Performing Incremental Bayesian Inference by Dynamic Model Counting

Wei Li and **Peter van Beek** and **Pascal Poupart**
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{w22li, vanbeek, ppoupart}@cs.uwaterloo.ca

Abstract

The ability to update the structure of a Bayesian network when new data becomes available is crucial for building adaptive systems. Recent work by Sang, Beame, and Kautz (AAAI 2005) demonstrates that the well-known Davis-Putnam procedure combined with a dynamic decomposition and caching technique is an effective method for exact inference in Bayesian networks with high density and width. In this paper, we define dynamic model counting and extend the dynamic decomposition and caching technique to multiple runs on a series of problems with similar structure. This allows us to perform Bayesian inference incrementally as the structure of the network changes. Experimental results show that our approach yields significant improvements over the previous model counting approaches on multiple challenging Bayesian network instances.

Introduction

Many real world Bayesian network applications need to update their networks incrementally as new data becomes available. For example, the capability of updating a Bayesian network is crucial for building adaptive systems. Many methods for refining a network have been proposed, both for improving the conditional probability parameters and for improving the structure of the network (see, e.g., (Buntine 1991; Lam & Bacchus 1994; Friedman & Goldszmidt 1997) and references therein). However, little attention has been directed toward improving the efficiency of exact inference in incrementally updated Bayesian networks.

Most methods for exact inference in Bayesian networks do not take advantage of previous computations when solving an incrementally updated Bayesian network. Methods based on join trees can efficiently update the conditional probability parameters for a fixed network structure. However, when the network structure changes, the join tree usually must be reconstructed from scratch. Recent work by Sang, Beame, & Kautz (2005) demonstrates that the well-known Davis-Putnam procedure combined with a dynamic decomposition and caching technique is an effective method for exact inference in computationally challenging Bayesian networks.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

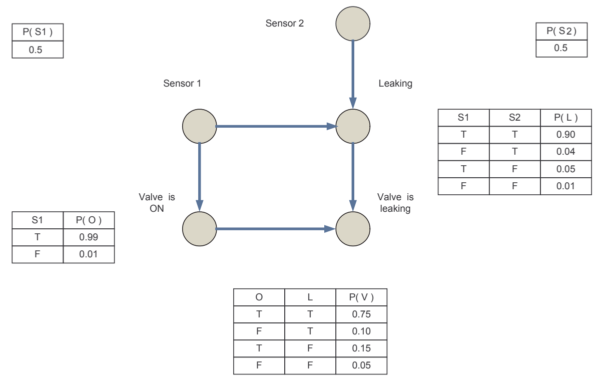
In this paper, we propose dynamic model counting (DynaMC). The philosophy behind dynamic model counting is intuitive. Bayesian networks can be efficiently compiled into CNFs (Darwiche 2002; Sang, Beame, & Kautz 2005). When two CNF formulas share many common clauses, it is possible to use the knowledge learned while counting the solutions of the first formula to simplify the solving and counting of the second formula. We show that component caching (good learning) can significantly improve Bayesian inference and other encoded dynamic model counting problems. The updating of Bayesian networks, such as adding an edge or deleting an edge, are presented as a sequence of regular model counting problems. We extend component (good) caching (Bacchus, Dalmao, & Pitassi 2003) to different runs of a series of changing problems. In each run, the previous problem evolves through local changes, and the number of models of the problem can be re-counted quickly based on previous results after each modification.

In this paper, our focus is to improve the efficiency of exact inference when the network structure is updated. We introduce DynaMC and perform experiments on real world problems to see how state-of-the art model counting algorithms handle DynaMC problems. The experiments show that our approach can solve various real world problems and improve the performance significantly.

Related Work

Darwiche (1998) proposes dynamic join trees, but in his framework the network structure is fixed and the query changes over time. Flores, Gámez, & Olesen (2003) propose the incremental compilation of a join tree in the case where the structure of the network changes. Their idea is to identify the parts of the join tree that are affected by the changes and only reconstruct those parts. However, join tree algorithms do not perform well on Bayesian networks with high density and large width. Hence, our interest in dynamic model counting approaches.

Dynamic versions of CSPs (Dechter & Dechter 1988; Schiex & Verfaillie 1994) and SAT (Hoos & O'Neill 2000; Jin & Somenzi 2005) have been proposed. However, the focus of previous work has been on the satisfiability problem—finding one solution or showing that no solution exists—and the learned knowledge is expressed in terms of conflict clauses (or nogoods). In nogood learn-



P(S1)
0.5

P(S2)
0.5

S1	S2	P(L)
T	T	0.90
F	T	0.04
T	F	0.05
F	F	0.01

S1	P(O)
T	0.99
F	0.01

O	L	P(V)
T	T	0.75
F	T	0.10
T	F	0.15
F	F	0.05

Figure 1: Oil tank monitoring system

ing, a constraint that captures the reason for a failure is computed from every failed search path. Both component caching and nogood learning collect information during search. However, it has been shown experimentally that good recording can be more effective than nogood recording for modeling counting (Bacchus, Dalmao, & Pitassi 2003; Sang *et al.* 2004). In our work, to maintain the partial solutions of the problems having similar structure, we generalize the caching (good learning) to multiple runs.

Background

The conventional model counting problem (#SAT) asks, given a Boolean formula F in CNF, how many of its assignments are satisfying? There are natural polynomial-time reductions between the Bayesian inference problem and model counting problems (Bacchus, Dalmao, & Pitassi 2003). Darwiche has described a method for compiling Bayesian networks as a set of multi-linear functions (Darwiche 2003). Recent work on compiling Bayesian networks reduces the problem to CNFs (Darwiche 2002; Littman 1999; Sang, Beame, & Kautz 2005). This allows advances in SAT solving techniques—such as non-chronological backtracking, clause learning and efficient variable selection heuristics—to be used.

A Bayesian network can be described as a weighted model counting problem (Sang, Beame, & Kautz 2005).

Definition 1 *A (static) weighted model counting problem consists of a CNF formula F with two kinds of variables: chance variables V_c and state variables V_s. A clause C in F is a propositional sentence over V_s and V_c. Both chance variables and state variables have weights, which satisfy the following constraints:*

$$weight(+v_c) + weight(-v_c) = 1$$
$$weight(+v_s) = weight(-v_s) = 1$$

The weight *of a single satisfying assignment is the product of the weights of the literals in that assignment. A* solution *of a weighted model counting problem is the sum of the weights of all satisfying assignments.*

We use the following example to show the method of encoding a Bayesian network into a weighted model counting

State Variables: O, L, V
Chance Variables:
$s_1(0.5), s_2(0.5), o_1(0.99), o_0(0.01), l_{11}(0.9), l_{01}(0.04), l_{10}(0.05), l_{00}(0.01), v_{11}(0.75), v_{01}(0.1), v_{10}(0.15), v_{00}(0.05)$
Node O Clauses:
$(-s_1, -o_1, O)(-s_1, o_1, -O)(s_1, -o_0, O)(s_1, o_0, -O)$
Node L Clauses:
$(-s_1, -s_2, -l_{11}, O)(-s_1, -s_2, l_{11}, -O)(s_1, -s_2, -l_{01}, O)$
$(s_1, -s_2, l_{01}, -O)(s_1, s_2, -l_{00}, O)(s_1, s_2, l_{00}, -O)$
$(-s_1, s_2, -l_{10}, O)(-s_1, s_2, l_{10}, -O)$
Node V Clauses:
$(-O, -L, -v_{11}, V)(-O, -L, v_{11}, -V)(O, -L, -v_{01}, V)$
$(O, -L, v_{01}, -V)(O, L, -v_{00}, V)(O, L, v_{00}, -V)$
$(-O, L, -v_{10}, V)(-O, L, v_{10}, -V)$

Figure 2: Variables and clauses for oil tank monitoring

problem. In an oil industry application, two sensors are deployed to monitor an oil tank. The Bayesian network in Figure 1 corresponds to the weighted model counting problem in Figure 2 (see (Sang, Beame, & Kautz 2005) for more information about the encoding).

Dynamic Model Counting

We define conventional model counting problems as static model counting problems.

Definition 2 *A dynamic model counting problem (DynaMC) is a sequence $M_0, M_1, M_2, \ldots,$ of conventional model counting (MC) problems, each one resulting from a change in the preceding one. As a result of such incremental change, the number of solutions of the problem may decrease (in which case it is considered a restriction) or increase (i.e., a relaxation).*

Solving a DynaMC problem consists of sequentially computing the number of models for each of the MCs $M_0, M_1, M_2, \ldots$. A naive approach is to successively apply an existing static model counting algorithm to each MC. Compared with updating model counts from scratch, a more efficient solution is to maintain the model counts for the subproblems of the previous MCs so that we only re-compute the part affected by the insertion or deletion of constraints. The hope is that the number of assignments that satisfy a component in formula M_{i-1} can be used to solve a new formula M_i, which has the same component.

Component (Good) Caching

The idea of recording good domain values was first applied to tree-structured problems (Bayardo & Miranker 1996). This algorithm keeps track of domain values that were previously used to completely instantiate an entire subtree of the problem. By skipping over the variables in these subtrees, good recording avoids solving the same subproblems multiple times. Bayardo and Miranker demonstrated that good recording improves the runtime complexity of tree-structured problems.

To avoid counting solutions of the same subproblems, Bayardo proposed "good learning". A "good" is a set of variable assignments for which the total solution number of the subproblem is greater than 0. Cachet (Sang *et al.* 2004) shows that good recording (component caching) can dramatically improve the performance of an MC solver for many real world problems.

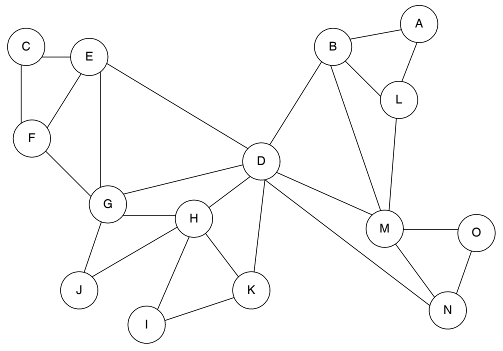


Figure 3: The primal graph of F, where $F = (\bar{A}, \bar{B}, \bar{L})\ (B, \bar{L}, M)\ (M, \bar{N}, O)\ (B, \bar{D}, M)\ (D, \bar{M}, \bar{N})\ (\bar{C}, E, F)\ (E, F, G)\ (E, G, \bar{D})\ (D, G)\ (J, \bar{G}, H)\ (\bar{H}, I, \bar{K})\ (G, \bar{D}, \bar{H})\ (D, H, K)\ (\bar{D}, H)$

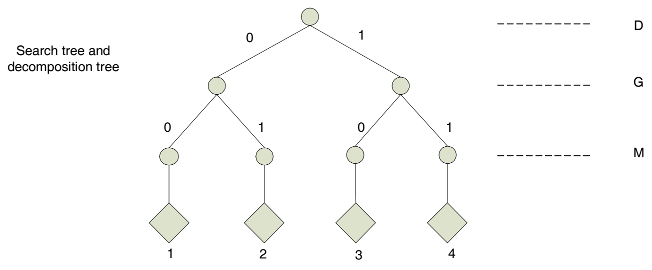


Figure 4: The search tree of variables D, G, M in F.

A component is defined relative to the residual formula (the reduced formula after unit propagation of the current partial assignment). The component caching algorithm takes the original CNF formula as the first component and keeps creating new components and counting each component until the number of satisfying assignments for each component has been stored in the cache.

Dynamic Decomposition

To save the partial solutions for each run, we decompose the initial SAT instance into subproblems recursively. Sinz (2004) shows that long implication chains exist in many real world instances. When solving such problems, most of the variables on the implication chains are instantiated after making a relatively small number of decisions, and the internal structure often changes dramatically in different parts of the search tree. Thus, we expect that a *dynamic* decomposition method can be used to effectively identify subproblems. Bayardo (2000) first proposed the idea of identifying connected components in the constraint graph of a CNF instance during DPLL.

As an example of dynamic decomposition, Figure 3 gives the primal constraint graph of a CNF formula F. Instead of using width-first-search as in Sang et al. (2005), we use depth-first-search to identify connected components after each variable's instantiation. Figure 4 shows the search tree of regular DPLL on F up to depth 3. Figure 5 and Figure 6 show the decomposition trees at leaf node 2 and 3 in Figure 4. The decompositions at leaf nodes are different due to the partial assignments and related unit propagations. In those diagrams, we can see that F is decomposed dynamically along each branch of the search tree. Compared with static decomposition, dynamic decomposition can capture more internal structure. Using DFS for component (good) detection has the advantage that we have more options for variable ordering heuristics inside each component.

We record the model count of every component and their parent-child relation for each run. If F is updated—for example, a clause is removed from the original component C_6—instead of recalculating every component, we only need to recalculate those components which belong to $ancestors(C_6)$. So, except for C_6, we need to recalculate C_5 and C_1. See Figure 5.

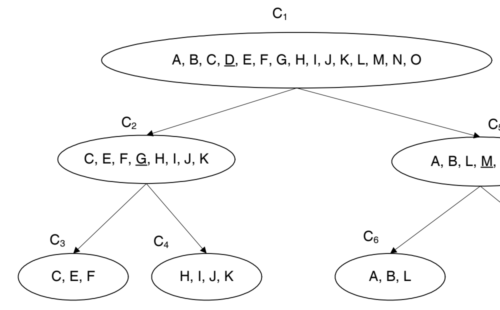


Figure 5: The decomposition tree of F at leaf node 2 of the search tree in Figure 4.

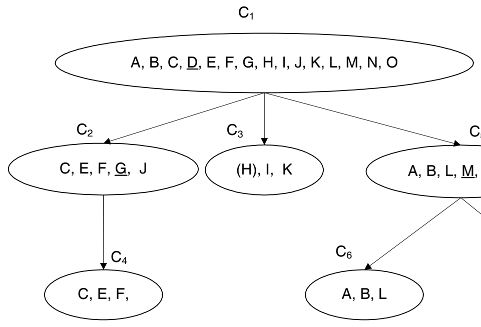


Figure 6: The decomposition tree of F at leaf node 3 of the search tree in Figure 4.

At each decision-making point, we need to detect new components in the top component of the branchable component stack. If the new components are found in the cache, then we can percolate up and update the total counting number. Otherwise, we keep analyzing the new component until its model count is calculated and we save the counting in the cache and component database for our next problem.

Basic Update Operations

Once we have the model count of the original problem, a relaxation will create more models, while a restriction will decrease the model count. In the following, we will discuss the possible updates of a CNF. There are several basic operations that could be used to update the problem sequence. Each of the possible modifications of a Bayesian network—add a node, delete a node, add an edge, delete an edge, change a CPT entry, and reverse an edge—can be expressed by a combination of these basic update operations on the CNF formula.

Removing existing clauses or removing existing literals from clauses. The basic modification of a CNF is to remove a literal from a clause. As a special case, the removal of a clause in a CNF will include removal of all literals in that clause and the variables of the removed clause are still in other clauses. If, for example, all the variables of a clause are in a component, the removal of a literal in that clause will lead to changes in the primal constraint graph. Since the structure of this component's children in the decomposition tree may also be changed, we not only need to recount the model number of this component, but this component's childrens' component model numbers should be updated as well. Considering the disconnectivity among its child components, only those components that have connection with the removed variable need to be recounted.

For purposes of explanation, assume that we follow the same variable ordering in each run and we remove only one literal from an existing clause. We can make the following observations. Here, a "clean" component is a component which does not include any literal of the updated clause.

Observation 1: If, in the updated clause, the removed literal and other literals have not been instantiated, then all the clean components created so far in the search tree will have the same structure and the same model count as the previous run. Consider the example in Figure 5. Both C_2 and C_5 are identified after variable D is instantiated. If the literal $-B$ is removed from $(-A, -B, -L)$ in C_5, the structure of C_2 remains the same. If there is any structural change in C_2, it must be caused either by the existing unit propagation chain connecting the updated clause and C_2 or by the instantiation of variable D. Since D is not in the updated clause and C_2 and C_5 are disconnected in the search tree, C_2 is "stable".

Observation 2: When any literal in the updated clause has been instantiated, the clean components which have been identified are stable. For example, if we delete literal D from $(B, -D, M)$. After $-D$ becomes instantiated, both C_2 (see Figure 5) and C_3 (see Figure 6) are clean components. If C_2 or C_3 are unstable due to the removal of $-D$, there must be at least one path, which does not include D, between C_2 (or C_3) and the other literals (B, M) in the updated clause. However, if such a path exists, C_2 and C_3 will not be disconnected from C_5 after $-D$ becomes instantiated.

Adding literals of existing variables to existing clauses. Adding a literal of an existing variable to a clause increases the number of models of the original problem. Since the new literals may connect components in the different branches of the decomposition tree, the related parts of the decomposition tree need to be updated. We get a similar conclusion as removing a literal. Here, the clean component does not include any of the original literals of the updated clause and the literals which share the same clause with the new literal.

Observation 3: In the search tree of an updated CNF, if the new literal and other literals which share the same clause with the new literal have not been instantiated, then all the clean components created so far in the search tree will have the same structure and the same number of models as the previous run.

Observation 4: If either the new literal or any related literal has been instantiated, all the existing clean components are stable.

Problem expansion: Adding literals of new variables. Adding new variables creates more satisfiable solutions. As discussed above, all clean components are stable.

Experimental Results

Our DynaMC program is built on Cachet, which is currently the fastest model counting solver. The difference between DynaMC and Cachet is that new components learned in each instance are saved in a database. The valid portion of the database is imported into the new instance before the beginning of each run. Also, we implemented a DynaMC compiler based on JavaBayes. A consistent variable numbering system can be maintained among compiled CNFs. In this way, adding or deleting variables or links in the original Bayesian network only generates local changes in each compiled CNF. The experiments were run on a Linux desktop with a 2.6GHz P4 processor and 1GB of memory, except for Experiment 2 which used a 3.0GHz P4 processor with 3GB of memory.

We compared DynaMC against Cachet. In our experiments, both programs compute the weight of the formula, although "computing all marginals is only about 10%-40% slower" (Sang, Beame, & Kautz 2005). We tested our approach over several kinds of networks: real networks taken from the repository of the Hebrew University[1], deterministic QMR networks, and artificially generated grid networks[2].

The intention of the experiments overall is to show that our method works well with scaling (Experiment 1), Bayesian network structure changes (Experiments 2 & 3), and Bayesian network parameter changes (Experiment 4). In Experiment 1, we made changes to the encoded weighted model counting problem. In Experiments 2–4, we made changes to the actual Bayesian networks. In all our experiments, the reported run-times assume that the component cache is memory-resident.

[1]http://compbio.cs.huji.ac.il/Repository/networks.html

[2]http://www.cs.washington.edu/homes/kautz/Cachet/

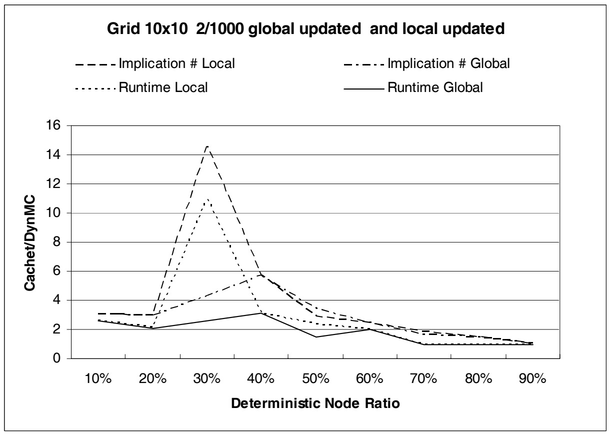


Figure 7: The ratio (Cachet/DynaMC) of runtime and implication number on 10 × 10 grid problems. 10 instances are tested on different percentage of deterministic nodes. We globally and locally delete 2/1000 literals on each instance.

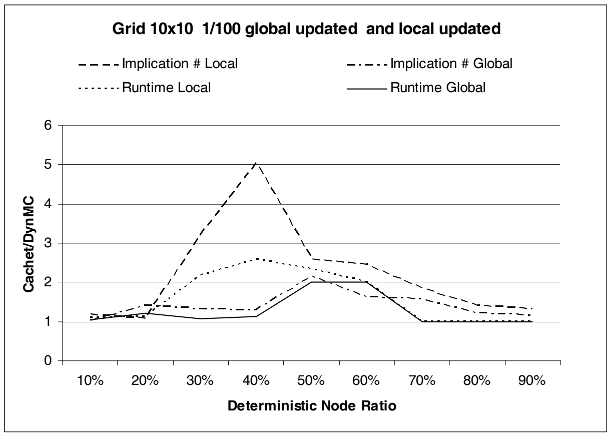


Figure 8: The ratio (Cachet/DynaMC) of runtime and implication number on 10 × 10 grid problems. 10 instances are tested on different percentage of deterministic nodes. We globally and locally delete 1/100 literals on each instance.

Experiment 1

In this experiment, we studied the effect of the number of changed literals and their relative positions (whether the changes occurred globally or locally) in compiled grid problems. Our approach to generating global modifications was to randomly select a clause from the original CNF and randomly add/delete a literal. To generate local modifications, we created a series of CNFs, $M_0, M_1, \ldots, M_n$. In M_i ($0 < i \leq n$), only the clauses which share at least one variable with the modified clauses in M_{i-1} are selected. In order to compare DynaMC and traditional MC, we collected the ratio of median runtime and the ratio of number of unit propagations (implications). The fraction of the nodes that are assigned deterministic conditional probability tables (CPTs) is a parameter, the deterministic ratio. The CPTs for such nodes are randomly filled in with 0 or 1; in the remaining nodes, the CPTs are randomly filled with values chosen uniformly in the interval (0, 1) (Sang, Beame, & Kautz 2005).

Figure 7 and 8 show the results obtained for 0.2% and 1% literal deletion of a 10 × 10 grid network compiled into a DynaMC problem. As discussed earlier, the larger the portion that is shared between successive problems, the more components we can save for achieving speed-ups. Now, as can be seen, the distribution of changes also plays an important role. DynaMC works much better when local changes are performed. Intuitively, we need to recompute more model counts for independent subproblems if modifications are distributed evenly in more components. At the low end of the deterministic ratio, the constraint graphs of compiled CNFs have very high width and density. So there is a low possibility of finding disconnected components while executing DPLL. The problems at the high end of deterministic ratio are relatively easy, so they can be solved without checking the component database.

In Figure 9, we mixed both insert literal and delete literal operations: the ten modifications included five insertions and five deletions. We also fixed the deterministic ratio as 75% and tested different problem sizes from 10 × 10 to 44 × 44. The experimental results show that DynaMC can be solved more efficiently than a set of independent regular MC problems.

Experiment 2

The grid networks in this experiment have 90% deterministic ratio. Our approach for generating modifications in Bayesian networks is based on the fact that the updates are usually concentrated on a limited region of the model. We use the following procedure to create $2m$ modifications in a sequence Seq. The procedure is similar to the procedure used by Flores, Gámez, & Olesen (Flores, Gámez, & Olesen 2004) in their experiments on incremental join tree compilation.

1. $DeleteSeq = \{\}$, $AddSeq = \{\}$
2. We randomly select a node V_i from the network B. Then we remove all the edges $e_i = \{E \mid V_{pi} \rightarrow V_i\}$ between V_i and its parents V_{pi} from B and add modification $delete(e_i)$ onto the end of DeleteSeq
3. Insert modification $add(e_i)$ into the front of AddSeq
4. All the remaining nodes linked to V_i are included in a set N_i. The next node V_{i+1} is randomly selected from N_i.
5. Return to Step 2, until we run m loops
6. Seq = $DeleteSeq$ CONCATENATE $AddSeq$

In Table 1, we tested 10 modifications for each problem size($m = 5$). Every modification includes 1-2 edges in grid networks depending on the location of the random selected node. The total runtime is the sum of runtime for solving the 10 modified networks. DynaMC is 2 times faster than Cachet in the best case.

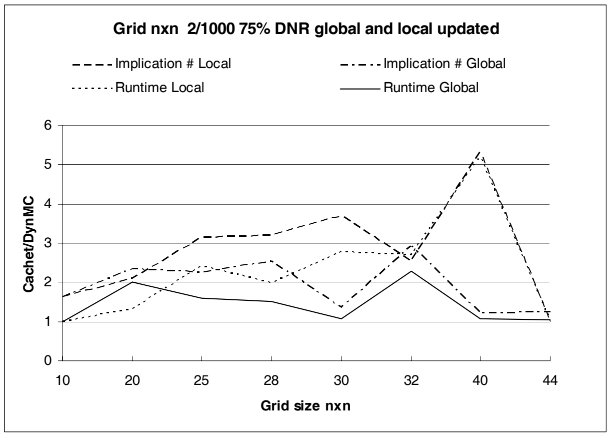


Figure 9: The ratio (Cachet/DynaMC) of runtime and implication number on N × N grid problems. 10 instances are tested on each problem size. We globally and locally insert and delete 2/1000 literals on each instance.

Table 1: Total runtime of 10 modifications for each grid network.

	Total Runtime Sec.		Improvement
Grid	DynaMC	Cachet	%
10×10	34	31	-8.8%
12×12	103	170	65.0%
14×14	182	228	25.3%
16×16	229	368	60.7%
18×18	292	597	104.5%
20×20	299	370	23.7%
21×21	488	616	26.2%
22×22	596	710	19.1%
24×24	1558	2067	32.7%

Experiment 3

DQMR is a simplified representation of the QMR-DT (Sang, Beame, & Kautz 2005). Each DQMR problem is a two-level multiply connected belief-network in which the top layer consists of diseases and the bottom layer consists of symptoms. If a disease may yield a symptom, there is an edge from the disease to the symptom. We test networks with 50 to 100 diseases and symptoms. The edges of the bipartite graph are randomly chosen. Each symptom is caused by three randomly chosen diseases. The problem consists of computing the weight of the encoded formula given a set of consistent observations of symptoms. In each instance of 50+50 networks, 10% disease nodes are randomly selected as observed. The observed nodes are 50% in 100+100 networks. Table 2 shows that the modifications on real Bayesian networks can be translated into DynaMC and solved more efficiently.

Experiment 4

In this experiment we analyzed the effect of using DynaMC when network determinism changed (see Table 3). We generated a sequence of 10 instance $M_0, M_1, \ldots, M_{10}$ for each network. $M_k (0 < k \leq 10)$ is generated by randomly selecting a node and making one entry of its CPT 1. The component library imported into M_k is generated in M_{k-1}.

Table 2: Total runtime and implication number of 10 DQMR instances for each network, where (e) indicates that an edge from a disease to a symptom was randomly selected and removed from the original DQMR network, and (n) indicates removing a randomly selected symptom node.

	Total Runtime Sec.		Implication#	
DQMR	DynaMC	Cachet	DynaMC	Cachet
50+50 (e)	194	334	4.3×10^6	8.0×10^6
100+100 (e)	22	48	1.6×10^6	3.7×10^6
50+50 (n)	34	63	8.2×10^5	1.3×10^6
100+100 (n)	101	172	5.6×10^6	1.1×10^7

Table 3: Total runtime and implication number of a sequence of 10 instances for real networks.

	Total Runtime Sec.		Implication#	
BN	DynaMC	Cachet	DynaMC	Cachet
Alarm	17	43	1.5×10^6	8.4×10^6
Insurance	360	1082	1.2×10^8	3.9×10^8
Asia	0.01	0.01	169	840
Car-starts	0.02	0.02	370	1690
Water	94	465	2.7×10^7	1.4×10^8

Due to the memory resource required by the large component library we skipped networks which could not be solved by both Cachet and DynaMC. It has been noted that for Bayesian networks which have variables with large cardinalities, very large CPTs, or a small amount of determinism, the general encoding method does not work well (Chavira & Darwiche 2005). Those encoded CNFs are simply too large to be quickly decomposed and quickly use up all available memory. In the same paper, Chavira and Darwiche propose a more efficient encoding.

For a few test instances, our method was slower than Cachet due to the overhead of querying the cache. In those "failed" cases, we found that the components imported from previous runs were extremely small. Usually, the average size of imported components in those cases was less than 10 literals. In many successful instances, the average literal number of imported components is more than 100. When the component database is full of small components, the overhead of checking each new generated component increases. Even if the correct component is found, only a few variables can be skipped in the search tree. If we limit the size components to import only "big" components, we can improve the performance of most "fail" instances. In practice, we expect the imported components to have at least 20-50 literals. However, we did not set a component limit in any of the experiments reported above. Another possible solution for the problem of overhead is to design a more accurate hash function to increase the hitting rate, so that when we search every new component in our database our query can return quickly.

Conclusions

There is obvious need for improving the performance of a model counting problem as incremental local change happens. Because of the errors in model construction and changes in the dynamics of the domains, we cannot ignore the new information. By maintaining the partial solutions of similar instances, we can achieve great speedups over current model counting algorithms. Cachet is currently the fastest model counting solver available. In grid and DQMR problems, Cachet dominates both join tree and previous state of the art conditioning algorithms. As compared with Cachet, we obtained significant improvements in runtime for most networks. Both the Grid and DQMR Bayesian networks have high density and tree-width. Those features challenge traditional Bayesian inference.

References

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. DPLL with caching: A new algorithm for #SAT and Bayesian inference. *Electronic Colloquium on Computational Complexity* 10(3).

Bayardo, R. J., and Miranker, D. P. 1996. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI-96*, 298–304.

Bayardo, R. J., and Pehoushek, J. D. 2000. Counting models using connected components. In *AAAI-00*, 157–162.

Buntine, W. L. 1991. Theory refinement of Bayesian networks. In *UAI-91*, 52–60.

Chavira, M., and Darwiche, A. 2005. Compiling Bayesian networks with local structure. In *IJCAI-2005*.

Darwiche, A. 1998. Dynamic jointrees. In *UAI-98*, 97–104.

Darwiche, A. 2002. A logical approach to factoring belief networks. In *KR-02*, 409–420.

Darwiche, A. 2003. A differential approach to inference in Bayesian networks. *J. ACM* 50(3):280–305.

Dechter, R., and Dechter, A. 1988. Belief maintenance in dynamic constraint networks. In *AAAI-88*, 37–42.

Flores, M. J.; Gámez, J. A.; and Olesen, K. G. 2003. Incremental compilation of Bayesian networks. In *UAI-03*, 233–240.

Flores, M. J.; Gámez, J. A.; and Olesen, K. G. 2004. Incremental compilation of Bayesian networks in practice. In *Proceedings of the Fourth International Conference On Intelligent Systems Design and Applications (ISDA 2004)*, 843–848.

Friedman, N., and Goldszmidt, M. 1997. Sequential update of Bayesian network structure. In *UAI-97*, 165–174.

Hoos, H. H., and O'Neill, K. 2000. Stochastic local search methods for dynamic SAT–an initial investigation. In *AAAI-2000 Workshop Leveraging Probability and Uncertainty in Computation*, 22–26.

Jin, H., and Somenzi, F. 2005. An incremental algorithm to check satisfiability for bounded model checking. *Electr. Notes Theor. Comput. Sci.* 119:51–65.

Lam, W., and Bacchus, F. 1994. Using new data to refine a Bayesian network. In *UAI-94*, 383–390.

Littman, M. L. 1999. Initial experiments in stochastic satisfiability. In *AAAI-99*, 667–672.

Sang, T.; Beame, P.; and Kautz, H. 2005. Solving Bayesian networks by weighted model counting. In *AAAI-05*, 1–10.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *SAT-04*.

Schiex, T., and Verfaillie, G. 1994. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools* 3:1–15.

Sinz, C. 2004. Visualizing the internal structure of SAT instances. In *SAT-04*.