother's genotype, is known to take on only a small fixed set of values—for example, $\texttt{GM}_1$, $\texttt{GM}_2$ and $\texttt{GM}_3$, namely "homozygous normal," "heterozygous" and "homozygous affected"—we may expand the expression
`(sum-over (gm) (* (p gp GENP gm gf) (p gd gm gf) (p PM gm) (p gm)))` to

```
(+ (* (p gp GENP GM1 gf) (p gd GM1 gf) (p PM GM1) (p GM1))
   (* (p gp GENP GM2 gf) (p gd GM2 gf) (p PM GM2) (p GM2))
   (* (p gp GENP GM3 gf) (p gd GM3 gf) (p PM GM3) (p GM3))).
```

After such *loop unrolling*, the execution time of the code controlling the loop has been eliminated, and further optimizations (e.g., constant folding) are possible because of the newly-introduced constants.

If applied throughout the program representing a large pedigree, the whole program would be translated into one enormous arithmetic expression consisting of sums of products of sums of products of ..., where the leaf terms would all be constants or references to the parameters of the problem. This structure has a number of significant benefits. For instance, compilers can generally produce very good code for such expressions, and implementation on fine-grained parallel machines can effectively exploit the obvious independence of different parts of the expression to overlap their computations. The major drawback is that such a fully unrolled expression for problems of actual interest might easily contain millions of terms, thus exacting a large space penalty for the improved speed and severely taxing the ability of real compilers to handle such large arithmetic expressions. One avenue of escape from this extreme is to unroll loops only in part, reducing both the speed gain and the space loss. Unrolling the loops furthest inside outer summations will generally make the greatest speed gains at the least space cost, but we have as yet no useful experience to report on how to make that tradeoff. It is also possible that even the extreme form of loop unrolling and the resulting huge expression are an appropriate way to organize computation for scalable supercomputer architectures; indeed, one such experiment has recently been reported for computational dynamics problems [**?**].

3 Discussion

We have reviewed the typical computational approach to calculating likelihoods over pedigrees, emphasizing that the pedigree is effectively translated into an equivalent mathematical formula which may be manipulated to exploit independence. In a similar way, we suggest that such a formula can then be translated into a computer program that computes its value. The resulting program may be further manipulated by techniques much like those applied in compiler optimization to exploit known characteristics (e.g., constants) of the problem and possible tradeoffs of the computing environment.

Optimizations such as these can achieve modest but significant improvements (perhaps an order of magnitude) in the run-time of large problems without introducing significant offsetting penalties. More extreme methods, such as complete loop unrolling, can gain larger improvements, but only at very large costs in space; such a tradeoff may only make sense in the context of some unusual computer architectures.

Ultimately, any optimization that gains a constant factor, no matter how large, will be overwhelmed by the exponential growth of computing cost as we increase the size and complexity of the pedigree or of the genetic model. This will most likely lead to another tradeoff: giving up exact solutions and instead applying sampling techniques based on variants of Monte Carlo simulations. Nevertheless, a significant constant improvement can bring some interesting, slightly larger problems within range of available computational techniques.

References

[1] Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23:25–37, Dec, 1990.

[2] Gregory F. Cooper. Bayesian belief-network inference using nested dissection. Technical Report KSL-90-05, Stanford University, Knowledge Systems Laboratory, Stanford, California, January 1990.

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

[4] Nomi Harris. Probabilistic belief networks for genetic counseling. *Computer Methods and Programs in Biomedicine*, Accepted for publication.

[5] K. Lange and R. C. Elston. Extensions to pedigree analysis. *Hum. Hered.*, 25:95–105, 1975.

[6] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[7] Ross D. Shachter. Evaluating influence diagrams. *Operations Research*, 34:871–882, 1986.

[8] David J. Spiegelhalter. Fast algorithms for probabilistic reasoning in influence diagrams, with applications in genetics and expert systems. In *Conference on Influence Diagrams*, Berkeley, May 1988. University of California.