Conditioning Graphs: Practical Structures for Inference in Bayesian Networks

Kevin Grant[1] and Michael C. Horsch[1]

Dept. of Computer Science, University of Saskatchewan, Saskatoon, SK, S7N 5A9
kjg658@mail.usask.ca, horsch@cs.usask.ca

Abstract. Programmers employing inference in Bayesian networks typically rely on the inclusion of the model as well as an inference engine into their application. Sophisticated inference engines require non-trivial amounts of space and are also difficult to implement. This limits their use in some applications that would otherwise benefit from probabilistic inference. This paper presents a system that minimizes the space requirement of the model. The inference engine is sufficiently simple as to avoid space-limitation and be easily implemented in almost any environment. We show a fast, compact indexing structure that is linear in the size of the network. The additional space required to compute over the model is linear in the number of variables in the network.

1 Introduction

When programmers wish to use a Bayesian network in their applications, the standard approach is to store the entire network, as well as an inference engine to compute posteriors from the model. Algorithms based on junction-tree message passing [7] or variable elimination [5, 12] have a high space requirement and are difficult to code. Furthermore, application programmers not wishing to implement their own version of inference must import large general-purpose libraries. There are few algorithms which can be simply implemented in limited space.

To overcome some of these difficulties, Darwiche and Provan developed Query-DAGs [4]. A Query-DAG (or Q-DAG) is a data structure that represents the desired posterior probabilities as an arithmetic equation (in graphical form) parameterized by evidence variables. Computing probabilities involves setting the appropriate evidence variables, and updating the graph, which involves a minimal set of multiplications and summations. That is, the Bayesian network is "compiled out;" the result is a structure consisting only of node pointers, floating point numbers and boolean variables, easily implementable on any machine. The evaluator for Q-DAGs is a small set of rules composed of elementary computational operations, such as pointer referencing, arithmetic operations, and variable modification. Together with its evaluation engine, a Q-DAG is self-contained. However, the size of a Q-DAG may be exponential in the size of the network it is derived from.

The technique of *conditioning* [2, 8, 9] for probabilistic inference requires only linear space. However, cutset conditioning [9] requires an implementation of the

message-passing algorithm, which is non-trivial to program. Recursive conditioning [2] is an inference engine that is easy to implement. However, it lacks the desirable properties of Q-DAGs; namely, a run-time structure with the details relevant to a specific query compiled away.

To overcome this problem, we present *conditioning graphs*. Conditioning graphs combine the linear space requirements of conditioning with the simplicity of Q-DAGs. Its components consist of simple node pointers and floating point values; no high-level elements of Bayesian network computation are included. As well, the evaluator for conditioning graphs is very simple: evaluating each node requires a series of arithmetic operations over floating point values.

The remainder of this paper is structured as follows. Section 2 gives some necessary background, and introduces *elimination trees*, which are the basis for conditioning graphs. Section 3 presents conditioning graphs, and demonstrates their construction from elimination trees. Section 4 shows how to optimize a structure when application-specific information is known. Section 6 outlines current and future research.

2 Elimination Trees

We denote a random variable with capital letters (eg. X, Y, Z), and sets of variables with boldfaced capital letters $\mathbf{X} = \{X_1, ..., X_n\}$. Each random variable V has an associated domain $\mathcal{D}(V) = \{v_1, ..., v_k\}$. An instantiation of a variable is denoted $X = x$, or x for short. A *context*, or instantiation of a set of variables, is denoted $\boldsymbol{X} = \boldsymbol{x}$ or $\boldsymbol{x}$. Given a set of random variables $\boldsymbol{V} = \{V_1, ..., V_n\}$ with domain function $\mathcal{D}$, a Bayesian network is a tuple $\langle \boldsymbol{V}, \boldsymbol{\Phi} \rangle$. $\boldsymbol{\Phi} = \{\phi_{V_1}, ..., \phi_{V_n}\}$ is a set of distributions with a one-to-one correspondence with the elements of $\boldsymbol{V}$. Each $\phi_{V_i} \in \boldsymbol{\Phi}$ is the conditional probability of V_i given its parents in the network (called *conditional probability tables* or CPTs). That is, if π_{V_i} represents the parents of V_i, then $\phi_{V_i} = P(V_i|\pi_{V_i})$. A variable in a Bayesian network is said to be *conditionally independent* of its non-descendents given its parents. This allows the joint probability to be factorized as:

$$P(\boldsymbol{V}) = \prod_{i=1}^{n} P(V_i|\pi_{V_i}) \ . \tag{1}$$

Figure 1 shows an example of a Bayesian network, and the CPTs associated with each variable, which we use as a running example.

A common inference problem in Bayesian networks is to compute posterior probabilities, which is NP-hard [1]. However, several algorithms give tractable run-times in many cases. The class of algorithms of interest is conditioning, specifically, recursive decomposition. Recursive decomposition [3,8,11] partitions a network by conditioning on a subset of its variables (such a subset of variables is deemed a *cutset*). Each of these components can be decomposed again, until each component in the final product is a single variable (with its associated distribution). Figure 2(a) shows a recursive decomposition for the *Fire* example.

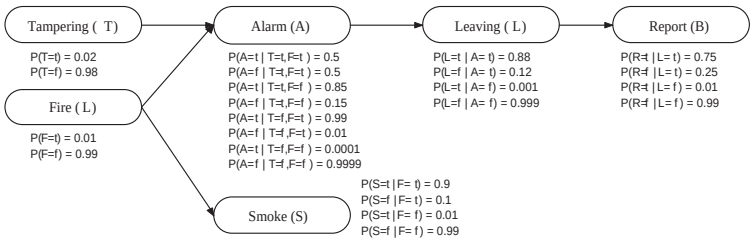
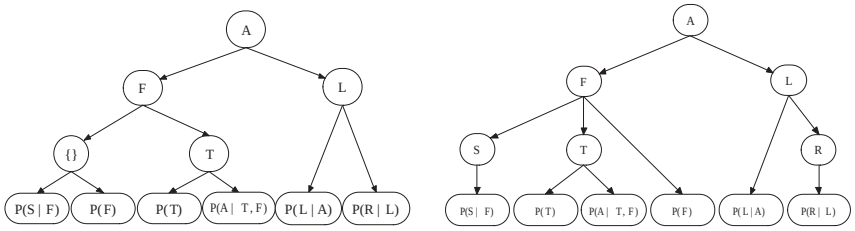


Fig. 1. The *Fire* Bayesian network (taken from Poole et al. [10])



(a) A recursive decomposition of the *Fire* network.

(b) An elimination tree for the *Fire* network.

Fig. 2. Two decompositions of the *Fire* network.

We begin by introducing elimination trees, which are another type of recursive decomposition. An *elimination tree* is a tree whose leaves and internal nodes correspond to the CPTs and variables of a Bayesian network, respectively. The tree is structured such that all CPTs containing variable V_i in their domain are contained in the subtree of the node labeled with V_i. Figure 2(b) shows a possible elimination tree for the *Fire* network.

There are two primary differences between elimination trees and other recursive decompositions (such as recursive conditioning [3]). The first is that the size of a cutset at any variable is restricted to exactly one variable. The other difference is that each variable of the Bayesian network must appear in a cutset, whereas leaf variables typically do not appear in the cutsets of recursive conditioning structures. These restrictions allow for simple low-level implementation, which is one of the goals of this project.

Elimination trees have a close correspondence with elimination algorithms [5, 12]. The algorithm for building an elimination tree parallels variable elimination, where an internal node represents the marginalization of its variable label, and the children of the node represent the distributions that would be multiplied together. Thus, an internal node is labeled with a variable, but represents a distribution. Figure 3 gives a simple algorithm for constructing an elimination tree from a Bayesian network $\langle \boldsymbol{V}, \boldsymbol{\Phi} \rangle$. In the algorithm, we use $dom(T)$ to represent the union of all CPT domains from the leaves of T's subtree.

Notice that the algorithm in Figure 3 returns a set of trees, rather than a single tree. In the event that the network is not connected, the number of disconnected components will correspond to the number of trees returned by *elimtree*. For the following discussion, we consider the case where the elimination tree is a single tree. Cases where multiple trees occur are examined in Section 4.

```
elimtree(⟨V, Φ⟩)
    T ← {}
    for each φ ∈ Φ do
        Construct a leaf node T_φ containing φ
        Add T_φ to T
    for each V_i ∈ V do
        Select the set T_i = {t ∈ T | V_i ∈ dom(t)}
        Remove T_i from T
        Construct a new internal node t_i whose children are T_i
        Label t_i with V_i, and add it to T
    return T
```

Fig. 3. The code for generating an elimination tree from a Bayesian network.

To calculate probabilities from an elimination tree, we define algorithm $\mathcal{P}$ (see Figure 4). $\mathcal{P}$ takes as parameters a node from an elimination tree and a context, and returns a distribution. We use the following notation: if T is a leaf node, then ϕ_T represents the CPT at T. If T is an internal node, V_T represents the variable labeling T, and ch_T represents its children.

```
P(T, c)
    if T is a leaf node
        return φ_T(c)
    elseif V_T is instantiated in c
        Total ← 1
        for each T' ∈ ch_T
            Total ← Total * P(T', c)
        return Total
    else
        Total ← 0
        for each v_T ∈ dom(V_T)
            Total ← Total + P(T, c ∪ {v_T})
        return Total
```

Fig. 4. The code for processing an elimination tree given a context.

The following theorem specifies the relationship between the probabilities of interest and the algorithm $\mathcal{P}$. Its correctness follows from the correctness of the

other recursive decomposition algorithms. A proof of the theorem can be found in Grant and Horsch [6].

Theorem 1. *Given a Bayesian network* $\langle \boldsymbol{V}, \boldsymbol{\Phi} \rangle$ *and an associated elimination tree* T*:*

$$P(x_q|\boldsymbol{c}) = \alpha \mathcal{P}(T, \{x_q\} \cup \boldsymbol{c}) \tag{2}$$

where $\alpha = P(\boldsymbol{c})^{-1}$ *is a normalization constant.*

The major advantage of recursive decompositions (and conditioning in general) is the linear space property of the algorithm. It is summarized in the following theorem, whose proof is also found in Grant and Horsch [6].

Theorem 2. *Given a Bayesian network and an corresponding elimination tree* T*,* $\mathcal{P}(T, \boldsymbol{C} = \boldsymbol{c})$ *makes* $\boldsymbol{O}(n \exp(d))$ *recursive calls and requires* $\boldsymbol{O}(d)$ *space, where* d *is the height of the elimination tree.*

Theorem 2 demonstrates the relationship between the depth of the tree and the complexity of the algorithm $\mathcal{P}$. The depth of the tree is a consequence of the order in which the variables are selected from the *elimtree* algorithm. Choosing an ordering that optimizes the depth of the tree is an open problem.

There are several optimizations that can be made to this structure. However, we first consider some implementation details for elimination trees - one that provides minimal indexing. Further optimization will be considered in subsequent sections.

3 Conditioning Graphs

In this section, we will give a low-level representation for a Bayesian network as an elimination tree, and a compact efficient implementation of the algorithm $\mathcal{P}$.

We implement $\mathcal{P}$ as a depth-first traversal. When we reach a leaf node, we need to retrieve the parameter that corresponds to the context. To do this, we assume that each CPT is stored as a linear array of parameters. Indexing a CPT assumes an ordering of its variables and the domain values of each variable. Let $\{C_1, ..., C_k\}$ be an ordering of the variables in a CPT. C_i is the ith variable in the ordering, and c_i is an integer specifying the $c_i th$ value of C_i's domain, where $0 \leq i < m$. We calculate the index of a context $\{c_1, ..., c_k\}$ as follows:

$$index(c_1, ..., c_k) = \sum_{i=1}^{k} c_i m^{k-i} \ . \tag{3}$$

A more efficient version of this function is the Horner form of the polynomial:

1. $index([]) = 0$
2. $index([c_1, ..., c_i]) = c_i + m(index([c_1, ..., c_{i-1}]))$

For any given ordering of the variables, we can index into a CPT using this function. If we choose an ordering that is consistent with the path from root to leaf in the elimination tree, then we can index the CPTs as the context is constructed, as we traverse the tree. However, to make the associations between variables and distributions, we require a second set of arcs at each internal node, referred to here as *secondary pointers* (call the original pointers *primary pointers*). The secondary arcs are added according to the following rule: *there is an arc from an internal node A to leaf node B iff the variable X associated with A is contained in the domain of the CPT associated with B*. The number of secondary arcs emitting from a node with variable V is equivalent to $|ch_V| + 1$, where ch_V refers to the number of arcs emitting from V in the Bayesian network. Cumulatively, the number of secondary arcs in the entire structure is $e+n$, where e is the number of arcs in the original network.

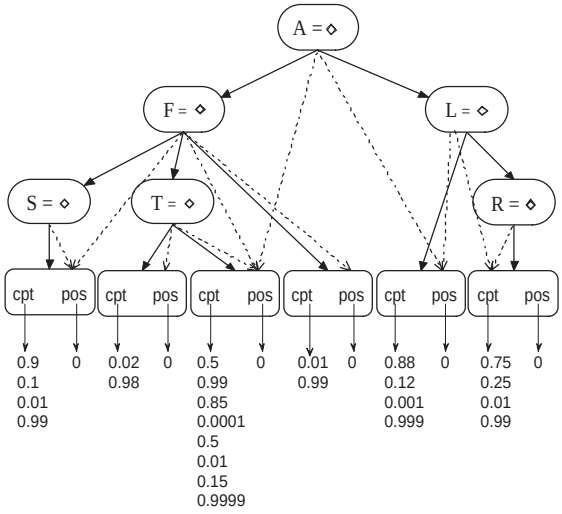


Fig. 5. The conditioning graph.

An example of the final structure is shown in Figure 5. We refer to this structure as a *conditioning graph*, as the secondary arcs destroy its tree properties. Note that at each leaf, we store the CPT as an array of values, and the index as an integer variable, which we call *pos*. In each internal node, we store a set of primary pointers (from the elimination tree), a set of secondary pointers, and an integer representing the current value of the node's variable.

We maintain one global context over all variables, denoted as $\boldsymbol{g}$. Each variable V_i is instantiated in $\boldsymbol{g}$ to a member of $\mathcal{D}(V_i) \cup \{\diamond\}$. The symbol $\diamond$ (borrowed from Darwiche and Provan [4]) is a special symbol that means the variable is unobserved. Initially, all nodes are assigned $\diamond$ in $\boldsymbol{g}$, as no variables have been instantiated. To calculate $P(E_1 = e_1, ..., E_k = e_k)$, we set $E_i = e_i$ in $\boldsymbol{g}$ for

$i = 1$ *to* k. While performing the algorithm, when conditioning a node to $V_i = v_i$, we set $V_i = v_i$. To 'uncondition' the variable (after conditioning on all values from its domain), we set $V_i = \diamond$ in $\boldsymbol{g}$.

Figure 6 shows an implementation of $\mathcal{P}$. Note that we use dot notation to refer to the members of the variables. For a leaf node N, we use $N.cpt$ and $N.pos$ to refer to the CPT and its current index, respectively. For an internal node N, we use $N.primary$, $N.secondary$, and $N.value$ to refer to the variables primary children, secondary children, and variable value, respectively. The member *value* at each internal node can also represent the input from the programmer. To set evidence $V = v_i$, the programmer would have to set $N.value$ to the appropriate value for the node N labeled with variable V.

```
Query(N)
    if N is a leaf node
        return N.cpt[N.pos]
    else if N.value ≠ ◇
        for each S' ∈ N.secondary do
            S'.pos ← S'.pos * m + N.value
        Total ← 1
        for each P' ∈ N.primary do
            Total ← Total * Query(P')
        for each S' ∈ N.secondary do
            S'.pos ← S'.pos/m
        return Total
    else
        Total ← 0
        for i ← 0 to m − 1 do
            N.value = i
            Total ← Total + Query(N)
        N.value = ◇
        return Total
```

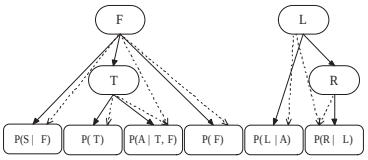
Fig. 6. The process algorithm.

The algorithm assumes that all variables are of size m. Extending conditioning graphs to variables of various sizes is easily accomplished with a little extra storage. If a node stores the size of its variable (as an integer value *size*) then we can replace all instances of m with $N.size$ in the algorithm, and it can handle multi-sized variables. A more interesting case is when the variables have sizes that are powers of two (eg. binary models). In this case, our multiplications and divisions become shift operations, which is much more efficient. In fact, if our secondary pointers can refer directly to their corresponding bits in the indexing variables, then shifting becomes unnecessary altogether, as does the requirement that the entries in the CPTs be ordered according to the global ordering (although they must be ordered according to some ordering). This optimization could be useful in some particular hardware implementations.

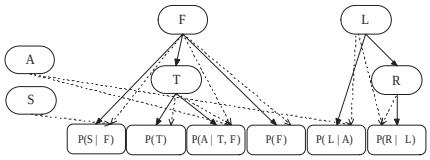
4 Sensor Models

It is well known that one can condition a Bayesian network on the evidence before performing inference. This reduces network connectivity, resulting in smaller cutset widths, and eliminates the evidence nodes from the CPTs, resulting in fewer marginalizations. If we know that some set of variables will always be observable, we can likewise modify the conditioning graph to be more efficient. This is a realistic situation: in any application, there typically exists at least a small subset of variables that are always observable. Examples of these include monitor output in medical patient monitoring, and sensor values in car diagnosis. We refer to these variables that can always be observed as *sensor variables*.

Considering the *Fire* model, suppose we know in advance that we will always be able to observe the state of the fire alarm, and whether or not there is smoke present (both are easily accomplished using sensors). Hence, our set of sensor variables is $\boldsymbol{E} = \{S, A\}$. We construct the elimination tree by removing $\boldsymbol{E}$ from the set of variables, and building the elimination tree over the variables that remain; all the CPTs are included in the tree. Essentially, this constructs a tree that does not marginalize S or A. See Figure 7(a). A conditioning graph is constructed from the elimination tree as before, with secondary arcs from each internal node to the appropriate leaf nodes. As well, the variables in $\boldsymbol{E}$ are included in the conditioning graph, with secondary arcs pointing to the appropriate leaf nodes, but they are not connected to the tree structure with any primary arcs. Figure 7(b) shows the resulting structure.



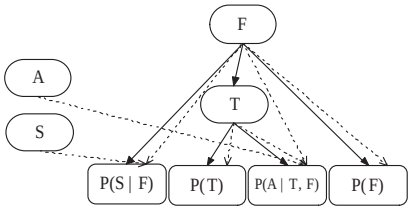
(a) The new conditioning graph. Note that *Alarm* and *Smoke* are never marginalized.



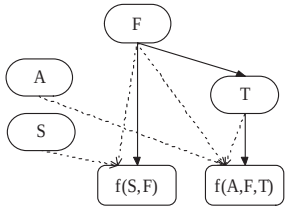
(b) The new conditioning graph, with nodes for *Alarm* and *Smoke*.

Fig. 7. The new conditioning graph, utilizing the evidence optimization. Note that for space consideration, we use the CPT notation, rather than listing the array of values explicitly.

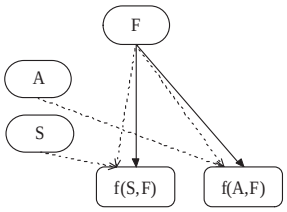
There are definite benefits to this separation of the evidence nodes from the conditioning graph. Leaving $\boldsymbol{E}$ out of the elimination tree may result in several distinct trees, each of which is smaller than if they were included. Computing $P(x_q|\boldsymbol{e})$ only requires processing the component containing X_q in its nodes. Thus, even though our conditioning graph is static at run-time, we are able to "prune" away irrelevant parts of the model during compilation. Note that this requires a pointer from each variable X_q to its corresponding elimination tree,



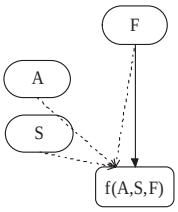
(a) The conditioning graph, leaving out *Report* and *Leaving*.



(b) The conditioning graph, with leaf nodes for each internal node compacted.



(c) The conditioning graph, with *Tampering* marginalized out.



(d) The final conditioning graph.

Fig. 8. Optimizing the conditioning graph.

but these pointers require only linear space to store. There are other advantages. Reducing the conditioning graph by leaving out the observable variables may reduce its depth, which can produce exponential speedup when computing probabilities. Plus, as long as the evidence remains the same, we need only process the relevant elimination tree to handle multiple queries.

5 Query Variables

In variable elimination, it is well known that eliminating barren variables can improve the time it takes to process a query. Also, any nodes in the Bayesian network that are d-separated from the query can be removed. These optimizations can also be used in conditioning graphs, if it is known in advance which variables will be queried, and which variables will be evidence variables. For example, if we knew that variables *Report* and *Leaving* would never be queried or observed, then that portion of the network need not even be stored. This eliminates approximately 33% of the space required for storage. Figure 8(a) shows the new structure.

If an internal node in a conditioning graph has several leaf nodes, the distributions can be multiplied at compile time, and the single distribution made the only child of the node. This will reduce the number of multiplications during inference, but has the potential to increase the space requirement of the problem. Thus it should only be performed if this increase in size is acceptable. On the other hand, it is possible that this operation may decrease the space required to store the conditioning graph.

From our previous example, we see that the internal node associated with *Tampering* has two leaf nodes, whose CPTs correspond to $P(A|T,F)$ and $P(T)$. Multiplying these two CPTs produces a factor over $\{A,T,F\}$, with 8 values. This operation does not add to the space requirements (in fact, it reduces them). Similarly, the node for *Fire* has two leaf nodes that can be multiplied with the same effects. Figure 8(b) shows the conditioning graph after these two optimizations are performed. Note that the size of the network and the number of operations necessary has been reduced.

We can take this optimization one step further considering that we know of variables that will never be observed or queried. If a subtree in the elimination tree contains only variables that will never be queried or observed, then we can compact that subtree into a single leaf node at compile time. This amounts to doing partial elimination, before we condition, and storing an intermediate distribution, rather than all CPTs from the original network. Once again, this step has the potential to increase the space requirements of the conditioning tree. However, we can calculate the size of the leaf node without actually performing the computation. This allows us to decide beforehand whether such an absorption is acceptable given our current size restrictions.

Continuing with the example, suppose that the need to query the *Tampering* node is now eliminated, and assume that it is not observable. Hence, we can multiply all of its children (there's only one in this example), and marginalize out the tampering variable. Figure 8(c) shows the system after we perform this step. Note that we have reduced the depth of the tree, decreasing our complexity by a factor of m (2, in this case). As well, the *Fire* variable now has two leaf nodes, that can be compacted without increasing the space complexity. Figure 8(d) shows the final product, an extremely small, efficient version of the original problem. In fact, we have reduced it to a simple lookup, given the values of the evidence and query. Note that such a reduction is not always possible, but it can reduce considerable portions of the network given the right variable ordering.

6 Conclusions and Future Work

This paper presents conditioning graphs, a low-level representation of inference in Bayesian networks. Conditioning graphs allow for Bayesian computation without storing the original model, or a large inference algorithm. We demonstrate their construction and operation, generate complexity results for their operation, and elicit some optimizations to improve their performance.

The system described in this paper is a generalization of conditioning over an elimination tree. Our work is inspired by the work on recursive conditioning [3] and adaptive conditioning [11]. However, rather than storing the original model and a complex inference engine, the abstraction converts the network to a very simple structure that allows us to compute posterior probabilities using a very simple, very small algorithm. The space required for storage and inference (over and above the storage for the parameters of the Bayesian network) is linear in the size of the model.

This paper presents the preliminary stages of this research project. Orderings considered good for standard inference are not necessarily good for the conditioning graphs. For example, consider a Bayesian network that is a chain of n variables. Summing out the variables in the order of the chain represents an optimal ordering in standard inference, but it represents the worst-case time complexity for the conditioning graph. Finding an optimal variable ordering is NP-hard, and the research community resorts to heuristics in many cases. Perhaps the nature of the problem may suggest heuristics that are tailored towards shallow conditioning graphs.

In our model, many values may be recomputed several times. Darwiche has demonstrated in his dtrees how simple caching procedures can increase the time performance of the algorithms exponentially, if the space is available. The caching procedure is somewhat involved, so porting it directly to conditioning graphs is not obvious, but marks an available area of research.

References

1. G. F. Cooper. The computational complexity of probabilistic inference using Bayesian Inference. *Artificial Intelligence*, 42:393–405, 1990.
2. A. Darwiche. Any-space probabilistic inference. In *Proceedings of the Sixteenth Conference on Uncertainty and Artificial Intelligence*, pages 133–142, 2000.
3. A. Darwiche. Recursive Conditioning: Any-space conditioning algorithm with treewidth-bounded complexity. *Artificial Intelligence*, pages 5–41, 2000.
4. A. Darwiche and G. Provan. Query dags: A practical paradigm for implementing belief network inference. In *Proceedings of the 12th Annual Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 203–210, San Francisco, CA, 1996. Morgan Kaufmann Publishers.
5. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.
6. K. Grant and M. Horsch. Conditioning Graphs: Practical Structures for Inference in Bayesian Networks. Technical Report 2005-04, Dept. of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada, 2005.
7. S. Lauritzen and D. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50:157–224, 1988.
8. S. Monti and G. F. Cooper. Bounded recursive decomposition: a search-based method for belief-network inference under limited resources. *Int. J. Approx. Reasoning*, 15(1):49–75, 1996.
9. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.
10. D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence* . Oxford University Press, 1998.
11. F. Ramos, F. Cozman, and J. Ide. Embedded Bayesian Networks: Anyspace, Anytime Probabilistic Inference. In *AAAI/KDD/UAI Workshop in Real-time Decision Support and Diagnosis Systems*, 2002.
12. N. Zhang and D. Poole. A Simple Approach to Bayesian Network Computations. In *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.