A Greedy Algorithm for Time–Space Tradeoff in Probabilistic Inference

David Allen and Adnan Darwiche
Computer Science Department
University of California
Los Angeles, CA 90095
{dlallen,darwiche}@cs.ucla.edu

James D. Park
Raytheon Missile Systems
TU, Bldg. M/02 T18
P.O. Box 11337
Tucson, AZ 85734
James_D_Park@raytheon.com

Abstract

Recursive Conditioning, RC, is an any–space algorithm for exact inference in Bayesian networks, which can trade space for time by varying its 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? Prior work has focused on finding the optimal cache allocation using branch-and-bound search. The cost of this search, however, can be significant, if not infeasible, for networks that contain a large number of variables. To address this, we propose in this paper a greedy algorithm which has quadratic time worst-case complexity and which appears empirically to produce near–optimal cache allocations on the networks tested. This new allocation algorithm allows RC to do efficient time–space tradeoff on substantially larger networks than it could previously.

1 Introduction

Recursive Conditioning, RC, is an any–space algorithm for exact inference in Bayesian networks (Darwiche, 2001). The algorithm works by using conditioning to decompose a network into smaller subnetworks that are then solved independently and recursively using RC. 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: "Given a limited amount of memory, which results should be cached in order to minimize the running time of the algorithm?"

Previous work has been done to find a memory allocation with an optimal running time by using systematic search techniques (Allen and Darwiche, 2003b; Allen and Darwiche, 2004). The main drawback of this approach, however, is that it can be very time consuming, if not infeasible, on large networks. We will therefore complement the previous technique by proposing a greedy algorithm which has a quadratic time worst-case complexity, and then show that on many published Bayesian networks the greedy algorithm produces results which are near–optimal. This new algorithm is useful in practice when there is not enough time for a lengthy preprocessing step to search for the optimal memory usage. It can also be useful in creating a good memory allocation to use as a seed for the branch-and-bound search. This new algorithm allows RC to be used for time–space tradeoff on networks which were too large for the previous allocation algorithm (thousands of nodes).

This paper is structured as follows. We begin in Section 2 with background on RC. The cache allocation problem and the new greedy memory allocation algorithm are then discussed in Section 3. Experimental results and time-space tradeoff graphs on published Bayesian networks and challenging bioinformatics networks are presented in Section 4, followed by some conclusions in Section 5.

2 Any–Space Inference

RC works by using conditioning and case analysis

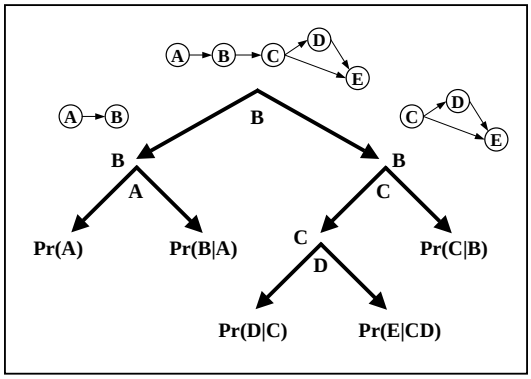

Figure 1: A dtree with the cutset labeled below each node and the context next to each node.

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. We will first review the dtree structure and then discuss RC.

2.1 Dtrees

Definition 1 *(Darwiche, 2001) 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. For an internal node t, with left child t_l and right child t_r,* $\mathsf{vars}(t) \stackrel{def}{=} \mathsf{vars}(t_l) \cup \mathsf{vars}(t_r)$.

Figure 1 depicts a simple dtree with the root node t representing 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 Figure 1. This decomposition process continues until a boundary condition, a subnetwork that has a single variable, is reached.

Each node in a dtree has three sets of variables associated with it. The first two of these sets are used by RC, 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. The context of node t in a dtree is:* $\mathsf{context}(t) \stackrel{def}{=} \mathsf{vars}(t) \cap \mathsf{acutset}(t)$. *The cluster of node t in a dtree is:* $\mathsf{cutset}(t) \cup \mathsf{context}(t)$ *if t is a non-leaf, and as* $\mathsf{vars}(t)$ *if t is a leaf. The maximal cluster size -1 is known as the* width.

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) indexed by 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. (Darwiche, 2001; Darwiche and Hopkins, 2001) discuss some dtree generation methods.

2.2 Recursive Conditioning

Given a Bayesian network and a corresponding dtree with root t, RC (Algorithms 1 and 2) can compute the probability of evidence $\mathbf{e}$[1] by first "recording" the instantiation $\mathbf{e}$ and then calling $\mathrm{RC}(t)$, which returns the probability of $\mathbf{e}$.

Lines 5 and 13 deal with RC's caching. 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 it has, it is simply fetched.

[1]This paper will use the following standard notational convention. Variables will be depicted by uppercase letters (A) and their values by lowercase letters (a). Sets of variables will be written in boldface uppercase (**A**) and sets of instantiated variables will be in boldface lowercase (**a**).

Algorithm 1 RC(t): Returns the probability of evidence $\mathbf{e}$ 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 unbound 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
```

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 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 the jointree algorithm, assuming that the dtree is generated from a jointree (Darwiche, 2001). When no computations are cached, the memory requirement 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 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. Therefore, we would like to choose that subset which minimizes the running time.

2.3 Computing Marginals Over Families

When RC is run on a dtree, it computes the probability of evidence, $Pr(\mathbf{e})$. However, another common query is to compute the posterior marginals over the variables or families in the network, for example $Pr(A, \mathbf{e})$. In order to compute these values, RC can be run on a different structure known as a decomposition graph (dgraph), which is a set of dtrees that share structure. We previously reported this ability, however the details were not presented (Allen and Darwiche, 2004).

Algorithm 3 Orient(t,u): Orient dtree with root node t with respect to leaf node u.

1: Remove the root t and replace it by an undirected edge between its children.
2: Remove the edge between the leaf node u and its parent v.
3: Add a new node t' with undirected edges to nodes u and v.
4: Direct all edges away from t', making t' the root of the oriented dtree.

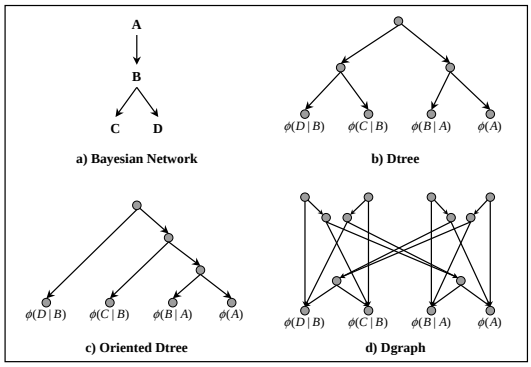

Figure 2: Converting a dtree into a dgraph.

Our first observation is that when RC is applied to a dtree under evidence $\mathbf{e}$, it not only computes the probability of $\mathbf{e}$, but also the marginals over variables in the cutset of the root. Specifically, if $\mathbf{U}$ is the root cutset, then as RC iterates over all possible instantiations $\mathbf{u}$ of $\mathbf{U}$, it computes $Pr(\mathbf{u}, \mathbf{e})$. From these values, the posterior marginals over $\mathbf{U}$ can be computed.

If the variables of interest do not appear in the root cutset, the dtree can be oriented so that they do, while maintaining RC's complexity. Pseudocode for the orientation process is given in Algorithm 3 and an example is shown in Figure 2.

Suppose that we have variable X and its parents $\mathbf{U}$ and that the marginal over this family is desired, $Pr(X, \mathbf{U}, \mathbf{e})$. This can be accomplished by taking a dtree t for the network and orienting it with respect to the leaf node containing the CPT of variable X. The root cutset in this new dtree t' will contain either X and its parents $\mathbf{U}$, or solely its parents $\mathbf{U}$. In the first case, Algorithm 1 computes $Pr(X, \mathbf{U}, \mathbf{e})$, the desired result. The second case only appears

when variable X is a leaf node in which case Algorithm 1 computes $Pr(\mathbf{U}, \mathbf{e})$. However, as X is a leaf node, $Pr(X, \mathbf{U}, \mathbf{e})$ is easily obtained from this.

By orienting the dtree towards each CPT in the network, we are guaranteed to obtain the marginals over every network family. The set of resulting dtrees will be called a *decomposition graph (dgraph)*, which is simply a set of dtrees that share structure (Figure 2). If we are only interested in marginals over variables (instead of over families), we only need to orient the dtree with respect to a subset of the CPTs, such that all variables appear in at least one root's cutset.

The following theorem shows that every (oriented) dtree that participates in the dgraph has no greater width than the original dtree, and that the total size of the dgraph is linear in the size of the original dtree. This proves that the complexity of RC on dgraphs is the same as its complexity on dtrees (under full caching).

Theorem 1 *Given a dtree with n nodes and width w, the dgraph generated by orienting the dtree with respect to each leaf node will have width w, $(5n - 7)/2$ nodes and $4(n - 2)$ edges for $n > 3$.*[2]

The dgraph version of RC uses more memory as it maintains more caches. These additional caches also make the cache allocation problem much harder as there are many more nodes in a dgraph where a decision needs to be made on whether to cache or not.

3 Cache Allocation Problem

The total number of computations that a dgraph (or dtree) node t desires 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 therefore must 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 Algorithm 1 assumes a discrete cache factor, which is captured by the flag $\mathsf{cache?}(t)$, indicating whether caching will take place at node t.

One can count the number of recursive calls made by RC given any discrete cache factor. This is significant as it is proportional to RC's time requirement in the worst case of having no evidence (as evidence allows inconsistent instantiations to be skipped). Specifically, if t^p denotes the parent of node t in a dtree, and $S^\#$ denotes the number of instantiations of variables S, the number of recursive calls made to node t is (Darwiche, 2001):

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

If the cache factor of t^p is 0, a call to t^p is then guaranteed to generate a call to child t for each instantiation of t^p's cutset. If the cache factor of t^p is 1, it will only make calls to child t to fill up its cache, but will then lookup further computations from the cache (therefore it makes one call for each instantiation of the cutset for each instantiation of its context). Equation 1 can be extended to dgraphs as follows:

$$calls(t) = \sum_{t^p} cutset(t^p)^\#[cf(t^p)context(t^p)^\# + (1 - cf(t^p))calls(t^p)]. \quad (2)$$

The only difference between Equations 1 and 2 is the summation over each parent. This is due to the fact that as RC is run on each root node, each parent will still make calls to its children based on Equation 1, and since this is independent for each parent of t, the number of calls from each parent can be summed to determine the total number of calls. If the cache factor is not discrete, the above formulas give the average number of recursive calls, since the actual number of calls will depend on the specific computations cached.

Previously, systematic search techniques were used to produce optimal time-space tradeoff curves using discrete cache factors (Allen and Darwiche, 2003b; Allen and Darwiche, 2004). However, some problems were too large for the search to finish in a reasonable amount of time. Also, in some situations (e.g. Bayesian network inference tools), it may be

[2] If $n \leq 3$, then $n = 3$ (two node network) or $n = 1$ (single node network) and any dtree on these networks can already compute all marginals.

Algorithm 4 GreedyMemoryAllocation().

1: $candidates \leftarrow$all internal dgraph nodes
2: $availableSpace \leftarrow$maximum number of caches entries allowed (total memory / memory per cache entry)
3: **while** $availableSpace > 0$ and $candidates \neq \emptyset$ **do**
4: Compute a score for each node in $candidates$
5: $maxNode \leftarrow$ candidate with largest score
6: Allocate memory to $maxNode$ and remove from $candidates$
7: Reduce $availableSpace$ by the size of $maxNode$'s context

preferable for the cache allocation algorithm to run very quickly, even if the resulting memory allocation is not optimal.

3.1 Greedy Memory Allocation Algorithm

This section proposes a greedy method for allocating memory, which runs in quadratic time. The memory allocations produced by this method are not guaranteed to be optimal, yet, it is shown experimentally in Section 4 that they appear to be near–optimal on the networks we experimented with.

The greedy method starts with no memory allocated to any of the dgraph caches. It then chooses (one at a time) a dgraph node t and allocates memory m (the size of t's cache) to t. Suppose that c_1 is the number of recursive calls made by RC before memory is allocated to the cache at node t. Suppose further that c_2 is the number of recursive calls made by RC after memory has been allocated to the cache at t ($c_2 \leq c_1$). The node t will be chosen greedily in order to maximize $(c_1 - c_2)/m$: the number of reduced calls per memory unit.

The pseudocode for this method is shown in Algorithm 4. The while–loop will execute $O(n)$ times, where n is the number of dgraph internal nodes. Note that Equation 2 can be evaluated for all nodes in $O(n)$ time, which gives us the total number of recursive calls made by RC under any cache factor cf. Hence, the score of each dgraph node t can be computed in $O(n)$ time by simply evaluating Equation 2 twice for all nodes: once while caching at t, and another time without caching at t. Under this method, Line 4 will take $O(n^2)$ time, leading to a total time complexity of $O(n^3)$.

We will now show, however, how to compute the scores of all candidates in only $O(n)$ time, leading to a total complexity of $O(n^2)$. The key idea is to maintain for each node t in the dgraph two auxiliary scores under the current cache factor cf. First, $calls^{cf}(t)$, which is the number of calls made to node t (under cache factor cf). Second, $cpc^{cf}(t)$, which is the number of calls made to descendants of t for each call made to t (inclusive of that call, under cache factor cf). We then have:

Theorem 2 *Let cf_2 be a cache factor which results from caching at node t in cache factor cf_1, and let m be the size of cache at node t. Let c_1 and c_2 be the total number of recursive calls made by* RC *under cache factor cf_1 and cf_2 respectively. The score of node t under cache factor cf_1 is then $score^{cf_1}(t) = \frac{c_1 - c_2}{m}$. Moreover, we have:*

$$score^{cf_1}(t) = [\mathsf{cutset}^{\#}(t) * calls^{cf_1}(t) - \mathsf{cutset}^{\#}(t) * \mathsf{context}^{\#}(t)] * [cpc^{cf_1}(t^l) + cpc^{cf_1}(t^r)]/[\mathsf{context}^{\#}(t)]. \quad (3)$$

Therefore, if we have $calls$ and cpc for each node in the dgraph, we can obtain the scores for all dgraph nodes in $O(n)$ time. We will now show that all such scores can be initialized in $O(n)$ time and that they can also all be updated in $O(n)$ time after caching at a particular node.

To initialize $calls$ for each node, we traverse the dgraph such that parent nodes are visited prior to their children. At each node t, we compute $calls(t)$ using Equation 2, which can be computed in constant time since the number of calls to each parent is already known. Now to initialize cpc for each node we reverse the order and visit children prior to their parents. At each node t, compute $cpc(t)$ using:

$$cpc(t) = 1 + \mathsf{cutset}(t)^{\#} * (cpc(t_l) + cpc(t_r)). \quad (4)$$

To update these numbers after caching at node t, we note that the only change in the cache factor occurs at t. Therefore, the only affected nodes are t's ancestors and descendants. Hence, the $scores$ and $calls$ for descendants of t and the $scores$ and cpc for ancestors of t need to be updated using Equations 2 and 4.

Some internal nodes have dead caches, which are never used (Allen and Darwiche, 2003a). The cached nodes are chosen only from those with useful caches, which can sometimes be significantly less than the total number of internal nodes. The outer loop of the greedy algorithm is actually linear only in the number of nodes cached. Therefore, a

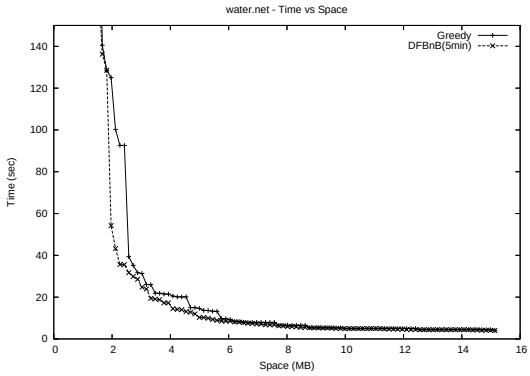

Figure 3: Time–space tradeoff on water.

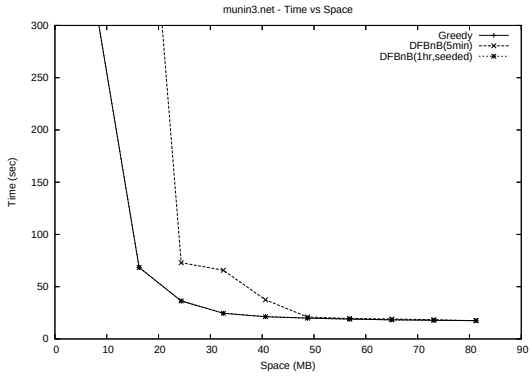

Figure 4: Time–space tradeoff on munin3.

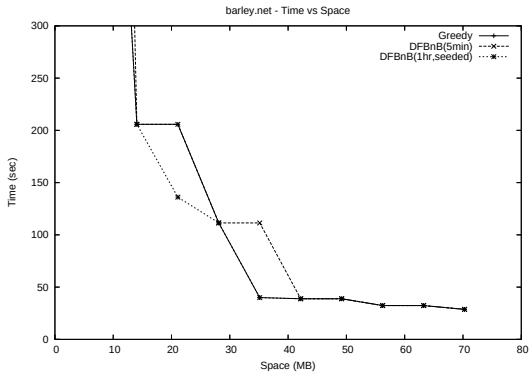

Figure 5: Time–space tradeoff on barley.

more refined complexity analysis would be $O(nm)$ where n is still the number of internal nodes in the dgraph and m is the number of cached nodes.

4 Time–Space Tradeoff

The main goal of this section is to present empirical evidence that the greedy memory allocation algorithm produces near–optimal results efficiently, making it significant for practical applications. Additionally, the branch-and-bound search is initialized with a seed, and since the greedy algorithm finds better results quicker, its results make good seed values. Experiments were run on networks from the Bayesian network repository (http://www.cs.huji.ac.il/labs/compbio/Repository/) and on genetic networks from the bioinformatics domain (Ott, 1999).[3]

As mentioned in Section 3, the number of recursive calls which will be made by RC in the worst case can easily be computed for any cache factor. This number is proportional to the running time of the algorithm.[4] These timings represent the worst case timings of RC, as RC will do less work given evidence.

For the networks from the repository, we generated a dgraph capable of computing marginals over all individual variables and then compared the cache allocation algorithms on this dgraph. We then plotted the results, where "Greedy" is the new greedy heuristic, "DFBnB(5min)" is the depth-first branch-and-bound algorithm run for 5 minutes (Allen and Darwiche, 2004), and "DFBnB(1hr,seeded)" is the search algorithm when seeded with the greedy result and run for up to 1 hour. Each data point on the plot shows how long it would take RC to run, based on the corresponding amount of memory (i.e. each data point is a solution to the cache allocation problem). All the DFBnB searches were also seeded with the result from the next lowest memory usage, as a solution using less memory would also be a solution to the problem with more memory.

Figures 3, 4, and 5 show these plots on the water, munin3, and barley networks. On all networks, the allocation problem was solved for 11 different memory constraints, with the exception of the water network where it was done for 101 constraints. There are two things to note. The first is shown on Figure 3, where the DFBnB finishes in under 6 seconds. Since it finished, the DFBnB curve is optimal

[3]The networks used are a subset of those used in (Fishelson and Geiger, 2002)

[4]On our 2.4 GHz Linux system, the JAVA implementation of RC executes approximately 6 million recursive calls per second. Therefore, our time–space tradeoff curves will use this constant to convert the number of calls into inference timings for each network. The bioinformatics networks require a special logarithmic version of RC and therefore the constant of 500,000 is used for them (Allen and Darwiche, 2003a).

Table 1: Timing of the greedy allocation algorithm on networks from the repository.

Network	Dgraph nodes	Useful caches	Timing (seconds) Max	Avg
Barley	190	77	0.04	0.02
Diabetes	1788	780	2.32	1.74
Link	2993	1268	7.86	5.72
Mildew	145	58	0.04	0.02
Munin1	775	332	0.43	0.29
Munin2	4158	1781	14.52	10.03
Munin3	4355	1885	17.57	13.56
Munin4	4285	1852	15.77	12.15
Pigs	1837	749	2.71	1.97
Water	121	41	0.03	0.01

and hence we don't need to run a 1 hour version. By comparing this optimal curve with the greedy curve we can see that the greedy results are not necessarily optimal, however they are close to the optimal curve. The second thing to note from these graphs is on Figure 4, where we see that the greedy algorithm found better values than the results when DFBnB was run for 5 minutes. Then when DFBnB was seeded with the greedy result, for each constraint, it was given an hour to find something better and was unable to improve on them.

Out of the 10 networks in the repository, when the DFBnB search was seeded with the greedy algorithm's result, it was unable to improve any of the results on 6 of the networks. On the other four, two of which are shown in Figures 3 and 5, the greedy still is close to the best cache allocation found.

Table 1 shows how long it took the greedy algorithm to calculate its memory allocation and the size of the corresponding dgraph. The number of useful caches is also reported, as this is a more accurate representation of the search space, as some nodes' caches are dead and should not be cached (Allen and Darwiche, 2003a). The table reports the maximum and average timings over the 11 different memory constraints. It shows that on most networks the greedy algorithm runs in just a few seconds and in that time it can be seen on the graphs that in many cases it finds better memory allocations than the DFBnB algorithm could in 5 minutes. Therefore, this shows that the greedy algorithm can be useful as a seed for the search, since it runs very fast and finds significantly better allocations quicker.

Some additional experiments were run on networks from the bioinformatics domain, which are used for doing genetic linkage analysis. These networks contain large numbers of variables and therefore have very large dtrees (Allen and Darwiche, 2003a). Genetic linkage analysis only requires $Pr(e)$ calculations, therefore these experiments were done on dtrees, instead of dgraphs. These dtrees still require significant amounts of memory, which has prompted another line of research into time–space tradeoffs (Fishelson and Geiger, 2002; Fishelson and Geiger, 2003). This other research attempts to run the variable elimination algorithm (Dechter, 1996) until it begins to run out of memory and then switches to conditioning (Pearl, 1988). As the main interest in the genetic linkage analysis domain is in running the algorithm without exhausting the system resources, no time–space tradeoff analysis has been published for differing amounts of memory.

Even on these very large networks, the greedy heuristic can run in a matter of seconds and still produce better results than DFBnB only. The timings of the greedy heuristic on the networks are shown in Table 2. For example, network EA11 has a dtree with $18,053$ nodes and the algorithm is still able

Table 2: Timing of the greedy allocation algorithm on the bioinformatics networks.

Network	Dtree nodes	Useful caches	Greedy Timing (secs) Max	Avg	RC Time (secs) $cf = 0.5$	$= 1.0$
EA5	4183	1063	0.88	0.49	0.4	0.3
EA6	4981	1263	1.01	0.77	0.9	0.6
EA7	5855	1512	1.76	1.08	2.2	1.8
EA8	7579	1883	2.37	1.80	5.7	4.1
EA9	15493	3763	9.20	7.81	8170.3	3439.1
EA10	15939	3879	10.21	8.22	2265.7	1762.4
EA11	18053	4391	13.62	10.78	3350.4	2377.3

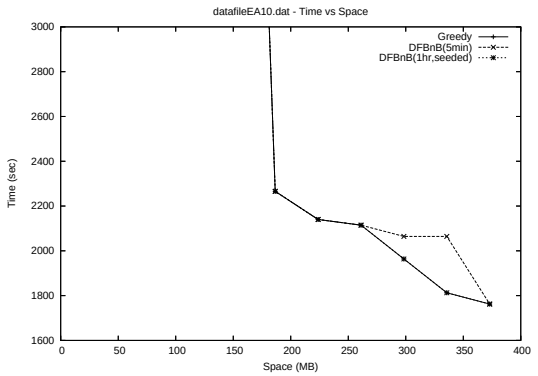

Figure 6: Time–space tradeoff on EA10.

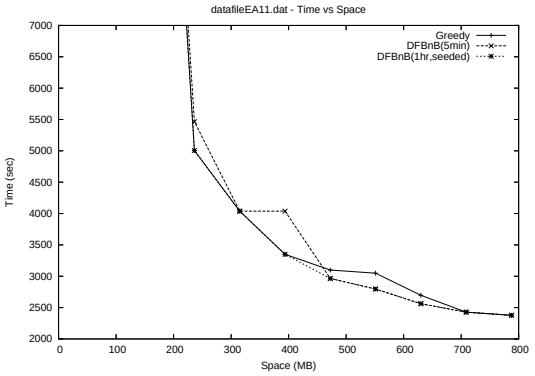

Figure 7: Time–space tradeoff on EA11.

to run in under 14 seconds. Two time–space tradeoff curves are shown in Figures 6 and 7. It can be seen on EA10 that the DFBnB could not improve upon the greedy results given an hour, and in fact it proved 4 of the data points were optimal. EA10 also shows that without the greedy seed, the DFBnB in 5 minutes wasn't able to do as well as the greedy did in under 11 seconds. On EA11, the one hour search only improved upon 3 of the greedy data points, while it proved 4 of the other greedy solutions were optimal.

Previously these genetic networks were too large to efficiently run the DFBnB algorithm on, however using the greedy cache allocation scheme opens these networks up to the domain of time-space tradeoff with RC. For example, Table 2 lists the amount of time RC would take using full caching and when this amount is reduced by half, when using the memory allocation from the greedy algorithm.[5] These results show that even on these large networks, RC can efficiently perform time–space tradeoff, which was previously not possible due to the memory allocation search.

5 Conclusions

We proposed in this paper a greedy algorithm for memory allocation within the framework of inference by recursive conditioning. The algorithm has quadratic time worst-case complexity, and appears to produce near–optimal results efficiently on our networks. The proposed method appears to be quite significant for the practice of time–space tradeoff in probabilistic reasoning.

The algorithms described in this paper have been implemented in the SAMIAM tool, which is available publicly (UCLA Automated Reasoning Group, http://reasoning.cs.ucla.edu/samiam).

[5] These timings each represent the exact time requirement in the worst-case of no evidence.

Acknowledgments

This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617. The work by James Park was done while at UCLA.

References

David Allen and Adnan Darwiche. 2003a. New advances in inference by recursive conditioning. In *UAI-2003*, pages 2–10.

David Allen and Adnan Darwiche. 2003b. Optimal time–space tradeoff in probabilistic inference. In *IJCAI-03*, pages 969–975.

David Allen and Adnan Darwiche, 2004. *Advances in Bayesian networks*, volume 146 of *Studies in Fuzziness and Soft Computing*, chapter Optimal Time–Space Tradeoff in Probabilistic Inference, pages 39–55. Springer–Verlag, New York.

Adnan Darwiche and Mark Hopkins. 2001. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *ECSQARU'01*, pages 180–191.

Adnan Darwiche. 2001. Recursive conditioning. *Artificial Intelligence*, 126:5–41, February.

Rina Dechter. 1996. Bucket elimination: A unifying framework for probabilistic inference. In *UAI-96*, pages 211–219.

Ma'ayan Fishelson and Dan Geiger. 2002. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(1):189–198.

Ma'ayan Fishelson and Dan Geiger. 2003. Optimizing exact genetic linkage computations. In *RECOMB'03*.

Jurg Ott. 1999. *Analysis of Human Genetic Linkage*. The Johns Hopkins University Press, Baltimore.

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Francisco.

UCLA Automated Reasoning Group. SamIam: Sensitivity Analysis, Modeling, Inference And More. http://reasoning.cs.ucla.edu/samiam.