Scaling Up MAP Search in Bayesian Networks Using External Memory

Heejin Lim, Changhe Yuan, and Eric A. Hansen
Department of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762

Abstract

State-of-the-art exact algorithms for solving the MAP problem in Bayesian networks use depth-first branch-and-bound search with bounds computed by evaluating a join tree. Although this approach is effective, it can fail if the join tree is too large to fit in RAM. We describe an external-memory MAP search algorithm that stores much of the join tree on disk, keeping the parts of the join tree in RAM that are needed to compute bounds for the current search nodes, and using heuristics to decide which parts of the join tree to write to disk when RAM is full. Preliminary results show that this approach improves the scalability of exact MAP search algorithms.

1 Introduction

State-of-the-art exact MAP algorithms for Bayesian networks use depth-first branch and bound (DF-BnB) search, and prune the search tree using bounds that are computed by evaluating a join tree (Park and Darwiche, 2003; Yuan and Hansen, 2009) or an arithmetic circuit (Huang et al., 2006). For large and complex Bayesian networks, however, the join tree or arithmetic circuit can be too large to fit in RAM, limiting scalability. In this paper, we focus on the approach that uses a join tree to compute bounds. The memory required to store the join tree is exponential in the treewidth of the Bayesian network.

We describe how to improve the scalability of a MAP search algorithm that evaluates a join tree to compute bounds by using external memory to store the join tree when it is too large to fit in RAM. The efficiency of this approach depends on the heuristics used to decide which parts of the join tree to write to disk when RAM is full. Our study shows that commonly used heuristics for external-memory algorithms, such as *least recently used* (LRU) and *least frequently used* (LFU), do not always perform well in MAP search. We introduce new heuristics that take into account the unique characteristics of the search algorithm. Preliminary results show that the approach can solve MAP problems that could not previously be solved due to memory limitations.

2 Background

We begin with a brief review of the MAP problem and algorithms for solving the MAP problem using branch-and-bound search.

2.1 The MAP problem

The Maximum a Posteriori assignment problem (MAP) is defined as follows. Let $\mathbf{M}$ be a set of *explanatory variables* in a Bayesian network; from now on, we call these the *MAP variables*. Let $\mathbf{E}$ be a set of *evidence variables* whose states have been observed. The remaining variables, denoted $\mathbf{S}$, are variables for which the states are unknown and not of interest. Given an assignment $\mathbf{e}$ for the variables $\mathbf{E}$, the MAP problem is to find an assignment $\mathbf{m}$ for the variables $\mathbf{M}$ that maximizes the joint probability $P(\mathbf{m}, \mathbf{e})$ (or, equivalently, the conditional probability $P(\mathbf{m}|\mathbf{e})$). Formally,

$$\hat{\mathbf{m}}_{MAP} = \arg\max_{\mathbf{M}} \sum_{\mathbf{S}} P(\mathbf{M}, \mathbf{S}, \mathbf{E} = \mathbf{e}) \ , \quad (1)$$

where $P(\mathbf{M}, \mathbf{S}, \mathbf{E} = \mathbf{e})$ is the joint probability distribution of the network given the assignment $\mathbf{e}$.

2.2 Join tree upper bound

In Equation (1), the maximization and summation operators are applied to different sets of variables. The MAP variables in $\mathbf{M}$ can be maximized in different orders, and the variables in $\mathbf{S}$ can be summed

out in different orders, without affecting the result. But the summations and maximizations are not commutable. As a result, variable elimination-based methods for solving MAP have a complexity that depends on the *constrained treewidth* of the network, and they are typically infeasible because they require too much memory.

If the ordering among the summations and maximizations is relaxed, however, an upper bound on the probability of a MAP solution is computed. The following theorem is due to Park and Darwiche (2003).

Theorem 1. *Let $\phi(\mathbf{M}, \mathbf{S}, \mathbf{Z})$ be a potential over the disjoint variable sets $\mathbf{M}$, $\mathbf{S}$, and $\mathbf{Z}$. For any instantiation $\mathbf{z}$ of $\mathbf{Z}$, the following inequality holds:*

$$\sum_S \max_M \phi(\mathbf{M}, \mathbf{S}, \mathbf{Z} = \mathbf{z}) \geq \max_M \sum_S \phi(\mathbf{M}, \mathbf{S}, \mathbf{Z} = \mathbf{z}) .$$

Based on this result, Park and Darwiche (2003) compute upper bounds for the MAP problem using the join tree algorithm, but with redefined messages. Each message is computed such that variables in $\mathbf{S}$ are summed over before MAP variables are maximized.

2.3 Solving MAP using DFBnB

Park and Darwiche (2003) use the join tree upper bound in a depth-first branch-and-bound (DFBnB) search algorithm to solve the MAP problem. Since a full evaluation of the join tree computes simultaneous upper bounds for all MAP variables, Park and Darwiche use dynamic variable ordering to speed up their search algorithm. Yuan and Hansen (2009) observe that, when a static variable ordering is used, it is only necessary to compute bounds for the next MAP variable to be instantiated at each step, and this only requires evaluating a small part of the join tree. They use a static ordering of MAP variables that is created from a post-order traversal of the MAP variables in the join tree. With this static ordering, the upper bounds needed for the next instantiating variable(s) in MAP search can be computed incrementally by message passing along a limited and fixed path in the join tree. During forward traversal of a branch of a search tree, it is only necessary to perform message passing once along this path in the join tree, broken up into separate steps for each instantiating variable. To allow efficient backtracking, the clique and separator potentials of the join tree that are changed during bounds computation are cached in the order that they are changed. During backtracking, the cached potentials can be used to efficiently restore the join tree to its previous state before one or more MAP variables were instantiated. The readers are referred to (Yuan and Hansen, 2009) for more details of the algorithm.

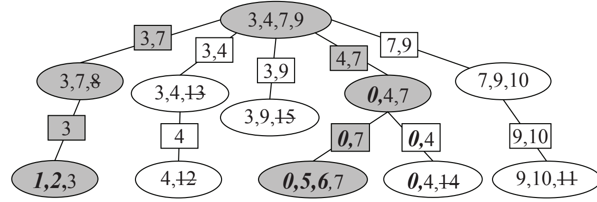

Figure 1: Example of a join tree for upper bound computation. The shaded nodes are nodes of the join tree that contain MAP variables.

We illustrate the idea with an example. Figure 1 shows the join tree of a Bayesian network based on the Hugin architecture (Jensen et al., 1990). The numbers are the indices of distinct variables in the network, and the numbers in bold-face italics represent the MAP variables, which are 0, 1, 2, 5, and 6. Let the static search ordering of the MAP variables be: 1,2,0,5,6. After MAP variables 1 and 2 are instantiated, their values are entered as evidence to clique $\{1, 2, 3\}$. Messages can then be sent to the other parts of the join tree to get upper bounds for the remaining MAP variables. However, since the next variable in the static ordering is 0, it is only necessary to send messages along the shaded path from clique $\{1, 2, 3\}$ to clique $\{0, 4, 7\}$. None of the other parts of the join tree need to be involved in the propagation. The path has a new set of clique and separator potentials as a result of the message propagation. The old potentials are cached before they are overwritten with the new potentials. If all the search nodes after instantiating variable 0 can be pruned using upper bounds, the search algorithm backtracks to the parent search node and retracts the join tree to the previous state. This can be achieved by simply restoring the cached potentials in reverse order and rolling back the changes.

3 External-memory MAP Search

The MAP search algorithms of Park and Darwiche (2003) and Yuan and Hansen (2009) use join tree evaluation to compute bounds for a depth-first branch-and-bound search. We use the Yuan and Hansen algorithm as the basis for our external-memory algorithm, in part because it has been shown to be more efficient than the Park and Darwiche algorithm when their internal-memory versions are compared, and in part because its incremental approach is easier to convert to an efficient external-memory algorithm. Each time the Park and Darwiche algorithm computes bounds for a node of the search tree, it must perform a full join tree evaluation; this could require copying the entire join tree into RAM for each search node, incurring a large amount of disk I/O. By contrast, each time the Yuan and Hansen algorithm computes bounds for a node of the search tree, it only needs to keep a small part of the join tree in RAM. As we will see, the locality that it exploits for incremental message propagation is also exploited by an external-memory version of the algorithm to minimize disk I/O.

3.1 Memory architecture

Figure 2 illustrates how our algorithm uses RAM and disk. We call the jointree without potentials the *skeleton*. The skeleton is typically small and always resides in RAM, as shown in Figure 2. Each clique of the skeleton has one *main pointer* that points to the clique's potential, which we call the *main potential*. The clique may also have one or more *cache pointers* that point to cached copies of the potential. The basic idea of the algorithm is to store some of the potentials in RAM and some on disk. When a potential is stored on disk, it is stored in a file named by a unique *tag* assigned to the potential. In Figure 2, arc 1 shows an example of a main pointer, and arc 2 shows an example of a cache pointer.

All computation on the jointree is based on the main potentials. The other potentials are only for caching purposes. A potential needed for computation should reside in RAM. When necessary, we can *write* a potential in RAM to a disk file named after the potential's tag, in order to free up space in RAM; we can also use a tag to find the appropriate disk file to *read* the potential back into RAM.

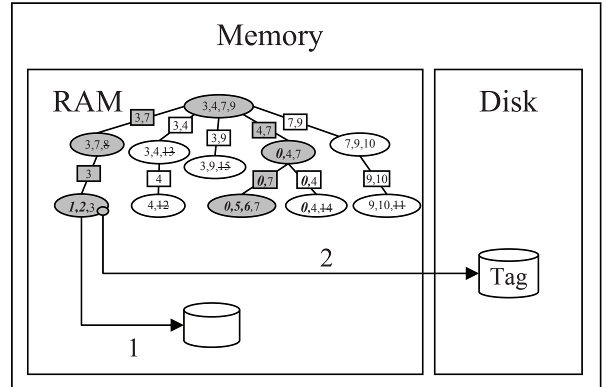

Figure 2: Memory architecture of the algorithm.

3.2 Initialize an upper-bound join tree

In (Yuan and Hansen, 2009), we initialize an upper-bound jointree for a Bayesian network completely in RAM. We first create its skeleton. We then initialize all the clique potentials with appropriate conditional probability tables of the Bayesian network. After entering evidence to the jointree, a full join tree propagation is performed to finish initializing the upper-bound join tree. A full join tree propagation involves two phases: *collect* and *distribute*. In the collect phase, all cliques send messages to their parents by combining messages sent from children cliques. After the root receives all messages from its children, the distribute phase starts, in which all cliques send messages to their children by combining messages from parents.

Several preprocessing methods can be used to reduce the size of an upper-bound jointree in order to delay the use of external memory. First, we use relevance reasoning (Lin and Druzdzel, 1997) to preprocess a Bayesian network based on the evidence and target variables of a MAP problem. This step reduces the size of the network by removing irrelevant variables such as barren variables. Second, we create an upper-bound join tree that is as small as possible without considering the quality of the bounds. Third, we can release clique potentials that are only needed during join tree initialization, and not during MAP search. By *releasing*, we mean that the potentials are deleted and their pointers are set to NULL. With incremental join tree bounds, only part of the join tree is involved in message propagation during MAP search. In Figure 1, the sec-

ond left-most branch has no MAP variables. Once a clique on this branch has sent a message to its parent, the clique can be released from memory. In fact, all non-shaded parts of the join tree in Figure 1 can be released before MAP search. Doing so not only postpones the use of external memory, it improves the efficiency of the algorithm, because there is no need to send messages to these parts of the join tree during the distribute phase.

Nevertheless, the final upper-bound join tree may still be too large to fit in RAM. In that case, we must construct the join tree incrementally and store parts of it on disk. We do so as follows. After constructing the skeleton, we initialize the jointree by *interleaving* potential initialization and message propagation via a *left-to-right, leaf-to-root* traversal of the jointree. For Figure 1, we would start by initializing the potential of clique $\{1,2,3\}$ in the left-most branch. We then initialize the potential for separator $\{3\}$ by calculating the message to be sent from $\{1,2,3\}$ to $\{3,7,8\}$. After that, we initialize the potential of clique $\{3,7,8\}$ and combine the message stored in separator $\{3\}$. To avoid exhausting RAM, we estimate the amount of additional RAM needed for each step of the algorithm and check if the increase is larger than the amount of available RAM. If there is not sufficient RAM available, we have to write to disk some cliques or separators that are already constructed and reside in RAM. In a later section of the paper, we introduce several heuristics for selecting which cliques and separators to write to disk. For now, it suffices to say that enough RAM will be freed so that the current step can be executed. We can use the above incremental scheme to complete the collect phase.

The collect phase traverses a join tree from left to right. After the collection phase finishes, the cliques stored on disk most likely come from the leftmost branches. Since the distribute phase may need to restore some clique potentials from disk to RAM, the distribute phase uses a *right-to-left, root-to-leaf* traversal of the join tree so as to use clique potentials that currently reside in RAM first. The leftmost branches are not read from disk until they are needed. After the distribute phase finishes, cliques from the rightmost branches are swapped out to disk. This improves the performance of MAP search because the search starts from the leftmost branches as well.

We may still be able to release part of the join tree during the distribute phase. When all of the MAP variables are in one branch of the join tree, we can release the root and its immediate successor cliques if they do not contain MAP variables.

Finally, calculating a message requires the potentials of at least one clique and one separator to be in RAM at the same time. For example, we need both clique $\{1,2,3\}$ and separator $\{3\}$ to be able to compute the message to be sent to $\{3,7,8\}$. Therefore, our method has a *minimal* memory requirement that is equal to the largest total size of any neighboring pair of clique and separator.

3.3 MAP search using external memory

Once the upper-bound join tree is initialized, we start the MAP search. At each search step, we need to use the join tree to compute the search bounds, which requires message propagation on the jointree. For each message propagation step, we check whether or not the potentials we need reside in RAM. If not, we check whether there is enough RAM available for reading them back in RAM from disk. If necessary, we use the heuristics described in Section 3.4 to select potentials to write from RAM to disk in order to free up enough RAM to continue. The strategy of using external memory is similar to the strategy used in the join tree initialization phase. There are, however, some important differences that warrant discussion.

For efficient backtracking, the Yuan and Hansen algorithm caches and restores potentials during the MAP search. The need for caching makes using external memory slightly more complicated. New strategies are needed, both during *forward search* and *backtracking*.

During *forward search*, we need to cache a potential before we set the state of a newly instantiated MAP variable as evidence to a clique potential, and before we update a clique or separator potential using incoming messages. There are two possibilities. One is that the potential to be cached is in RAM. If enough additional RAM is available, we make a copy in RAM immediately. If not, we free up space by writing some cliques from RAM to disk before making the copy in RAM. The second possibility is that the potential is on disk. Since we need the po-

tential in the next operation, we read a copy of the potential from disk to RAM and swap the main and cache pointers so that the main pointer points to the copy in RAM.

When *backtracking*, we need to restore the join tree to a previous state by restoring some cached potentials. A potential and its cached copy can be in RAM and/or disk. No matter where they are, we simply delete the current main potential and redirect the main pointer to point to the cached copy. Therefore, no disk I/O is needed during backtracking. We only read potentials from disk to RAM when they are needed during forward search.

The strategies described above allow a potential that is once written to disk to remain on disk and to be repeatedly used until it is not needed anymore. This helps to limit the number of times the same potential is written to disk or read from disk.

3.4 Heuristics

It is critical for our MAP algorithm to have a good heuristic for selecting which cliques and separators to write to disk when RAM is full, in order to keep the amount of disk I/O as low as possible.

Commonly-used heuristics include storing the *least recently used* (LRU) or the *least frequently used* (LFU) data in external memory. We implemented both and found that the LRU heuristic outperforms the LFU heuristic because message propagation follows a fixed post-traversal order of the join tree. We also tested two other heuristics. One heuristic that we call *largest first* (LF) selects the largest cliques to store in external memory. The other heuristic that we call *largest but least frequently used* (LLF) integrates the LF and LFU heuristics. LLF selects the largest cliques, but decreases the priority of a clique if it is selected too often. More formally, we use the following formula to order the candidate cliques,

$$priority = \frac{size}{\#I/O}, \tag{2}$$

where $size$ is the size of a clique and $\#I/O$ is the number of times the clique is written to disk or read from disk. The clique that has the largest $priority$ value is selected as the next clique to write to disk.

4 Empirical evaluation

We tested our external-memory algorithm (DFBnB+EM) by comparing it to the original internal-memory MAP algorithm (DFBnB) developed by Yuan and Hansen (2009). Experiments were performed on a 2.1GHz processor with 4GB of RAM running a 32-bit version of Windows Vista. The user is only allowed to use 2GB of the memory address in a 32-bit environment. The magnetic disk used for external memory operates at 5400RPM and its interface is SATA2 (Serial Advanced Technology Attachment).

4.1 Benchmarks and experimental design

Algorithm performance was tested on a set of benchmark Bayesian networks. Two of the networks, BN-43 and BN-44, are from the UAI-06 inference competition. For these two networks, the MAP problem cannot be solved without using disk. This shows that the new algorithm increases the range of problems for which the MAP problem can be solved exactly.

The other networks have been previously used to test the internal-memory version of the MAP search algorithm. They allow us to compare the performance of the internal-memory MAP-search algorithm to the external-memory algorithm under artificial memory limits. For each Bayesian network, we set the *low* memory limit based on the minimal memory requirement for the join tree of the network. We set the *high* memory limit by averaging the low memory limit and the total memory used by the internal-memory MAP search algorithm. For the BN-43 and BN-44 networks, we set the low memory limit in the same way but set the high memory limit to be all available RAM.

Since the artificial memory limits are set based on the join tree size, they do not consider other memory requirements of the MAP algorithm, such as memory incurred during search. Even when the initial join tree fits in physical memory, caching potentials during MAP search can significantly increase the amount of memory needed. Yuan and Hansen (2009) found that the increase ranges from slightly larger to several times larger. Therefore, MAP search may still fail if external memory is not used. Finally, the Windows OS typically allocates

Network	DFBnB							
	#Nodes	#Backtracks	Build(ms)	Search(ms)	Largest clique	Jointree	Avg memory	Max memory
Hailfinder	9,902	61,721	15	457	26K	98K	231K	231K
Water	5	14	1,683	372	41M	70M	126M	140M
Munin4	195	104	57,332	567	8M	155M	180M	408M
Barley	282	2,524	8,949	62,858	100M	230M	345M	475M
Mildew	1,135	6,102	5,490	13,684	43M	104M	166M	167M
Andes	124,971	975,861	743	80,522	2M	5M	14M	14M
Pigs	75,627	660,986	2,313	150,206	4M	11M	32M	33M
Diabetes	11,783	161,262	7,629	204,837	2M	89M	144M	150M
BN-43	5,520	54,700	-	-	512M	1,008M	-	-
BN-44	1,292	16,276	-	-	258M	890M	-	-

Table 1: Results for the DFBnB algorithm of Yuan and Hansen (2009). '#Nodes' is the number of search nodes. '#Backtracks' is the number of backtracking steps. 'Largest clique' is the largest clique size, and 'Jointree' is the jointree size. 'Avg memory' is the average total memory of the test cases, while 'Max memory' is the maximum total memory. 'K' means kilobytes, and 'M' megabytes. Since the internal-memory algorithm could not solve the MAP problem for networks BN-43 and BN-44, the partial results in the table are from the external-memory algorithm.

a significant portion of memory to service applications. Therefore, not all RAM is available for use by the MAP algorithm.

For each of the benchmark Bayesian networks, we generated 10 random test cases with as many root nodes as MAP variables and with all leaf nodes as evidence variables so that they are solvable within reasonable time. Tables 1 and 2 report the average results for these test cases.

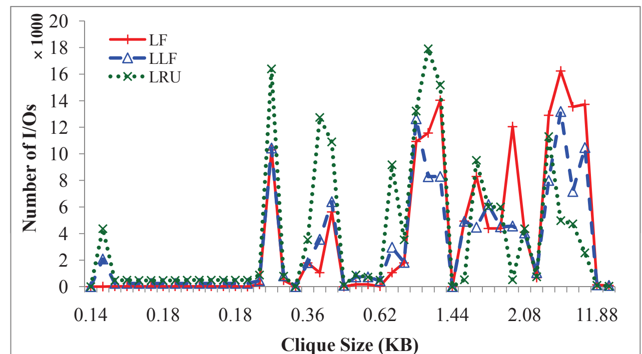

Figure 3: Number of times chosen for I/O (in thousands) versus clique size for different heuristics on the Hailfinder network.

4.2 Analysis of results

Table 1 shows the results for the DFBnB algorithm and Table 2 shows the results for the DFBnB+EM algorithm. However, note that the numbers of search nodes (#Nodes) and backtracks (#Backtracks) shown in Table 1, as well as the size of the largest clique and the size of the join tree, are the same for both algorithms.

Unsurprisingly, there is more disk I/O (both reads and writes) when there is more backtracking. But interestingly, the amount of data read from disk to RAM is often larger than the amount of data written from RAM to disk. During backtracking, we need to restore some cached potentials. If they are in external memory, we do not need to immediately read them from disk to RAM; we just pass their tags to the potential pointers. When forward search resumes, copies are then read from disk to RAM for message propagation, and main and cache pointers are swapped so that the cache pointer points to the external-memory copy. No write is necessary during this process, unless memory is running low. This is desirable because, for the same amount of data, writing to disk takes more time than reading from disk. This can be explained as follows. When there is not much backtracking, however, as for the Munin network, it is possible for more data to be written to disk than read from disk.

Table 2 compares the performance of the three heuristics described in Section 3.4: LRU, LF, and LLF. The results indicate that LRU typically incurs the least amount of I/O. A join tree typically has smaller leaf cliques and larger inner cliques, and inner cliques are accessed more often during back-

Network	DFBnB+EM							
	Memory limit	Heuristics	Jointree building			MAP search		
			Time	Write	Read	Time	Write	Read
Hailfinder	40K	LF	836	303K	227K	147,730	58M	97M
		LLF	881	303K	227K	**91,539**	33M	73M
		LRU	1,218	302K	209K	156,263	**9M**	**62M**
	135K	LF	16	0K	0K	**900**	507K	**8M**
		LLF	19	0K	0K	1,915	298K	20M
		LRU	19	0K	0K	3,293	**267K**	27M
Water	55M	LF	2,951	180M	164M	**1,767**	**223M**	**234M**
		LLF	2,972	180M	164M	2,199	**223M**	**234M**
		LRU	3,187	186M	168M	2,034	237M	240M
	90M	LF	1,758	0M	0M	**957**	93M	**93M**
		LLF	1,766	0M	0M	1,108	**62M**	132M
		LRU	1,778	0M	0M	1,296	91M	151M
Munin4	10M	LF	67,879	697M	534M	**2,940**	334M	333M
		LLF	69,180	697M	534M	8,311	333M	332M
		LRU	81,317	570M	410M	20,562	**239M**	**234M**
	95M	LF	59,037	186M	123M	2,305	102M	93M
		LLF	59,870	248M	180M	**2,138**	114M	98M
		LRU	74,017	281M	206M	10,581	**84M**	**66M**
Barley	150M	LF	11,906	363M	293M	266,415	24G	**30G**
		LLF	11,605	363M	293M	**259,678**	**22G**	**30G**
		LRU	11,522	313M	275M	291,873	24G	31G
	250M	LF	8,859	0M	0M	229,783	21G	27G
		LLF	8,923	0M	0M	113,009	6G	17G
		LRU	8,869	0M	0M	**88,168**	**249M**	**15G**
Mildew	50M	LF	8,065	335M	272M	46,589	5G	7G
		LLF	7,859	335M	272M	47,537	5G	7G
		LRU	8,331	344M	280M	**40,764**	**2G**	**4G**
	110M	LF	5,643	0M	0M	16,395	**55M**	2G
		LLF	5,588	0M	0M	**16,373**	**55M**	2G
		LRU	5,680	0M	0M	18,595	66M	**606M**
Andes	5M	LF	782	2M	1M	139,772	3G	12G
		LLF	775	1M	1M	**120,408**	**977M**	9G
		LRU	818	0M	0M	362,868	1G	**8G**
	10M	LF	772	0M	0M	**84,679**	319M	**1G**
		LLF	757	0M	0M	87,456	**13M**	5G
		LRU	765	0M	0M	175,501	51M	7G
Pigs	10M	LF	2,528	10M	8M	877,444	77G	86G
		LLF	2,558	15M	12M	**399,216**	23G	47G
		LRU	6,251	9M	6M	1,866,499	**8G**	**42G**
	20M	LF	2,384	0M	0M	**184,497**	523M	29G
		LLF	2,363	0M	0M	197,263	**368M**	27G
		LRU	2,443	0M	0M	1,163,290	3G	**23G**
Diabetes	5M	LF	10,821	155M	115M	3,445,326	115G	124G
		LLF	11,213	159M	119M	**2,275,228**	62G	77G
		LRU	11,475	146M	106M	2,661,619	**44G**	**62G**
	75M	LF	7,694	0M	0M	238,486	147M	26G
		LLF	7,680	0M	0M	238,031	51M	30G
		LRU	7,682	0M	0M	**227,471**	**34M**	**20G**
BN-43	600M	LF	81,276	4G	4G	1,760,086	41G	46G
		LLF	82,602	4G	4G	1,804,655	41G	46G
		LRU	71,151	3G	3G	**1,114,261**	**24G**	**29G**
	Available	LF	13,962	0M	0M	514,972	10G	20G
		LLF	13,775	0M	0M	481,041	**5G**	16G
		LRU	13,962	0M	0M	**451,230**	**5G**	**15G**
BN-44	350M	LF	70,731	4G	3G	3,513,469	251G	254G
		LLF	69,006	4G	3G	3,472,347	239G	245G
		LRU	77,144	3G	3G	**3,219,000**	**208G**	**214G**
	Available	LF	18,426	0M	0M	537,318	16G	64G
		LLF	17,984	0M	0M	**407,260**	**1G**	**50G**
		LRU	18,225	0M	0M	510,685	12G	56G

Table 2: Results of external-memory MAP search. Running time is measured in milliseconds. 'K' means kilobytes, 'M' means megabytes, and 'G' means gigabytes.

tracking search. Therefore, LRU tends to select smaller cliques to write to disk. By contrast, LF and LLF select larger cliques to write to disk. This result is clearer in Figure 3, which plots the number of disk I/O operations versus clique size in solving the MAP problem for the Hailfinder network. The two largest cliques are not selected often by LF and LLF because they are stored in external memory early in the search, and they remain there most of the time. They are only occasionally copied from disk back to RAM during backtracking.

However, the amount of disk I/O is not directly proportional to running time. Although LRU has the least amount of I/O and is faster than LF and LLF on some networks, it can be much slower on some other networks, such as Munin4 and Pigs. One reason for this is the small cliques chosen by LRU may not completely fill the I/O buffers, leading to more I/O operations. If all cliques are large, as they are for BN-43 and BN-44, LRU may perform better.

The LF heuristic may repeatedly select the same large cliques to write to disk because it only considers clique size. The LLF heuristic decreases the frequency with which the same large cliques are selected by prioritizing cliques based on dividing clique size by the number of times the clique has been written to or read from disk. Results show that when there is a lot of backtracking and not much RAM available, LLF can be significantly faster than LF. This is illustrated by the results for Hailfinder and Barley. Otherwise, LF and LLF demonstrate similar behavior in most cases.

We can use the ratio of the size of the join tree to the size of the largest clique to estimate the improvement in scalability from using our external-memory algorithm. For the benchmark networks, the ratio ranges from several times to around 45 times. The ratio is even higher if we take into account all memory used by the original MAP algorithm, since caching potentials can increase the size of a join tree.

5 Conclusion

We have introduced an external-memory approach to scaling up a depth-first branch-and-bound algorithm for solving the MAP problem that uses incremental join tree bounds (Yuan and Hansen, 2009). Our results show that the external-memory approach improves scalability and allows MAP problems to be solved exactly that could not be solved before due to memory limitations.

The minimum memory requirement of our algorithm is the amount of memory needed to store any neighboring pair of clique and separator. We plan to address this limitation by allowing just part of a clique to be stored in RAM while the rest is stored on disk (Kask et al., 2010). We will try to develop better heuristics for selecting potentials to write to external memory. Although we already use relevance reasoning to exploit evidence-based independence, we will consider whether the lazy propagation architecture proposed in (Madsen and Jensen, 1999) can improve the efficiency of our algorithm.

Acknowledgments This research was supported by NSF grants IIS-0953723, EPS-0903787, and IIS-0812558.

References

Jinbo Huang, Mark Chavira, and Adnan Darwiche. 2006. Solving map exactly by searching on compiled arithmetic circuits. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, page 143148.

Finn V. Jensen, Steffen L. Lauritzen, and Kristian G. Olesen. 1990. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282.

Kalev Kask, Rina Dechter, and Andrew Gelfand. 2010. BEEM: Bucket elimination with external memory. In *Proceedings of The 26th Conference on Uncertainty in Artificial Intelligence (UAI-10)*, AUAI Press Corvallis, Oregon.

Yan Lin and Marek J. Druzdzel. 1997. Computational advantages of relevance reasoning in Bayesian belief networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 342–350, San Francisco, CA. Morgan Kaufmann Publishers, Inc.

Anders L. Madsen and Finn V. Jensen. 1999. Lazy propagation: A junction tree inference algorithm based on lazy evaluation. *Artificial Intelligence*, 113(1-2):203–245.

James D. Park and Adnan Darwiche. 2003. Solving MAP exactly using systematic search. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI–03)*, pages 459–468, Morgan Kaufmann Publishers San Francisco, California.

Changhe Yuan and Eric A. Hansen. 2009. Efficient computation of jointree bounds for systematic MAP search. In *Proceedings of 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1982–1989, Pasadena, CA.