

# Efficient Caching in Elimination Trees

Kevin Grant and Michael C. Horsch

Department Of Computer Science

University of Saskatchewan

176 Thorvaldson Building, 110 Science Place

Saskatoon, Saskatchewan S7N 5C9

{kjpg658,horsch}@cs.usask.ca

## Abstract

In this paper, we present *subcaching*, a method for reducing the size of the caches in the recursive decomposition while maintaining the runtime of recursive conditioning with complete caching. We also demonstrate a heuristic for constructing recursive decompositions that improves the effects of subcaching, and show empirically that the savings in space is quite substantial, with very little effect on the runtime of recursive conditioning.

## Introduction

Recently, we proposed *elimination trees*, and their simple extensions, *conditioning graphs* as runtime representations of belief networks (Grant & Horsch 2005). These are novel variants of the dtree structure used in recursive conditioning (Cooper 1990; Darwiche 2000). The term *recursive decompositions* refers to this class of recursive compilations.

Conditioning graphs (CGs) have a number of properties that make them ideal for use in embedded systems or multi-agent architectures, where memory limitations prevent the use of junction-tree message passing (JTP) (Lauritzen & Spiegelhalter 1988) or variable elimination (VE) (Zhang & Poole 1994; Dechter 1996). First, they require only linear space: the space needed is the same order as the memory needed to store the Bayesian network itself. Second, a CG consists of simple node pointers and floating point values: sophisticated data structures are not needed. As well, the inference algorithm for CGs is a small recursive algorithm, easily implementable on any architecture.

These minimal memory requirements are achieved at the cost of increased run-time. The time complexity of inference using a conditioning graph is exponential on the height of its underlying elimination tree. Algorithms have been proposed to minimize the height of these trees (Grant & Horsch 2006b), but for a given network, the height of the tree can still be substantially larger than the treewidth of the network. It is possible to restrict computation to relevant portions of the network based on current query and evidence (Grant & Horsch 2006a), but this does little in situations where the size of the relevant network is large.

The technique of *caching* is a well-known optimization for recursive decompositions (Darwiche & Hopkins 2001; Allen & Darwiche 2003b; Allen, Darwiche, & Park 2004), and is applicable to elimination trees. Caching reduces runtime by storing values required for intermediate calculation to avoid recomputing them. Caching all possible intermediate values reduces the runtime in recursive decompositions to that of JTP and VE, while at the same time increasing its space requirements to JTP and VE. Caching in recursive decompositions has two distinct advantages over other algorithms. First, the technique of caching allows for partial caching schemes, where only a subset of the values are cached, making the algorithm any-space (Darwiche 2000). Secondly, certain values (those in *dead caches*, discussed in the next section) are only used in calculation once, and therefore do not need to be cached (Allen & Darwiche 2003a). Identifying these values and excluding them from the cache gives a decisive advantage over JTP and VE in terms of memory, while still allowing it a runtime that is exponential on treewidth (Allen & Darwiche 2003a).

In this paper, we extend the state of the art in caching for recursive decompositions with a technique called *subcaching*, which reduces the size of the caches in elimination trees by reusing memory in the cache during inference. We present a heuristic for constructing elimination trees that make good reuse of memory in the cache (though not necessarily optimal), and show empirically that the savings in space is quite substantial. Our work is presented in terms of elimination trees, but can be simply extended to other recursive decompositions.

## Background

We denote random variables with capital letters (eg.  $X$ ,  $Y$ ,  $Z$ ), and sets of variables with boldfaced capital letters  $\mathbf{X} = \{X_1, \dots, X_n\}$ . Each random variable  $V$  has an associated domain  $\mathcal{D}(V) = \{v_1, \dots, v_k\}$ , which we assume is finite and discrete. An instantiation of a variable is denoted  $V=v$ , or  $v$  for short. A *context*, or instantiation of a set of variables, is denoted  $\mathbf{X}=\mathbf{x}$  or simply  $\mathbf{x}$ . Given a set of random variables  $\mathbf{V} = \{V_1, \dots, V_n\}$  with domain function  $\mathcal{D}$ , a Bayesian network is a tuple  $\langle \mathbf{V}, \Phi \rangle$ .  $\Phi = \{\phi_{V_1}, \dots, \phi_{V_n}\}$  is a set of distributions with a one-to-one correspondence with the elements of  $\mathbf{V}$ . A Bayesian network has an associated DAG, and each  $\phi_{V_i} \in \Phi$  is the conditional probability of  $V_i$

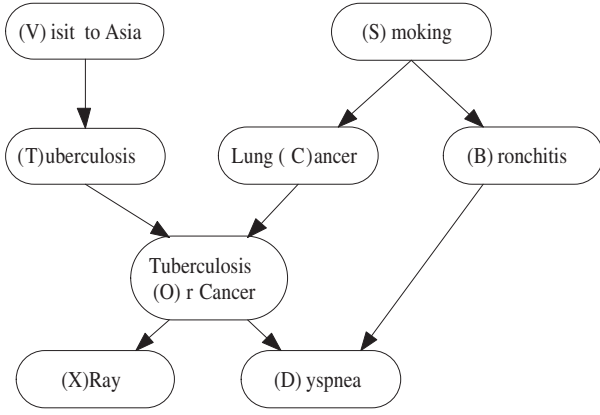


Figure 1: An example Bayesian network.

given its parents in the DAG (called *conditional probability tables* or CPTs). That is, if  $\pi_{V_i}$  represents the parents of  $V_i$ , then  $\phi_{V_i} = P(V_i | \pi_{V_i})$ . The *family* of a variable  $V_i$  is the set  $\{V_i\} \cup \pi_{V_i}$ , which is also the *domain* of the CPT for  $V_i$ .<sup>1</sup>

A variable in a Bayesian network is said to be *conditionally independent* of its non-descendants given its parents. This allows the joint probability to be factorized as:

$$P(\mathbf{V}) = \prod_{i=1}^n P(V_i | \pi_{V_i}) \quad (1)$$

Figure 1 shows the DAG of a Bayesian network, taken from (Lauritzen & Spiegelhalter 1988).

An *elimination tree* (etree) (Grant & Horsch 2005) 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 shows one of the possible etrees for the Bayesian network of Figure 1.

The function  $\mathcal{P}$  (Figure 3) computes the probability of a given context and etree. By construction, a depth-first traversal of an etree defines an elimination ordering. Whereas standard VE could be implemented in a bottom-up computation in an etree, algorithm  $\mathcal{P}$  recursively computes the sums and products of variable elimination using a top-down approach. The base case looks up values in the CPTs of the Bayesian network. We use the following notation: if  $T$  is a leaf node, then  $\phi_T$  represents the CPT at  $T$ . If  $T$  is an internal node  $V_T$  represents the variable labeling  $T$ , and  $ch_T$  represents its children. Finally, if an etree  $\mathcal{T}$  is used in the context of a treenode, then we are referring to the root node of  $\mathcal{T}$ .

The time complexity of  $\mathcal{P}$  over an etree  $T$  is exponential on  $T$ 's height, while the space complexity is exponential only on the largest family in the Bayesian network (Grant & Horsch 2005).

<sup>1</sup>The term *domain* is overloaded here, but usage should be clear from the context.

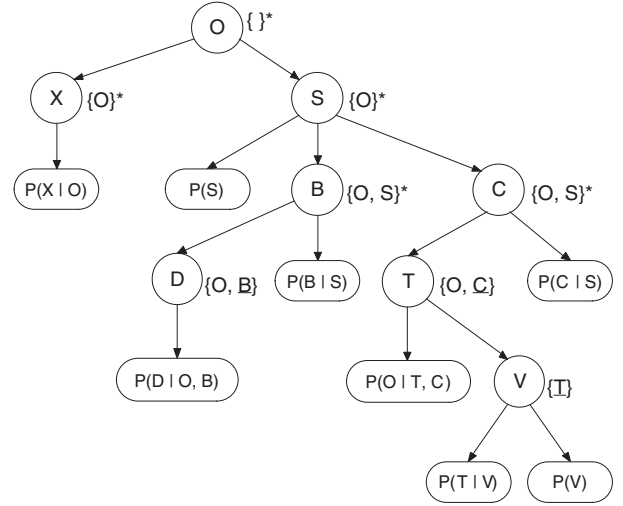


Figure 2: The network from Figure 1, arranged in an etree.

```

 $\mathcal{P}(T, \mathbf{c})$ 
1.  if  $T$  is a leaf node
2.    return  $\phi_T(\mathbf{c})$ 
3.  elseif  $V_T$  is instantiated in  $\mathbf{c}$ 
4.    Total  $\leftarrow 1$ 
5.    for each  $T' \in ch_T$  while Total > 0
6.      Total  $\leftarrow$  Total *  $\mathcal{P}(T', \mathbf{c})$ 
7.  else
8.    Total  $\leftarrow 0$ 
9.    for each  $v_T \in \mathcal{D}(V_T)$ 
10.     Total  $\leftarrow$  Total +  $\mathcal{P}(T, \mathbf{c} \wedge \{v_T\})$ 
11.  return Total

```

Figure 3: Code for processing an etree given a context.

## Caching

The runtime of computing over a dtree can be reduced through the use of caching (Darwiche 2000; Allen & Darwiche 2003a). The same techniques can be applied to etrees, without significant modification. Let  $N_V$  denote the internal node in an etree that is labeled with variable  $V$ . Consider the tree from Figure 2, and consider calling node  $N_T$  when  $O = o_0$ ,  $S = s_0$ , and  $C = c_0$ :

$$\mathcal{P}(N_T, \{o_0 \wedge s_0 \wedge c_0\}) = \sum_T P(o_0 | T, c_0) \sum_V P(T | V) P(V) \quad (2)$$

Notice that this equation does not depend on the value of  $S$ . Hence, when  $O = o_0$ ,  $C = c_0$  and  $S = s_1$ , the value returned from node  $N_T$  will be exactly the same as the return value of Equation 2. By caching this value at node  $N_T$ , it needs only be calculated when  $S = s_0$ , and retrieved when  $S = s_1$ .

Define the *a-cutset* of node  $N$  to be the set of variables labeling the nodes in  $N$ 's ancestry; the *cache-domain* of  $N$  (denoted  $CD(N)$ ) is the intersection of  $N$ 's a-cutset and the domains of the CPTs in  $N$ 's subtree. The cache-domains of

each node in Figure 2 are shown in curly braces to the right of each node. The return value from  $N$  depends only on the assignment to its cache-domain, and not its a-cutset. This is clearly demonstrated in the previous example: the cache-domain of node  $N_T$  is  $\{O, C\}$ , since the values of this node did not depend on  $S$ .

The algorithm for calculating probabilities from an etree (Figure 3) can be very simply modified to perform caching. As with dtrees, when a value is calculated for a particular assignment of the cache-domain, it is stored in the cache at that node. When a node is visited, we check to see if the corresponding value is cached. If it is, the cached value is returned; if not, the value is calculated, cached, and returned.

**Theorem 1.** (Grant 2006) *The complete cache space required by an etree is  $O(n \exp(w))$ , where  $n$  is the number of variables in the tree, and  $w$  is the width of the variable ordering used to construct the etree. The time required for algorithm  $\mathcal{P}$  to compute a value from an etree that caches is  $O(n \exp(w))$ .*

*Proof Sketch:* To sketch the space complexity result, note that the variables in a node’s cache-domain are exactly the variables over which VE would build an intermediate distribution, given the variable ordering of an etree. For time complexity, we note that the top down process computes cache elements exactly once.

The time savings due to caching can be substantial. Consider again the etree shown in Figure 2. Without caching, computation requires 165 total recursive calls. When all possible values are cached, the same computation requires only 85 recursive calls.

Caching allows a time-for-space trade-off. Darwiche (Darwiche 2000) demonstrated an any-space algorithm, which used a partial-caching technique (only some nodes are allowed to cache). The same technique can be applied to etrees (Grant 2006).

## Dead Caches

Dead caches are caches whose values are only generated and never queried (Allen & Darwiche 2003a). Consider the etree in Figure 2; in particular, consider node  $N_B$ . The cache-domain at this node is  $\{O, S\}$ . When caching is employed, Algorithm  $\mathcal{P}$  visits a node once for each assignment of its parent’s cache-domain and labeling variable. As this is the case, node  $N_B$  is visited only once for each assignment of its own cache-domain. Therefore, the cache values are never actually used, only set. Dead caches can be removed from recursive decompositions with no runtime consequence. In Figure 2, the dead caches are labeled with an asterisk.

Dead caches can be identified in dtrees as a cache whose context is a superset of its parent’s context (Allen & Darwiche 2003a). While this definition suffices in etrees as well, the restriction of exactly one variable per node allows us to identify dead caches without performing set comparison. Let  $\text{var}(N)$  represent the variable labeling node  $N$ , and let  $N^P$  refer to node  $N$ ’s parent in its etree. We define a *proper etree* as follows:

**Definition 1.** *An etree is **proper** if, for all nodes  $N$ , the subtree of node  $N$  contains a CPT with  $\text{var}(N^P)$  in its domain,*

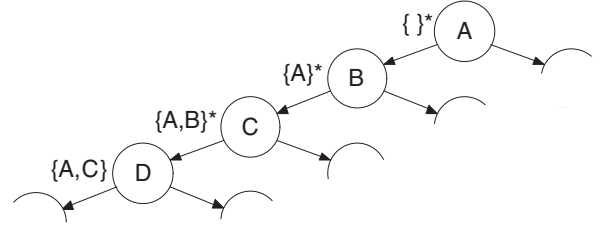


Figure 4: A partial etree, with caches shown to the left of the nodes. Dead caches are marked with an asterisk.

for all  $N$ .

A proper etree is guaranteed using previously published construction methods (Grant & Horsch 2005; 2006b), but Algorithm  $\mathcal{P}$  is correct even for improper etrees. For the remainder of the document, we will assume that our etrees are proper, unless otherwise stated. It is possible to prove the following theorem for proper etrees:

**Theorem 2.** (Grant 2006) *If  $CD(N) \supset CD(N^P)$ , then  $CD(N) = CD(N^P) \cup \{\text{var}(N^P)\}$ .*

From Theorem 2, we can identify a dead cache in a proper etree as *any cache whose cardinality is larger than the cardinality of the cache of its parent node*. This is a slightly simpler criterion than the one used for dtrees (Allen & Darwiche 2003a).

Figure 2 demonstrates the advantage of removing dead caches, as it removes the number of cache values from 23 to 10. We provide more empirical demonstration of the memory saved by eliminating dead caches in a subsequent section; briefly, the results in etrees are very similar to those obtained by Allen and Darwiche (Allen & Darwiche 2003a).

For the remainder of the document, we will refer to computing with dead caches removed as *live caching* (not removing dead caches will be referred to as *complete caching*).

## Subcaching

While live caching requires much less space than complete caching, we can improve on the space requirements further, by noting that while a cache may not be dead, there exist cases where only certain parts of it are live at any moment. Consider a portion of an etree, shown in Figure 4. The caches are shown to the left of the node, with dead caches marked with an asterisk. There is one live cache at node  $N_D$ , caching values over the variables  $\{A, C\}$ .

A trace of the visits to node  $N_D$  in Figure 4 reveals two important points:

1. The cache values corresponding to  $A = 0$  are never reused after  $A$  becomes 1 (that is, after visit 4).
2. The cache values corresponding to  $A = 1$  are only calculated after  $A$  becomes 1.

In other words, the portion of the cache corresponding to  $A = 0$  is dead following  $A$ ’s being set to 1, so its memory

can be reused. As well, the portion of the cache corresponding to  $A = 1$  has yet to be calculated following  $A$ 's conditioning to 1. Therefore, *these values can occupy the same memory*. The new cache will be indexed only on the variable  $C$ , and will be reset each time the value of  $A$  changes. While the computation at node  $D$  has not changed, we have reduced its cache memory requirements by 50%.

This form of caching is not partial caching, as we do eventually cache all of the values that live caching does. Since we only cache a subset of the entire cache at a time, we refer to this as *subcaching*. The subset of the cache-domain of node  $N$  that will define the cache will be referred to as an *effective cache-domain*.

The effective cache-domain can be defined as follows: let  $\rho = [A_1, \dots, A_q]$  be the cache-domain of node  $N$  ordered according to the etree (variables whose nodes are closer to the root come first in the ordering). Let  $\rho' = [B_1, \dots, B_r, \text{var}(N^P)]$  be the cache-domain of the parent node of  $N$ , ordered according to the etree, and appended with the variable labeling  $N^P$ . Let  $\rho[i]$  denote the  $i$ th variable in  $\rho$  according to the said ordering. The effective cache-domain of  $N$ , denoted  $ECD(N)$ , is equal to  $[A_i, \dots, A_q]$ , where  $\rho[i] \neq \rho'[i]$  and  $\forall j < i \ \rho[j] = \rho'[j]$ . The cache will be reset each time the value of  $A_{i-1}$  changes (if it exists). We will denote this variable as the *reset variable* of  $N$ .

There are two special cases that the above specification of effective cache-domains does not consider:

1.  $A_1 \neq B_1$ . This means that the cache-domain of  $N$  is equivalent to the effective cache-domain of  $N$ , in which case the cache is never reset. Caching proceeds as normal.
2.  $A_i = B_i, \forall i$ . This means that the effective cache-domain of  $N$  is empty. However, this also means that  $CD(N) = CD(N^P) \cup \{\text{var}(N^P)\}$ , which we proved previously indicates a dead cache. Hence, an empty effective cache-domain indicates a dead cache.

The following theorem proves the correctness of reusing memory in subcaching:

**Theorem 3.** *When the value of  $N$ 's reset variable changes, no current cache values at  $N$  will ever be queried again.*

*Proof.* Let  $\rho = [A_1, \dots, A_q]$  be an ordering over the cache-domain of  $N$ , and let  $\rho' = [B_1, \dots, B_r, \text{var}(N^P)]$  be an ordering over  $N^P$ 's cache-domain and  $N^P$ 's variable, ordered as defined above. Note that  $\rho'$  is a superset of  $\rho$ . Let  $A_{i-1}$  be the reset variable for node  $N$ . Let  $\mathbf{p}' \in \mathcal{D}(\rho')$  be a context over the variables in  $\rho'$ . We will define the projection of a context  $\mathbf{p}' \in \mathcal{D}(\rho')$  to the variables in  $\rho$ , denoted  $\Downarrow_\rho \mathbf{p}'$  as the context  $\mathbf{p}$  such that  $X \in \rho$  and  $(X = x) \in \mathbf{p}' \Rightarrow (X = x) \in \mathbf{p}$ .

In order for a cache value to be successfully hit, there must exist two contexts  $\mathbf{p}'_1$  and  $\mathbf{p}'_2 \in \mathcal{D}(\rho')$  such that  $\mathbf{p}'_1 \neq \mathbf{p}'_2$  and  $\Downarrow_\rho \mathbf{p}'_1 = \Downarrow_\rho \mathbf{p}'_2$ . This means that there exists a variable  $Y \in \rho' - \rho$  such that  $\Downarrow_{\{Y\}} \mathbf{p}'_1 \neq \Downarrow_{\{Y\}} \mathbf{p}'_2$ . For two contexts to be the same subsequent to a change in the value of  $A_{i-1}$ , one of these variables that have different values in  $\mathbf{p}'_1$  and  $\mathbf{p}'_2$  must exist in the ancestry of  $A_{i-1}$ 's node. However, no variable exists that can meet all these criteria at once: if  $Y$  is in  $CD(N^P)$  and is before  $A_{i-1}$  in the ordering, then it

Table 1: The amount of memory required for caching over networks from the Bayesian network repository.

Network	Complete (MB)	Live (MB)	Sub (MB)
Barley	16.14	7.737	0.4580
Diabetes	4.172	2.252	0.6250
Link	1475	16.40	12.20
Mildew	1.497	0.4219	0.1058
Munin1	170.7	90.06	41.81
Munin2	3.494	2.065	0.4560
Munin3	3.605	1.879	0.6227
Munin4	19.17	6.793	0.4271
Pigs	1.052	0.4860	0.1234
Water	10.32	2.170	1.885

must also exist in  $CD(N)$ , which contradicts the statement  $Y \in \rho' - \rho$ . Hence, we assume that the two contexts  $\mathbf{p}'_1$  and  $\mathbf{p}'_2$  cannot be generated across the changing of the value of  $A_{i-1}$ .  $\square$

Algorithm  $\mathcal{P}$  requires very little modification to accommodate subcaching. We need to know when the value of the reset variable of node  $N$  changes. Hence, by storing the current value of this variable at  $N$ , we can easily determine when it changes, and reset the cache appropriately. At each node, we store the variable  $R_N$  (the reset variable of  $N$ ), as well as  $r_N$ , the current value of  $R_N$ . When that value changes, we call *ResetCache* on node  $N$ , that resets the value of  $N$ 's cache.

The memory savings due to subcaching are quite substantial. Consider the etree in Figure 2. Of the three remaining live caches, the subcaches of nodes  $N_D$ ,  $N_T$ , and  $N_V$  are  $\{B\}$ ,  $\{C\}$ , and  $\{T\}$ , respectively (shown in the Figure as underlined variables). In other words, the variable  $O$  does not belong to any of the subcaches. This pruning means that only 6 caches values are needed to compute over this network, without sacrificing the runtime of algorithm  $\mathcal{P}$ .

Table 1 compares the memory requirements of complete caching, live caching, and subcaching over several well-known networks from the Bayesian network repository.<sup>2</sup> The networks were constructed using a variable ordering generated by the *Netica* software package, which uses a search-based method to find orderings of low-width.<sup>3</sup> The results empirically demonstrate that subcaching reduces the overall size of the caches considerably, even from live caching. Seven of the ten networks required less than 1 MB of cache storage. Again, this reduction in space does not affect the time complexity of algorithm  $\mathcal{P}$  – it remains  $\mathcal{O}(n \exp(w))$ .

## Exploiting Subcaching

The amount of cache memory in an etree constructed from a Bayesian network depends on a given elimination ordering over the network's variables. An ordering of low treewidth results in smaller cache-domains in the nodes of an etree,

<sup>2</sup><http://www.cs.huji.ac.il/labs/compbio/Repository/>.

<sup>3</sup>*Netica* is a software package distributed by *Norsys Software Corporation*.

which results in smaller memory requirements. Choosing an optimal ordering is NP-hard; however, several good heuristics exist (Kjaerulff 1990). We used the popular *min-fill* heuristic to generate the variable ordering for constructing the etree in Figure 2. The *min-fill* heuristic (Kjaerulff 1990) is lightweight, and in practice yields good variable orderings. The algorithm for computing an elimination ordering can be simply modified to build a etree (Grant & Horsch 2005; 2006b).

In any algorithm for computing a variable ordering, selecting a next variable often requires tie-breaking. The *min-size* heuristic (Kjaerulff 1990) is often used to break ties, although this is typically not enough to resolve every tie. In practice, some random tie-breaking procedure must be used. This means that there is more than one possible elimination ordering for each Bayesian network, even using these heuristics. Often however, breaking the ties randomly does not affect the width of the ordering considerably, and there is typically no preference between two variable orderings of the same width.

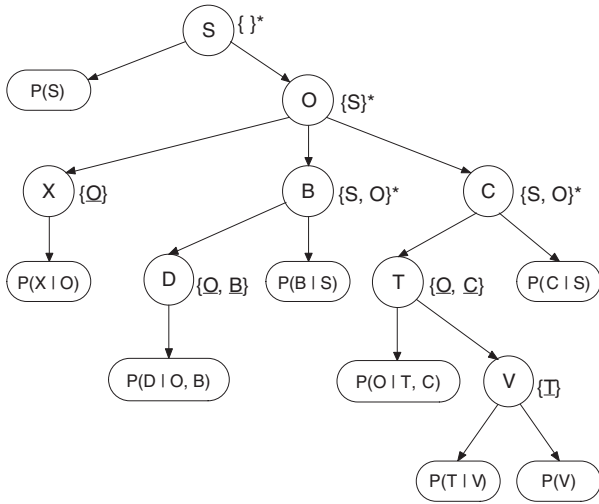


Figure 5: The network from Figure 1, arranged in a different etree.

However, the situation is different when subcaching is to be used. Consider the etree in Figure 5, which was constructed using the *min-fill* heuristic. The variable ordering used to construct this tree has the same width as the one to construct the etree in Figure 2. However, this network has one more live cache, and the variable  $O$  becomes part of the subcache in the caches of  $N_D$  and  $N_T$ . This doubles the amount of memory required for caching. In other words, while the random tie-breaking procedures do not seem to affect the width of the variable ordering significantly, it does seem to have a great impact on the amount of memory required for subcaching.

To further test the extent of this effect, we generated 50 etrees over each repository Bayesian network using the *min-fill* heuristic, breaking ties with *min-size*, and breaking any further ties randomly. We then measured the width of the variable ordering, and the storage requirements for caching.

Table 2: Memory requirements of etrees constructed using *min-fill* and *min-size*.

Network	$\mu$ (MB)	$\sigma$ (MB)	Min-Max (MB)	Width
Barley	3.118	6.176	0.1954-42.34	7-7
Diabetes	1.124	0.1298	0.8623-1.449	4-4
Link	2.643	1.899	0.9588-10.16	13-17
Mildew	0.3848	0.4033	0.0738-1.943	4-4
Munin1	21.01	29.92	1.865-197.9	11-11
Munin2	0.3522	0.0921	0.2282-0.7729	7-8
Munin3	0.4842	0.3635	0.1304-1.336	7-7
Munin4	0.9766	0.5155	0.5010-2.955	8-8
Pigs	0.1835	0.1138	0.0535-0.5349	10-10
Water	0.2027	0.1554	0.0224-0.6195	10-10

Table 2 shows the results of this test. The first two columns show the mean and standard deviation of the memory required for caching over the generated etrees for each network. The next column shows the range of the cache memory requirements over the generated etrees. The final column shows the range of the widths of the computed variable orderings.

From the table, we can see that breaking ties randomly has almost no effect on the width of the final ordering (with the exception of the *Link* network). However, the variance in the memory requirements of caching is quite high. In seven of the ten networks, the ratio between the smallest tree and largest tree (in terms of memory requirements) is over 10; in *Barley* and *Munin1*, this ratio exceeds 100. These results suggest that while the mentioned heuristics provide low-width orderings (and thus good runtimes) pretty reliably, they cannot by themselves reliably provide low-memory etrees.

To address this problem, we propose another heuristic when breaking ties between variables. Where *min-fill* and *min-size* attempt to minimize the width of the variable ordering, the new heuristic, which we will call *min-cache*, will attempt to minimize the amount of memory required for caching. We cannot directly calculate the amount of cache memory that will be required in the subtree below node  $N_V$ , since this depends on the order in which the variables in  $N_V$ 's cache-domain are eliminated, which is unknown at the time of  $V$ 's selection. However, if we assume for each node in  $N_V$ 's subtree that the chosen ordering is optimal with respect to the number of variables excluded from the subcache, then we can easily calculate a lower bound on the amount of cache memory in  $N_V$ 's subtree. This lower bound will be the *min-cache* value of variable  $V$ . Hence, when choosing between variables with equivalent *min-fill* and *min-size* values, we will choose the variable with the smallest *min-cache* value. Any further ties will be broken randomly.

To test the effectiveness of *min-cache*, we performed the same experiments as those that generated the data in Table 2, using *min-cache* to further reduce ties. Table 3 summarizes the results.

The table demonstrates that *min-cache* improves the mean memory requirements for each network except for *Diabetes*. This improvement has no effect on the width of the order-

Table 3: Memory requirements of etrees constructed using *min-fill*, *min-size*, and *min-cache*.

Network	$\mu$ (MB)	$\sigma$ (MB)	Min-Max (MB)	Width
Barley	0.6281	0.0004	0.6276-0.6285	7-7
Diabetes	1.663	0.0214	1.620-1.714	4-4
Link	1.858	0.3671	1.227-2.266	15-15
Mildew	0.1978	0.0309	0.1608-0.2381	4-4
Munin1	0.6186	0.4556	0.1373-1.704	11-11
Munin2	0.3134	0.0062	0.3012-0.3278	8-8
Munin3	0.1429	0.0021	0.1390-0.1470	7-7
Munin4	0.6230	0.0523	0.5205-0.7463	8-8
Pigs	0.0410	0.0001	0.0408-0.0412	10-10
Water	0.1525	0.0487	0.0996-0.2290	10-10

ings, meaning that we obtain these smaller memory requirements on average without significantly affecting the runtime of the algorithm.

In all cases, the variance of the memory requirements was reduced considerably by using *min-cache*. The ratio between the smallest and largest etrees (in terms of cache memory) has been reduced for all networks, and is less than 2 for all but two networks. Combined with the lower average memory requirements, this means that we can expect to obtain a good etree in fewer tries, on average.

## Conclusions

In this paper, we have presented *subcaching*, a new technique for reducing the memory requirements of caching in etrees. The technique demonstrates how two values in a node's cache can occupy the same memory, without collision, based on their non-overlapping lifetimes. Exploiting this property empirically showed a substantial improvement in the amount of caching memory required by the etrees computed from our test networks, without affecting the runtime of the algorithm for computing over these etrees.

We also presented a heuristic, *min-cache*, for computing variable elimination orderings that attempts to reduce the overall memory requirements of an etree's cache. *min-cache* was used in conjunction with the *min-fill* and *min-size* heuristic, to resolve ties that occur during variable selection. The heuristic breaks ties by choosing the variable that minimizes the lower bound on the amount of memory required for caching in that variable's subtree. Our experiments show that in almost all cases, the average cache size of the etrees for each network was reduced when compared to using *min-fill* and *min-size* alone. Amongst all experimental networks, the variance on the amount of memory required was reduced considerably when using *min-cache*. This means that by using *min-cache*, the chance of obtaining an etree with large memory requirements is smaller than when it is not used.

For future work, note that while the values in a particular cache could share a spot in memory, two values from different caches could not share memory. However, it should be no surprise that there are pairs of cache values between caches whose lifetimes do not overlap with each other as well. Thus, by sharing memory *between* nodes, rather than simply *within* nodes, we hope to reduce the memory require-

ments of computing in etrees even further.

Finally, while *min-cache* chose smaller-memory etrees on average, the smallest etrees overall were still found without using this heuristic. Hence, we will continue to explore other options for heuristics that will not only provide the best result on average, but also find the smallest tree overall, while still maintaining the low-variance results of *min-cache*.

## References

- Allen, D., and Darwiche, A. 2003a. New advances in inference by recursive conditioning. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, 2–10.
- Allen, D., and Darwiche, A. 2003b. Optimal time–space tradeoff in probabilistic inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 969–975.
- Allen, D.; Darwiche, A.; and Park, J. D. 2004. A greedy algorithm for time-space tradeoff in probabilistic inference. In *Proceedings of the Second European Workshop on Probabilistic Graphical Models*, 1–8.
- Cooper, G. F. 1990. Bayesian belief-network inference using recursive decomposition. Technical Report KSL-90-05, Knowledge Systems Laboratory, Stanford, CA, 94305, USA.
- Darwiche, A., and Hopkins, M. 2001. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI*, 2143. Springer-Verlag, 180–191.
- Darwiche, A. 2000. Recursive Conditioning: Any-space conditioning algorithm with treewidth-bounded complexity. *Artificial Intelligence* 5–41.
- Dechter, R. 1996. Bucket Elimination: A Unifying Framework for Probabilistic Inference Algorithms. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 211–219.
- Grant, K., and Horsch, M. 2005. Conditioning Graphs: Practical Structures for Inference in Bayesian Networks. In *Proceedings of the The Eighteenth Australian Joint Conference on Artificial Intelligence*, 49–59.
- Grant, K., and Horsch, M. 2006a. Exploiting Dynamic Independence in a Static Conditioning Graph. In *Proceedings of the Nineteenth Canadian Conference on Artificial Intelligence*.
- Grant, K., and Horsch, M. 2006b. Methods for Constructing Balanced Elimination Trees and Other Recursive Decompositions. In *Proceedings of the the Nineteenth International Florida Artificial Intelligence Research Society Conference*.
- Grant, K. 2006. *Conditioning Graphs: Practical Structures for Inference in Bayesian Networks*. Ph.D. Dissertation, University of Saskatchewan, Computer Science Department.
- Kjaerulff, U. 1990. Triangulation of graphs - algorithms giving small total state space. Technical Report R 90-09, Dept. of Mathematics and Computer Science, Strandvejan, DK 9000 Aalborg, Denmark.
- Lauritzen, S., and Spiegelhalter, D. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society* 50:157–224.
- 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.