Efficient Indexing for Recursive Conditioning Algorithms

Kevin Grant
Department Of Math and Computer Science
University of Lethbridge
4401 University Drive
Lethbridge, Alberta, T1K 3M4
kevin.grant2@uleth.ca

Abstract

In this paper, we consider efficient indexing methods for recursive conditioning algorithms. We compare two well-known methods for indexing, a *top-down* method and a *bottom-up* method, and discuss the redundancy that each of these suffer from. We also present a new method for indexing that is a hybrid of these models. Using this new approach shows an improvement in the amount of indexing operations by 55% or more in our test networks, as well as a reduction in the cumulative time for inference by 33% or more.

Introduction

Computing posterior probabilities from a Bayesian network (henceforth referred to as *inference*) is an NP-hard task (Cooper 1990). However, modern algorithms are able to feasibly perform this computation on many networks. The standard algorithms, such as Junction Tree Processing (JTP) and Variable Elimination (VE), can compute in time and space exponential on the induced width of the associated elimination ordering of the variables, the best known complexity for inference known to date (Lauritzen & Spiegelhalter 1988; Zhang & Poole 1994).

Within this asymptotic complexity lies some performance variation. For example, the probability distributions of Bayesian networks are typically implemented as linear arrays. Given this representation, the inference algorithm must be able to compute the array index that corresponds to a given context of the variables defining the distribution (henceforth referred to as *indexing*). Using a straightforward indexing approach, it can be shown that for inference operations such as multiplication and marginalization, the number of mathematical operations required for correctly indexing the arrays far exceeds the number of mathematical operations that occur on the probabilities in the arrays. Several algorithms for efficient indexing have been presented for the standard inference methods such as JTP and VE (Huang & Darwiche 1996; Arias & Diez 2007), which reduce the amount of computation required for indexing during the course of inference. While these methods work well for standard inference algorithms, it is not clear how these methods translate to conditioning methods, in particular, recursive conditioning models (Darwiche 2000; Grant 2006). These recursive algorithms do not employ multiplication and marginalization in the same way as the standard elimination algorithms do, and therefore their indexing requirements can differ considerably.

In this paper, we consider efficient methods for indexing distributions in recursive conditioning models. We focus on conditioning graphs (Grant 2006), as the indexing structure for these data structures is explicit, rather than implied. The main contribution of this paper is a new bidirectional indexing model for conditioning graphs that is a hybrid of two known indexing methods. We show that this new approach reduces the number of indexing operations in a conditioning graph by 55% or more compared to current methods, and a corresponding time reduction of 33% or more. The methods discussed are applicable to other recursive conditioning algorithms (Darwiche 2000; Ramos & Cozman 2005), where efficient indexing methods have not been formally considered.

Background and Previous Work

We denote random variables with capital letters (e.g. X, Y, Z), and sets of variables with boldfaced capital letters $\mathbf{X} = \{X_1, ..., X_n\}$. Each random variable V has a discrete domain of finite size $|V|$ containing values $\mathcal{D}(V) = \{0, ..., |V| - 1\}$. An instantiation of a variable is denoted *V=v*, or v for short. A *context*, or instantiation of a set of variables, is denoted **X=x** or simply **x**. Given a set of random variables $\mathbf{V} = \{V_1, ..., V_n\}$ with domain function $\mathcal{D}$, a Bayesian network is a tuple $\langle \mathbf{V}, \mathbf{\Phi} \rangle$. $\mathbf{\Phi} = \{\phi_{V_1}, ..., \phi_{V_n}\}$ is a set of probability distributions with a one-to-one correspondence with the elements of $\mathbf{V}$. A Bayesian network has an associated DAG, and each $\phi_{V_i} \in \mathbf{\Phi}$ is the conditional probability of V_i given its parents in the DAG (called *conditional probability tables* or CPTs). That is, if π_{V_i} represents the parents of V_i, then $\phi_{V_i} = P(V_i|\pi_{V_i})$. The *definition* of a discrete variable function is the set of variables over which it is defined. Figure 1 shows the DAG of a Bayesian network, taken from (Lauritzen & Spiegelhalter 1988).

An *elimination tree* (etree) (Grant 2006) 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 definition are contained in the subtree of the node labelled

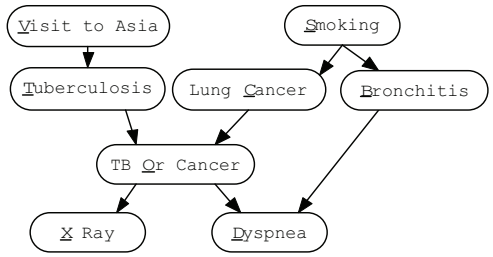

Figure 1: An example Bayesian network.

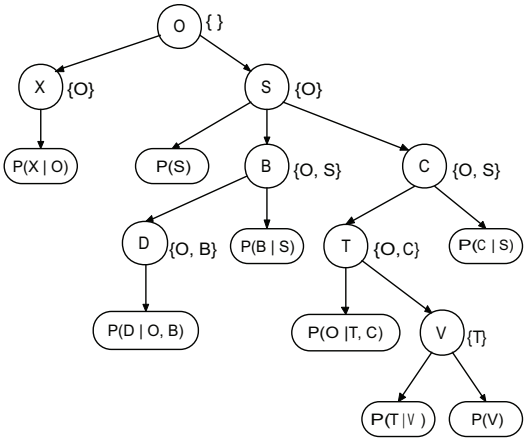

Figure 2: The network from Figure 1, arranged in an etree.

```
SetEvidence(N, i)
 1.  if i ≠ N.value
 2.    diff ← i − N.value {⋄ = 0 in this equation}
 3.    for each S′ ∈ N.secondary do
 4.     S′.pos ← S′.pos + scalar(N, S′) ∗ diff
 5.   N.value ← i
QueryA(N)
 1.  if N.cpt[N.pos] ≠ ∅
 2.    return N.cpt[N.pos]
 3.  else if N.value <> ⋄
 4.    Total ← 1
 5.    for each P′ ∈ N.primary while Total > 0
 6.      Total ← Total ∗ QueryA(P′)
 7.  else
 8.    Total ← 0
 9.    for i ← 0 to N.size − 1 do
10.      SetEvidence(N, i)
11.      Product ← 1
12.      for each P′ ∈ N.primary while Product > 0
13.        Product ← Product ∗ QueryA(P′)
14.      Total ← Total + Product
15.    SetEvidence(N, ⋄)
16.  N.cpt[N.pos] ← Total
17.  return Total
```

Figure 3: Query algorithm for computing probabilities from a conditioning graph.

with V_i. Figure 2 shows one of the possible etrees for the Bayesian network of Figure 1.

In order to avoid recomputing values, each internal node maintains a cache of its computed values. The cache of N is a function over N's *cache-domain*, defined as the intersection of the variables in the CPTs of N's subtree, and the variables labelling the path from root to N (called a *context* by Darwiche (2000)). The cache-domains in Figure 2 are shown in curly braces to the right of each node.

A *conditioning graph* (Grant 2006) is a low-level representation of an elimination tree. At each node of the conditioning graph, we store its CPT or cache as a linear array, and the current index of this array as an integer or pointer. At 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 associations between variables in the graph and the CPTs and caches that depend on them. The secondary arcs are added according to the following rule: *there is an arc from an internal node A to node B iff the variable X labelling A is contained in the definition of the CPT or cache associated with B*. Each secondary arc has an associated scalar value, which indicates the change that occurs to the CPT or cache index relative to a change in the variable's value.

Computing a probability from a conditioning graph, called a *query*, is implemented as a depth-first traversal over the underlying elimination tree. When an internal node is reached, we condition over its labelling variable. Each time the value of a variable is updated, the position of each associated CPT and cache index is updated as well, using the secondary arcs. When a leaf node is reached, or an internal node with a non–empty cache value is reached, the value of the array at its corresponding index is returned. The low-level, simple design of the conditioning graph means that this inference algorithm for computing probabilities is compact, efficient, and written entirely using low-level operations such as simple arithmetic and pointer manipulation. Space restrictions preclude a more detailed introduction to conditioning graphs; further details can be found in the related literature. Figure 3 shows an implementation of the query algorithm. For a leaf node N, we use $N.cpt$ and $N.pos$ to refer to the CPT and its current index, respectively. We also overload these terms to refer to an internal node's cache and cache index, as they are used in an equivalent manner. The value $\emptyset$ in a cache location means the value has not yet been set. For an internal node N, we use $N.primary$, $N.secondary$, $N.value$, and $N.size$ to refer to the node's primary children, secondary children, variable value, and variable size, respectively. The function $scalar$ labels each secondary arc with its scalar value. The variable's value represents the evidence, if any. To set the evidence $V = i$, the application would set $V.value = i$. When the value of a variable is $\diamond$, it is unknown. 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.

Recursive inference algorithms such as conditioning graphs offer several advantages over other inference methods. The primary advantage is their ability to trade-off space for time, which is particularly useful in memory-constrained applications. Furthermore, the algorithm very naturally exploits zeroes that occur during inference. An internal node in the tree calls its children sequentially, accumulating its value as a product. In the event that one of its children returns a zero, the node can abort these calls to the remainder of its children. This very simple optimization is trivial to implement (the *while* condition on Lines 5 and 12), yet can potentially yield exponential reductions in computation. While such an optimization is also available to other algorithms (such as VE), it is much more difficult to exploit, as these algorithms compute and pass entire distributions rather than single values.

Indexing. Suppose we have a function $f(\mathbf{X})$ defined over discrete variables $\mathbf{X} = \{X_1, ..., X_k\}$. Storing this function as a linear array requires a mapping from each context of $\mathbf{X}$ to an integer in the array. Typically, an ordering over the variables is selected (for simplicity, assume that the index of the variables in our example also defines that variable's position in the ordering). Given a context $\mathbf{x} = \{x_1, ..., x_k\}$, we can compute the associated index using the formula:

$$index(\mathbf{x}) = \sum_{i=1}^{k} C_i * x_i \quad (1)$$

where each $C_i = \prod_{j=i+1}^{k} |X_i|$ is the *offset* associated with variable X_i. Hence, given that we know the offsets (which can be computed offline), computing an index from a context of k variables requires roughly $\mathbf{O}(k)$ operations.

For a single context, Equation (1) is a reasonable choice, as the values of all variables must be considered in order to determine an index. However, when multiple queries to the same distribution are made, applying this formula to each context can be inefficient. For example, suppose we query f with two different contexts of $\mathbf{X}$ that differ only in their values of $X_i \in \mathbf{X}$. Once the first index (call it j_1) is computed, the second index (call it j_2) could be computed as $j_2 = j_1 + C_i * \delta_i$, where δ_i is the difference in the value of X_i between each context. Hence, we require a context number of arithmetic operations, a distinct improvement over Equation (1), especially when the number of variables is large.

Conditioning graphs attempt to resolve some of this inefficiency by only updating indices in response to the value of a variable changing. For instance, when a variable X changes values, then the indices that correspond to those caches and CPTs defined over X are updated through the secondary arcs. The scalar values attached to the secondary arcs are exactly the offset values C_i from Equation (1). The efficiency of this model is discussed in the next section.

Recently, Arias and Diez (2007) published a method based on *accumulated offsets*. Briefly, these accumulated offsets are an array associated with each distribution. Using these offsets, they show how to index over a sequence of contexts for a distribution using a constant time operation per context. However, this technique requires complete regularity in the order in which each value in a discrete function is accessed. Such regularity is often not present in recursive conditioning algorithms, as recursive inference algorithms can use computed zeroes to prune away large sections of the problem space. This optimization rules out the use of accumulated offsets, because the regularity requirement is not met. It should be noted that if no zeroes occur in the network distributions, then accumulated offsets could be used in a conditioning graph in much the same manner as the normal offsets are; this is discussed further in the last section.

Another model for fast indexing in discrete functions is *cluster-sepset mapping* (Huang & Darwiche 1996), which uses discrete functions that map the indices of one function to another. These mappings are constructed offline, so there is no consequence to the online runtime of the algorithm. For recursive inference structures such as those discussed, it can be shown that for a node N, the index of the cache or CPT at node N is a function of the cache index and variable value at N's parent, so a similar mapping system could be employed. However, the memory required by these maps is at least as large as the size of the caches and CPTs combined; storing these maps would roughly double the memory requirements of the algorithm. Furthermore, while caches can be pruned without affecting program correctness, the index maps could not be simply pruned away without some secondary device to correctly compute the index values.

As noted by Arias and Diez (2007), there are efficient indexing designs implemented in commercial inference systems, the details of which are typically not made public. We are unaware of any indexing model that is similar to the one that we present here.

Indexing In Conditioning Graphs

Before considering ways to efficiently implement indexing, we demonstrate that the cumulative costs of indexing can represent a non-trivial portion of the overall costs of inference. To do this, we first define the notion of an *indexing operation*. An indexing operation represents the cumulative computation that takes place on the index of a CPT or cache as a result of the value of a *single* variable. For example, considering Equation (1), since we compute the index based on the value of k variables, we say that k indexing operations took place, where each indexing operation requires a multiplication and an addition. One of the metrics that we will use to measure the quality of an indexing model is the number of indexing operations that it incurs, as we have found that this provides a reasonably good indicator of the true cost of indexing during inference.

Table 1 shows the results of an empirical evaluation of the cost of indexing over several well-known benchmark networks.[1] Each table entry represents a query over the network given no evidence. For each network, the table shows

[1]Networks obtained from the Bayesian Network Repository: http://compbio.cs.huji.ac.il/Repository/

	Ops ($\times 10^6$)		Runtime (s)	
Network	**Data**	**Index**	**Indexing**	**No Indexing**
Barley	61.66	201.8	18.22	6.29
Diabetes	27.72	179.8	14.38	2.86
Insurance	0.1322	0.5771	0.05	0.01
Mildew	10.85	48.75	4.12	1.01
Munin2	7.031	51.32	4.14	0.77
Munin3	9.535	36.74	3.64	0.98
Munin4	32.6	143.1	13.31	3.37
Pigs	2.693	15.95	1.24	0.29
Water	2.537	15.93	1.34	0.29

Table 1: Evaluating the cumulative cost of indexing operations on the inference process.

the number of multiplications and additions that take place on the actual probabilistic data, the number of indexing operations that take place, and the CPU time required for each run. The final column is an approximation of the runtime with the indexing operations removed. To simulate this, we first run the program, and precompute the sequence of array indices that will be used during the course of the operation. We then use this sequence in the place of the normal indexing operations used by conditioning graphs.

The table demonstrates several points. First, the number of indexing operations can be significant, especially by comparison to the number of operations that take place on the data. Second, the cost of indexing in conditioning graphs is significant, representing the majority of the time taken by each inference operation (this was also recognized in Koller and Friedman (2009)). Given that indexing plays such a substantial role in the overall performance of inference, it seems worthwhile to optimize its behaviour as much as possible.

Top-Down vs. Bottom-Up Indexing. The current indexing model for conditioning graphs can be described as a *top-down* approach. As each variable is conditioned to a particular value, information is passed down via secondary arcs to all related index pointers. Of course, it is not necessary to follow a top-down approach in a conditioning graph. Rather, computing the index for a distribution at a particular node could be delayed until that node is actually reached. Upon reaching a node, we could use Equation (1) to compute the respective index. We call this a *bottom-up* approach, as each index is calculated from its respective node by looking up its respective variable values above it in the tree. The structural modifications required for a bottom-up approach are minimal – we simply reverse the directions of the secondary arcs. The weights for the secondary arcs remain the same.

When no caching is used, the top-down method is clearly superior to the bottom-up approach. Denote by $d(N)$ the number of nodes above N in the elimination tree. Consider a CPT or cache at node N defined over $\{X_1, ..., X_k\}$. Let N_i denote the node labeled by X_i, for $1 \leq i \leq k$. The number of indexing operations performed by the top-down approach will be $\sum_{i=1}^{k} \exp(d(N_i))$, whereas the bottom-up approach requires $k \exp(d(N))$ indexing operations. Because $k < d(N)$, the top-down approach is never worse than the bottom-up approach, and is typically much better. Intuitively, the top-down approach is superior because it avoids redundant indexing operations. Consider the example network shown in Figure 2, in particular, the leaf variable representing $P(D|B, O)$. Using the bottom-up approach, we compute its index each time the node is visited, by considering the value of each variable. However, between the first and second visit, only the value of D changes, yet we repeat the entire index computation over each variable. By contrast, the top-down approach avoids this redundant computation, only changing the index relative to variable B when B changes value.

Network	**Top-Down**		**Bottom-Up**		**Bidirectional**	
	Ops	**RT**	**Ops**	**RT**	**Ops**	**RT**
Barley	201.8	18.22	296.8	26.08	79.69	12.07
Diabetes	179.8	14.38	107.5	10.71	46.86	7.05
Insurance	0.5771	0.05	0.6687	0.06	0.2376	0.03
Link	9512	666.9	4735	336.59	649.8	99.72
Mildew	48.75	4.12	44.26	4.05	17.19	2.48
Munin1	1911	186.28	1930	204.7	647.9	106.74
Munin2	51.32	4.14	39.48	3.6	13.78	2.07
Munin3	36.74	3.64	30.83	3.62	13.71	2.4
Munin4	143.1	13.31	154.1	14.83	56.29	8.61
Pigs	15.95	1.24	11.72	1.09	4.895	0.63
Water	15.93	1.34	18.23	1.54	6.796	0.86

Table 2: Op counts ($\times 10^6$) and runtimes (secs) of the different indexing models.

While top-down indexing is superior in a no-caching model, the choice becomes less obvious when caching is employed. Consider again the example network shown in Figure 2, in particular, the leaf node representing $P(T|V)$. Each time the value of variable T changes, the index for $P(T|V)$ will be updated. A trace of the query operation shows that the value of variable T will be changed 8 times. However, subsequent to the first two changes, the cache at V will be filled, so $P(T|V)$ will no longer be queried. It is therefore unnecessary to adjust its index after this point, so the last six updates simply waste computation.

Hence, in a caching environment, both top-down and bottom-up have potential for redundancy. In top-down, we see unnecessary updating of indexes for obsolete caches and CPTs, while in bottom-up, we see unnecessary computation over unchanged variables. The question now becomes: which method incurs the least amount of redundancy? Clearly, there is no winner at a node level, as we can show that for some nodes, the top-down approach is superior, and vice versa for other nodes. What about cumulatively over an entire query operation? Table 2 shows the number of operations and time required for a query operation over the benchmark networks, under a full-caching model. The first two columns represent the number of indexing operations and the runtime using the top-down model, while the next two columns summarize the same data for the bottom-up model.

The results show two important points. First, in terms of the number of indexing operations, there is no clear winner between the two models in a caching environment. Second, a discrepancy exists between the runtime improvement and indexing operation reduction between the two models. For instance, consider network Munin3. While the number of indexing operations required by bottom-up is considerably less than that of top-down, the time difference was negligible. We attribute this to the difference in overhead costs between the two models. Both top-down and bottom-up indexing require looping constructs to iterate over lists of secondary pointers. The number of required looping constructs is proportional to the number of visits made to the node. In the top-down approach, this excludes leaf-node visits, while in the bottom-up approach, this includes leaf node visits. Consequently, we find that bottom-up indexing incurs a slightly larger overhead.

Bidirectional Indexing

In both the top-down and bottom-up indexing models, the weight associated with each secondary arc is consistent. This means that a variable's effect on an index is the same in both models – the only difference is when the index is adjusted relative to a variable's value. Therefore, it should be straightforward to modify the algorithm so that both top-down and bottom-up arcs can be used to index a particular distribution in the network.

To determine the direction that an arc should go, we need to define a few values. First, given a non-root node N with parent N_p in the elimination tree, the number of recursive calls made to N depends on whether or not N_p is caching. If N_p is caching, then the number of recursive calls made to N is defined as $calls(N) = cachesize(N_p) \times size(N_p)$, where *cachesize(N)* is the number of entries in N's cache, and $size(N)$ is the size of the variable labeling N. If N_p is not caching, then the number of recursive calls made to N is defined recursively as $calls(N) = calls(N_p) \times size(N_p)$. These formulae are modified from Darwiche (2000). For a root node N, $calls(N) = 1$.

Given a variable X labeling node N, the number of times the value of the variable will change also depends on whether N is caching its values. If N is caching, then the number of times the variable will change will be defined as $changes(X) = size(X) \times cachesize(N)$. To see this, note that we condition over X only when a call to N results in a cache miss. On the other hand, if N is not caching, then the number of times the variable will change is defined as $changes(X) = size(X) \times calls(N)$ as each call to N results in conditioning over X.

Consider a node N_X labeled by variable X, and a node N_f whose CPT or cache is defined by X. Which direction should the secondary arc between N_X and N_f go? Suppose it is a top-down arc. Then the number of times the index of f will be adjusted according to the value of X equals the number of times the value of that variable will change. On the other hand, if it is a bottom-up arc, then the number of times the index will be adjusted relative to X will be the number of calls made to N_f. Therefore, we orient the arc in the direction that results in the fewest number of index adjustments relative to a particular variable.

Consider Figure 2, and the secondary arc between T and $P(T|V)$, and assume full caching. The number of times T will change during the query process is 8. On the other hand, the number of times $P(T|V)$ will be called is 4. Hence, we make this arc bottom-up. On the other hand, consider the arc between B and $P(D|B, O)$. The number of times B will change is 2, and the number of times $P(D|B, O)$ is called is 8. Therefore, the arc becomes a top-down arc.

In the event of a tie, we favour a top-down arc over a bottom-up arc. This stems from discussion in the previous section, where we noted that the overhead involved with bottom-up is slightly higher than for top-down. As a side-effect, favouring the top-down means that in a no-caching model, all arcs become top-down, and the modified model degrades to the original conditioning graph model.

The overall cost of determining arc direction is linear in the number of secondary arcs in the network, assuming that we know how many times each node will be called, which is linear in the number of nodes in the network. The number of secondary arcs is bounded by $\mathbf{O}(nw)$, where n is the number of nodes, and w is the network width. The cost of orienting the arcs is therefore trivial by comparison to the actual inference operation. Furthermore, we only assign direction to each arc once, so the orientation process can be moved offline, where the computational cost is less important.

Figure 4 shows the modified version of the query algorithm that accommodates both top-down and bottom-up indexing. We assume that the secondary list at each node has been replaced by two different lists: *topdown* and *bottomup*, which store the top-down arcs and bottom-up arcs emitting from a node, respectively. *SetEvidence* is almost identical in this version as in Figure 3 (and therefore omitted to conserve space), with the exception that the iteration on Line 3 iterates over *N.topdown* rather than *N.secondary*. The function *ComputeIndex* computes the adjustment to the index of a particular node based on the information from its bottom-up arcs. This information is combined to produce a final index (Line 1 of *Query*).

Note that the information from the bottom-up arcs of a node N is computed during the call to N's parent (Lines 4 and 5 of *Query*), prior to N being called. If the rules for orienting the direction of the arcs are followed, then any arc between N and its parent is guaranteed to be a top-down arc, and therefore any change to the variable labeling N's parent will not affect the outcome of N's bottom-up computation. Therefore, calling *ComputeIndex* from the parent variable rather than the child does not affect the correctness of the program, but reduces the number of calls to *ComputeIndex* and therefore provides a slight runtime optimization.

The final two columns of Table 2 summarize the performance of the bidirectional indexing model on the benchmark networks. In terms of indexing operations, the data shows that the bidirectional model compares favourably to both the top-down and bottom-up approach. We see over a 55% reduction in operations over all networks, and nearly an order of magnitude reduction in the *Link* network. In terms of runtime, the bidirectional model also compares favourably,

```
ComputeIndex(N)
 1.  N.pos2 ← 0
 2.  for each P' ∈ N.bottomup do
 3.    N.pos2 ← N.pos2 + scalar(P', N) * P'.value
QueryB(N)
 1.  pos ← N.pos + N.pos2
 2.  if N.cpt[pos] ≠ ∅
 3.    return N.cpt[pos]
 4.  for each P' ∈ N.primary do
 5.    ComputeIndex(P')
 6.  if N.value <> ⋄
 7.    Total ← 1
 8.    for each P' ∈ N.primary while Total > 0
 9.      Total ← Total * QueryB(P')
10.  else
11.    Total ← 0
12.    for i ← 0 to N.size − 1 do
13.      SetEvidence(N, i)
14.      Product ← 1
15.      for each P' ∈ N.primary while Total > 0
16.        Product ← Product * QueryB(P')
17.      Total ← Total + Product
18.    SetEvidence(N, ⋄)
19.  N.cpt[pos] ← Total
20.  return Total
```

Figure 4: Modified query algorithm, utilizing both top-down and bottom-up secondary arcs.

improving performance in all cases by at least 33%, and up to as much as 70%.

A final note regarding evidence should be made. In the original conditioning graph algorithm, evidence was entered via the *SetEvidence* function, which immediately adjusted any indices connected to the evidence variables. This still works for distributions connected to the evidence variables via top-down arcs, but not for those connected through bottom-up arcs. In order to optimize the efficiency of the bottom-up computation, the adjustments to indices based on the evidence should be done prior to query. This is easily accommodated by a pre-query traversal of the bottom-up arcs.

Conclusions and Future Work

In this paper, we considered efficient indexing methods for conditioning graphs. We compared two well-known methods for indexing, a *top-down* method and a *bottom-up* method, and discussed the redundancy that each of these suffers from. We presented a hybrid of these models, that uses a bidirectional indexing approach that attempts to minimize the redundancy. Using this new approach reduced the number of indexing operations in all test networks by 55%, as well as reduced the overall time of inference by over 33% in all tests. While the methods of this paper focused on conditioning graphs, the results can be easily extended to similar recursive inference methods (e.g. Recursive Conditioning).

There are several extensions to this project that we are exploring. First, we mentioned the use of accumulated offsets in conditioning graphs, which is possible under certain conditions, but fails when we begin pruning parts of the search space. We speculate that perhaps a hybrid approach exists, where accumulated offsets could be used in parts of the query process where pruning will not occur. Another potential improvement is a more sophisticated decision process for the direction of the arcs. It was shown that the top-down arcs incur less overhead per call than the bottom-up arcs, however, we treated top-down calls to be computationally equivalent to the bottom-up calls when selecting arc direction. We speculate that by weighting these calls appropriately to account for overhead, we can realize a further improvement in the overall runtime of the algorithm, even if it means a slightly larger number of indexing operations. Finally, the bidirectional model of indexing presented here attempts to minimize the number of redundant computations that occur during indexing. However, it does not completely *eliminate* the redundant computations. We have attempted several sophisticated indexing techniques that attempt to eliminate all unnecessary operations, but to date, the overhead computation required by these methods negates any realized performance benefit. We will continue to search for new lightweight methods to eliminate redundancy.

Acknowledgements

Thanks to the anonymous reviewers for their suggestions regarding the paper. A special thanks goes out to Michael Horsch for his helpful comments. This research is funded by NSERC; their support is gratefully acknowledged.

References

Arias, A., and Diez, F. 2007. Operating with Potentials of Discrete Variables. *International Journal of Approximate Reasoning* 46:166–187.

Cooper, G. F. 1990. The Computational Complexity of Probabilistic Inference using Bayesian Belief Networks. *Artificial Intelligence* 42:393–405.

Darwiche, A. 2000. Recursive Conditioning: Any-space conditioning algorithm with treewidth-bounded complexity. *Artificial Intelligence* 5–41.

Grant, K. 2006. *Conditioning Graphs: Practical Structures for Inference in Bayesian Networks*. Ph.D. Dissertation, University of Saskatchewan, Computer Science Department.

Huang, C., and Darwiche, A. 1996. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning* 15(3):225–263.

Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Lauritzen, S., and Spiegelhalter, D. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statistical Society* 50:157–224.

Ramos, F., and Cozman, F. 2005. Anytime anyspace probabilistic inference. *International Journal of Approximate Reasoning* 38(1):53 – 80.

Zhang, N., and Poole, D. 1994. A Simple Approach to Bayesian Network Computations. In *Proceedings of the Tenth Canadian Conference on Artificial Intelligence*, 171–178.