Recursive Conditioning

Any–space conditioning method with treewidth–bounded complexity

Adnan Darwiche
Computer Science Department
University of California
Los Angeles, CA 90095
darwiche@cs.ucla.edu

October 3, 2000

Abstract

We introduce an any–space algorithm for exact inference in Bayesian networks, called recursive conditioning. On one extreme, recursive conditioning takes $O(n)$ space and $O(n \exp(w \log n))$ time—where n is the size of a Bayesian network and w is the width of a given elimination order—therefore, establishing a new complexity result for linear–space inference in Bayesian networks. On the other extreme, recursive conditioning takes $O(n \exp(w))$ space and $O(n \exp(w))$ time, therefore, matching the complexity of state–of–the–art algorithms based on clustering and elimination. In between linear and exponential space, recursive conditioning can utilize memory at increments of X-bytes, where X is the number of bytes needed to store a floating point number in a cache. Moreover, the algorithm is equipped with a formula for computing its average running time under any amount of space, hence, providing a valuable tool for time–space tradeoffs in demanding applications. Recursive conditioning is therefore the first algorithm for exact inference in Bayesian networks to offer a smooth tradeoff between time and space, and to explicate a smooth, quantitative relationship between these two important resources.

1 Introduction

Conditioning algorithms have been of major interest to the Bayesian network community since the introduction of cutset conditioning as one of the first methods for inference in multiply–connected networks [25, 26]. Most of the interest in conditioning methods stems from their intuitive appeal as they are based on the principle of reasoning by cases—a very common form of human reasoning. Reasoning by cases is amenable to parallelization, facilitates time–space tradeoff [12], and appears to be best positioned for exploiting context–specific independence [2].

The best known conditioning method is *cutset conditioning,* which is also known as the *loop–cutset method* [25, 26, 28]. The best known fact about this method is its linear space complexity, which is very attractive when compared to the exponential space complexity (in treewidth) of state–of–the–art algorithms based on clustering [19, 30, 18, 17] and elimination [29, 20, 11, 32]. The worst known fact about cutset conditioning is its time complexity, which is exponential in the size of loop–cutset. The loop–cutset can be quite large, even for networks which can be solved in linear time and space using other methods.

There have been improvements and variations on cutset conditioning which reduce its running time under certain conditions [16, 5, 13]. For example, dynamic conditioning [5] and local conditioning [13] are known to take linear time on some networks for which cutset conditioning is known to be exponential. But both methods lose the linear space complexity of cutset conditioning. Moreover, neither the time nor the space complexity of these methods are bounded formally. Bounded conditioning [16] will also reduce the running time of cutset conditioning, but at the expense of returning approximate answers.

In this paper, we introduce a method for conditioning, *recursive conditioning,* which is characterized by its *any–space* behavior as it can use as much space as is made available to it.

- On one extreme, recursive conditioning uses linear space, leading to a time complexity of $O(n \exp(w \log n))$, where n is the number of network nodes and w is the width of a given elimination order.

- On the other extreme, recursive conditioning uses $O(n \exp(w))$ space, leading to a time complexity of $O(n \exp(w))$.

Therefore, if given enough space, recursive conditioning will match the space and time complexity of clustering and elimination methods. However, if less space is given to recursive conditioning, its running time will increase until it hits $O(n \exp(w \log n))$ with linear space, which is a new complexity result for inference in Bayesian networks. Interestingly enough, this running time is incomparable to the running time of cutset conditioning, $O(n \exp(c))$, where c is the loop–cutset size.

To introduce the key intuition underlying recursive conditioning, we note that the main power of conditioning is its ability to reduce network connectivity. In cutset conditioning, this power is exploited for *reducing* a multiply–connected network into a singly–connected network that can be solved using the (linear) polytree algorithm. In recursive conditioning, however, this power is exploited to *decompose* a network into smaller subnetworks that can be solved independently. Each of these subnetworks is then solved recursively using the same method, until we reach boundary conditions where we try to solve single–node networks.

The decomposition of a problem into smaller problems is a classical example of the divide–and–conquer computational paradigm. Alan George is credited for having used this technique in 1973 to solve systems of linear equations [15]. His algorithm, termed *nested dissection,* was later generalized [21] and applied to other problems, such as network reliability [27]. The application of nested dissection to inference in Bayesian networks was also investigated independently by Gregory Cooper in [3], under the name *recursive decomposition.* We will have more to say about the relationship between recursive conditioning and previous works in Section 6.4.

A key concept in cutset conditioning is that of a *loop–cutset,* which is a set of nodes that when instantiated will render the network singly–connected. In recursive conditioning, the conditioning process is driven by a new graphical structure, which we call a *dtree.* This tree specifies many cutsets, each to be used at a different level of the recursive conditioning process. As we shall see later, a dtree is a simple concept and it is extremely easy to construct one. There are typically many dtrees for a Bayesian network, any of which can be used to drive recursive conditioning. Some of these dtrees, however, will lead to more work than others. The quality of a dtree is measured by its *width,* which we also introduce in this paper, and the different complexities of recursive conditioning will be expressed in terms of the width of used dtree. We shall show that given an elimination order of width w for a Bayesian network, we can construct in linear time a corresponding dtree of width $\leq w$.

One of the more interesting things about recursive conditioning is how it utilizes space. Recursive conditioning solves a network by decomposing it into smaller, independent subnetworks. A close examination of the algorithm reveals that it can solve some subnetworks many times, therefore, leading to many redundant computations. By caching the solutions of subnetworks, recursive conditioning will avoid such redundancy. This will reduce its running time, but will also increase its space requirements. When all redundancies are avoided, recursive conditioning will run in $O(n \exp(w))$ time, but it will also take that much space to store the solutions of subnetworks. What is important, however, is that we can cache as many results as our available memory would allow, leading to any–space behavior. Moreover, we shall provide a formula which can be used to compute (in linear time) the average running time of recursive conditioning under any amount of available space. This equips the algorithm with a very important tool for time–space tradeoff, which appears to be necessary for certain applications:

- One class of such applications involves computationally demanding networks, whose memory requirements (using classical algorithms) exceed existing resources. Not only can one use recursive conditioning on these networks, but one can also compute the extra time entailed by running under the limited memory. This extra time can then be used to make a decision on whether to acquire more memory.

- Another class of applications involves embedded systems, where only a fixed amount of memory (typically modest) is available for the Bayesian network application. Here, one can subtract the memory needed to store the recursive conditioning code and associated network, and then run the algorithm under the remaining amount of memory. Recall that recursive conditioning allows one to tradeoff space

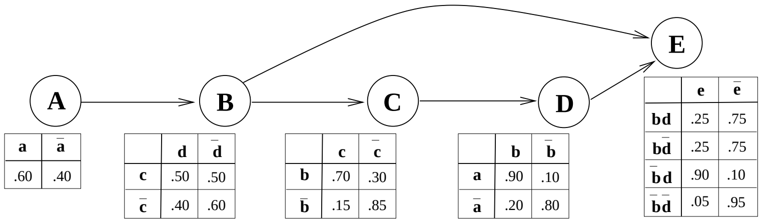


Figure 1: A Bayesian network.

at increments of X–bytes, where X is the number of bytes needed to store a floating point number in a cache.

- Finally, applications in which Bayesian network inference runs as a background process in an operating system are becoming more popular. In these applications, memory usage should be as invisible as possible to ensure transparency with respect to end–users. A theory of any–space reasoning appears essential in reducing the used memory in these applications.

This paper is structured as follows. In Section 2, we discuss the basic intuition behind conditioning and illustrate its computational power. We also contrast cutset conditioning with recursive conditioning, therefore, introducing the key principle behind our presented method. In Section 3, we discuss the linear–space version of recursive conditioning and introduce the concept of a dtree. In Section 4, we introduce the exponential–space version of recursive conditioning and provide a number of its properties. We then introduce the any–space version of recursive conditioning in Section 5 and show that the first two versions are only extremes. In Section 6, we study the relationship between dtrees, elimination orders and jointrees, showing polynomial time transformations from one to the other. We finally close in Section 7 with some concluding remarks.

2 The Computational Power of Making Assumptions

A very common form of human reasoning—which is dominant in mathematical proofs—is that of reasoning by cases or assumptions. To solve a complicated problem, we try to simplify it by considering a number of cases which correspond to a set of mutually exclusive and exhaustive assumptions. We then solve each of the cases under its corresponding assumption, and combine the results to obtain a solution to the original problem. In probabilistic reasoning, this is best illustrated by the identity:

$$\mathsf{Pr}(\mathbf{x}) = \sum_{\mathbf{c}} \mathsf{Pr}(\mathbf{x}, \mathbf{c}), \tag{1}$$

where $\mathbf{x}$ and $\mathbf{c}$ are instantiations of variables $\mathbf{X}$ and $\mathbf{C}$, respectively.[1] Here, we try to compute the probability of instantiation $\mathbf{x}$ by considering a number of cases, each corresponding to an assumption $\mathbf{c}$ (an instantiation of variables $\mathbf{C}$). We then compute the probability of $\mathbf{x}$ under each assumption $\mathbf{c}$, and add up the results to get the probability of $\mathbf{x}$.

In general, solving a problem becomes easier when we make assumptions and probabilistic reasoning is no exception. Consider, for example, the multiply–connected network $\mathcal{N}$ of Figure 1. Suppose further that our goal is to compute the probability of some event, say e, with respect to this network, denoted $\mathsf{Pr}^{\mathcal{N}}(e)$. If we perform this computation under the assumption b, therefore computing $\mathsf{Pr}^{\mathcal{N}}(e, b)$, we can use the singly–connected network $<\mathcal{N}, b>$ of Figure 2 instead. That is, we are guaranteed to obtain the same probability for e, b with respect to either network.

[1] We are using the standard notation: variables are denoted by upper–case letters and their values by lower–case letters. Sets of variables are denoted by bold–face upper–case letters and their instantiations are denoted by bold–face lower–case letters. For variable A and value a, we often write a instead of $A{=}a$. We will use the same convention for variables $\mathbf{A}$ and their instantiation $\mathbf{a}$. For a variable A with values true and false, we use a to denote $A{=}\mathsf{true}$ and $\overline{a}$ to denote $A{=}\mathsf{false}$.

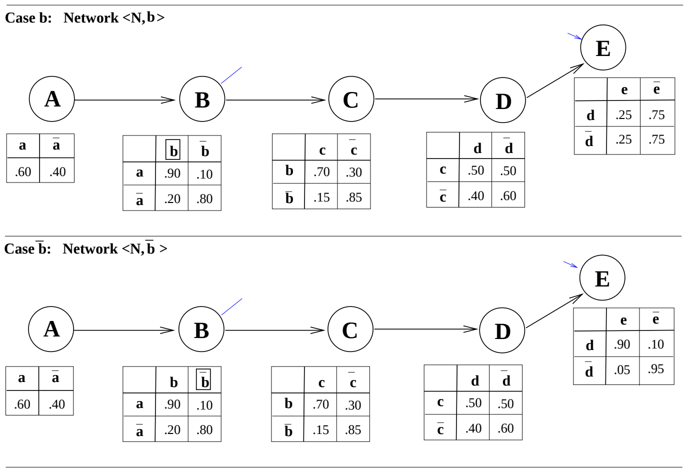


Figure 2: Instantiating variables to render the network singly–connected.

Note that network $<\mathcal{N}, b>$ of Figure 2 was obtained using a *local transformation* to network $\mathcal{N}$ of Figure 1:

1. we deleted the edge $B \rightarrow E$;[2]
2. we reduced the conditional probability table (CPT) of node E from

	e	$\overline{e}$
bd	.25	.75
$b\overline{d}$	.25	.75
$\overline{b}d$	.90	.10
$\overline{b}\overline{d}$	.05	.95

to

	e	$\overline{e}$
d	.25	.75
$\overline{d}$	.25	.75

in order to reflect the assumption that B is instantiated to b.

3. we recorded the observation b (shown pictorially by a box around the value b in the CPT for B).

Note that the result of the above *instantiation operation* is not simply a Bayesian network, but a Bayesian network together with some associated evidence.

In general, we will use the term *instantiated network* to refer to the pair $<\mathcal{N}, \mathbf{e}>$, which results from instantiating $\mathbf{e}$ in network $\mathcal{N}$ as indicated above. Moreover, we will write $\mathsf{Pr}^{<\mathcal{N},\mathbf{e}>}(\mathbf{x})$ to refer to the probability of instantiation $\mathbf{e}, \mathbf{x}$ with respect to the instantiated network $<\mathcal{N}, \mathbf{e}>$. For example, in Figure 2, we will write $\mathsf{Pr}^{<\mathcal{N},b>}(e)$ to mean the probability of b, e with respect to the instantiated network $<\mathcal{N}, b>$. This means that when we instantiate evidence $\mathbf{e}$ in network $\mathcal{N}$, we will only be computing probabilities of events of the form $\mathbf{e}, \mathbf{x}$ with respect to the instantiated network.

An instantiated network $<\mathcal{N}, \mathbf{e}>$ will always be

- equivalent to network $\mathcal{N}$ as far as computing the probability of any instantiation $\mathbf{e}, \mathbf{x}$.[3]
- less connected than network $\mathcal{N}$ (unless every variable in $\mathbf{e}$ is a leaf node in network $\mathcal{N}$).

[2] We can also delete the edge $B \rightarrow C$ from network $<\mathcal{N}, b>$, simplifying it even further.

[3] A common confusion is that if Pr is the probability distribution specified by the original network $\mathcal{N}$, then $\mathsf{Pr}(. \mid \mathbf{e})$ is the probability distribution specified by the instantiated network $<\mathcal{N}, \mathbf{e}>$. This is not true! The only relation between the two distributions is that they agree on the probability of any instantiation $\mathbf{e}, \mathbf{x}$ [5]. We believe that part of the confusion stems from the term *conditioned network*, which we and others have used in the past to refer to $<\mathcal{N}, \mathbf{e}>$. This is why we avoid the term in this paper, and use *instantiated network* instead.

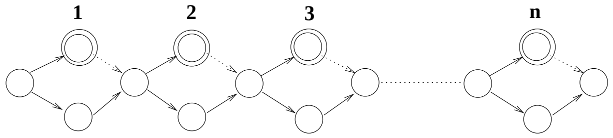


Figure 3: A Bayesian network with a large loop–cutset.

This very important result, which formalizes the computational power of making assumptions, is the key result on which conditioning methods are based. To state this result more formally, we have:

$$\mathsf{Pr}^{<\mathcal{N},\mathbf{e}>}(\mathbf{x}) = \mathsf{Pr}^{\mathcal{N}}(\mathbf{e},\mathbf{x}), \qquad \textbf{(Conditioning)} \qquad (2)$$

for any instantiation $\mathbf{x}$.

Cutset conditioning was the first method to identify this computational power and it exploited it by identifying a set of variables $\mathbf{C}$, known as a *loop–cutset*, which when instantiated will render the network singly–connected. It then performed a case analysis on the instantiations of $\mathbf{C}$, reducing each case to that of solving a singly–connected network (using the polytree algorithm). Specifically, to compute the probability of any evidence $\mathbf{e}$, cutset conditioning uses Equation 2 as follows:

$$\mathsf{Pr}^{\mathcal{N}}(\mathbf{e}) = \sum_{\mathbf{c}} \mathsf{Pr}^{\mathcal{N}}(\mathbf{e},\mathbf{c}) = \sum_{\mathbf{c}} \mathsf{Pr}^{<\mathcal{N},\mathbf{c}>}(\mathbf{e}). \qquad \textbf{(Cutset Conditioning)} \qquad (3)$$

This leads to $O(\exp(\mid \mathbf{C} \mid))$ calls to the polytree algorithm, one call for each singly–connected network $<\mathcal{N},\mathbf{c}>$ ($\mid \mathbf{C} \mid$ is the number of variables in the loop–cutset $\mathbf{C}$).

The main problem with cutset conditioning is that a large loop–cutset will lead to a blow up in the number of cases that it has to consider. In Figure 3, for example, the loop–cutset contains n variables, leading cutset conditioning to consider 2^n cases (when all variables are binary). It is worth mentioning, however, that clustering and elimination methods can solve this network in linear time. Again, although a number of improvements have been suggested to reduce the number of cases considered by cutset conditioning [16, 5, 13, 2], the best bound we currently have on the worst–case complexity of any linear–space conditioning method is the one stating that complexity is exponential in the size of loop–cutset.

In this paper, we propose another conditioning method which *exploits assumptions differently than they are exploited by cutset conditioning.* Specifically, instead of using assumptions to *singly–connect* a Bayesian network, we will use such assumptions to *decompose* a Bayesian network. By decomposition, we mean the process of splitting the network into smaller, disconnected pieces that can be solved independently. Consider again the network $\mathcal{N}$ in Figure 1. Figure 4 shows how we can decompose network $\mathcal{N}$ into two subnetworks, $\mathcal{N}^l$ and $\mathcal{N}^r$, by instantiating variable B. Moreover, Figure 5 shows how we can further decompose network $\mathcal{N}^r$ into two subnetworks, $\mathcal{N}^{rl}$ and $\mathcal{N}^{rr}$, by instantiating variable C. Note that subnetwork $\mathcal{N}^{rl}$ contains a single–node and cannot be decomposed further.

We can always use this recursive decomposition process to reduce the computation of $\mathsf{Pr}^{\mathcal{N}}(\mathbf{e})$ into the computation of probabilities with respect to single–node networks. Specifically, let $\mathbf{C}$ be a set of variables such that the instantiated network $<\mathcal{N},\mathbf{c}>$ is decomposed into two disconnected subnetworks, $<\mathcal{N},\mathbf{c}>^l$ and $<\mathcal{N},\mathbf{c}>^r$. We then have:

$$\mathsf{Pr}^{\mathcal{N}}(\mathbf{e}) = \sum_{\mathbf{c}} \mathsf{Pr}^{<\mathcal{N},\mathbf{c}>}(\mathbf{e}) = \sum_{\mathbf{c}} \mathsf{Pr}^{<\mathcal{N},\mathbf{c}>^l}(\mathbf{e}_l)\mathsf{Pr}^{<\mathcal{N},\mathbf{c}>^r}(\mathbf{e}_r), \qquad \textbf{(Recursive Conditioning)} \qquad (4)$$

where $\mathbf{e}_l$ and $\mathbf{e}_r$ are the subsets of instantiation $\mathbf{e}$ pertaining to subnetworks $<\mathcal{N},\mathbf{c}>^l$ and $<\mathcal{N},\mathbf{c}>^r$, respectively. This is the characteristic equation of recursive conditioning, which parallels Equation 3 of cutset conditioning. Note that each of the queries $\mathsf{Pr}^{<\mathcal{N},\mathbf{c}>^l}(\mathbf{e}_l)$ and $\mathsf{Pr}^{<\mathcal{N},\mathbf{c}>^r}(\mathbf{e}_r)$ can be decomposed using the same method recursively, until we reach queries with respect to single–node networks.

This is a very simple, universal process which can be used to compute the probability of any instantiation. It is a nondeterministic process though since there are many ways in which we can decompose a Bayesian

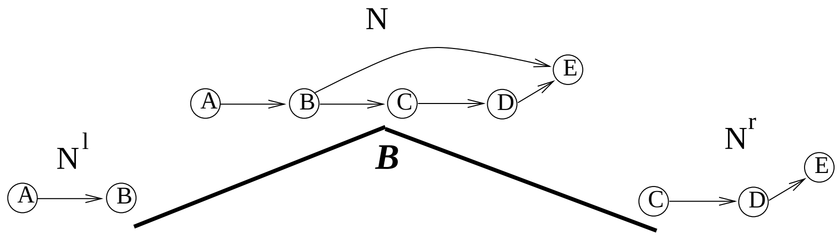


Figure 4: Decomposing a Bayesian network by instantiating variable B.

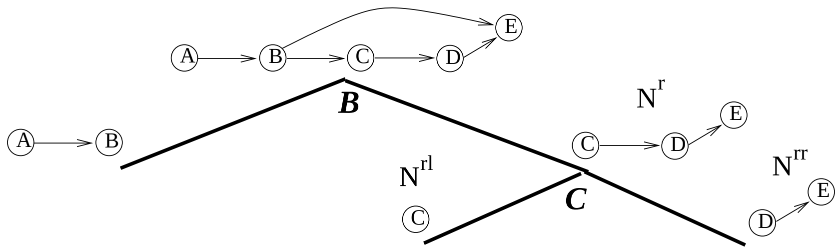


Figure 5: Decomposing a Bayesian network by instantiating variables B and C.

network into disconnected subnetworks. The question then is: which decomposition should we use? As it turns out, any decomposition will be valid, but some decompositions will lead to less work than others. The key is therefore to choose decompositions that will minimize the amount of work done, and to bound it in some meaningful way. We will address this issue later but we first provide a formal tool for capturing a certain decomposition policy, which is the subject of the following section.

Before we conclude this section, we highlight three key differences between cutset conditioning and recursive conditioning. First, the role of a cutset is different: in cutset conditioning, it is used to singly–connect a network; in recursive conditioning, it is used to decompose a network into disconnected subnetworks. In Figure 1, for example, variable C constitutes a valid loop–cutset since it would render the network singly–connected when instantiated. However, instantiating variable C will not decompose the network into smaller subnetworks; hence, C is not a valid cutset in recursive conditioning. Next, there is a single cutset in cutset conditioning, which is used at the very top level to generate a number of singly–connected networks. But there are many cutsets in recursive conditioning, each of which is used at a different level of the recursion. Finally, the boundary condition in cutset conditioning is that of reaching a singly–connected network, but the boundary condition in recursive conditioning is that of reaching a single–node network.

3 Inference by Recursive Conditioning

The method of recursive conditioning is quite simple in concept: we condition on a cutset to decompose the Bayesian network into smaller, disconnected subnetworks and then solve each of the subnetworks recursively using the same method. This method is an example of the classical divide–and–conquer paradigm, which is quite prevalent in computer algorithms. The effectiveness of this method, however, is very much dependent on our choice of cutsets at each level of the recursive process. Recall that the number of cases we have to consider at each level is exponential in the size of used cutset. Therefore, we want to choose our cutsets in order to minimize the total number of considered cases.

Before we can address this issue, however, we need to introduce a formal tool for capturing the collection

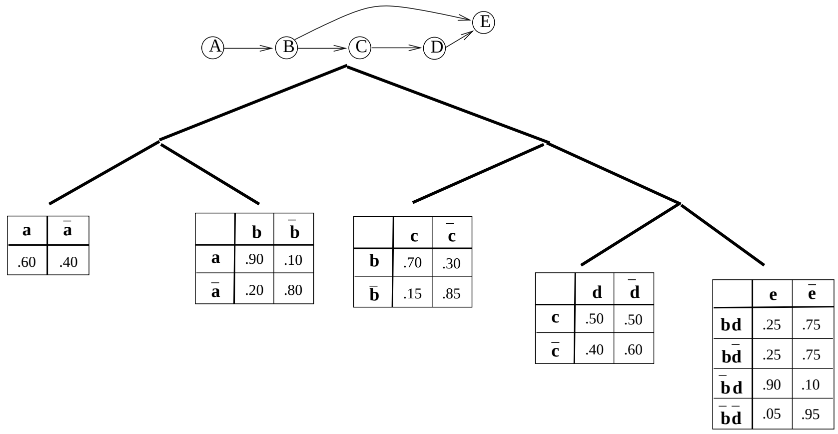


Figure 6: A dtree for the Bayesian network in Figure 1.

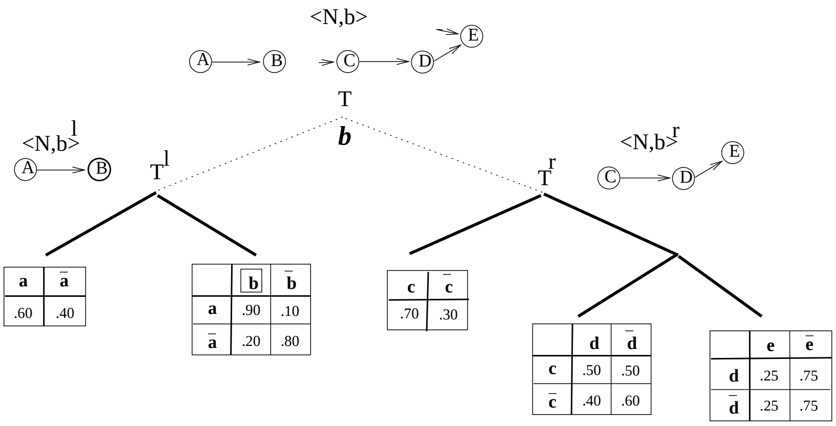


Figure 7: Instantiating variable B to b in a dtree. Note how the CPTs of variables C and E (children of variable B) have been reduced.

of cutsets employed by recursive conditioning. We call this tool a dtree:

Definition 1 *A <u>dtree</u> for a Bayesian network is a full binary tree, the leaves of which correspond to the network CPTs.*

Recall that a full binary tree is a binary tree where each node has 2 or 0 children. Figure 6 depicts a dtree for the Bayesian network in Figure 1.

It is important to note that a dtree T for a Bayesian network $\mathcal{N}$ is simply a more structured representation of the network $\mathcal{N}$. That is, it contains all the information available in $\mathcal{N}$, and imposes in addition a tree structure on the CPTs of $\mathcal{N}$. Following standard conventions on binary trees, we will often not distinguish between a node and the dtree rooted at that node. Therefore, T will refer both to a dtree and the root of that dtree.

A dtree T suggests that we decompose its associated Bayesian network by instantiating variables that are shared by its left and right subtrees, T^l and T^r, which are denoted by $\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)$. In Figure 6, B is the only variable shared by the left and right subtrees. Figure 7 shows the result of instantiating B=b in the dtree. As a result of this instantiation, the instantiated network $<\mathcal{N}, b>$ was decomposed into two disconnected subnetworks, $<\mathcal{N}, b>^l$ and $<\mathcal{N}, b>^r$, each of which can be solved independently. What is most

Algorithm RC1

```
RC1(T)
  01. if T is a leaf node,
  02.  then return LOOKUP(T)
  03.  else p ← 0
  04.       for each instantiation c of uninstantiated variables in cutset(T) do
  05.           record instantiation c
  06.           p ← p + RC1(T^l)RC1(T^r)
  07.           un-record instantiation c
  08.       return p

LOOKUP(T)
  01. φ ← CPT of variable X associated with leaf T
  02. if X is instantiated,
  03.   then x ← recorded instantiation of X
  04.        p ← recorded instantiation of X's parents
  05.        return φ(x | p)      // φ(x | p) = Pr(x | p)
  06.   else return 1
```

Figure 8: Pseudocode for recursive conditioning.

important though, is that subtrees T^l and T^r are guaranteed to be dtrees for the subnetworks $<\mathcal{N}, b>^l$ and $<\mathcal{N}, b>^r$, respectively. Therefore, each of these subnetworks can be decomposed recursively using these subtrees. The process continues until we reach single–node networks, which cannot be decomposed further.

Figure 8 provides the pseudocode for algorithm RC1, which is an implementation of Equation 4 that uses dtree T to direct the decomposition process. There are two key observations about this algorithm. First, it does not compute cutsets dynamically, but it assumes that they have been precomputed as follows.

Definition 2 *The* <u>*cutset*</u> *of internal node T in a dtree is defined as follows:*

$$\mathsf{cutset}(T) \stackrel{def}{=} \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r) - \mathsf{acutset}(T),$$

where $\mathsf{acutset}(T)$*, called the a-cutset of T, is the union of cutsets associated with ancestors of node T in the dtree.*[4]

For the root T of a dtree, $\mathsf{cutset}(T)$ is simply $\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)$. But for a non-root node T, the cutsets associated with ancestors of T are excluded from $\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)$ since these cutsets are guaranteed to be instantiated when RC1 is called on node T.

The second observation about algorithm RC1 is that it does not really reduce CPTs when variables are instantiated. It simply "records" that variables have been instantiated, and "un-records" that when variables are de-instantiated. Given the implementation of LOOKUP, such recording/un-recording is all we need.

Theorem 1 (Soundness) *Suppose that T is a dtree for Bayesian network $\mathcal{N}$. Then* RC1$(T) = \mathsf{Pr}^{\mathcal{N}}(\mathbf{e})$*, where* $\mathbf{e}$ *is the instantiation recorded before* RC1 *is called.*

Therefore, to compute the probability of instantiation $\mathbf{e}$ with respect to network $\mathcal{N}$, all we have to do is construct a dtree T of network $\mathcal{N}$, compute the cutset for each node in T as given in Definition 2, record the instantiation $\mathbf{e}$, and finally call RC1(T).

Note that the more variables we instantiate before calling RC1, the less work it will do since that would reduce the number of instantiations it has to consider on line 04. Therefore, the worst case complexity for

[4] $\mathsf{acutset}(T) = \emptyset$ when T is a root node.

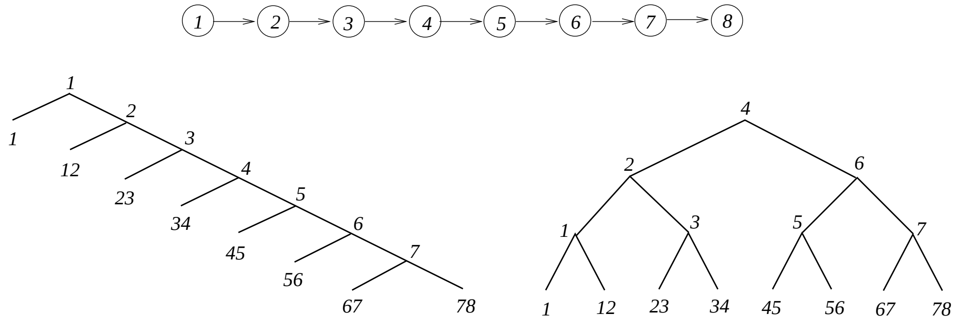


Figure 9: Two dtrees for a chain network, with their cutsets explicated. We are only showing the variables of CPTs as their probabilities do not matter for computing cutsets.

RC1 is when $\mathbf{e} = \mathsf{true}$. *In fact, in all of the following complexity analyses, we do assume that* $\mathbf{e} = \mathsf{true}$ *for which* RC1(T) *will simply return* $1 = \mathsf{Pr}^{\mathcal{N}}(\mathsf{true})$.

Clearly, the only space used by algorithm RC1 is that needed to store the dtree, which is linear in the network size. So what about the time complexity of RC1? We can measure this by counting the number of recursive calls made by RC1 as this number is proportional to its running time. Note that each call RC1(T), where T is an internal node, will generate two recursive calls for each instantiation of $\mathsf{cutset}(T)$. We can therefore count the number of recursive calls made by RC1 as follows.

Definition 3 *The* *__cutset width__* *of a dtree is the size of its largest cutset. The* *__a–cutset width__* *of a dtree is the size of its largest a–cutset.*

From now on, we will use $\mathbf{X}^{\#}$ to denote the number of instantiations of variables $\mathbf{X}$.

Theorem 2 *The total number of recursive calls made by* RC1 *to node* T *is* $\mathsf{acutset}(T)^{\#}$. *Moreover,* $\mathsf{acutset}(T)^{\#} = O(\exp(dw_c))$, *where* w_c *is the cutset width, and* d *is the depth of node* T.

In Figure 9, the cutset width of each dtree is 1. However, the a-cutset width is 7 for the first dtree and is 3 for the second. In general, for a chain of n variables, both dtrees will have a cutset width of 1, but the unbalanced dtree will have an a–cutset width of $O(n)$, while the balanced dtree will have an a–cutset width of $O(\log n)$. Therefore, RC1 will make $\Omega(\exp(n))$ recursive calls to some nodes in the first dtree, but will make $O(n)$ recursive calls to each node in the second dtree.[5]

This example illustrates the significance of used dtree on the complexity of recursive conditioning. In particular, we want to use a dtree which a–cutset width is minimal. We will provide in Section 6 two algorithms:

1. EL2DT: converts an elimination order of width w into a dtree with cutset width no greater than w.
2. BAL-DT: balances a dtree while keeping its cutset width $\leq w + 1$.

These two algorithms, and Theorem 2, lead to the following complexity of recursive conditioning:

Theorem 3 *Given an elimination order of width* w *and length* n, *and given a balanced dtree based on the order (using* EL2DT *and* BAL-DT*), the total number of recursive calls made by* RC1 *is* $O(n \exp(w \log n))$ *and the space it consumes is* $O(n)$.

This is basically the running time of recursive conditioning under linear space. We have a number of observations about this complexity. First, Appendix A discusses two experiments, each involving 1000

[5]This is worse than any of the known algorithms, which can solve this network in linear time under linear space. We shall see later, however, that RC2, the second version of recursive conditioning, will solve this network in linear time using a linear amount of caching (space).

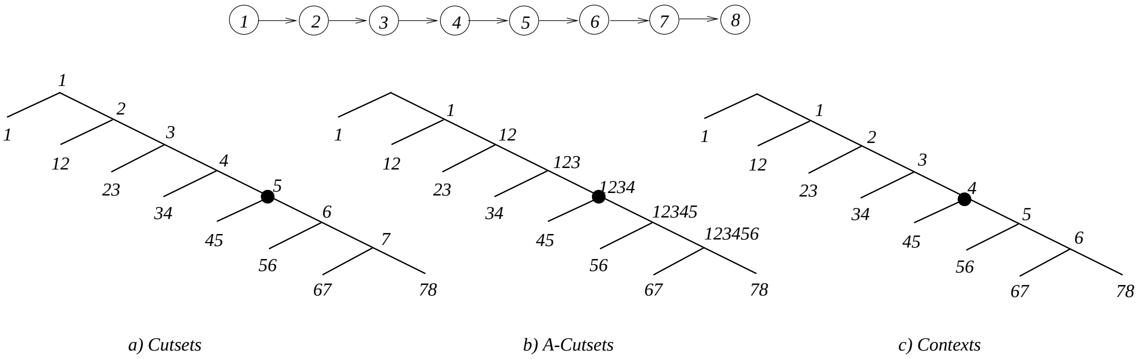


Figure 10: Cutsets, a–cutsets and contexts of a dtree.

random networks. For the first set of networks, SET-A, which contain 100-node networks with elimination-order width ≤ 20, the a-cutset width divided by elimination-order width was 3.5 on average. For the second set of networks, SET-B, which contain 150-node networks with elimination-order width ≤ 50, this average was 2.4. This gives an idea of what the constant factors in $\exp(w \log n)$ are for this class of networks. Second, the $O(n \exp(w \log n))$ time complexity is not comparable to that of cutset conditioning. However:

- When treewidth is bounded, $n \exp(w \log n)$ becomes bounded by a polynomial. Therefore, recursive conditioning takes polynomial time on any network with bounded treewidth. On the other hand, it is well known that many networks with bounded treewidth can have unbounded loop–cutsets. The network in Figure 3 is an example.

- The constant factors in recursive conditioning are expected to be much lower than those of cutset conditioning. Recall that with a loop–cutset of size c, cutset conditioning must solve $O(\exp(c))$ singly–connected networks, each taking $O(n)$ time. Therefore, the constant factor here is that associated with each run of the polytree algorithm. In recursive conditioning, however, the constant factor is that associated with making a recursive call.

4 Remembering Previous Computations

The time complexity of RC1 is clearly not optimal. This is best seen by observing RC1 run on the dtree in Figure 10. Consider the subtree T rooted at the bullet $\bullet$, which corresponds to subnetwork $4 \rightarrow \ldots \rightarrow 8$. RC1 will be called on this subtree sixteen different times, once for each instantiation of $\mathsf{acutset}(T) = 1234$. Note, however, that only variable 4 appears in the subtree T and its corresponding subnetwork $4 \rightarrow \ldots \rightarrow 8$. Hence, the sixteen calls to T correspond to only two different instances of this subnetwork and RC1 is solving each one of these instances eight different times!

In general, each node T in a dtree corresponds to a number of subnetwork instances. Each of these instances share the same structure, which is determined by T. But each instance has a different quantification/evidence, which is determined by the instantiation of $\mathsf{vars}(T) \cap \mathsf{acutset}(T)$; that is, variables in T which are guaranteed to be instantiated when T is called. The set $\mathsf{vars}(T) \cap \mathsf{acutset}(T)$ is so important that we give it a special name:

Definition 4 *The* <u>*context*</u> *of node T in a dtree is defined as follow:*

$$\mathsf{context}(T) \stackrel{def}{=} \mathsf{vars}(T) \cap \mathsf{acutset}(T).$$

Moreover, the <u>*context width*</u> *of a dtree is the size of its maximal context.*

Figure 10(c) depicts the context of each node in the given dtree.

```
                    Algorithm RC2

RC2(T)
  if T is a leaf node,
  then return LOOKUP(T)
  else y ← recorded instantiation of context(T)
       if cache_T[y] ≠ nil, return cache_T[y]
       else p ← 0
            for each instantiation c of uninstantiated variables in cutset(T) do
                record instantiation c
                p ← p + RC2(T^l)RC2(T^r)
                un–record instantiation c
            cache_T[y] ← p
            return p
```

Figure 11: Pseudocode for recursive conditioning. All cache entries must be initialized to nil.

RC1 will solve each subnetwork instance represented by node T a number of times which equals to $(\mathsf{acutset}(T) - \mathsf{vars}(T))^{\#}$, although it can afford to solve such an instance only once. To avoid the redundant computations, however, RC1 needs to remember the solutions of different instances. Since each instance is characterized by an instantiation of $\mathsf{context}(T)$, all RC1 needs to do is save the result of solving each instance, indexed by the characterizing instantiation of $\mathsf{context}(T)$. Any time a subnetwork instance is to be solved, RC1 will check its memory first to see if it has solved this instance before. If it did, it will simply return the remembered answer. If it did not, it will recurse on T, saving its computed solution at the end.[6]

This simple remembering mechanism will actually drop the number of recursive calls made by recursive conditioning from $O(n \exp(w \log n))$ to only $O(n \exp(w))$. But as should be clear, this improvement in running time comes at the expense of memory used to remember previous computations. In fact, as we shall now present, avoiding all redundancies will require that we remember $O(n \exp(w))$ solutions.

Figure 11 presents the second version of recursive conditioning which remembers its previous computations. All we had to do is include a cache with each node T in the dtree. This cache is used to store the answers returned by calls to T. RC2 will not recurse on a node T before it checks the cache at T first.

It should be clear that the size of cache_T in RC2 is bounded by $\mathsf{context}(T)^{\#}$. In Figure 10, the cache stored at each node in the dtree will have at most two entries. Therefore, RC2 will consume only a linear amount of space in addition to what is consumed by RC1. Interestingly enough, this additional, linear space will drop the complexity of recursive conditioning from exponential to linear on this network.

Theorem 4 *The number of recursive calls made to a non-root node T by* RC2 *is $\mathsf{cutset}(T^p)^{\#}\mathsf{context}(T^p)^{\#}$, where T^p is the parent of node T.*

In Figure 10, each cutset has one variable and each context has no more than one variable. Therefore, RC2 will make no more than 4 recursive calls to each node in the dtree.

Algorithm EL2DT, which we present in Section 6, has the following property: When EL2DT constructs a dtree based on an elimination order of width w, $\mathsf{cutset}(T)^{\#}\mathsf{context}(T)^{\#} = O(\exp(w))$ will hold for every node T in the dtree. Hence, the following result.

Theorem 5 *Given an elimination order of width w and length n, and given a dtree based on the order (using* EL2DT*), the number of recursive calls made by* RC2 *is $O(n \exp(w))$ and the space it consumes is $O(n \exp(w))$.*

This is basically the best complexity result we currently have for exact inference in Bayesian networks. It is also the complexity of state–of–the–art algorithms based on clustering and elimination.

[6]This technique is known as *memoization* in the dynamic programming literature and has also been employed in [3].

Algorithm RC

```
RC(T)
 01. if T is a leaf node,
 02.  then return LOOKUP(T)
 03.  else y ← recorded instantiation of context(T)
 04.       if cache_T[y] ≠ nil, return cache_T[y]
 05.       else p ← 0
 06.            for each instantiation c of uninstantiated variables in cutset(T) do
 07.                record instantiation c
 08.                p ← p + RC(T^l)RC(T^r)
 09.                un–record instantiation c
 10.            when cache?(T, y), cache_T[y] ← p
 11.            return p
```

Figure 12: Pseudocode for an any–space version of recursive conditioning.

5 Any–Space Inference

We have presented two extremes of recursive conditioning thus far. On one extreme, no computations are remembered, leading to a space complexity of $O(n)$ and a time complexity of $O(n \exp(w \log n))$. On the other extreme, all previous computations are remembered, dropping the time complexity to $O(n \exp(w))$ and increasing the space complexity to $O(n \exp(w))$.

These behaviors of recursive conditioning are only two extremes of an any–space version, which can use as much space as is made available to it. Specifically, recursive conditioning can remember as many computations as available space would allow and nothing more. By changing one line in RC2, we obtain an any–space version, which is given in Figure 12. In this version, we included an extra test on line 10, which is used to decide whether to remember a certain computation. One of the simplest implementations of this test is based on the availability of global memory. That is, $\mathsf{cache?}(T, \mathbf{y})$ will succeed precisely when global memory has not been exhausted and will fail otherwise.

A more refined scheme will allocate a certain amount of memory to be used by each cache. We can control this amount using the notion of a cache factor.

Definition 5 *A cache factor for a dtree is a function* cf *that maps each internal node* T *in the dtree into a number* $0 \leq \mathsf{cf}(T) \leq 1$.

The intention here is for $\mathsf{cf}(T)$ to be the fraction of cache_T which will be filled by algorithm RC. That is, if $\mathsf{cf}(T) = .2$, then we will only use 20% of the total storage required by cache_T. Note that algorithm RC1 corresponds to the case where $\mathsf{cf}(T) = 0$ for every node T. Moreover, algorithm RC2 corresponds to the case where $\mathsf{cf}(T) = 1$. For each of these cases, we provided a count of the recursive calls made by recursive conditioning. The question now is: What can we say about the number of recursive calls made by RC under a particular cache factor cf?

As it turns out, the number of recursive calls made by RC under the memory committed by cf will dependent on the particular instantiations of $\mathsf{context}(T)$ that will be cached on line 10. However, if we assume that any given instantiation $\mathbf{y}$ of $\mathsf{context}(T)$ is equally likely to be cached, then we can compute the average number of recursive calls made by RC and, hence, its average running time. Note that we can enforce the assumption that any given instantiation $\mathbf{y}$ of $\mathsf{context}(T)$ is equally likely to be cached by randomly choosing the instantiations to be cached.

Theorem 6 *If the size of* cache_T *is limited to* $\mathsf{cf}(T)$ *of its full size, and if each instantiation of* $\mathsf{context}(T)$ *is equally likely to be cached on line 10 of* RC*, the average number of calls made to a non–root node* T *in algorithm* RC *is*

$$\mathsf{ave}(T) = \mathsf{cutset}(T^p)^{\#} \left[\mathsf{cf}(T^p)\mathsf{context}(T^p)^{\#} + (1 - \mathsf{cf}(T^p))\mathsf{ave}(T^p)\right].$$

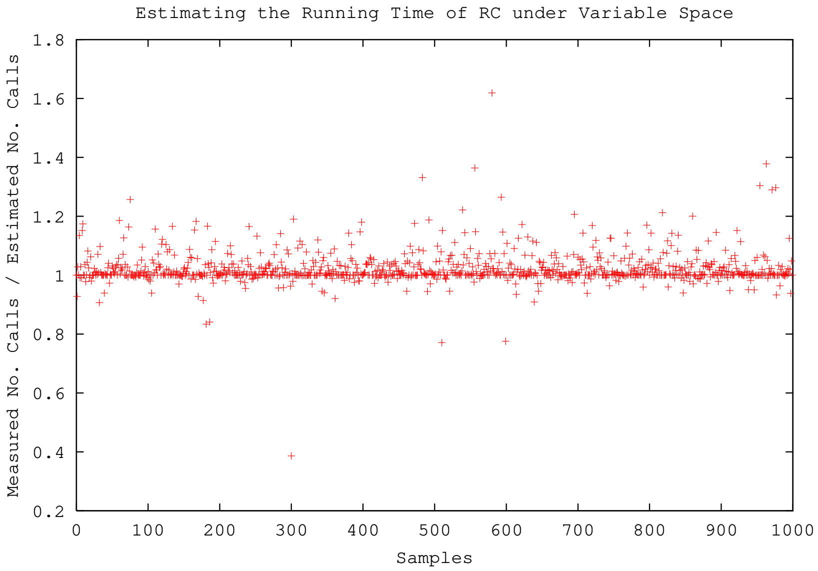


Figure 13: The average ratio of measured over estimated number of calls is 1.02 and the standard deviation is 0.06. The correlation coefficient between measured and estimated calls is 0.99. The networks in SET-A were used in this experiments—see Appendix A.

This theorem is quite important practically as it allows one to estimate the running time of RC under any given memory configuration. All we have to do is add up $\mathsf{ave}(T)$ for every node T in the dtree. Note that once $\mathsf{ave}(T^p)$ is computed, we can compute $\mathsf{ave}(T)$ in constant time. Therefore, we can compute and sum $\mathsf{ave}(T)$ for every node T in the dtree in time linear in the dtree size.

Before we further discuss the practical utility of Theorem 6, we mention two important points. First, when the cache factor is discrete ($\mathsf{cf}(T) = 0$ or $\mathsf{cf}(T) = 1$), Theorem 6 provides an exact count of the number of recursive calls made by RC. In fact, the running time of RC1 and RC2 follow as corollaries of Theorem 6:

- When $\mathsf{cf}(T) = 0$ for all T:
$$\mathsf{ave}(T) = \mathsf{cutset}(T^p)^{\#}\mathsf{ave}(T^p),$$
and the solution to this recurrence is $\mathsf{ave}(T) = \mathsf{acutset}(T)^{\#}$. This is basically the result of Theorem 2.
- When $\mathsf{cf}(T) = 1$ for all T:
$$\mathsf{ave}(T) = \mathsf{cutset}(T^p)^{\#}\mathsf{context}(T^p)^{\#},$$
which is the result of Theorem 4.

When the cache factor is not discrete, Theorem 6 allows us to compute the average number of recursive calls made by RC. Figure 13 depicts the result of an experiment for computing such averages using Theorem 6. We generated 1000 random networks, each of which containing a 100 variables (SET-A in Appendix A), and then generated a random cache factor for each network. We then used Theorem 6 to estimate the number of recursive calls which will be made by RC under that factor. We also ran RC and measured the actual number of recursive calls. Figure 13 reports the ratio of measured to estimated calls for each network. As is clear from the figure, the correlation factor is .99 between estimated and measured. This is very good since we only ran each network once with respect to a given cache factor.

One of the most practical aspects of Theorem 6 is that it allows us to produce a time–space tradeoff curve, which can be used to make decisions on how to allocate resources when using recursive conditioning on computationally demanding networks. We have applied the theorem to some realistic networks from the UC Berkeley Repository (*http://www-nt.cs.berkeley.edu/home/nir/public-html/repository/index.htm*), which are also provided with elimination orders that we utilized in our experiments. We depict three of these networks in Figures 14. Each of the given plots corresponds to one network using both a balanced and an unbalanced dtree. To produce each plot, we simply varied the cache factor and computed the corresponding number of recursive calls. Space is recorded as $\log_2$ of the maximum cache size (cache width) and time is recorded as $\log_2$ of the maximum number of recursive calls that any node receives (recursive calls width).[7]

A number of observations are in order about these figures:

- When we are close to linear space, it is betters to use a balanced dtree for the time–space tradeoff. When we are close to exponential space, it is better to use the original, unbalanced dtree.
- The difference between the balanced and unbalanced dtrees is quite significant for the *Diabetes* network. This is not surprising if one examines the structure of this network, as it looks very similar to the structure of the ladder network in Figure 3.[8]
- In many parts of the time–space curve, cutting the space by half leads to approximately doubling the time.

One of the key questions relating to recursive conditioning is that of identifying the cache factor which would minimize the running time according to Theorem 6. Specifically, let us define the *effect* of a cache factor cf as the number of cache entries that it will utilize:

$$\mathsf{effect}(\mathsf{cf}) \stackrel{def}{=} \sum_{T} \mathsf{cf}(T)\mathsf{context}(T)^{\#}.$$

Given that available memory will only accomodate m cache entries, a key question relating to recursive conditioning is the identification of a cache factor with effect m that would minimize the running time according to Theorem 6. This crucial question, however, is outside the scope of this paper and is the subject of current research.

The issue of time–space tradeoff has been receiving increased interest in the context of Bayesian network inference, due mostly to the observation that state–of–the–art algorithms tend to give on space first. The key existing proposal for such tradeoff is based on realizing that the space complexity of clustering algorithms is exponential only in the size of separators, which are typically smaller than clusters [12]. Therefore, one can always trade time for space by using a jointree with smaller separators, at the expense of introducing larger clusters [12]. This method, however, can generate very large clusters which can render the time complexity very high. To address this problem, a hybrid algorithm is proposed which uses cutset conditioning to solve each enlarged cluster, where the complexity of this hybrid method can sometimes be less than exponential in the size of enlarged clusters [12].

There are two key differences between this proposal and ours. First, the proposal is orthogonal to our notion of a cache factor, as it can be realized during the construction phase of a dtree. That is, we may decide to construct a dtree with smaller caches, yet larger cutsets. But once we have committed to a particular dtree, the cache factor can be used to control the time–space tradeoff at a finer level as suggested above. The second key difference between the proposal of [12] and ours is that when the hybrid algorithm of [12] is run in linear space, it will reduce to cutset conditioning since the whole jointree will be combined into a single cluster. In our proposal, linear space leads to algorithm RC1 which has a different time complexity than cutset conditioning.

6 Relation to Elimination and Clustering

The main purpose of this section is to show how to construct good dtrees, those with small width. A secondary objective is to relate the complexity of recursive conditioning to the complexity of elimination and clustering algorithms. Both objectives will be achieved by studying the relationship between dtrees, which drive recursive conditioning, and

[7]This is how the cache factor was varied in this experiment. Let s be the size of the largest cache in the dtree; that is, $s = \max_T s_T$, where s_T is the size of cache at node T. For a given x ranging from 0 to s, the factor $\mathsf{cf}(T)$ for each node T was chosen as large as possible such that $\mathsf{cf}(T)s_T \leq x$.

[8]A postscript figure depicting the structure of this network is available in the UC Berkeley site.

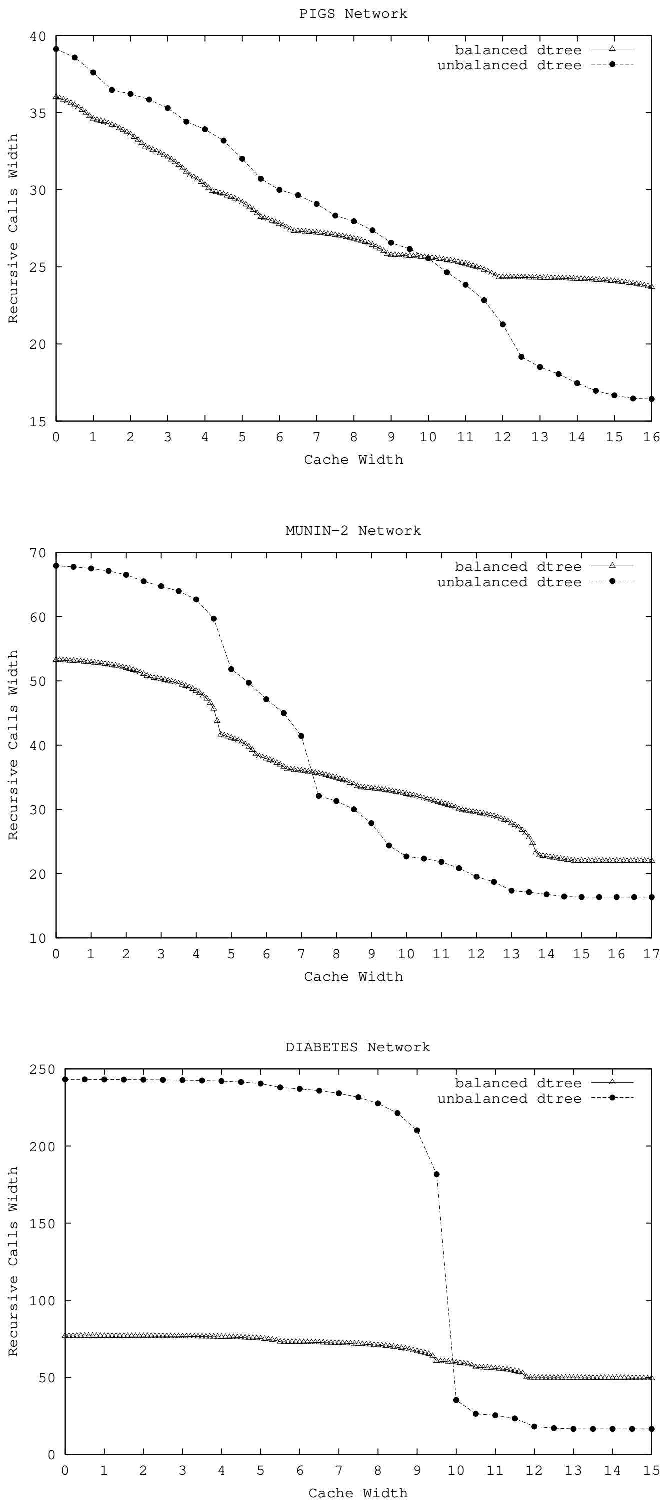


Figure 14: Time–space tradeoff in realistic networks.

- jointrees, which drive clustering methods; and
- elimination orders, which drive elimination methods.

The quality of both elimination orders and jointrees is measured by their width. The core of this section is therefore two linear time transformations that achieve the following:

- Given a dtree of width w for a Bayesian network, construct a jointree of width w for the same network.
- Given an elimination order of width w for a Bayesian network, construct a dtree of width $\leq w$ for the same network.

Given existing transformations between elimination orders of width w and jointrees of the same width [10, 18, 11], the results of this section allow for linear, width–preserving transformations between any pair of graphical structures.[9] There are several implications of these transformations:

- Any good method for constructing elimination orders or jointrees is immediately a good method for constructing dtrees. This means that recursive conditioning can capitalize on the good heuristics already established in the literature, such as the mindegree heuristic.
- Since the treewidth of a Bayesian network is defined as the width of its best elimination order (or jointree), treewidth can also be defined as the width of the network's best dtree.
- If a Bayesian network has a small treewidth, then an optimal elimination order (jointree) can be constructed in linear time [1, 9]. This means that an optimal dtree can also be constructed in such a case.

6.1 From Dtrees to Jointrees

A jointree for Bayesian network $\mathcal{N}$ is a labeled tree (T, C), where T is a tree and C is a labeling function that maps each node i in T into a set of variables $C(i)$ in $\mathcal{N}$, such that:

1. every family (a node and its parents) in $\mathcal{N}$ belongs to some label $C(i)$;
2. if a variable belongs to two labels $C(i)$ and $C(j)$, its must belong to every label $C(k)$, where k is on the path connecting i and j in T.

The label $C(i)$ is typically called a *cluster* or *clique* of the jointree. Moreover, the set $C(i) \cap C(j)$, where (i, j) is an edge in T, is called the *separator* of clusters $C(i)$ and $C(j)$. The width of a jointree is defined as the size of its maximal cluster minus one. The separator width of a jointree is defined as the size of its maximal separator. The time complexity of a clustering method is exponential only in the jointree width, and its space complexity is exponential only in its separator width.

Definition 6 *Let T be a node in a dtree. The cluster of T is defined as follows:*

$$\mathsf{cluster}(T) = \begin{cases} \mathsf{vars}(T), & \textit{if } T \textit{ is leaf;} \\ \mathsf{cutset}(T) \cup \mathsf{context}(T), & \textit{otherwise.} \end{cases}$$

The width of a dtree is defined as the size of its maximal cluster minus one.

As it turns out, the clusters of a dtree already form a jointree.

Theorem 7 *Let $\mathcal{N}$ be a Bayesian network and let T be a corresponding dtree of width w. Then $(T, \mathsf{cluster})$ is a jointree of width w for network $\mathcal{N}$. Moreover, for any node T and its parent T^p, we have $\mathsf{cluster}(T) \cap \mathsf{cluster}(T^p) = \mathsf{context}(T)$.*

[9] There are direct transformations from dtrees to elimination orders, and from jointrees to dtrees, but we omit them here to simplify the discussion [7].

Algorithm EL2DT

```
EL2DT(𝒩, π)
 Σ ← {LEAF(φ) : φ is a CPT in 𝒩}
 for i ← 1 to length of order π do
     let T_1, ..., T_n be trees in Σ which contain variable π(i)
     remove T_1, ..., T_n from Σ
     add COMPOSE(T_1, ..., T_n) to Σ
 COMPOSE and return the trees in Σ.
```

Figure 15: Pseudocode for transforming an elimination order into a dtree. LEAF(ϕ) creates a leaf node and associates CPT ϕ with it.

That is, the clusters of a dtree T form a jointree, where the contexts represent the jointree separators. The jointree induced by a dtree is special in two ways: (1) the CPTs are assigned to leaf clusters only and (2) each cluster has at most three neighbors. Therefore, this theorem shows a very close connection between RC2 and clustering methods. It also shows that if a network has treewidth w, then the width of any of its dtrees will be $\geq w$. In the next section, we will show that if a network has treewidth w, then it must have a dtree of width $\leq w$. The two results lead to a new, alternative definition of treewidth: it is the width of the best dtree for the Bayesian network.

6.2 From Elimination Orders to Dtrees

Strictly speaking, elimination orders are defined for undirected graphs in the graph–theoretic literature. Therefore, when we say an elimination order for a Bayesian network, we mean an elimination order for the moral graph of that network.[10]

An elimination order for an undirected graph G is simply a total order $\pi(1), \pi(2), \ldots, \pi(n)$ of the n variables (nodes) in G. One of the simplest ways for defining the width w of order π is constructively. Simply eliminate variable $\pi(1), \pi(2), \ldots, \pi(n)$ from G in that order, connecting all neighbors of a variable before eliminating it. The maximum number of neighbors that any eliminated variable has is then the width of order π. Moreover, the treewidth of a graph is the width of its best elimination order (the one with the smallest width).

Given a Bayesian network $\mathcal{N}$ and a corresponding elimination order π of width w, we want to construct a dtree for $\mathcal{N}$ of width $\leq w$. This can be easily achieved using the COMPOSE operator, which takes a set of binary trees $T_1, \ldots, T_n$ and connects them (arbitrarily) into a single binary tree COMPOSE($T_1, \ldots, T_n$). We start initially by constructing a set of dtrees, each containing a single node and corresponding to one of the CPTs in network $\mathcal{N}$. We then consider variables $\pi(1), \pi(2), \ldots, \pi(n)$ in that order. Each time we consider a variable $\pi(i)$, we compose all binary trees which mention $\pi(i)$. We finally return the composition of all remaining binary trees. This procedure is given in Figure 15, and two examples of its applications are depicted in Figure 16. In the first example, we use the order $\pi = < D, F, E, C, B, A >$, which has width 3, to generate a dtree of width 2. In the second example, we use the elimination order $\pi = < F, E, A, B, C, D >$ of width 2 and generate a dtree of the same width. Note that algorithm EL2DT is not deterministic since the COMPOSE procedure is not deterministic. Therefore, different dtrees could have been generated using the above orders, but all of them are guaranteed to have width which is no greater than the width of used elimination order.

Algorithm EL2DT can be implemented in time which is linear in the size of given Bayesian network.[11] Its

[10] The moral graph of a Bayesian network is an undirected graph. It is obtained by connecting every pair of parents in the network and then dropping out the directionality of edges.

[11] This can be done using *buckets* [11]. That is, we construct a bucket i for each variable $\pi(i)$. A tree T belongs to bucket i if variable $\pi(i)$ appears in T and comes first in the order among all other variables in T. We start initially by placing each leaf tree LEAF(ϕ) in its corresponding bucket. As we process variable $\pi(i)$, we compose all trees in bucket i and place the resulting tree in its corresponding bucket.

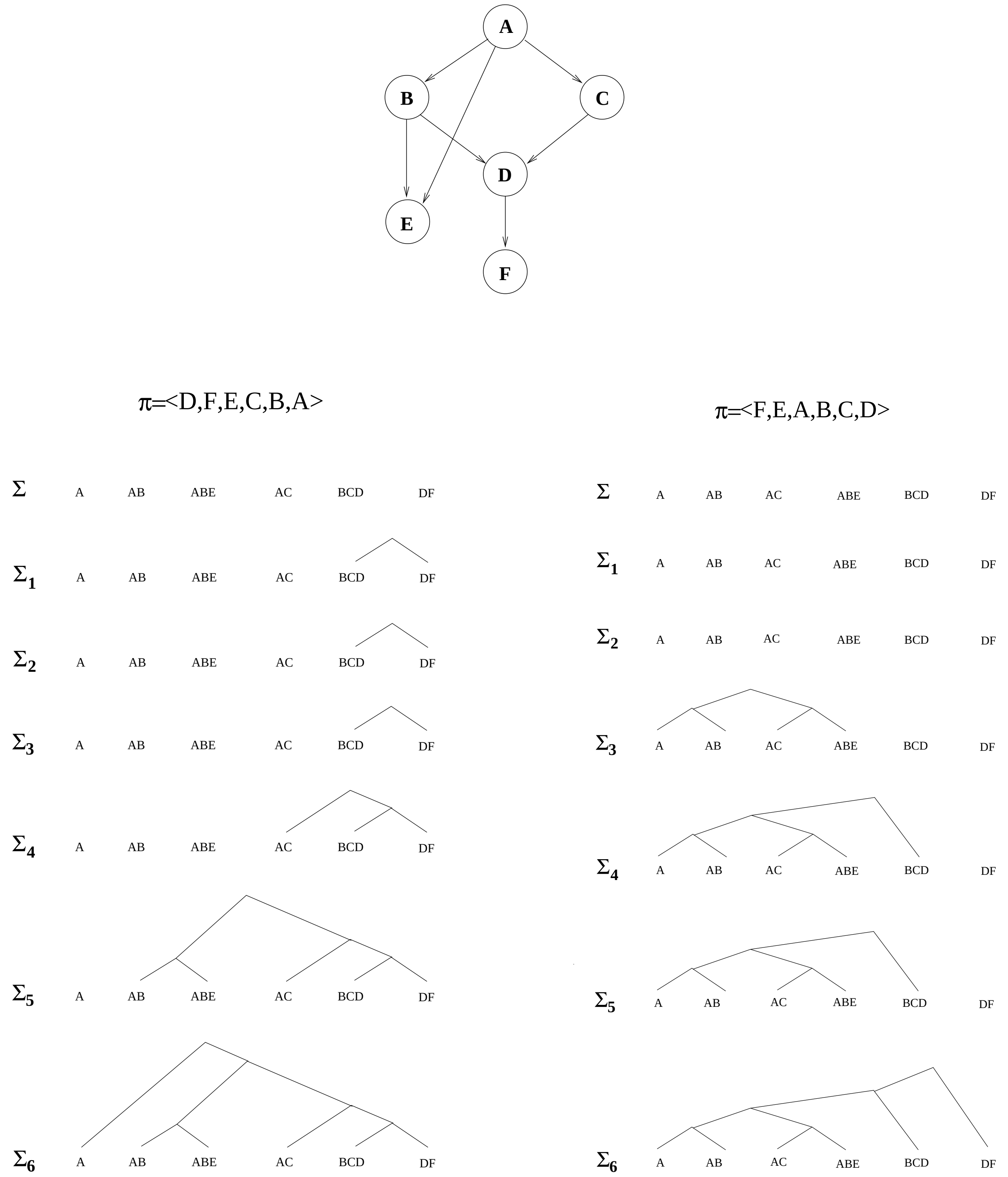


Figure 16: A step–by–step construction of dtrees for the above Bayesian network, using algorithm EL2DT, with respect to two different elimination orders. Each step i depicts the trees present in Σ of algorithm EL2DT after having processed variable $\pi(i)$.

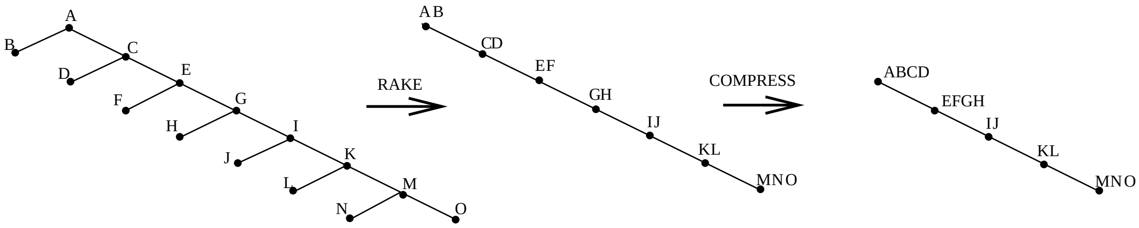


Figure 17: Demonstrating the CONTRACT operation of [24].

Algorithm BAL-DT

```
BAL-DT(T)
  for each internal node N in T, LABEL(N) ← empty dtree
  for each leaf node N in T, LABEL(N) ← dtree N
  OP ← COMPOSE
  R ← final node resulting from successive applications of CONTRACT to T
  return LABEL(R)
```

Figure 18: Pseudocode for balancing a dtree.

soundness is established below:

Theorem 8 *Let $\mathcal{N}$ be a Bayesian network and let π be a corresponding elimination order of width w. The call* EL2DT$(\mathcal{N}, \pi)$ *will return a dtree of width $\leq w$ for network $\mathcal{N}$.*

6.3 Balancing Dtrees

We now present an algorithm for balancing a dtree while increasing its width by no more than a constant factor. The algorithm is similar to EL2DT except that the composition process is not driven by an elimination order. Instead, it is driven by applying the CONTRACT operation of [24] to the given dtree. We need to explain this operation first.

CONTRACT is an operation which is applied to a tree. It simply *absorbs* some of the tree nodes into their neighbors, therefore, producing a smaller tree. To absorb node N_1 into node N_2 is to make the neighbors of N_1 into neighbors of N_2 and to remove node N_1 from the tree. CONTRACT works by applying a RAKE operation to the tree, followed by a COMPRESS operation. The RAKE operation is simple: it absorbs each leaf node into its parent. The COMPRESS operation is more involved: it identifies maximal chains $N_1, N_2, \ldots, N_k$ and then absorbs N_i into N_{i+1} for odd i. The sequence $N_1, N_2, \ldots, N_k$ is a chain if N_{i+1} is the only child of N_i for $1 \leq i < k$, and if N_k has exactly one child and that child is not a leaf. Typically, each tree node N will have an application–specific label, LABEL(N). When node N_1 is absorbed into its neighbor N_2, the label of N_2 is updated as follows: LABEL$(N_2) \leftarrow$ LABEL(N_1) OP LABEL(N_2) where OP is an application–specific operation. One of the key applications of CONTRACT is in evaluating arithmetic–expression trees. In this application, the label of a node is a number and the operation OP is either addition or multiplication.

Figure 17 depicts an example where CONTRACT is applied to a tree, where the labels of nodes are strings and OP is string concatenation. The main property of CONTRACT is that any tree can be reduced to a single node by only applying CONTRACT $O(\log n)$ times, where n the size of given tree [24].

We will use CONTRACT to balance a dtree T as follows. First, we label each internal node in T with the empty dtree. Second, we label each leaf node of T with itself. We then choose the operation OP to be COMPOSE, defined in Section 6.2. Finally, we apply CONTRACT successively to T until it is reduced to a single node and return the label of the final node. This algorithm is given in Figure 18. Its properties follow:

Theorem 9 *Let T be a dtree of context width w for a Bayesian network $\mathcal{N}$ with n nodes.* BAL-DT*(T) will take $O(n \log n)$ time and will return a dtree for $\mathcal{N}$ of height $O(\log n)$, cutset width $\leq w$, context width $\leq 2w$ and width $\leq 3w - 1$.*

The experimental results in Appendix A provide a sense of the constant factors involved in this theorem. For example, the width is increased by 2.1 for SET-A networks and by 1.6 for SET-B networks after balancing using algorithm BAL-DT.

The important aspect of Theorem 9 is that balancing a dtree will increase each of its widths by no more than a constant factor. In fact, the cutset width will never exceed the context width of unbalanced dtree after applying BAL-DT.

6.4 Decomposition by Graph Separators

One of the key differences between recursive conditioning and previous work on nested dissection (including the work of Cooper on recursive decomposition [3]) is the manner in which a problem is decomposed into smaller problems, and the formal guarantees provided on the quality of such a decomposition. Previous works have appealed to the notion of *graph separators* to recursively decompose a graph into smaller subgraphs [14]. A graph separator is a set of nodes $\mathbf{C}$ that partitions the graph into three sets $\mathbf{A}, \mathbf{B}, \mathbf{C}$, such that no node in $\mathbf{A}$ is adjacent to a node in $\mathbf{B}$. In finding separators, one tries to minimize the size of separator $\mathbf{C}$, while keeping the sizes of $\mathbf{A}$ and $\mathbf{B}$ as close as possible. That is, the emphasis is on minimizing separators, while keeping the decomposition balanced. In our framework, this corresponds to generating balanced dtrees that have a minimal cutset width. But this does not necessarily lead to minimizing dtree width, which is the parameter that governs the complexity of recursive conditioning. In fact, balanced decompositions tend to have larger widths than unbalanced ones.

Central to the work on graph separation is the notion of an $f(n)$*–separator theorem.* A class of graphs is said to have an $f(n)$–separator theorem iff there exists constants $\alpha < 1$ and $\beta > 0$, such that if G is a graph in the class with n nodes, then G can be partitioned into sets $\mathbf{A}, \mathbf{B}, \mathbf{C}$ such that no node in $\mathbf{A}$ is adjacent to a node in $\mathbf{B}$, neither $\mathbf{A}$ nor $\mathbf{B}$ contains more than αn nodes, and $\mathbf{C}$ contains no more than $\beta f(n)$ nodes.

An $f(n)$–separator theorem for a class of graph allows one to guarantee the quality of recursive decompositions obtained for that class of graphs. For example, planar graphs have $\sqrt{n}$–separator theorem [22], and at least half a dozen other classes of graphs are known to have similar separator theorems [14].

One of the main differences between decomposing a graph using a separator theorem, versus decomposing it using a dtree, is that the decompositions generated by separator theorems are balanced, while decompositions induced by dtrees can be either balanced or unbalanced. As we have seen earlier, balanced decompositions are preferred if recursive conditioning is to run under linear space (or close to linear space). However, balanced decompositions have a bigger width than unbalanced ones, and are not preferred if one is running under $O(n \exp(w))$ space (or in that region).

The term *decomposition tree* have been used in many places in the literature to denote different notions of recursive decomposition. It is used in [31] to denote a recursive decomposition of a graph into atoms; it is used in [23] to denote a recursive decomposition of a database schema; it is also used in [3] to denote a recursive decomposition of a Bayesian network. We have chosen the term *dtree* in this paper to distinguish our decomposition trees from previous ones.

The recursive–decomposition algorithm of [3] is similar to RC2, except that the decomposition tree employed is quite different from our dtree. With each node in a decomposition tree of [3], four sets of variables are associated: a summation set, an instantiation set, an evaluation set and a variable set. Summation sets represent graph separators and play the role of cutsets in our framework. Instantiation sets are used to cache results and, hence, play the role of contexts in our frameworks. No guarantees, however, are provided on the sizes of these sets in terms of network width. Moreover, evaluation and variable sets are specific to the given construct proposed in [3] and seem to play no role in our framework.

Therefore, aside from a new complexity result for Bayesian network inference under linear space (that is, $O(n \exp(w \log n))$); and a refined, formal theory of any–space reasoning; one of our key contributions here is the introduction of dtrees as a new device for inducing recursive decompositions on directed acyclic graphs.[12] Beyond their simplicity, and admitting balanced/unbalanced decompositions, the significance of

[12] We also show in [7] how to decompose undirected graphs using dtrees.

dtrees stems from the explication of their four parameters (a-cutset width, cutset width, context width, and width) and the bounding of these parameters by treewidth.

Dtrees and their various properties are not specific to probabilistic reasoning, but are applicable to other forms of decompositional reasoning. Preliminary versions of recursive conditioning using dtrees have already been applied to model–based diagnosis [8] and to propositional–logic compilation [6, 4]. In both cases, dtrees were used to recursively decompose a problem into subproblems that can be solved independently. The any–space behavior of recursive conditioning, however, and its time complexity under linear space have yet to be investigated in non–probabilistic reasoning.

7 Conclusion

Recursive conditioning is an any–space algorithm for exact inference in Bayesian networks. On one extreme, recursive conditioning takes $O(n)$ space and $O(n \exp(w \log n))$ time—where n is the size of Bayesian network and w is the width of a given elimination order—therefore, establishing a new complexity result for linear–space inference in Bayesian networks. On the other extreme, recursive conditioning takes $O(n \exp(w))$ space and $O(n \exp(w))$ time, therefore, matching the complexity of state–of–the–art algorithms based on clustering and elimination. In between linear and exponential space, recursive conditioning can utilize memory at increments of X-bytes, where X is the number of bytes needed to store a floating point number in a cache. Moreover, the algorithm is equipped with a formula for computing its average running time under any amount of space, hence, providing a valuable tool for time–space tradeoffs in demanding applications. Recursive conditioning is therefore the first algorithm for exact inference in Bayesian networks to offer a smooth tradeoff between time and space, and to explicate a smooth, quantitative relationship between these two important resources.

Acknowledgement

I wish to thank Rina Dechter for inspiring the analysis of recursive conditioning under linear space and for various helpful discussions; Gregory Cooper for making his unpublished work available; James Park for commenting on an earlier draft of this paper; Judea Pearl for many valuable discussions; and Stuart Russell for suggesting the relevance of graph separators to recursive conditioning.

A Further Experimental Results

We used two sets of networks in our experiments:

- Set-A: Each network in this set contains a 100 nodes and the width of its corresponding dtrees is ≤ 20. The set is depicted in Figure 19, together with some further statistics relating to different width parameters.
- Set-B: Each network in this set contains a 150 nodes and the width of its corresponding dtrees is ≤ 50. The set is depicted in Figure 20, together with some further statistics relating to different width parameters.

For each network, we computed an elimination order based on the following heuristic: always eliminate a variable which leads to adding the smallest number of edges to the moral graph. We then computed a dtree based on this order using algorithm el2dt of Figure 15. The width reported in Figures 19 and 20 refers to the width of computed elimination order. This is at least equal to the treewidth of given network, but can be larger. Note that computing treewidth is an NP–hard problem.

The networks were generated randomly as follows. On average, 20% of the nodes are root, 10% have a single parent, 20% have two parents, 25% have three parents, 20% have four parents and 5% have five parents. We assumed that nodes are numbered from 0 to n. The parents of each node i have been chosen randomly from the set $0, \ldots, i-1$. Moreover any parent of node i was not to be less than $i - w$ for a certain constant w. This constant allows us to control the connectivity of generated network; the bigger w is, the more connected the network is. In the first set of networks, we chose w randomly for each network so it ranges between 2 and 35. In the second set, it ranged between 2 and 75.

B Proofs

Lemma 1 *The following relationships hold:*

(a) $\mathsf{cutset}(T) \cap \mathsf{context}(T) = \emptyset$.

(b) $\mathsf{context}(T) \subseteq \mathsf{cutset}(T^p) \cup \mathsf{context}(T^p) = \mathsf{cluster}(T^p)$.

(c) $\mathsf{cutset}(T^p) \subseteq \mathsf{context}(T)$.

(d) $\mathsf{cutset}(T_1) \cap \mathsf{cutset}(T_2) = \emptyset$ *when* T_1 *is an ancestor of* T_2.

(e) $\mathsf{context}(T) = \mathsf{cluster}(T) \cap \mathsf{cluster}(T^p)$.

Proof of Lemma 1

(a) If $X \in \mathsf{context}(T)$, then $X \in \mathsf{acutset}(T)$ since $\mathsf{context}(T) = \mathsf{acutset}(T) \cap \mathsf{vars}(T)$. Then X cannot belong to $\mathsf{cutset}(T)$, which is equal to $\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r) - \mathsf{acutset}(T)$. The other direction follows similarly.

(b) Suppose $X \in \mathsf{context}(T)$. Then $X \in \mathsf{acutset}(T) \cap \mathsf{vars}(T)$ and, hence, $X \in \mathsf{vars}(T^p)$. We have two cases.

- $X \in \mathsf{acutset}(T^p)$: Then $X \in \mathsf{context}(T^p)$.
- $X \notin \mathsf{acutset}(T^p)$: Then $X \in \mathsf{cutset}(T^p)$ since $X \in \mathsf{acutset}(T)$.

Therefore, $X \in \mathsf{context}(T^p)$ or $X \in \mathsf{cutset}(T^p)$.

(c) Let T^s be the sibling of T and suppose $X \in \mathsf{cutset}(T^p)$. Then $X \in \mathsf{vars}(T) \cap \mathsf{vars}(T^s)$ by definition of a cutset. Therefore, $X \in \mathsf{vars}(T)$, $X \in \mathsf{acutset}(T)$ and, hence, $X \in \mathsf{context}(T)$.

(d) We have $\mathsf{cutset}(T_1) \subseteq \mathsf{acutset}(T_2)$ by definition of $\mathsf{acutset}$. We also have $\mathsf{cutset}(T_2) \cap \mathsf{acutset}(T_2) = \emptyset$ by definition of cutset. Hence, $\mathsf{cutset}(T_1) \cap \mathsf{cutset}(T_2) = \emptyset$.

(e) By definition of context, we have $\mathsf{context}(T) \subseteq \mathsf{cluster}(T)$. By (b), we have $\mathsf{context}(T) \subseteq \mathsf{cluster}(T^p)$. Hence, $\mathsf{context}(T) \subseteq \mathsf{cluster}(T) \cap \mathsf{cluster}(T^p)$. Suppose that $X \in \mathsf{cluster}(T) \cap \mathsf{cluster}(T^p)$. Then $X \in \mathsf{vars}(T)$ since $X \in \mathsf{cluster}(T)$. Since $X \in \mathsf{cluster}(T^p)$, we have two cases. Case 1: $X \in \mathsf{cutset}(T^p)$. Then $X \in \mathsf{context}(T)$ by (c). Case 2: $X \notin \mathsf{cutset}(T^p)$. Then $X \in \mathsf{context}(T^p)$ by (a); hence, $X \in \mathsf{acutset}(T^p)$ and $X \in \mathsf{vars}(T^p)$. Therefore, $X \in \mathsf{acutset}(T)$ and $X \in \mathsf{context}(T)$. □

Lemma 2 *Let* $\mathsf{vars}^{\uparrow}(T)$ *denote* $\cup_{T'} \mathsf{vars}(T')$, *where* T' *is a leaf connected to node* T *through its parent. Then*

$$\begin{aligned}
\mathsf{cutset}(T) &= \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r) - \mathsf{vars}^{\uparrow}(T) \\
\mathsf{context}(T) &= \mathsf{vars}(T) \cap \mathsf{vars}^{\uparrow}(T) \\
\mathsf{cluster}(T) &= (\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)) \cup (\mathsf{vars}(T^l) \cap \mathsf{vars}^{\uparrow}(T)) \cup (\mathsf{vars}(T^r) \cap \mathsf{vars}^{\uparrow}(T)).
\end{aligned}$$

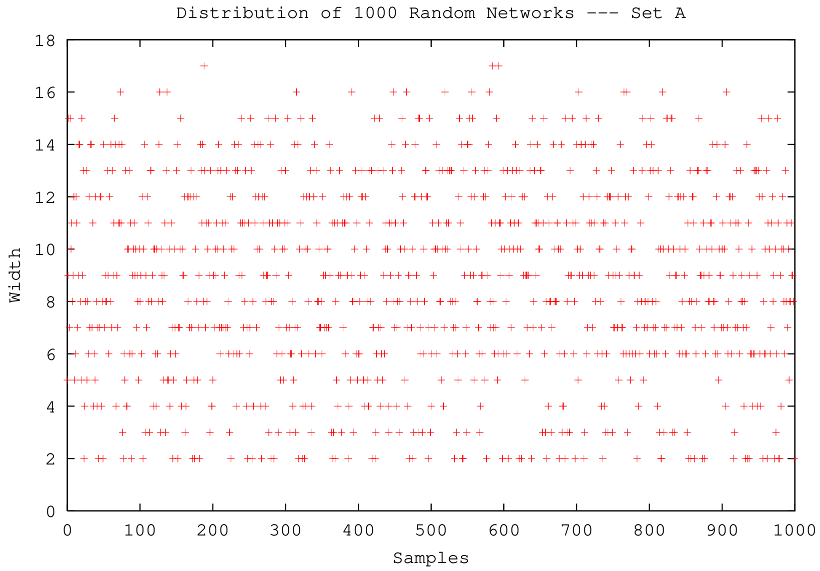
Proof of Lemma 2

If $X \in \mathsf{vars}(T)$, then $X \in \mathsf{vars}^{\uparrow}(T)$ iff $X \in \mathsf{acutset}(T)$. This immediately leads to $\mathsf{cutset}(T) = \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r) - \mathsf{vars}^{\uparrow}(T)$ and $\mathsf{context}(T) = \mathsf{vars}(T) \cap \mathsf{vars}^{\uparrow}(T)$.

Suppose that

$$X \in (\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)) \cup (\mathsf{vars}(T^l) \cap \mathsf{vars}^{\uparrow}(T)) \cup (\mathsf{vars}(T^r) \cap \mathsf{vars}^{\uparrow}(T)).$$

If $X \in (\mathsf{vars}(T^l) \cap \mathsf{vars}^{\uparrow}(T)) \cup (\mathsf{vars}(T^r) \cap \mathsf{vars}^{\uparrow}(T))$, then $X \in \mathsf{vars}(T) \cap \mathsf{vars}^{\uparrow}(T) = \mathsf{context}(T) \subseteq \mathsf{cluster}(T)$. If $X \notin (\mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)) \cup (\mathsf{vars}(T^r) \cap \mathsf{vars}^{\uparrow}(T))$ and $X \in \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)$, then $X \notin \mathsf{vars}^{\uparrow}(T)$ and, hence, $X \in \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r) - \mathsf{vars}^{\uparrow}(T) = \mathsf{cutset}(T) \subseteq \mathsf{cluster}(T)$.



Unbalanced Dtrees

Parameter	Ave	Std	Min	Max
Width	8.8	3.7	2.0	17.0
Cutset Width	5.2	2.2	2.0	13.0
Context Width	9.0	3.2	2.0	16.0
A-Cutset Width	51.8	15.4	7.0	93.0

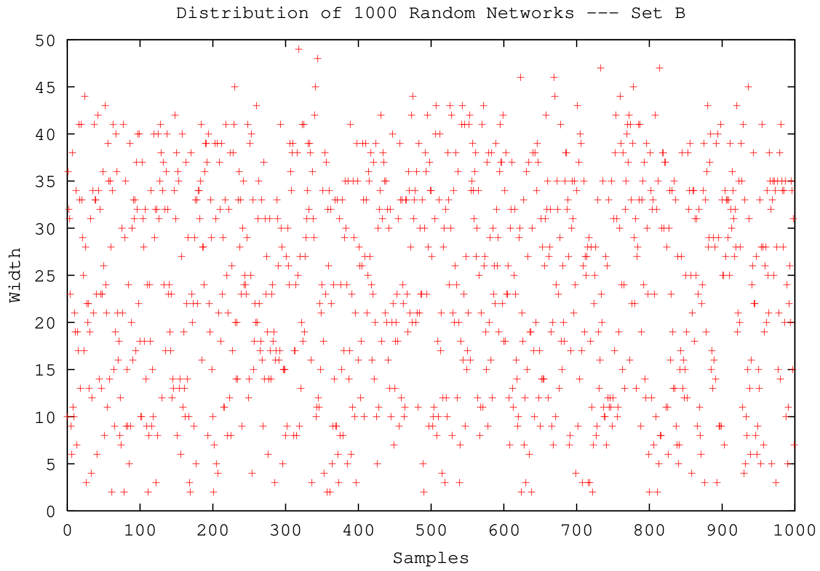
Balanced Dtrees

Parameter	Ave	Std	Min	Max
Width	17.5	6.6	3.0	34.0
Cutset Width	7.5	2.6	2.0	15.0
Context Width	14.5	5.4	3.0	27.0
A-Cutset Width	28.4	8.8	5.0	50.0

Balanced / Unbalanced Ratio

Ratio	Ave	Std	Min	Max
Width / Width	2.1	0.3	1.1	3.0
Cutset Width / Cutset Width	1.5	0.4	0.9	3.5
Context Width / Context Width	1.6	0.2	1.0	2.0
A-Cutset Width / A-Cutset Width	0.6	0.2	0.2	1.0
Cutset Width / Width	0.6	0.2	0.5	1.5
A-Cutset Width / Width	3.5	0.7	2.0	6.0

Figure 19: SET-A networks.



Unbalanced Dtrees

Parameter	Ave	Std	Min	Max
Width	24.2	11.5	2.0	49.0
Cutset Width	12.4	6.4	2.0	32.0
Context Width	23.2	10.6	3.0	46.0
A-Cutset Width	62.4	15.4	10.0	138.0

Balanced Dtrees

Parameter	Ave	Std	Min	Max
Width	33.5	10.5	3.0	55.0
Cutset Width	17.1	7.8	2.0	43.0
Context Width	29.9	10.4	4.0	55.0
A-Cutset Width	47.9	11.3	7.0	73.0

Balanced / Unbalanced Ratio

Ratio	Ave	Std	Min	Max
Width / Width	1.6	0.5	1.0	3.0
Cutset Width / Cutset Width	1.5	0.6	0.9	4.8
Context Width / Context Width	1.4	0.3	0.9	2.0
A-Cutset Width / A-Cutset Width	0.8	0.2	0.2	1.0
Cutset Width / Width	0.7	.33	0.4	1.5
A-Cutset Width / Width	2.4	1.0	1.2	5.8

Figure 20: Set-B networks.

Suppose that

$$X \in \mathsf{cluster}(T) = \mathsf{cutset}(T) \cup \mathsf{context}(T).$$

If $X \in \mathsf{context}(T) = \mathsf{vars}(T) \cap \mathsf{vars}^{\uparrow}(T)$, then $X \in \mathsf{vars}(T^l) \cap \mathsf{vars}^{\uparrow}(T)$ or $X \in \mathsf{vars}(T^r) \cap \mathsf{vars}^{\uparrow}(T)$. If $X \in \mathsf{cutset}(T) = \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r) - \mathsf{vars}^{\uparrow}(T)$, and $X \notin \mathsf{context}(T)$, then $X \notin \mathsf{vars}^{\uparrow}(T)$ and, hence, $X \in \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)$. $\square$

Proof of Theorem 1

First, we need to show that if variables in $\mathbf{C} = \mathsf{vars}(T^l) \cap \mathsf{vars}(T^r)$ are instantiated, then the CPTs of T^l and T^r will not share any variables. Suppose that $X \in \mathbf{C}$. The CPT of X may belong to either T^l or T^r but not both. Suppose it belongs to T^l. Once we instantiate x, variable X will disappear from all CPTs in T^r. Therefore, X will not appear in $\mathsf{vars}(T^r)$ and, hence, the CPTs of T^l and T^r will not share any variables.

Second, we need to pretend that each time instantiation $\mathbf{c}$ is recorded on line 05, then CPTs are actually reduced. And that this process is reversed on line 07. In this case, each time we reach a leaf node T, the table ϕ associated with T is guaranteed to be reduced to table ϕ' which contains only one variable X. This follows because (1) every variable $Y \neq X$ which appears in CPT ϕ must also appear in $\mathsf{acutset}(T)$ and (2) when RC1 is called on T, $\mathsf{acutset}(T)$ is guaranteed to be instantiated. The base case follows since LOOKUP(T) will return $\mathsf{Pr}^{\mathcal{N}}(\mathbf{e})$, where $\mathcal{N}$ is the network consisting of the single node X and its reduced CPT ϕ', and $\mathbf{e}$ is the evidence available on X. The inductive step follows from Equation 4. $\square$

Proof of Theorem 2

The first part of this theorem follows as a corollary of Theorem 6—see discussion after the statement of Theorem 6.

To show $\mathsf{acutset}(T)^{\#} = O(\exp(dw_c))$, we note the following:

- The cutsets associated with the ancestors of T are pairwise disjoint by Lemma 1(d).
- The size of any of these cutsets is no greater than w_c.
- $\mathsf{acutset}(T)$ is the union of $\mathsf{cutset}(T')$, where T' is an ancestor of T.

Hence, the size of $\mathsf{acutset}(T)$ is bounded by dw_c. $\square$

Proof of Theorem 3

That RC1 consumes $O(n)$ space follows immediately from the statement of the algorithm.[13]

Given an elimination order of width w, EL2DT will construct a dtree of cutset width $\leq w$ (Theorem 8). BAL-DT will balance the dtree, while ensuring that its cutset width is $\leq w$ (Theorem 9). Since the height of the balanced dtree is $O(\log n)$, its a-cutset width must be $O(w \log n)$ by Theorem 2. Therefore, the number of recursive calls made by RC1 to node T is $O(\exp(w \log n))$. The total number of recursive calls made by RC1 is then $O(n \exp(w \log n))$. $\square$

Proof of Theorem 4

Follows as a corollary of Theorem 6—see discussion after the statement of Theorem 6. $\square$

Proof of Theorem 5

We have $O(n)$ caches and the size of each cache_T is $\leq \mathsf{context}(T)^{\#}$. Since the dtree is constructed using EL2DT, $\mathsf{context}(T)^{\#} = O(\exp(w))$ (Theorem 8). Hence, the size of all caches is $O(n \exp(w))$.

By Theorem 4, the number of recursive calls to each node T is $\mathsf{cutset}(T^p)^{\#}\mathsf{context}(T^p)^{\#}$. Since the dtree is constructed using EL2DT, $\mathsf{cutset}(T^p)^{\#}\mathsf{context}(T^p)^{\#} = O(\exp(w))$ (Theorem 8). Hence, the total number of recursive calls is $O(n \exp(w))$. $\square$

[13] We are assuming that cutset and context sizes are bounded by constants.

Proof of Theorem 6

The central concept in this proof is the notion of a T–type for a given node T in the dtree. This is basically the set of all calls to node T that agree on the instantiation of $\mathsf{context}(T)$ at the time the calls are made. Calls in a particular T–type are guaranteed to return the same probability. In fact, the whole purpose of cache_T is to save the result returned by one member of each T–type so the result can be looked up when other calls in the same T-type are made. Each T–type is identified by a particular instantiation $\mathbf{y}$ of $\mathsf{context}(T)$. Hence, there are $\mathsf{context}(T)^{\#}$ different T–types, each corresponding to one instantiation of $\mathsf{context}(T)$. We further establish the following definitions and observations:

- A T–type $\mathbf{y}$ is either *cached* or *non–cached* depending on whether the test $\mathsf{cache?}(T,\mathbf{y})$ succeeds on line 10.
- $\mathsf{acpt}(T)$ is the average number of calls in a T–type.
- $\mathsf{ave}(T)$ is the average number of calls to node T and equals $\mathsf{ave}(T) = \mathsf{acpt}(T)\mathsf{context}(T)^{\#}$.
- We have $\mathsf{cf}(T)\mathsf{context}(T)^{\#}$ cached T–types and $(1-\mathsf{cf}(T))\mathsf{context}(T)^{\#}$ non–cached T–types.[14]
- A T^p–type $\mathbf{x}$ is *consistent* with T–type $\mathbf{y}$ iff instantiations $\mathbf{x}$ and $\mathbf{y}$ agree on the values of their common variables $\mathsf{context}(T^p) \cap \mathsf{context}(T)$. Calls in a particular T–type $\mathbf{y}$ will be generated recursively only by calls in a consistent T^p–type $\mathbf{x}$.
- There are $(\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}$ T^p–types which are consistent with a given T–type $\mathbf{y}$. On average,
  - $\mathsf{cf}(T^p)(\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}$ of them are cached, and
  - $(1-\mathsf{cf}(T^p))(\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}$ are non–cached.

  This follows because each T^p–type is equally likely to be cached. Moreover,

  - A cached T^p–type $\mathbf{x}$ will generate $\mathsf{cutset}(T^p)^{\#}$ calls to node T since $\textsc{rc}(T^p)$ will recurse on *only one* call per cached T^p–type. Only one of these calls is consistent with T–type $\mathbf{y}$ since $\mathsf{cutset}(T^p) \subseteq \mathsf{context}(T)$ by Lemma 1(c).
  - A non–cached T^p–type $\mathbf{x}$ will generate $\mathsf{acpt}(T^p)\mathsf{cutset}(T^p)^{\#}$ calls to node T since $\textsc{rc}(T^p)$ will recurse on *every* call in a non–cached T^p–type. Only $\mathsf{acpt}(T^p)$ of these calls are consistent with T–type $\mathbf{y}$.
- $\mathsf{acpt}(T)$ equals the sum of calls in some T–type $\mathbf{y}$ which are generated by each T^p–type consistent with $\mathbf{y}$. Therefore,

$$\begin{aligned}\mathsf{acpt}(T) &= \underbrace{\mathsf{cf}(T^p)(\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}}_{\text{(no. cached } T^p\text{–types consistent with } \mathbf{y})} \underbrace{1}_{\text{(no. calls in } T\text{–type } \mathbf{y} \text{ each generates)}} + \\ &\quad \underbrace{(1-\mathsf{cf}(T^p))(\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}}_{\text{(no. non–cached } T^p\text{–types consistent with } \mathbf{y})} \underbrace{\mathsf{acpt}(T^p)}_{\text{(no. calls in } T\text{–type } \mathbf{y} \text{ each generates)}} \\ &= (\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}\left[\mathsf{cf}(T^p) + (1-\mathsf{cf}(T^p))\mathsf{acpt}(T^p)\right].\end{aligned}$$

Hence,

$$\begin{aligned}\mathsf{ave}(T) &= (\mathsf{context}(T^p) - \mathsf{context}(T))^{\#}\left[\mathsf{cf}(T^p) + (1-\mathsf{cf}(T^p))\mathsf{acpt}(T^p)\right]\mathsf{context}(T)^{\#} \\ &= (\mathsf{cluster}(T^p) - \mathsf{context}(T))^{\#}\left[\mathsf{cf}(T^p) + (1-\mathsf{cf}(T^p))\mathsf{acpt}(T^p)\right]\mathsf{context}(T)^{\#}, \text{ by Lemma 1(b,c)} \\ &= \mathsf{cluster}(T^p)^{\#}\left[\mathsf{cf}(T^p) + (1-\mathsf{cf}(T^p))\mathsf{acpt}(T^p)\right], \text{ by Lemma 1(b)} \\ &= \mathsf{cutset}(T^p)^{\#}\mathsf{context}(T^p)^{\#}\left[\mathsf{cf}(T^p) + (1-\mathsf{cf}(T^p))\mathsf{acpt}(T^p)\right], \text{ by Lemma 1(a,b)} \\ &= \mathsf{cutset}(T^p)^{\#}\left[\mathsf{cf}(T^p)\mathsf{context}(T^p)^{\#} + (1-\mathsf{cf}(T^p))\mathsf{acpt}(T^p)\mathsf{context}(T^p)^{\#}\right] \\ &= \mathsf{cutset}(T^p)^{\#}\left[\mathsf{cf}(T^p)\mathsf{context}(T^p)^{\#} + (1-\mathsf{cf}(T^p))\mathsf{ave}(T^p)\right]. \ \square\end{aligned}$$

[14] In algorithm $\textsc{rc}1$, all T–types are non–cached ($\mathsf{cf}(T) = 0$). In $\textsc{rc}2$, all T–types are cached ($\mathsf{cf}(T) = 1$).

Proof of Theorem 7

That $\mathsf{cluster}(T) \cap \mathsf{cluster}(T^p) = \mathsf{context}(T)$ follows from Lemma 1(e).

It also follows from the definition of a dtree that the clusters of leaf nodes correspond to the families of Bayesian network. Therefore, each family is contained in some dtree cluster.

To prove the jointree property, we will use Lemma 2. Suppose that L, M and N are three nodes in dtree T. Suppose further that L is on the path connecting M and N. Let X be a node in $\mathsf{cluster}(M) \cap \mathsf{cluster}(N)$. We want to show that X belongs to $\mathsf{cluster}(L)$. We consider two cases.

Case: M is an ancestor of N. Then L is an ancestor of N. Since $X \in \mathsf{cluster}(N)$, then $X \in \mathsf{vars}(N)$ and, hence, $X \in \mathsf{vars}(L)$. Since $X \in \mathsf{cluster}(M)$, then either $X \in \mathsf{cutset}(M)$ or $X \in \mathsf{context}(M)$. If $X \in \mathsf{cutset}(M)$, then $X \in \mathsf{vars}(M^l)$ and $X \in \mathsf{vars}(M^r)$. If $X \in \mathsf{context}(M)$, then $X \in \mathsf{vars}^{\uparrow}(M)$. In either case, we have $X \in \mathsf{vars}^{\uparrow}(L)$, $X \in \mathsf{vars}(L) \cap \mathsf{vars}^{\uparrow}(L) = \mathsf{context}(L)$ and, hence, $X \in \mathsf{cluster}(L)$.

Case: M is not an ancestor of N. Then we must have a common ancestor O of both M and N. Moreover, either $O = L$ or O is an ancestor of L. Therefore, it suffice to show that $X \in \mathsf{cluster}(O)$ (given the above case). Without loss of generality, suppose that M is in the left subtree of O and N is in the right subtree. Since $X \in \mathsf{vars}(M)$, then $X \in \mathsf{vars}(O^l)$. Since $X \in \mathsf{vars}(N)$, then $X \in \mathsf{vars}(O^r)$. Therefore, $X \in \mathsf{cluster}(O)$ by Lemma 2. □

Proof of Theorem 8

We need a couple of lemmas first.

Lemma 3 *When processing variable $\pi(i)$ in* EL2DT*, the cluster of any node N which is added in the process of composing trees $T_1, \ldots, T_n$ must be included in $\mathsf{vars}(T) \cap \{\pi(i), \ldots, \pi(n)\}$, where $T = \text{COMPOSE}(T_1, \ldots, T_n)$.*[15]

Suppose that a variable X belongs to $\mathsf{cluster}(N)$. Then, by Lemma 2, X must either belong to two trees in $T_1, \ldots, T_n$, or belong to a tree in $T_1, \ldots, T_n$ and another tree in $\Sigma - \{T_1, \ldots, T_n\}$. In either case, X cannot belong to $\{\pi(1), \ldots, \pi(i-1)\}$ since these variables have already been processed, so each can belong only to a single tree in Σ. Therefore, X must belong to $\pi(i), \ldots, \pi(n)$. Moreover, X must belong to at least one tree in $T_1, \ldots, T_n$. Hence, X must belong to T and $X \in \mathsf{vars}(T) \cap \{\pi(i), \ldots, \pi(n)\}$. □

Lemma 4 *Let Γ be a collection of sets $S_1, \ldots, S_n$, where S_i is the family of variable $\pi(i)$ in network $\mathcal{N}$. To eliminate variable $\pi(i)$ from Γ is to replace the sets S_k containing $\pi(i)$ by the set $(\bigcup_k S_k) - \{\pi(i)\}$. Now, if we start eliminating variables according to the order π, concurrently, from the moral graph G of $\mathcal{N}$ and from the collection Γ, we find the following. As we are about to eliminate variable $\pi(i)$, the set $(\bigcup_k S_k) - \{\pi(i)\}$ will contain exactly the neighbors of $\pi(i)$ in graph G.*

It suffices to show that two nodes appear in the same set in Γ iff they are connected by an edge in G. This follows initially, before any variable is eliminated. Moreover, it is easy to show that it continues to hold after a variable has been eliminated. □

Now algorithm EL2DT$(\mathcal{N}, \pi)$ can be viewed as performing variable elimination on a collection of sets, which initially contains the families of $\mathcal{N}$. We need to establish this correspondence first in order to prove our theorem. After processing variable $\pi(i)$ in algorithm EL2DT, the set of variables represented by tree T in Σ is

$$\mathsf{set}(T) \stackrel{def}{=} \mathsf{vars}(T) \cap \{\pi(i+1), \ldots, \pi(n)\};$$

that is, variables in T that have not been processed yet.

Initially, the trees in Σ represent the families in $\mathcal{N}$. As we process variable $\pi(i)$, we collect all trees $T_1, \ldots, T_n$ such that $\pi(i) \in \mathsf{set}(T_1), \ldots, \mathsf{set}(T_n)$ and replace them by the tree $\text{COMPOSE}(T_1, \ldots, T_n)$. It follows that

$$\mathsf{set}(\text{COMPOSE}(T_1, \ldots, T_n)) = \mathsf{set}(T_1) \cup \ldots \cup \mathsf{set}(T_n) - \{\pi(i)\},$$

and hence the correspondence we are seeking.

[15] We are referring to the cluster of N in the final dtree returned by EL2DT.

From this correspondence, and Lemma 4, we conclude that when processing variable $\pi(i)$, the tree $T = \text{COMPOSE}(T_1, \ldots, T_n)$, which is added to Σ, is such that $\mathsf{set}(T)$ contains exactly the neighbors of variable $\pi(i)$ in the moral graph G of $\mathcal{N}$ after having eliminated $\pi(1), \ldots, \pi(i-1)$ from it. This means that the size of $\mathsf{set}(T) = \mathsf{vars}(T) \cap \{\pi(i+1), \ldots, \pi(n)\}$ is $\leq \mathit{width}(\pi)$ and, hence, the size of $\mathsf{vars}(T) \cap \{\pi(i), \ldots, \pi(n)\}$ is $\leq \mathit{width}(\pi) + 1$.

Given Lemma 3, this means that the cluster of any node which is added as a result of composing $T_1, \ldots, T_n$ cannot be bigger than $\mathit{width}(\pi) + 1$. This proves that the width of constructed dtree is no more than the width of order π. $\square$

Proof of Theorem 9

That $\text{BAL-DT}(T)$ takes $O(n \log n)$ time and returns a binary tree of height $O(\log n)$ follows immediately from the properties of the CONTRACT operation [24]. That $\text{BAL-DT}(T)$ is a dtree follows from the way we initialized the labels of nodes in T.

To prove the results on widths, we need to introduce some new notation. Since the call $\text{BAL-DT}(T)$ modifies dtree T using the CONTRACT operation, we will use $T_0, T_1, T_2, \ldots$, where $T_0 = T$, to denote the modified dtrees after each RAKE or COMPRESS operation. Moreover, we will use N_i to denote node N in dtree T_i.

We will use $\mathit{Lvars}(N)$ to denote the variables appearing in dtree $\text{LABEL}(N)$; $\mathit{Lvars}^{\downarrow}(N)$ to denote variables appearing in dtrees $\text{LABEL}(M)$, where $M = N$ or M is a descendent of N; $\mathit{Lvars}^{\uparrow}(N)$ to denote variables appearing in dtrees $\text{LABEL}(M)$, where M is connected to N through its parent.

We first prove two lemmas.

Lemma 5 *We have* $\mid \mathit{Lvars}^{\downarrow}(N_i) \cap \mathit{Lvars}^{\uparrow}(N_i) \mid \leq w$.

This holds in T_0 since $\mathit{Lvars}^{\downarrow}(N_0) \cap \mathit{Lvars}^{\uparrow}(N_0) = \mathsf{context}(N_0)$ by Lemma 2, which size is $\leq w$. We need to prove that the RAKE and COMPRESS operations preserve this invariant.

- COMPRESS: after absorbing ${N_i}^p$ into N_i to yield N_{i+1}, we have $\mathit{Lvars}^{\downarrow}(N_{i+1}) = \mathit{Lvars}^{\downarrow}({N_i}^p)$ and $\mathit{Lvars}^{\uparrow}(N_{i+1}) = \mathit{Lvars}^{\uparrow}({N_i}^p)$. Therefore, $\mathit{Lvars}^{\downarrow}(N_{i+1}) \cap \mathit{Lvars}^{\uparrow}(N_{i+1}) = \mathit{Lvars}^{\downarrow}({N_i}^p) \cap \mathit{Lvars}^{\uparrow}({N_i}^p)$ and the invariant holds in T_{i+1} given that it holds in T_i.

- RAKE: after absorbing the children ${N_i}^l$ and ${N_i}^r$ into N_i to yield N_{i+1}, $\text{LABEL}(N_{i+1})$ will be the composition of $\text{LABEL}(N_i)$, $\text{LABEL}({N_i}^l)$ and $\text{LABEL}({N_i}^r)$. Therefore, $\mathit{Lvars}^{\downarrow}(N_{i+1}) = \mathit{Lvars}^{\downarrow}(N_i)$ and $\mathit{Lvars}^{\uparrow}(N_{i+1}) = \mathit{Lvars}^{\uparrow}(N_i)$ and the invariant holds in T_{i+1} given that it holds in T_i. $\square$

Lemma 6 *If N_i is a node with two children, then $\text{LABEL}(N_i)$ is the empty dtree.*

If N_i has two children, then $N_0, N_1, \ldots, N_{i-1}$ have two children each since CONTRACT cannot add children to nodes. By construction, $\text{LABEL}(N_0)$ must be the empty dtree. Suppose that $\text{LABEL}(N_i)$ is not the empty dtree. Then a node must have been absorbed into N in some dtree $T_0, \ldots, T_i$. This is impossible though since N cannot be part of any chain in these dtrees, and N is not a leaf in any of these dtrees. Therefore, neither COMPRESS nor RAKE could have altered the label of N in dtrees $T_0, \ldots, T_i$. $\square$

We now proceed to prove the rest of this theorem. Initially, the dtrees in the labels of T_0 represent leaf nodes in the final dtree returned by BAL-DT. Since these nodes are leaves, they do not have cutsets. That the context and cluster sizes of these nodes have the claimed sizes in the final dtree returned by BAL-DT follows immediately from the fact that they correspond to the leaves in dtree T_0.

There are three ways in which COMPOSE can add a new dtree node d to combine two dtrees together. We will show that the cutset, context and cluster of each added node d will have the claimed size in the final dtree returned by BAL-DT. In what follows, $\mathsf{cutset}(d)$, $\mathsf{context}(d)$ and $\mathsf{cluster}(d)$ refer to the cutset, context and cluster of node d in the final dtree returned by BAL-DT.

Case 1. We have a chain $N_i - O_i - P_i$, where N_i is absorbed into child O_i by COMPRESS, creating dtree $d = \text{LABEL}(O_{i+1}) = \text{COMPOSE}(\text{LABEL}(N_i), \text{LABEL}(O_i))$. Then

$$\begin{aligned} \mathsf{cutset}(d) &\subseteq \mathit{Lvars}(N_i) \cap \mathit{Lvars}(O_i) \\ &\subseteq \mathit{Lvars}^{\uparrow}(O_i) \cap \mathit{Lvars}^{\downarrow}(O_i), \end{aligned}$$

which size is $\leq w$ by Lemma 5. Moreover, by Lemma 2,

$$\begin{aligned}\mathsf{context}(d) &= (Lvars(N_i) \cup Lvars(O_i)) \cap \bigcup_{K_i \neq N_i, K_i \neq O_i} Lvars(K_i) \\ &\subseteq (Lvars^{\uparrow}(N_i) \cap Lvars^{\downarrow}(N_i)) \cup (Lvars^{\uparrow}(P_i) \cap Lvars^{\downarrow}(P_i)),\end{aligned}$$

which size is $\leq 2w$ by Lemma 5. Finally, since $\mathsf{cluster}(d) = \mathsf{cutset}(d) \cup \mathsf{context}(d)$, we have $|\ \mathsf{cluster}(d)\ | \leq 3w$.

Case 2. Node N_i has a single child O_i, where O_i is a leaf. Node O_i is absorbed into parent N_i by RAKE, creating dtree $d = \text{LABEL}(N_{i+1}) = \text{COMPOSE}(\text{LABEL}(N_i), \text{LABEL}(O_i))$. We have

$$\begin{aligned}\mathsf{cutset}(d) &\subseteq Lvars(N_i) \cap Lvars(O_i) \\ &\subseteq Lvars^{\uparrow}(O_i) \cap Lvars^{\downarrow}(O_i),\end{aligned}$$

which size is $\leq w$. Moreover,

$$\begin{aligned}\mathsf{context}(d) &= (Lvars(N_i) \cup Lvars(O_i)) \cap \bigcup_{K_i \neq N_i, K_i \neq O_i} Lvars(K_i) \\ &\subseteq Lvars^{\uparrow}(N_i) \cap Lvars^{\downarrow}(N_i),\end{aligned}$$

which size is $\leq w$. Finally, since $\mathsf{cluster}(d) = \mathsf{cutset}(d) \cup \mathsf{context}(d)$, we have $|\ \mathsf{cluster}(d)\ | \leq 2w$.

Case 3. Node N_i has two children O_i and P_i, which are leaves. Nodes O_i and P_i are absorbed into parent N_i by RAKE, creating dtree $d = \text{LABEL}(N_{i+1}) = \text{COMPOSE}(\text{LABEL}(O_i), \text{LABEL}(P_i))$ since $\text{LABEL}(N_i)$ is the empty dtree by Lemma 6. We have

$$\begin{aligned}\mathsf{cutset}(d) &\subseteq Lvars(O_i) \cap Lvars(P_i) \\ &\subseteq Lvars^{\uparrow}(O_i) \cap Lvars^{\downarrow}(O_i),\end{aligned}$$

which size is $\leq w$. Moreover,

$$\begin{aligned}\mathsf{context}(d) &= (Lvars(O_i) \cup Lvars(P_i)) \cap \bigcup_{K_i \neq O_i, K_i \neq P_i} Lvars(K_i) \\ &\subseteq Lvars^{\uparrow}(N_i) \cap Lvars^{\downarrow}(N_i),\end{aligned}$$

which size is $\leq w$. Finally, since $\mathsf{cluster}(d) = \mathsf{cutset}(d) \cup \mathsf{context}(d)$, we have $|\ \mathsf{cluster}(d)\ | \leq 2w$.

Therefore, the size of every cutset is $\leq w$, the size of every context is $\leq 2w$ and the size of every cluster is $\leq 3w$. This means that cutset width, context width, and width are $\leq w, 2w, 3w-1$, respectively. $\square$

References

[1] Hans. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal of Computing*, 25(6):1305–1317, 1996.

[2] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context–specific independence in bayesian networks. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 115–123, 1996.

[3] Gregory F. Cooper. Bayesian belief-network inference using recursive decomposition. Technical Report KSL-90-05, Knowledge Systems Laboratory, Stanford, CA 94305, 1990.

[4] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*. To appear.

[5] Adnan Darwiche. Conditioning algorithms for exact and approximate inference in causal networks. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 99–107, 1995.

[6] Adnan Darwiche. Compiling knowledge into decomposable negation normal form. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 284–289, 1999.

[7] Adnan Darwiche. Dtrees: A new graphical model for structure–based reasoning. Technical Report D–107, Computer Science Department, UCLA, Los Angeles, Ca 90095, 1999.

[8] Adnan Darwiche. Utilizing device behavior in structure–based diagnosis. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1096–1101, 1999.

[9] R. Dechter. Constraint networks. *Encyclopedia of Artificial Intelligence*, pages 276–285, 1992.

[10] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, pages 353–366, 1989.

[11] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 211–219, 1996.

[12] Rina Dechter. Topological parameters for time-space tradeoff. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 211–219, 1996.

[13] F. J. Díez. Local conditioning in bayesian networks. *Artificial Intelligence*, 87(1):1–20, 1996.

[14] H. N. Djidjev and J. R. Gilbert. Separators in graphs with negative and multiple vertex weights. *Algorithmica*, 23:57–71, 1999.

[15] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10(2):345–363, 1973.

[16] E.J. Horvitz, H.J. Suermondt, and G.F. Cooper. Bounded conditioning: Flexible inference for decisions under scarce resources. In *Proceedings of Conference on Uncertainty in Artificial Intelligence, Windsor, ON*, pages 182–193. Association for Uncertainty in Artificial Intelligence, Mountain View, CA, August 1989.

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

[18] F. V. Jensen, S.L. Lauritzen, and K.G. Olesen. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

[19] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of Royal Statistics Society, Series B*, 50(2):157–224, 1988.

[20] Z. Li and B.D. D'Ambrosio. Efficient Inference in Bayes Networks as a Combinatorial Optimization Problem. *International Journal of Approximate Reasoning*, 11:55–81, 1994.

[21] Richard Lipton, Donald Rose, and Robert Andre Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16(2):346–358, 1979.

[22] Richard Lipton and Robert Andre Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36(2):177–189, 1979.

[23] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

[24] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 478–489, Portland, OR, 1985.

[25] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[26] Mark A. Peot and Ross D. Shachter. Fusion and propagation with multiple observations in belief networks. *Artificial Intelligence*, 48(3):299–318, 1991.

[27] Arnie Rosenthal. Computing the reliability of complex networks. *SIAM Journal of Applied Mathematics*, 32(2):384–393, 1977.

[28] R. Shachter, S.K. Andersen, and P. Szolovits. Global Conditioning for Probabilistic Inference in Belief Networks. In *Proc. Tenth Conference on Uncertainty in AI*, pages 514–522, Seattle WA, 1994.

[29] R. Shachter, B.D. D'Ambrosio, and B. del Favero. Symbolic Probabilistic Inference in Belief Networks. In *Proc. Conf. on Uncertainty in AI*, pages 126–131, 1990.

[30] Prakash P. Shenoy. A valuation–based language for expert systems. *International Journal of Approximate Reasoning*, 5(3):383–411, 1989.

[31] Robert E. Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55:221–232, 1985.

[32] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.