Exploiting Dynamic Independence in a Static Conditioning Graph

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

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

Abstract. A conditioning graph (CG) is a graphical structure that attempt to minimize the implementation overhead of computing probabilities in belief networks. A conditioning graph recursively factorizes the network, but restricting each decomposition to a single node allows us to store the structure with minimal overhead, and compute with a simple algorithm. This paper extends conditioning graphs with optimizations that effectively reduce the height of the CG, thus reducing time complexity exponentially, while increasing the storage requirements by only a constant factor. We conclude that CGs are frequently as efficient as any other exact inference method, with the advantage of being vastly superior to VE and JT in terms of space complexity, and far simpler to implement.

1 Introduction

Recently, we proposed *conditioning graphs* (CGs) which are runtime representations of belief networks [6]. CGs have a number of important properties. First, they require only linear space, in terms of the size of the original network, whereas a join tree for example, requires space that is exponential in the width of the network. Second, a CG consists of simple node pointers and floating point values; no high-level elements of belief network computation are included. As well, inference algorithms for conditioning graphs are small recursive algorithms, easily implementable on any architecture, without requiring monolithic runtime libraries, or worse, the implementation of complex inference techniques such as variable elimination [4, 15] or junction tree propagation [9].

Conditioning graphs are a form of recursive factorization of belief networks. Recursive decomposition [10] and recursive conditioning [2] restrict the number of children at each internal node to two, and no restriction is made on the number of variables at each internal node. In contrast, conditioning graphs have exactly one variable at each internal node, and no restriction on the number of children. This difference simplifies the implementation of inference substantially.

Conditioning graphs are also related to Query-DAGs [3] in which simple formulae are precomputed and stored as DAGs. The evaluation engine for this approach is very lightweight, reducing system overhead substantially. However, the size of a Q-DAG may be exponential in the size of the network.

Inference in belief networks allows the calculation of posterior probabilities while considering only essential information. Any information deemed irrelevant to the current query is ignored by certain inference algorithms (such as Variable Elimination (VE) [15]). Such pruning can provide enormous efficiency gain in application, both space and time-wise. The complexity of pruning is linear on the size of the network model, making it fast in comparison to inference.

Because precompiled structures like conditioning graphs must be general enough to allow any query, they do not inherently exploit the use of variables that are irrelevant for a given query. In previous work, we exploited certain domain-dependent observation variables for faster calculation [6]. In this paper, we show how to exploit irrelevant variables for a given query. We show that with a small amount of additional memory, we can achieve exponential speedup for inference using conditioning graphs. In some cases, the time complexity is very competitive with other exact methods such as VE and JTP, with the advantage of requiring only linear space, and being very simple to implement.

The remainder of this paper is as follows. Section 2 reviews conditioning graphs and their methods. Sections 3 and 4 present two improvements to the basic algorithm. Section 5 shows empirical analysis of these improvements over some well-known networks. Section 6 summarizes current and future research.

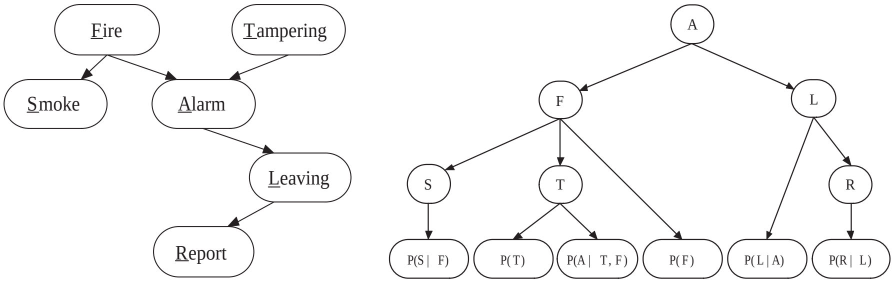
2 Elimination Trees and Conditioning Graphs

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 X has an associated domain of size m_X, and can be assigned a value or *instantiated*. An instantiation of a variable is denoted $X = x$, or x for short, where $x \in \{0, ..., m_X - 1\}$. A *context*, or instantiation of a set of variables, is denoted $\boldsymbol{X} = \boldsymbol{x}$ or $\boldsymbol{x}$.

An *elimination tree* [6] over a belief network is a tree in which each leaf corresponds to a conditional probability table (CPT) in the network, and each non-leaf corresponds to a random variable from the network. The tree is structured such that for any non-leaf node N in the tree, the variable at N and its ancestor variables in the tree d-separate all variables of one subtree directly below N from all variables in another subtree below N. An elimination tree can be derived from an elimination ordering using a modified version of elimination [15] (see Grant & Horsch [6, 7] for details). Figure 1(b) shows the elimination tree for the $Fire$ example, shown in Figure 1(a)

An algorithm for computing probabilities in elimination trees is presented in Figure 2. At each internal node T, we condition over its variable (denoted by V_T), unless it is observed. To compute probability $P(\boldsymbol{e})$ from elimination tree T, we call $\mathcal{P}(\mathcal{T}, \boldsymbol{e})$. The context is extended as the tree is traversed in a depth-first manner, and when a leaf node T is reached, its CPT (denoted by ϕ_T) is indexed by that context.

A conditioning graph [6] is a low-level representation of an elimination tree. The abstract algorithm in Figure 2 is given a compact efficient implementation



(a) The fire network [12].

(b) An elimination tree for the fire network.

Fig. 1. Elimination tree construction.

```
$\mathcal{P}(T, \boldsymbol{c})$
if $T$ is a leaf node
    return $\phi_T(\boldsymbol{c})$
elseif $V_T$ is instantiated in $\boldsymbol{c}$
    $Total \leftarrow 1$
    for each $T' \in ch_T$
        $Total \leftarrow Total * \mathcal{P}(T', \boldsymbol{c})$
    return $Total$
else
    $Total \leftarrow 0$
    for each $v_T \in dom(V_T)$
        $Total \leftarrow Total + \mathcal{P}(T, \boldsymbol{c} \cup \{v_T\})$
    return $Total$
```

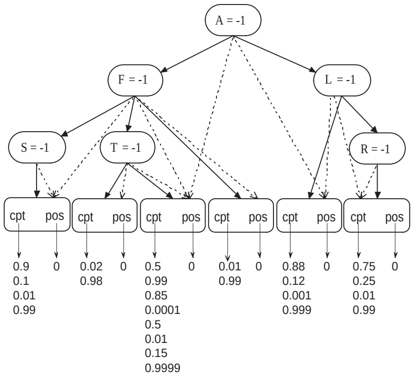
Fig. 2. Algorithm $\mathcal{P}$, for processing an elimination tree given a context.

in terms of conditioning graphs, and primitive computational operations such as arithmetic and pointer manipulation.

An example of a conditioning graph is shown in Figure 3(a). Note that at each leaf, we store the CPT as an array of values, and an index as an integer variable, which we call *pos*. In each internal node, we store a set of primary arcs, a set of secondary arcs, and an integer representing the current value of the node's variable. The primary arcs are used to direct the recursive computation, and are obtained from the elimination tree. The secondary arcs are used to make the associations between variables in the graph and the CPTs that depend on them. 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 definition of the CPT associated with B.*

We implement $\mathcal{P}$ as a depth-first traversal. When we reach a leaf node, we need to retrieve the CPT parameter that corresponds to the context. To do this, we store each CPT as a linear array of parameters, as follows. Let $\{C_1, \cdots, C_k\}$ be the variables of the CPT ϕ, ordered according to the order of their depth in the tree. The index of $\phi(c_1, \cdots, c_k)$ is calculated as follows:

$$\begin{aligned} index([]) &= 0 \\ index([c_1, \cdots, c_k]) &= c_k + m_k \times index([c_1, \cdots, c_{k-1}]) \end{aligned} \tag{1}$$



(a) Conditioning graph.

```
Query(N)
if $N$ is a leaf node
    return $N.cpt[N.pos]$
else if $N.value \neq \diamond$
    for each $S' \in N.secondary$ do
        $S'.pos \leftarrow S'.pos * N.m + N.value$
    $Total \leftarrow 1$
    for each $P' \in N.primary$ do
        $Total \leftarrow Total * Query(P')$
    for each $S' \in N.secondary$ do
        $S'.pos \leftarrow S'.pos / N.m$
    return $Total$
else
    $Total \leftarrow 0$
    for $i \leftarrow 0$ to $N.m - 1$ do
        $N.value \leftarrow i$
        $Total \leftarrow Total + Query(N)$
    $N.value \leftarrow \diamond$
    return $Total$
```

(b) Algorithm.

Fig. 3. Conditioning graph of the *Fire* example and the algorithm for computing probabilities from it.

where m_i is the size of the domain of variable C_i. By choosing an ordering that is consistent with the path from root to leaf in the elimination tree, we can compute the CPT's index as the context is constructed, that is, while we traverse the tree.

Inference in a CG consists of summing out 'hidden' variables. Variables that are either being queried or used as evidence are instantiated in advance of calling $\mathcal{P}$. To do this, we maintain one global context over all variables, denoted $\boldsymbol{g}$. 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 [3]) is a special symbol that means the variable is unobserved (we use -1 in our implementation). Initially, all nodes are assigned $\diamond$ in $\boldsymbol{g}$, as no variables have been instantiated. To calculate $P(E_1 = e_1, \cdots, E_k = e_k)$, we set $E_i = e_i$ in $\boldsymbol{g}$ for $i = 1 \dots k$. While performing the algorithm, when conditioning a node to $V_i = v_i$, we set $V_i = v_i$. To reset the variable (after conditioning on all values from its domain), we set $V_i = \diamond$ in $\boldsymbol{g}$.

Figure 3(b) shows *Query*, the final low-level implementation of $\mathcal{P}$. We use dot notation to refer to the data 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*, *N.value*, and *N.m* to refer to the node's primary children, secondary children, variable value, and variable size, respectively. The variable's value represents the evidence, if any. To set the evidence $V = v_i$, the application would set *V.value* $= i$. It is assumed that a constant-time mapping exists between the variable and the node that contains it: such a mapping can be constructed during compilation of the graph.

To avoid confusion regarding the notions of parents and children in the various graphs and trees, we refer to the parents (children) of a variable in the belief network as its *network parents (children)*, while those in the conditioning graph will be *graph parents (children)*.

3 Optimizing indexing

Conditioning graphs index CPTs as variables are instantiated using a depth-first traversal. For each variable that has been observed or conditioned, the indices for its CPTs (linked through secondary pointers) are updated (Line 4 and 5 of the *Query* algorithm). These values must be unset once the child values have been calculated (Line 9 and 10 of the *Query* algorithm). This linear-time indexing occurs once for each time the node is visited; the number of times a variable is visited in exponential in the depth of the variable in the elimination tree. This approach is simple to implement, but inefficient. We can dramatically improve the efficiency of indexing by precomputing some of parameters involved, at a small cost in terms of memory.

The function *index* takes a context over the variables of a CPT and returns a unique index for that context's entry in the CPT. We showed *index* in its Horner form (Equation 1), but we can also represent it as a linear function over its parameters. Let $M_i = \prod_{j=i+1}^{k} m_j$. This means that $index(c_1, \cdots, c_k) = \sum_{i=1}^{k} c_i M_i$. The cardinality of a variables never changes during inference, so M_i is a constant that can be calculated during the compilation of the conditioning graph. The commutativity of addition means that we can add the terms in the above equation in any order. Consequently, evidence values can be determined and their effect on the indexing computation is independent of any query. Furthermore, evidence only needs to be set once. This is in contrast to the original algorithm, where the evidence was factored into the index when an evidence variable was visited in the traversal, and evidence variables were reset when the traversal of the subtree was completed. Hence, the number of times the evidence is set and reset reduces from exponential to constant (per query). This decrease in the number of operations is exponential on the height of the tree, although this is not evident in terms of asymptotic complexity. If the evidence remains unchanged over multiple queries, then the savings propagates over these queries as well.

Figure 4 gives the new algorithm for updating evidence, and querying the graph. We represent the scalar value between a node N and a respective secondary child S using the function $scalar(N, S)$. Notice that the query algorithm does not compute over the secondary links for an observed variable.

4 Relevant Variables

When computing a posterior probability, the variables in the belief network can be classified into three sets.

1. The *query* variables, including the variable over which a posterior distribution is to be computed, as well as all the evidence variables.
2. The *relevant* variables, whose CPTs must be included.
3. The *irrelevant* variables, whose CPTs may be safely left out.

The irrelevant variables include *barren* variables [13] and *d-separated* variables [5]. Barren variables are variables whose marginalization would produce intermediate distributions full of 1s. Barren variables often comprise a considerable

```
$SetEvidence(N, i)$
    $diff \leftarrow i - N.value$ {$\diamond = 0$ in this equation}
    for each $S' \in N.secondary$ do
        $S'.pos \leftarrow S'.pos + scalar(N, S') * diff$
    $N.value \leftarrow i$

$Query2(N)$
    if $N$ is a leaf node
        return $N.cpt[N.pos]$
    else if $N.value <> \diamond$
        $Total \leftarrow 1$
        for each $P' \in N.primary$ do
            $Total \leftarrow Total * Query2(P')$
        return $Total$
    else
        $Total \leftarrow 0$
        for $i \leftarrow 0$ to $N.m - 1$ do
            $SetEvidence(N, i)$
            $Total \leftarrow Total + Query2(N)$
        $SetEvidence(N, \diamond)$
        return $Total$
```

Fig. 4. Algorithms for setting evidence and querying, given that secondary scalar values are used.

portion of the belief network, especially when the observations and queries are localized to a particular section of the network, and even more so when those observations/queries are shallow (closer to the root than the leaves). D-separated variables are variables in the belief network that are irrelevant to the current query given the current evidence. These variables can also be ignored.

Finding barren and d-separated variables requires traversal through the belief network, but the conditioning graph does not store the belief network in a convenient manner for this. Two possibilities are immediately apparent:

1. At each node, store two tertiary sets of pointers, that correspond to the original belief network. That is, node N storing variable V would have two sets, *parents* and *children*, that point to the nodes containing V's network parents and network children, respectively.
2. Make the secondary arcs bi-directional. In other words, each leaf node in the conditioning graph stores pointers up to its variables in the conditioning graph. As the leaf node stores a CPT for a variable V, and a CPT represents a relationship between V and its network parents, every leaf node has a distinguished arc to V (called a root arc), and a set of pointers to V's network parents (a non-root arc).

Tertiary pointers are more intuitive, and require only one step to traverse to a neighbour (rather than the two step process of traversing to a tree leaf first). However, including tertiary pointers is more space-expensive than making existing secondary arcs bidirectional. In a highly connected graph, the difference can be substantial. For simplicity, we will use the first option, but the algorithms are easily modified to use the second option if space is limited.

There exist several algorithms for finding nodes that are relevant to the query. One of the more recent ones, the Bayes-ball algorithm [14], finds both d-separated and barren variables simultaneously, and is a very good choice. However, since barren variables can be identified prior to the query, our algorithm performs these tasks separately: first non-barren variables are identified, and from these, the set of dependent variables are found.

The simplest definition of a barren variable is recursive: a variable in a belief network is barren if (a) it is not observed or part of the query and (b) either it is a leaf node, or all of its children are barren. For our algorithm, we maintain the collection of non-barren variables dynamically, as follows: whenever a barren variable becomes observed (or part of a query), then it becomes non-barren, and notifies its network parents of its non-barren state. This process continues in a recursive manner. Conversely, when a non-barren variable becomes unobserved, it checks whether or not its children are all barren. If they are, it becomes barren, and notifies its parents of its barren-ness. To accomplish this in a timely fashion, each internal node in the conditioning graph maintains an integer, *nonbarren*, that represents the number of *nonbarren* children that variable has in the network. When a variable becomes non-barren, it notifies its network parents, which update their *nonbarren* status by incrementing it. The opposite process occurs when a non-barren node becomes barren. A variable is barren if it is not observed and its *nonbarren* value is 0. Figure 5 shows $SetEvidence2$, our new evidence entry method that maintains barren variables. Note that $SetEvidence2$ is called whenever the observed value of a variable changes, independent of any query.

```
SetEvidence2($N, i$)
    SetEvidence(N, i)
    if $i \neq \diamond$
        ResetBarren(N)
    else
        SetBarren(N)

ResetBarren($N$)
    if $N.barren = true$
        $N.barren \leftarrow false$
        for each $Pa \in N.parents$ do
            $Pa.nonbarren \leftarrow Pa.nonbarren + 1$
            ResetBarren(Pa)

SetBarren($N$)
    if $N.barren = false$ AND $N.nonbarren = 0$ AND $N.value = \diamond$
        $N.barren \leftarrow true$
        for each $Pa \in N.parents$ do
            $Pa.nonbarren \leftarrow Pa.nonbarren - 1$
            SetBarren(Pa)
```

Fig. 5. Algorithm for setting the evidence, maintaining labeling of barren nodes.

From the set of nonbarren variables, we can select the relevant information. The relevant information of a query in a belief network is information that is not independent of the query; it is not *d-separated* from the query [11]. Space precludes a detailed discussion on d-separation, however, it suffices to say that a query is *dependent* on a variable if there exists at least one (undirected) path between the query and that variable that is not blocked by the evidence.

A variable is *relevant* if its local distribution is relevant to the query. Given that all non-barren nodes have been identified, relevant variables can be identified recursively (we assume that a query variable is not observed), using the rules of d-separation [11]:

1. A query variable is marked as a relevant variable.
2. An unmarked barren variable is marked as an irrelevant variable.
3. Given a relevant variable, its unmarked, unobserved parents are relevant.
4. Given a relevant unobserved variable, its unmarked children are relevant.

It must be noted that the above definition of a relevant variable only applies if the barren variables are identified. This simple recursive definition allows us to write a depth-first search algorithm for marking the relevant nodes. This algorithm, *SetRelevant*, is given in Figure 6. To identify relevant variable, a boolean value *relevant* is attached to each node, and is given the value *true* for each graph node which contains a relevant variable.

```
SetRelevant(N)
    for each node N' in the conditioning graph
        N'.relevant ← N'.active ← false
    MarkRelevant(N,N)

MarkRelevant(N, Q)
    N.relevant ← true
    MarkActive(N.root)
    for each P ∈ N.pa s.t. P.barren = false AND P.relevant=false AND P.value= ◇ do
        MarkRelevant(P, Q)
    if N = Q OR N.value ≠ ◇
        for each C ∈ N.ch s.t. C.barren=false AND C.relevant=false do
            MarkRelevant(C, Q)

MarkActive(N)
    N.active ← true
    if N.parent.active = false
        MarkActive(N.parent)
```

Fig. 6. The *SetRelevant* algorithm, which marks the active part of the conditioning graph for processing a particular query.

In addition to marking the relevance of each node, we need to mark the active paths through the conditioning graph. A leaf node is active if the query is dependent on its CPT. An internal node is active iff (a) the query is dependent on its variable or (b) it has a dependent primary child. Only the active nodes are traversed, the rest are ignored. In addition, the active nodes that are not dependent are treated as observed nodes: they are not conditioned over, they only combine results from their active children. We identify each active node in the conditioning graph by setting a value *active=true*. We use the *MarkActive* algorithm in Figure 6 to mark the active nodes in the graph as we identify relevant information. Note that *MarkActive* requires that each node N have a pointer to its parent node, which we identify as *N.parent* in the algorithm. As well, we denote N's root arc (described previously) as *N.root*.

Given that we have marked the active and relevant nodes in the conditioning graph (that is, we have called *SetRelevant* on the query node), *Query3* in Figure 7 shows the new query algorithm. The new query algorithm only traverses the active part of the network. It only conditions over relevant nodes. Each node now additionally stores pointers to the nodes containing its network parents and children, and maintains *nonbarren*, *relevant*, and *active* flags. These additions cumulatively contribute a constant factor to the current network storage.

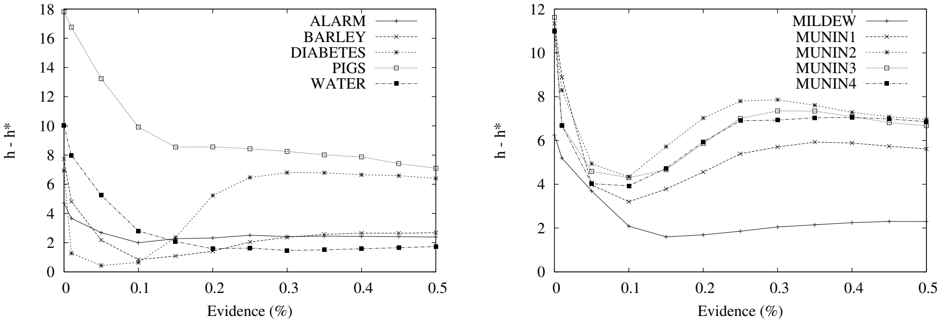
```
Query3(N)
1.    if $N$ is a leaf node
2.       return $N.cpt[N.pos]$
3.    else if $N.value \neq \diamond$ OR $N.relevant = false$
4.       $Total \leftarrow 1$
5.       for each $P' \in N.primary$ s.t. $P'.active = true$ do
6.          $Total \leftarrow Total * Query3(P')$
7.       return $Total$
8.    else
9.       $Total \leftarrow 0$
10.      for $i \leftarrow 0$ to $N.m - 1$ do
11.         $SetEvidence(N, i)$
12.         $Total \leftarrow Total + Query3(N)$
13.      $SetEvidence(N, \diamond)$
14.      return $Total$
```

Fig. 7. The Query algorithm, utilizing active and relevant nodes (Lines 03 and 05).

5 Results

Conditioning graphs offer linear-space computation, and easy portability to any architecture. However, they have a worst-case time complexity that is exponential on the size of the network. Methods for balancing elimination trees have been developed [7], however, the subsequent heights are still a function of network size. Elimination methods, on the other hand, compute in time exponential on the tree-width of the network [4]. This value is typically small in comparison to the network size, so elimination methods will typically be quicker to answer queries than conditioning methods, but they require much more space. In this section, we show that the proposed optimizations provide considerable speedup in inference, and that the inference times are reasonable compared to elimination.

We refer to the height of a conditioning graph as its *actual height* h, while its height after ignoring irrelevant nodes will be its *effective height* h^*. We will refer to the *effective conditioning graph* as the conditioning graph with its irrelevant nodes ignored. To draw a comparison between conditioning graph methods and elimination methods, we compare the effective height h^* of the conditioning



(a)

(b)

Fig. 8. Height difference between actual and relevant conditioning graph.

graph to the width w^* of the network generated using the min-fill heuristic [8]. By comparing the CG height to induced width, we are comparing the complexity of inference in CGs with the complexity of inference in VE and JTP, by looking at the exponent involved in the worst case analysis.

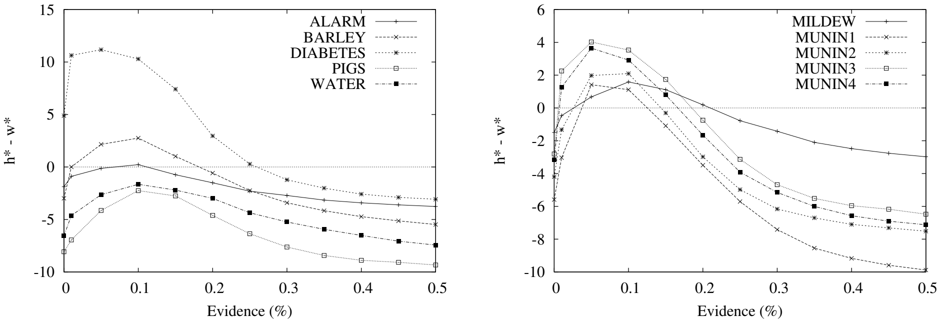
We compared the approaches over ten well-known networks, obtained from the Bayesian network repository.[1] We tested the algorithms using different percentages of evidence variables (ranging from $0 - 50\%$ of the variables in the network). For each test, we generated 100 random sets of evidence, and tested 50 different query variables on for each set of evidence, for a total of 5000 runs *per evidence set size*, per network.

Figure 8 shows the difference $h - h^*$ for each network (for readability, we have presented the results in two graphs). The graphs show that ignoring the irrelevant information of the network offers a substantial speedup over computing over it. The speedup is most prominent when there is no evidence; there is also a tendency for the difference to increase when the amount of evidence is greater than 20%. An explanation for these results is offered below.

We next compare the height of the effective relevant graph to the width of the network (generated using the standard min-fill algorithm), i.e., $h^* - w^*$. Figure 9 shows the result of this comparison. While the actual height of the conditioning graph is typically much worse than the width of the network, the effective height of the relevant conditioning graph is not that much worse than the network width - in fact, it's typically *better* when the amount of evidence is greater than 20%. The curves are similar for all graphs: an initial growth, followed by a decline. This shows that in many cases, the complexity of recursive decompositions is within the width of the network, meaning that we obtain reasonable time while maintaining the benefits of conditioning graphs, namely, linear space implementation and portability.

The results for both sets of graphs are easily explained by considering where the hardest inference problems are in terms of amount of evidence. When a

[1] http://www.cs.huji.ac.il/labs/compbio/Repository/.



(a) (b)

Fig. 9. Difference between relevant height of conditioning graph and network width.

network has no evidence, the number of barren variables is typically high, so the complexity is low. As evidence is added, the number of barren variables declines, increasing the complexity. However, this increase in the number of variables is eventually offset by the number of d-separated variables, so the complexity begins to decline. Hence, the hardest problems for inference in our example networks occur when the amount of evidence is greater than 0% and less than 20%.

6 Conclusions and Future Work

This paper presents two optimizations to conditioning graphs, to improve their efficiency while still maintaining linear space. The first optimization improved the efficiency of indexing in the CPTs of the conditioning graph. The second optimization demonstrated how to leave irrelevant variables out of the conditioning technique. These optimizations required simple extensions to the original code which are consistent with the original goal of CGs: easily implementable, making them universally portable. The optimizations attempt to avoid repeat calculation and irrelevant information. They take advantage of current model state.

The first optimization saves us an exponential number of arithmetic operations for a given query, and these savings can be realized across queries in cases where the evidence remains the same. For the second optimization, we measured its performance according to the effective height of the conditioning graph (the maximum number of *relevant* non-observed variables along any path). We observed that the effective height of the network is typically better than the actual height, which means an exponential speedup in the run-times of conditioning graphs. We also observed that this speedup allows conditioning graphs to be competitive in runtime to elimination algorithms in certain cases, especially when the percentage of observed nodes does not fall between 5% and 20%. Both of these optimizations increase the storage requirements of the algorithm by only a constant factor.

While these optimizations provide some speedup, providing caching of intermediate values at internal nodes ultimately produces the fastest recursive structures [2]. Caching is easily implemented in conditioning graphs (caches are indexed the same as distributions). However, naive caching seems to require exponential space. Darwiche et al. have provided good methods for optimal caching given limited space for d-trees [1]. We are currently investigating the most effective use of space in a conditioning graph.

References

1. D. Allen and A. Darwiche. Optimal time–space tradeoff in probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 969–975, 2003.
2. A. Darwiche. Recursive Conditioning: Any-space conditioning algorithm with treewidth-bounded complexity. *Artificial Intelligence*, pages 5–41, 2000.
3. 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*, pages 203–210, 1996.
4. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.
5. D. Geiger, T. Verma, and J. Pearl. Identifying independence in Bayesian networks. *Networks*, 20:507–534, 1990.
6. K. Grant and M. Horsch. Conditioning Graphs: Practical Structures for Inference in Bayesian Networks. In *Proceedings of the The 18th Australian Joint Conference on Artificial Intelligence*, pages 49–59, 2005.
7. K. Grant and M. Horsch. Methods for Constructing Balanced Elimination Trees and Other Recursive Decompositions. *Proceedings of the the 19th International Florida Artificial Intelligence Research Society Conference (To Appear)*, 2006.
8. U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical report, Dept. of Mathematics and Computer Science, Strandvejan, DK 9000 Aalborg, Denmark, 1990.
9. 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.
10. 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.
11. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., 1988.
12. D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence.* Oxford University Press, 1998.
13. R. D. Shachter. Evaluating influence diagrams. *Oper. Res.*, 34(6):871–882, 1986.
14. R. D. Shachter. Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, pages 480–487, 1998.
15. N. Zhang and D. Poole. A Simple Approach to Bayesian Network Computations. In *Proc. of the 10th Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.