Optimal Time–Space Tradeoff In Probabilistic Inference

David Allen and Adnan Darwiche

University of California, Los Angeles CA 90025, USA

Abstract. Recursive Conditioning, RC, is an any–space algorithm for exact inference in Bayesian networks, which can trade space for time in increments of the size of a floating point number. This smooth tradeoff is possible by varying the algorithm's cache size. When RC is run with a constrained cache size, an important problem arises: Which specific results should be cached in order to minimize the running time of the algorithm? RC is driven by a decomposition structure (a dtree or dgraph). In this research we examine the problem of searching for an optimal caching scheme for a given decomposition structure and present several time–space tradeoff curves for published Bayesian networks. Our results show that the memory requirements of these networks can be significantly reduced with only a minimal cost in time, allowing for exact inference in situations previously impractical. They also show that probabilistic reasoning systems can be efficiently designed to run under varying amounts of memory.

1 Introduction

Recursive Conditioning, RC, was recently proposed as an any–space algorithm for exact inference in Bayesian networks [5]. The algorithm works by using conditioning to decompose a network into smaller subnetworks that are then solved independently and recursively using RC. It turns out that many of the subnetworks generated by this decomposition process need to be solved multiple times redundantly, allowing the results to be stored in a cache after the first computation and then subsequently fetched during further computations. This gives the algorithm its any–space behavior since any number of results may be cached. This also leads to an important question, which is the subject of this research: "Given a limited amount of memory, which results should be cached in order to minimize the running time of the recursive conditioning algorithm?"

We approach this problem by formulating it as a systematic search problem. We then use the developed method to construct time–space tradeoff curves for some real–world Bayesian networks, and put these curves in perspective by comparing them to the memory requirements of state–of–the–art methods based on jointrees [11,15]. The curves produced illustrate that a significant amount of memory can be reduced with only a minimal cost in time. In fact, for much of their domains, the time–space curves we produce appear close to linear, with exponential behavior appearing only near the extreme

case of no caching. This dramatic space reduction, without a significant time penalty, allows one to practically reason with Bayesian networks that would otherwise be impractical to handle or in situations where the system memory is constrained.

This chapter is structured as follows. We start in Sect. 2 by providing some background on recursive conditioning and the cache allocation problem. We then formulate this problem in Sect. 3 as a systematic search problem. Time–space tradeoff curves for several published Bayesian networks are then presented in Sect. 4. Finally, in Sect. 5, we provide some concluding remarks.

2 Any–Space Inference

The RC algorithm for exact inference in Bayesian networks works by using conditioning and case analysis to decompose a network into smaller subnetworks that are solved independently and recursively. The algorithm is driven by a structure known as a decomposition tree (dtree), which controls the decomposition process at each level of the recursion. The RC algorithm has also been extended to work on a decomposition graph (dgraph), which has the ability to answer more queries than the dtree version [6,3]. RC can also be augmented to take advantage of determinism in networks by dynamically using logical techniques in the context of conditioning, which in some cases can significantly speedup the inference [2]. We will begin with a review of the dtree structure and then discuss RC.

2.1 Dtrees

Definition 1 *[5] A dtree for a Bayesian network is a full binary tree, the leaves of which correspond to the network conditional probability tables (CPTs). If a leaf node t corresponds to a CPT ϕ, then* $\mathsf{vars}(t)$ *is defined as the variables appearing in CPT ϕ.*

Figure 1 depicts a simple dtree. The root node t of the dtree represents the entire network. To decompose this network, the dtree instructs us to condition on variable B, called the cutset of root node t. Conditioning on a set of variables leads to removing edges outgoing from these variables, which for a cutset is guaranteed to disconnect the network into two subnetworks, one corresponding to the left child of node t and another corresponding to the right child of node t; see Fig. 1. This decomposition process continues until a boundary condition is reached, which is a subnetwork that has a single variable.

We will now present some notation needed to define additional concepts with regard to a dtree. The notation t_l and t_r will be used for the left child and right child of node t, and the function vars will be extended to internal nodes t: $\mathsf{vars}(t) \stackrel{def}{=} \mathsf{vars}(t_l) \cup \mathsf{vars}(t_r)$. Each node in a dtree has three more

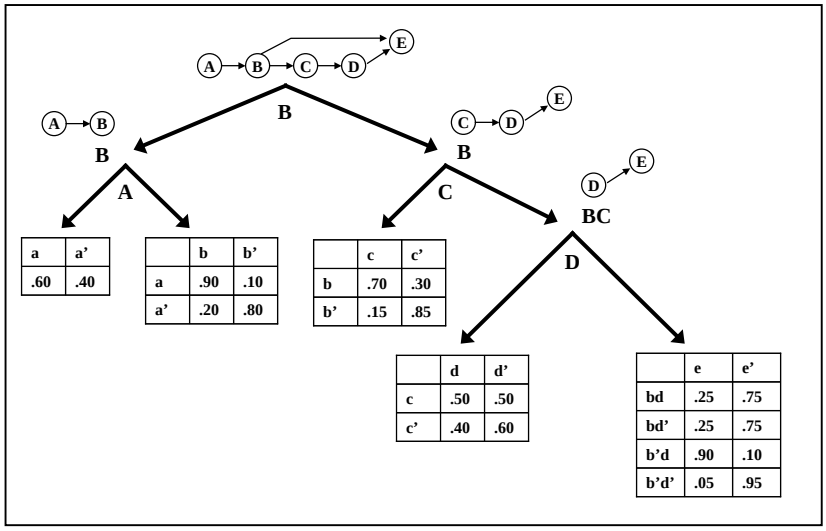

Fig. 1. An example dtree with the cutset labeled below each node and the context next to each node

sets of variables associated with it. The first two of these sets are used by the RC algorithm, while the third set is used to analyze the complexity of the algorithm.

Definition 2 *The cutset of internal node t in a dtree is:* $\mathsf{cutset}(t) \stackrel{def}{=} \mathsf{vars}(t_l) \cap \mathsf{vars}(t_r) - \mathsf{acutset}(t)$, *where* $\mathsf{acutset}(t)$ *is the union of cutsets associated with ancestors of node t in the dtree.*

Definition 3 *The context of node t in a dtree is:* $\mathsf{context}(t) \stackrel{def}{=} \mathsf{vars}(t) \cap \mathsf{acutset}(t)$.

Definition 4 *The cluster of node t in a dtree is:* $\mathsf{cluster}(t) \stackrel{def}{=} \mathsf{cutset}(t) \cup \mathsf{context}(t)$ *if t is a non-leaf, and as* $\mathsf{vars}(t)$ *if t is a leaf.*

The *width* of a dtree is the size of its maximal cluster -1. Figure 1 labels the cutset variables under each dtree node and the context variables beside them.

The cutset of a dtree node t is used to decompose the network associated with node t into the smaller networks associated with the children of t. That is, by conditioning on variables in $\mathsf{cutset}(t)$, one is guaranteed to disconnect the network associated with node t. The context of dtree node t is used to cache results: Any two computations on the network associated with node t will yield the same result if these computations occur under the same instantiation of variables in $\mathsf{context}(t)$. Hence, a cache is associated with each dtree node t, which stores the results of such computations (probabilities)

Algorithm 1 RC(t): Returns the probability of evidence **e** recorded on the dtree rooted at t

```
1: if t is a leaf node then
2:    return LOOKUP(t)
3: else
4:    y ← recorded instantiation of context(t)
5:    if cache?(t) and cache_t[y] ≠ nil then
6:       return cache_t[y]
7:    else
8:       p ← 0
9:       for instantiations c of uninstantiated vars in cutset(t) do
10:         record instantiation c
11:         p ← p + RC(t_l)RC(t_r)
12:         un–record instantiation c
13:      when cache?(t), cache_t[y] ← p
14:      return p
```

Algorithm 2 LOOKUP(t)

```
φ ← CPT of variable X associated with leaf t
if X is instantiated then
   x ← recorded instantiation of X
   u ← recorded instantiation of X's parents
   return φ(x|u)        // φ(x|u) = Pr(x|u)
else
   return 1
```

indexed by instantiations of $\mathsf{context}(t)$. This means that the size of a cache associated with dtree node t can grow as large as the number of instantiations of $\mathsf{context}(t)$.

For a given Bayesian network, many different dtrees exist and the quality of the dtree significantly affects the resource requirements of RC. The width is one important measure of this, as RC's time complexity is exponential in this value. The construction of dtrees is beyond the scope of this chapter, but in [5,7] it was shown how to create them from elimination orders, jointrees, or directly by using the hMeTiS [12] hypergraph partitioning program. It should also be pointed out that the clusters of a dtree actually form a binary jointree, which was shown to be more efficient than standard jointrees for the Shenoy–Shafer algorithm, at the expense of additional memory [16].

2.2 Recursive Conditioning

Given a Bayesian network and a corresponding dtree with root t, the RC algorithm given in Algorithms 1 and 2 can be used to compute the probability of evidence **e** by first "recording" the instantiation **e** and then calling RC(t), which returns the probability of **e**.

Our main concern here is with Line 5 and Line 13 of the algorithm. On Line 5, the algorithm checks whether it has performed and cached this computation with respect to the subnetwork associated with node t. A computation is characterized by the instantiation of t's context, which also serves as an index into the cache attached to node t. If the computation has been performed and cached before, its result is simply fetched. Otherwise, the computation is performed and its result is possibly cached on Line 13.

When every computation is cached, RC uses $O(n \exp(w))$ space and $O(n \exp(w))$ time, where n is the number of nodes in the network and w is the width of the dtree. This corresponds to the complexity of jointree algorithm, assuming that the dtree is generated from a jointree [5]. When no computations are cached, the memory requirement of RC is reduced to $O(n)$, in which case the time requirement increases to $O(n \exp(w \log n))$. Any amount of memory between these two extremes can also be used in increments of the size of a floating point number, a cache value.

Suppose now that the available memory is limited and we can only cache a subset of the computations performed by RC. The specific subset that we cache can have a dramatic effect on the algorithm's running time. A key question is then to choose that subset which minimizes the running time, which is the main objective of this research. We refer to this as the *secondary optimization problem,* with the first optimization problem being that of constructing an optimal dtree.

Most of our results in this chapter are based on a version of RC which not only computes the probability of evidence $\mathbf{e}$, but also posterior marginals over families and, hence, posterior marginals over individual variables. This version of RC uses a *decomposition graph* (dgraph), which is basically a set of dtrees that share structure.

2.3 DGraphs

A dgraph can be constructed from a dtree by orienting the dtree with respect to each of its root nodes [6]. This can be done while maintaining the width, as each of the oriented dtrees will have a width no greater than the original. Figure 2 graphically depicts this process, beginning with the Bayesian network and constructing a dtree, a dtree oriented with respect to leaf $\phi(D|B)$, and finally a complete dgraph. It should be noted that each of the four root nodes in the dgraph corresponds to a valid dtree, so this dgraph actually contains four dtrees which share a significant portion of their structure.

The code in Algorithms 1 and 2 is also used in the dgraph version of RC, where RC(t) is called once on each root t of the dgraph. As a side effect to computing the probability of evidence, RC using a dgraph also computes the posterior marginal of each family in the network [6]. This version of RC uses more memory as it maintains more caches, but it is more meaningful when it comes to comparing our time–space tradeoff curves with the memory

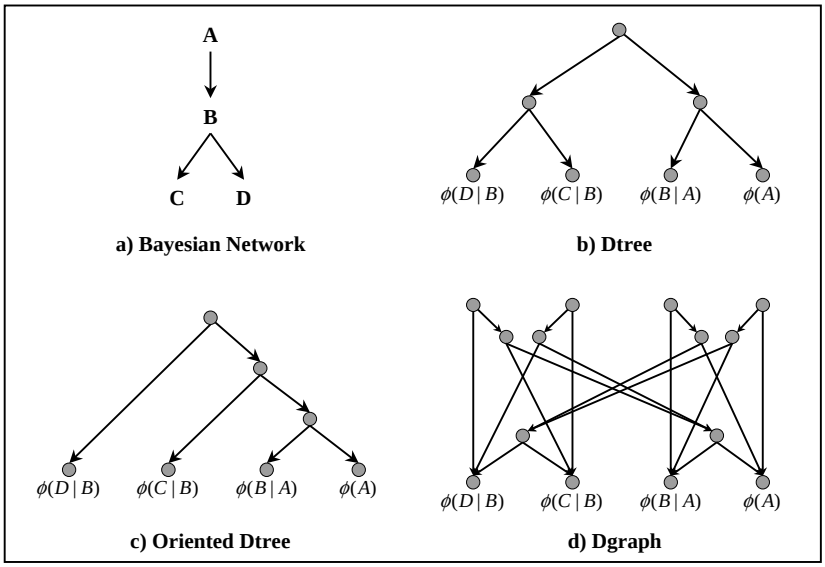

Fig. 2. A Bayesian network and some decomposition structures

requirements of jointree algorithms, as this version of RC is equally powerful to these algorithms.

3 The Cache Allocation Problem

The total number of computations that a dgraph (or dtree) node t needs to cache equals the number of instantiations of $\mathsf{context}(t)$. Given a memory constraint, however, one may not be able to cache all these computations, and we need a way to specify which results in particular to cache. A *cache factor cf* for a dgraph is a function which maps each internal node t in the dgraph into a number $cf(t)$ between 0 and 1. Hence, if $cf(t) = .75$, then node t can only cache 75% of these total computations. A *discrete* cache factor is one which maps every internal dgraph node into either 1 or 0: all of the node's computations are cached, or none are cached. The RC code in Algorithms 1 and 2 assumes a discrete cache factor, which is captured by the flag $\mathsf{cache?}(t)$, indicating whether caching will take place at dgraph node t.

One can count the number of recursive calls made by RC (and, hence, compute its running time) given any discrete cache factor. Specifically, if t^p denotes a parent of node t in a dgraph, and $S^\#$ denotes the number of instantiations of variables S, the number of recursive calls made to node t is [5,6]:

$$calls(t) = \sum_{t^p} cutset(t^p)^\# [cf(t^p) context(t^p)^\# + (1 - cf(t^p)) calls(t^p)]. \quad (1)$$

If the cache factor is not discrete, the above formula gives the average number of recursive calls, since the actual number of calls will depend on the specific computations cached. This equation is significant as it can be used to predict the worst-case expected time requirement of RC under a given caching scheme. RC runs significantly faster as more evidence is set on the network. For example, on Munin1 we have seen instances which would require 12 minutes with no evidence run in just 23 seconds with evidence set on the network.

RC can additionally determine how many times a cache value will be used. This is important because not every cache is useful, as some are never looked up; these are referred to as *dead caches*. Dead caches are those whose context is a superset of its parents context, and can be determined before the cache allocation search. On a dtree, each node only has a single parent and therefore dead caches can be determined by comparing the context of a node with that of its parent. In dgraphs, which are composed of multiple dtrees sharing structure, we will differentiate between *dtree dead caches* and *dgraph dead caches*. Dtree dead caches are those caches which for any individual dtree in the dgraph would not be used. However, when computing all posterior marginals on dgraphs, where nodes may have multiple parents, some of these dead caches would not be useful for any single dtree, but can be useful because one dtree could fill it and another dtree could lookup the stored values. Therefore, a dgraph dead cache is located at a node with only one parent in addition to the context for the node being a superset of its parent.

We focus in this research on searching for an optimal discrete cache factor, given a limited amount of memory, where optimality is with respect to minimizing the number of recursive calls. To this end, we will first define a search problem for finding an optimal discrete cache factor and then develop a depth–first branch–and–bound search algorithm. We will also use the developed algorithm to construct the time–space tradeoff curves for some published Bayesian networks from various domains, and compare these curves to the memory demands and running times of jointree algorithms.

3.1 Cache Allocation as a Search Problem

The cache allocation problem can be phrased as a search problem in which *states* in the search space correspond to partial cache factors that do not violate the given memory constraint, and where an *operator* extends a partial cache factor by making a caching decision on one more dgraph node. The *initial state* in this problem is the empty cache factor, in which no caching decisions have been made for any nodes in the dgraph. The *goal states* correspond to complete cache factors, where a caching decision has been made for every dgraph node, without violating the given memory constraint. Suppose for example that we have a dgraph with three internal nodes t_1, t_2, t_3. This will then lead to the search tree in Fig. 3. In this figure, each node n in the

search tree represents a partial cache factor cf. For example, the node in bold corresponds to the partial cache factor $cf(t_1) = 0$, $cf(t_2) = 1$ and $cf(t_3) =?$. Moreover, if node n is labeled with a dgraph node t_i, then the children of n represent two possible extensions of the cache factor cf: one in which dgraph node t_i will cache all computations (1–child), and another in which dgraph node t_i will cache no computations (0–child).

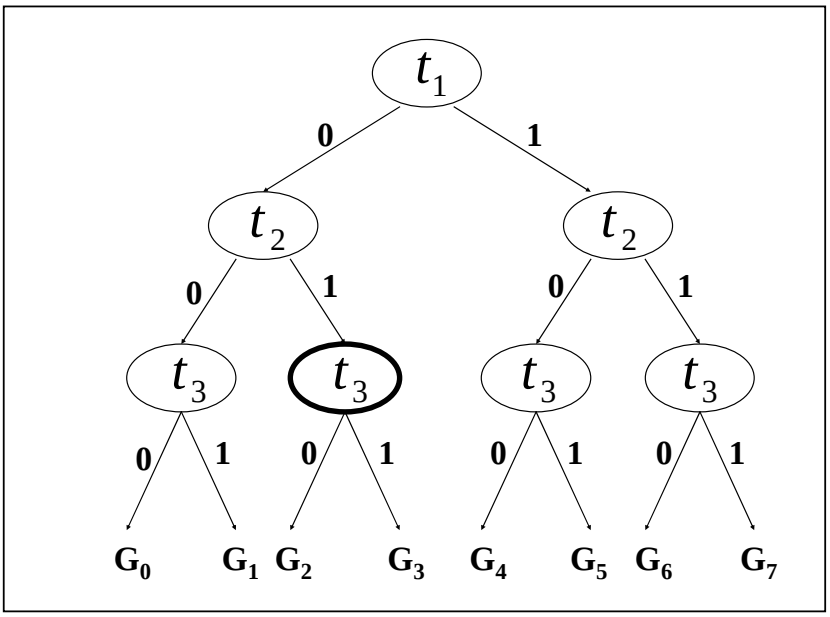

Fig. 3. Search tree for a dgraph with 3 internal nodes

According to the search tree in Fig. 3, one always makes a decision on dgraph node t_1, followed by a decision on dgraph node t_2, and then node t_3. A fixed ordering of dgraph nodes is not necessary, however, as long as the following condition is met: A decision should be made on a dgraph node t_i only after decisions have been made on all its ancestors in the dgraph. We will explain the reason for this constraint later on when we discuss cost functions.

In the search tree depicted in Fig. 3, the leftmost leaf (G_0) represents no caching, while the rightmost leaf (G_7) represents full caching. The search trees for this problem have a maximum depth of d, where d is the number of internal nodes in the dgraph. Given this property, depth–first branch–and–bound search is a good choice given its optimality and linear space complexity [14]. It is also an anytime algorithm, meaning that it can always return its best result so far if interrupted, and if run to completion will return the optimal solution. Hence, we will focus on developing a depth–first branch–and–bound search algorithm.

It should be noted that this search for a cache allocation only needs to be done once, while the user will usually be interested in running multiple probability calculations based on the results.

3.2 Cost Functions

The depth–first branch–and–bound (DFBnB) algorithm requires a cost function f which assigns a cost $f(n)$ to every node n in the search tree. The function $f(n)$ estimates the cost of an optimal solution that passes through n. The key here is that $f(n)$ must not overestimate that cost; otherwise, one loses the optimality guarantee offered by the search algorithm. We will now develop such a cost function $f(n)$ based on the following observations. Since each node n represents a partial cache factor cf, function $f(n)$ must estimate the number of recursive calls made to RC based on an optimal completion of cache factor cf. Consider now the completion cf' of cf in which we decide to cache at each dgraph node that cf did not make a decision on. This cache factor cf' is the best completion of cf from the viewpoint of running time, but it may violate the constraint given on total memory. Yet, we will use it to compute $f(n)$ as it guarantees that $f(n)$ will never overestimate the cost of an optimal completion of cf.

One important observation in this regard is that once the caching decision is made on the ancestors of dgraph node t, we can compute exactly the number of recursive calls that will be made to dgraph node t (see Equation 1). Therefore, when extending a partial cache factor, we will always insist on making a decision regarding a dgraph node t for which decisions have been made on all its ancestors. This improves the quality of the estimate $f(n)$ as n gets deeper in the tree. It also allows us to incrementally compute this estimate based on the estimate of n's parent in the search tree.

3.3 Pruning

As depicted by the search tree in Fig. 3, there is potentially an exponential number of goal nodes in the search tree and the combinatorial explosion of exhaustive search can become unmanageable very quickly. Hence the search algorithm must eliminate portions of the search space while still being able to guarantee an optimal result. One of the key methods of doing this is by pruning parts of the search tree which are known to contain non-optimal results. The DFBnB algorithm does this by pruning search tree nodes when the cost function $f(n)$ is larger than or equal to the current best solution. Hence, more accurate cost functions will allow more pruning. Another major source of pruning is the given constraint on total memory. This is accomplished by pruning a search tree node and all its descendants once it attempts to assign more memory to caches than is permitted by the memory constraint.

3.4 Search Decisions

Now that we have chosen a cost function, we are still left with two important choices in our search algorithm: (1) which child of a search tree node to expanded first, and (2) in what order to visit dgraph nodes during search. Expanding the 1–child first is a greedy approach, as it attempts to fully cache at a dgraph node whenever possible. Results on many different networks have shown that in many cases, expanding the 1–child before the 0–child appears to be equal to or better than the opposite [1], and it is this choice that we adopt in our experiments. The specific order in which we visit dgraph nodes in the search tree turns out to have an even more dramatic effect on the efficiency of search. Even though we make caching decisions on parent dgraph nodes before their children, there is still a lot of flexibility. Our experimentation on many networks has shown that choosing the dgraph node t with the largest $\mathsf{context}(t)^{\#}$ is orders of magnitude more efficient than some other basic ordering heuristics [1]. This choice corresponds to choosing the dgraph node with the largest cache, and it is the one we use in our search algorithm.

4 Time–Space Tradeoff

The main goal of this section is to present time–space tradeoff curves for a number of benchmark Bayesian networks, some of which are obtained from [4] and others are included in the distributions of [9,10]. The main points to observe with respect to each curve is the slope of the curve, which provides information on the time penalty one pays when reducing space in probabilistic inference. The second main point is to compare the produced curves with the time and space requirement of jointree methods, as the version of RC we are using provides the same functionality as these algorithms (that is, probability of evidence and posterior marginals over variables and families). This baseline comparison is important as it places our results in the context of state–of–the–art inference systems. The results presented here use the same datasets as those in [3]. This version however ignores only dgraph dead caches, while the version in [3] ignored dtree dead caches. Ignoring only dgraph dead caches adds more memory under full caching, but runs faster under constrained memory.

Time–space tradeoff curves. Figures 4 and 5 depict optimal discrete time–space tradeoff curves for two networks. These curves were generated as follows. A jointree was first generated for the network using Hugin.[1] The jointree was then converted into a dtree as described in [5]. The dtree was finally converted into a dgraph as described in [6]. Two sets of results were then generated:

[1] We used Hugin's default setting: the minimum fill–in weight heuristic in conjunction with prime component analysis.

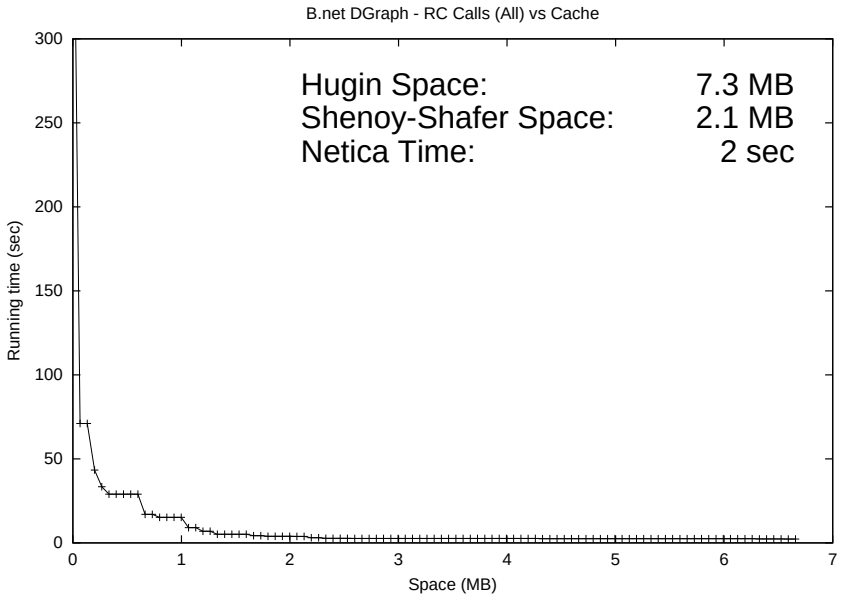

Fig. 4. Time–space tradeoff on B

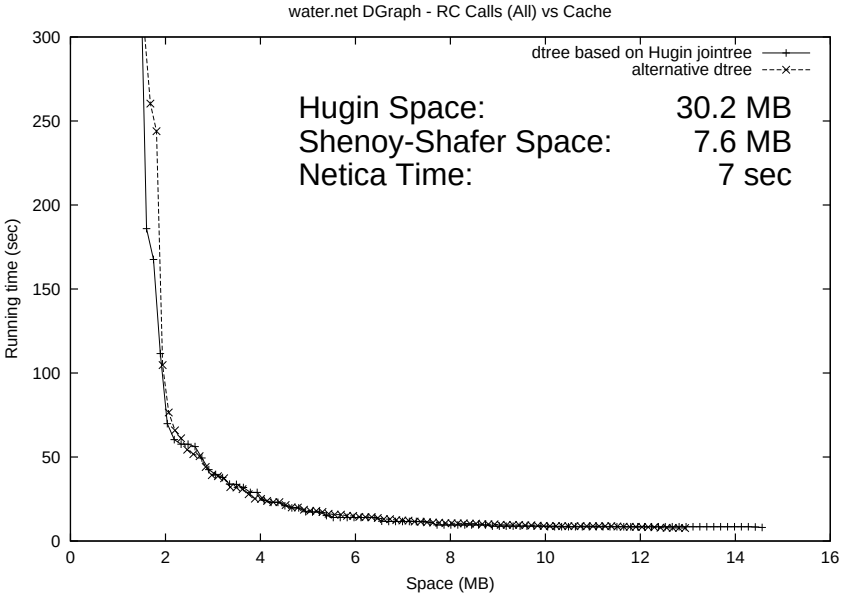

Fig. 5. Time–space tradeoff on Water

- We computed the space requirements for jointree algorithms, using both the Hugin [11] and Shenoy–Shafer [15] architectures (on the non–binary jointree). For the first architecture, we assumed one table for each clique and one table for each separator. For the second, we assumed two tables for each separator (no tables for cliques). We also performed propagation on the jointree using Netica [13], which implements the Hugin algorithm, and recorded the running time.
- We then ran our search algorithm to find an optimal cache factor under different memory constraints, where we generated 100 data points for each curve. For each caching factor that we identified, we computed the number of recursive calls that will be made by RC under that factor and converted the calls to seconds.[2]

A number of observations are in order here. First, RC is using memory efficiently, and would use a similar amount of memory as the Shenoy–Shafer algorithm would if run on the binary jointree determined by the dtree. Second, the curves show that a significant amount of memory can sometimes be reduced from full caching with only a limited increase in the time required; in fact, the exponential growth appears to be occurring only near the lower extreme of no caching. The space requirement for Water (Fig. 5), for example, can be reduced to 30% while only increasing the running time by a factor of 2.9. Moreover, the space requirements for B (Fig. 4) can be reduced to about 3% while increasing the running time by a factor of 19. Finally, we note that each optimal search for the B network took less than a second and for Water took less than three minutes. We stress though that such searches need to be done only once for a network, and their results can then be used for many further queries.

Non–optimal tradeoffs. On some networks, the search space is too large to solve the cache allocation optimally using our search algorithm, but the anytime nature of the algorithm allows us to interrupt the search at any point and ask for the best result obtained thus far. Figures 6, 7, and 8 were generated by allowing the search to run for ten minutes. Even though these curves are not optimal, they are useful practically. For example, according to these curves, the memory requirement of Barley can be reduced from about 54 MB to about 8 MB while only increasing the running time from about 1 to 3 minutes. Moreover, the space requirement of Munin1 can be reduced from about 450 MB to 180 MB, while increasing the running time from about 13 minutes to about 3.5 hours. Encouraged by such results, we are planning to investigate other (non–optimal) search methods, such as local search.

Dtrees vs dgraphs. Running RC on a dtree takes less space than running it on a dgraph, but produces much less information (probability of evidence instead of posterior marginals). To illustrate this difference concretely,

[2] Our Java implementation of RC on a Sun Ultra 10, 440 MHz computer with 256 MB of RAM, makes an average number of three million recursive calls per second.

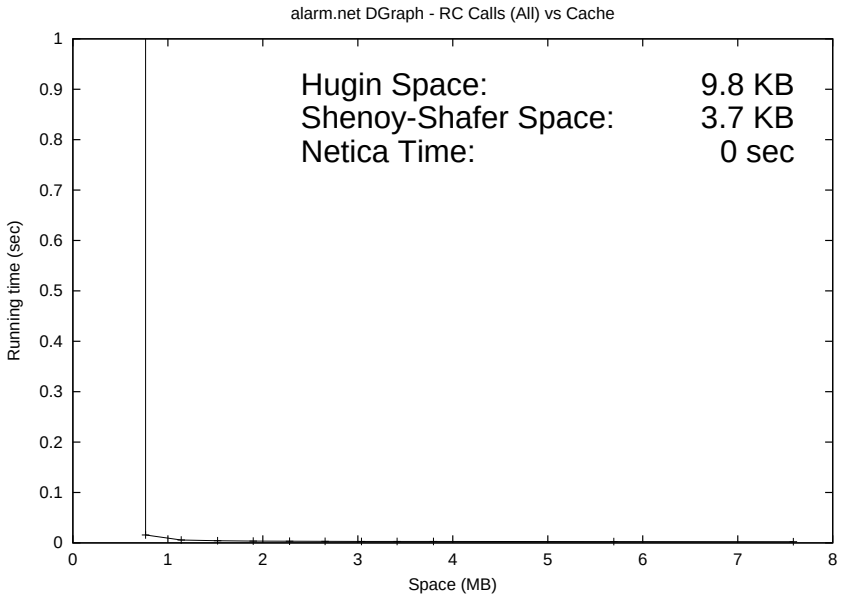

Fig. 6. Time–space tradeoff on Alarm

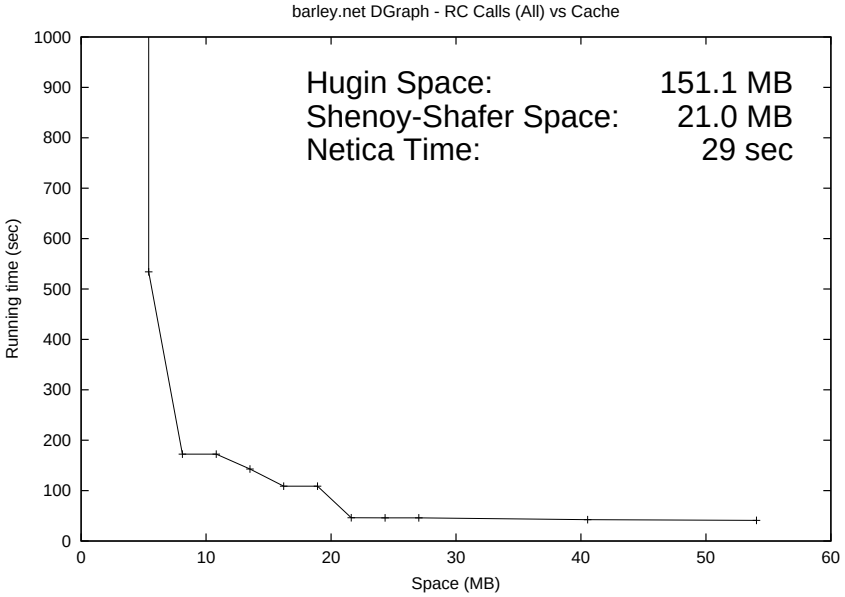

Fig. 7. Time–space tradeoff on Barley

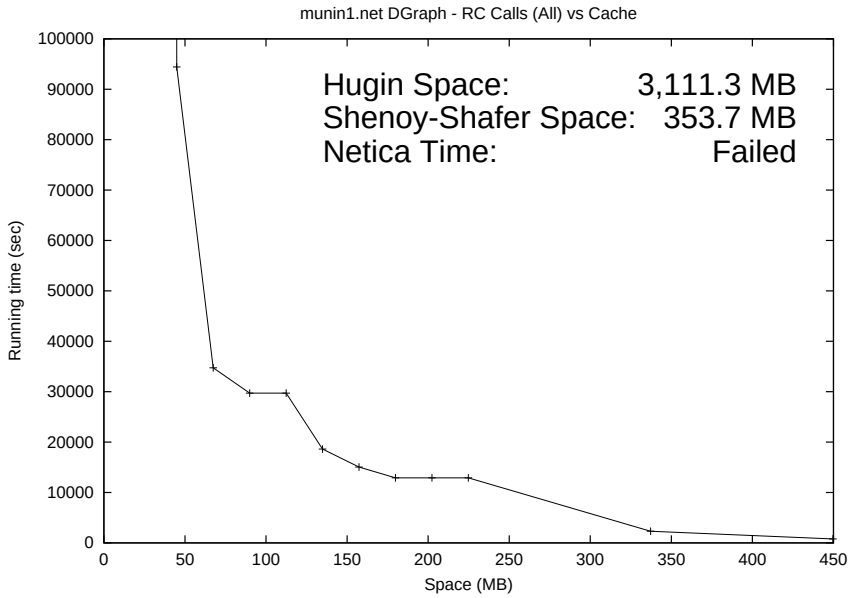

Fig. 8. Time–space tradeoff on Munin1

we present in Fig. 9 two tradeoff curves for the Water network, assuming a dtree version of RC, which require much less memory compared to the curves in Fig. 5. Suppose now that we only have 1 MB of memory, instead of the 7.6 MB or 30.2 MB required by jointree algorithms, and we want to compute the posterior marginals for all variables. According to Fig. 5, we can do this in 693 seconds using the dgraph version of RC. The dtree version takes 16.3 seconds to compute the probability of evidence under this amount of memory, and we would have to run it 85 times to produce all posterior marginals for the Water network (given variable cardinalities in Water).

Effect of dtree/dgraph on tradeoff. Our notion of optimality for tradeoff is based on a given dtree/dgraph; hence, generating different decomposition structures could possibly lead to better time–space tradeoff curves. To illustrate this point, we generated tradeoff curves for the Water network based on multiple dtrees/graphs, as shown in Figs. 5 and 9. One observation that we came across is that dtrees/graphs that are based on jointrees tend to require less time under full caching, but are not necessarily best for tradeoff towards the no caching region; see Fig. 9 for an example. Yet, we used such dtrees/graphs in this chapter in an effort to provide a clear baseline for comparison with jointree methods. If we relax this constraint, however, we can obtain better tradeoff curves than is generally reported here, as illustrated by Figs. 5 and 9. The specific way in which properties of a dtree/dgraph influence the quality of corresponding time–space tradeoff curves is not very

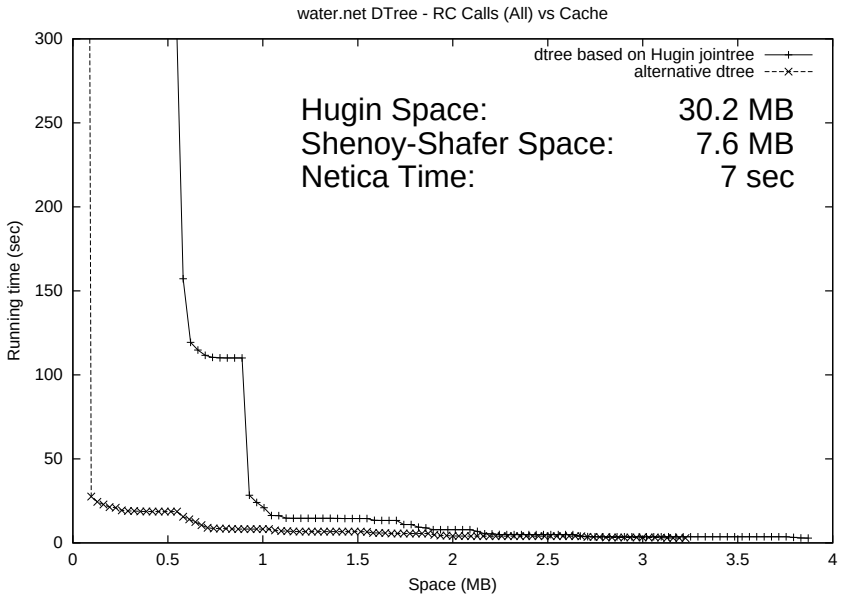

Fig. 9. Time–space tradeoff on Water for computing probability of evidence

well understood, however, and we hope to shed more light on this in future work.

Size of search space. It should be noted that the difficulty of obtaining an optimal time–space tradeoff curve on some networks is not due to a large space requirement, but is due mostly to the number of nodes in the Bayesian network as that is what decides the size of search space. To further illustrate this point, we generated a network randomly with 40 nodes (many of them non–binary), 86 edges, and a width of 14. This network requires extensive memory but has a relatively small number of variables. In fact, both Netica and Hugin were unable to compile the network requiring about 6 GB and 11 GB respectively. We were able, however, to produce an optimal time–space tradeoff curve for this network. The curve for the dtree version of RC is shown in Fig. 10. According to this curve, we can compute the probability of any evidence on this network in about 2 hours using only about 75 MB.

Related work. We close this section by a note on related work for time–space tradeoff in probabilistic reasoning, which takes a different approach [8]. In this work, large separators in a jointree are removed by combining their adjacent clusters, which has the effect of reducing the space requirements of the Shenoy–Shafer architecture (as we now have fewer separators), but also increasing its running time (as we now have larger clusters). The tradeoffs permitted by this approach, however, are coarser than those permitted by RC as discussed in [5]. Furthermore, the secondary optimization problem of which

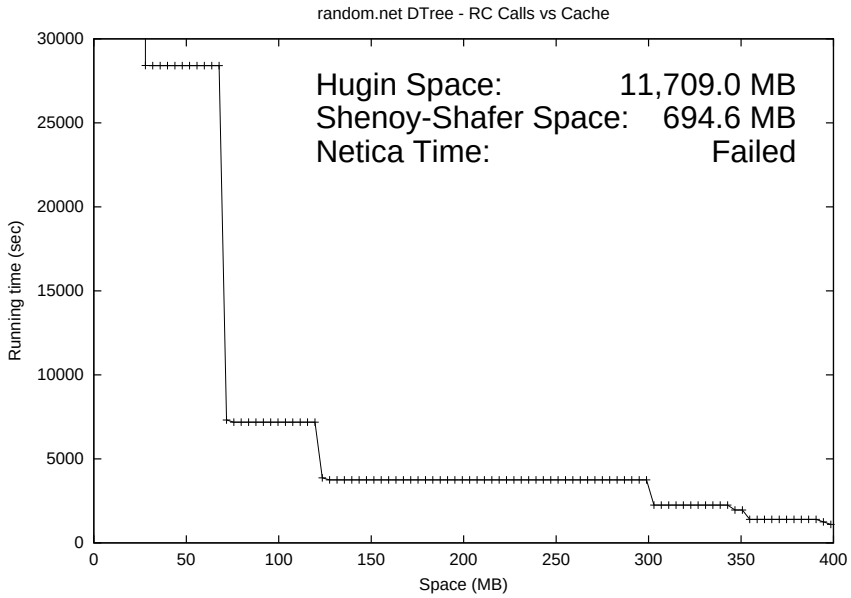

Fig. 10. Time–space tradeoff on Random for computing probability of evidence

separators to remove in order to minimize running time is not addressed in [8] for the proposed approach, as we do in this chapter for the RC approach.

5 Conclusions

The main contribution of this research is a formal framework, and a corresponding working system, for trading space for time when designing probabilistic reasoning systems based on Bayesian networks. The research is based on the algorithm of recursive conditioning, and is accompanied with a set of experimental results showing that a significant amount of memory can sometimes be reduced while only incurring a reasonable penalty in running time. The proposed framework is then beneficial for designing reasoning systems with limited memory, as in embedded systems, and for reasoning with challenging networks on which jointree algorithms can exhaust the system memory.

Recursive conditioning and the described time–space tradeoff system have been implemented in JAVA in the SamIam tool, which is available publically [17].

Acknowledgments

This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

References

1. Allen, D., and Darwiche, A. Optimal time–space tradeoff in probabilistic inference. In *Proceedings of the First European Workshop on Probabilistic Graphical Models* (2002), pp. 1–8.
2. Allen, D., and Darwiche, A. New advances in inference by recursive conditioning. To appear in *Uncertainty in Artificial Intelligence: Proceedings of the Nineteenth Conference (UAI)* (2003).
3. Allen, D., and Darwiche, A. Optimal time–space tradeoff in probabilistic inference. To appear in *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)* (2003).
4. Bayesian Network Repository. http://www.cs.huji.ac.il/labs/compbio/Repository/, URL.
5. Darwiche, A. Recursive conditioning. *Artificial Intelligence 126* (February 2001), 5–41.
6. Darwiche, A. Decomposition graphs. Tech. Rep. D-134, UCLA, 2002.
7. Darwiche, A., and Hopkins, M. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *Proc. 6th European Conf. on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU'01, Toulouse, France)* (2001), pp. 180–191.
8. Dechter, R., and Fattah, Y. E. Topological parameters for time-space tradeoff. *Artificial Intelligence 125*, 1-2 (2001), 93–118.
9. GeNIe. http://www2.sis.pitt.edu/∼genie/, URL.
10. Hugin Expert. http://www.hugin.com/, URL.
11. Jensen, F. V., Lauritzen, S. L., and Olesen, K. G. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly 4* (1990), 269–282.
12. Karypis, G., and Kumar, V. Hmetis: A hypergraph partitioning package. http://www.cs.umn.edu/karypis, 1998.
13. Norsys Software Corp. http://www.norsys.com/, URL.
14. Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization.* Dover Publications, Inc., 1998.
15. Shafer, G. R., and Shenoy, P. P. Probability propagation. *Annals of Mathematics and Artificial Intelligence 2* (1990), 327–352.
16. Shenoy, P. P. Binary join trees for computing marginals in the shenoy-shafer architecture. *International Journal of Approximate Reasoning 17* (1997), 239–263.
17. UCLA Automated Reasoning Group. SamIam: Sensitivity Analysis, Modeling, Inference And More. http://reasoning.cs.ucla.edu/samiam, URL.

Index