# Solving Bayesian Networks by Weighted Model Counting

**Tian Sang, Paul Beame, and Henry Kautz**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{sang,beame,kautz}@cs.washington.edu

## Abstract

Over the past decade general satisfiability testing algorithms have proven to be surprisingly effective at solving a wide variety of constraint satisfaction problem, such as planning and scheduling (Kautz and Selman 2003). Solving such NP-complete tasks by "compilation to SAT" has turned out to be an approach that is of both practical and theoretical interest. Recently, (Sang *et al.* 2004) have shown that state of the art SAT algorithms can be efficiently extended to the harder task of *counting* the number of models (satisfying assignments) of a formula, by employing a technique called *formula caching*. This paper begins to investigate the question of whether "compilation to model-counting" could be a practical technique for solving real-world #P complete problems. We describe an efficient translation from Bayesian networks to *weighted* model counting, extend the best model-counting algorithms to weighted model counting, develop an efficient method for computing all marginals in a single counting pass, and evaluate the approach on computationally challenging reasoning problems.

## Introduction

In recent years great strides have been made in the development of efficient satisfiability solvers. Programs such as zChaff (Zhang *et al.* 2001) and Berkmin (Goldberg and Novikov 2002) are routinely used in industry and academia to solve difficult problems in hardware verification, planning, scheduling, and experiment design (Kautz and Selman 2003). Such practical success is quite surprising, since all known complete SAT algorithms run in worst-case exponential time, a situation unlikely to change, given that satisfiability testing is NP-complete. Although these solvers are all based on the original DPLL backtracking SAT procedure (Davis *et al.* 1962), they incorporate a number of techniques — in particular, non-chronological backtracking (Dechter 1990), clause learning (Bayardo Jr. and Schrag 1997; Marques-Silva and Sakallah 1996) and variable selection heuristics (Cook and Mitchell 1997) — that tremendously improve performance.

Any backtracking SAT algorithm can be trivially extended to one that counts the number of satisfying assignments by simply forcing it to backtrack whenever a solution is found. Such a simple approach, however, is infeasible for all but the smallest problem instances. Building on previous work on model-counting by (Bayardo Jr. and Schrag 1997) and theoretical work on formula-caching proof systems (Majercik and Littman 1998; Beame *et al.* 2003; Bacchus *et al.* 2003a), the creators of Cachet (Sang *et al.* 2004) built a system that scales to problems with thousands of variables by combining clause learning, formula-caching, and decomposition into connected components.

Model-counting is complete for the complexity class #P, which also includes problems such as computing the permanent of a Boolean matrix and performing inference in Bayesian networks. The power of programs such as Cachet raises the question of whether various real-world #P problems can be exactly solved in practice by translation to model-counting and the application of a general model-counting algorithm. This paper provides initial evidence that the answer is affirmative: such a translation approach can indeed be effective for interesting classes of hard problems that cannot be solved by previously known exact methods.

This paper examines the problem of computing the posterior probability of a query given evidence in a Bayesian network. Such Bayesian inference is well known to be #P complete (Roth 1996), and both Bayesian inference and #SAT are instances of a more general counting problem called "sum-product" (Dechter 1999; Bacchus *et al.* 2003a). However, there has been little previous work on *explicitly* translating Bayesian networks to instances of #SAT. (Littman 1999) briefly sketches a reduction, and (Darwiche 2002; Chavira *et al.* 2004) describe a method for encoding a Bayesian network as a set of propositional clauses, where certain variables are associated with the numeric values that appear in the original conditional probability tables. We employ a translation from Bayesian networks to weighted model-counting problem that is similar but smaller both in terms of the number of clauses and the total sum of the lengths of all clauses. We also describe the relatively minor modifications to Cachet that are required to extend it to handle weighted model-counting.

Many approaches to Bayesian inference, such as join tree algorithms (Spiegelhalter 1986), calculate the marginals of all variables in one pass. A translation approach, therefore, would be at a serious disadvantage, if such a calculation required a separate translation and query for *each* variable. We therefore further extended our model-counting algorithm so that all marginals can be computed efficiently in one pass. In addition to calculating the number of models which satisfy a formula, the extended algorithm calculates, for each variable, the number of satisfying models in which that variable is true. These additional statistics can be kept with usually insignificant overhead.

We present experimental results on three families of computationally challenging Bayesian networks, grid networks, plan recognition problems, and diagnostic networks. These

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

domains exhibit high density and tree-width, features that are problematic for many previous approaches. Our experiments show that as problem size and the fraction of deterministic nodes increases, the translation approach comes to dominate both join tree and previous state of the art conditioning algorithms.

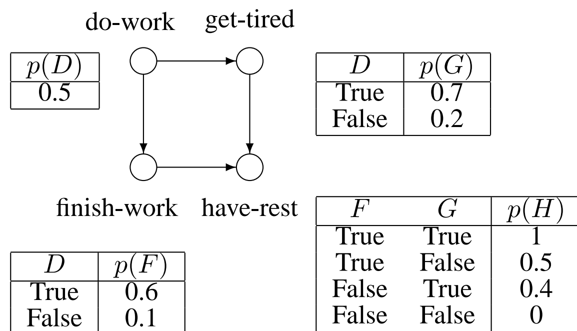
## Related Work

As (Sang *et al.* 2004) demonstrate, Cachet is currently the fastest model-counting system available. Its backtracking DPLL-style search is essentially a form of reasoning by conditioning (Dechter 1999). We now briefly compare the operation of the Cachet-based model-counting approach (MC) with similar conditioning algorithms, in particular recursive conditioning (RC) (Darwiche 2001; Allen and Darwiche 2003), value elimination (VE) (Bacchus *et al.* 2003b), and classical cutset conditioning (CC) (Dechter 1990).

The basic idea of MC, RC, and VE is to to recursively decompose a problem (break it into disconnected components) by branching on variables, though only MC works on CNF encodings. The basic idea of CC is to simplify (not necessarily decompose) a problem so that it contains no loops. RC always branches on (sequences) of variables that partition a problem; CC always branches on a variable that breaks a loop; while MC and VE can branch on any variable chosen heuristically. RC, CC and VE determine a static variable ordering before branching begins, while MC pick variables dynamically. MC, RC, and VE cache the results of evaluated subproblems. MC and VE use a dynamic cache management strategy; while RC tries to allocate enough space to cache all subproblems, but if that is not available, only caches a random fraction of all subproblems. For MC only, cache hits can occur between any subproblems which correspond to the same CNF formula, even if they are derived from different substructures of the original problem. Finally, only MC and VE cache inconsistent subsets of assigned variables (learned clauses, or nogoods) as well as subproblems, but they differ in details of nogood(clause) learning and caching.

## Encoding Bayesian Networks

### Boolean Bayesian Networks




| $p(D)$ |
|---|
| 0.5 |

| $D$ | $p(G)$ |
|---|---|
| True | 0.7 |
| False | 0.2 |

| $D$ | $p(F)$ |
|---|---|
| True | 0.6 |
| False | 0.1 |

| $F$ | $G$ | $p(H)$ |
|---|---|---|
| True | True | 1 |
| True | False | 0.5 |
| False | True | 0.4 |
| False | False | 0 |

Figure 1: The work-rest Bayesian Network

We illustrate the approach with the 4 node Bayesian network in Fig. 1. Fig. 2 shows the encoding for this example. We use two types of variables: *chance* variables that encode entries in CPTs and *state* variables for the values of nodes. Each row of each CPT has an associated chance variable whose weight is the probability given in the True column

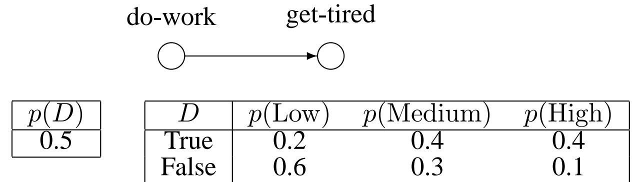
**State variables:** $G$, $F$, $H$
**Chance variables (weights in parentheses):**
at do-work: $d$ (0.5)
at get-tired: $g_1$ (0.7), $g_0$ (0.2)
at finish-work: $f_1$ (0.6), $f_0$ (0.1)
at have-rest: $h_{10}$ (0.5), $h_{01}$ (0.4)
**clauses for node get-tired**
$(\neg d, \neg g_1, G)(\neg d, g_1, \neg G)(d, \neg g_0, G)(d, g_0, \neg G)$
**clauses for node finish-work**
$(\neg d, \neg f_1, F)(\neg d, f_1, \neg F)(d, \neg f_0, F)(d, f_0, \neg F)$
**clauses for node have-rest**
$(\neg F, \neg G, H)(\neg F, G, \neg h_{10}, H)(\neg F, G, h_{10}, \neg H)$
$(F, \neg G, \neg h_{01}, H)(F, \neg G, h_{01}, \neg H)(F, G, \neg H)$

Figure 2: Variables and clauses for the work-rest Bayesian Network

of that row of the CPT. Source nodes have only one row in their CPTs so their state variables are superfluous and we identify them with the corresponding chance variables. Each CPT row yields two clauses which determine the weight of the node's value assignment as a function of the parent node values and the weight of the CPT entry. For example, at the CPT of node get-tired, when its parent do-work is True, the conditions are equivalent to the following two clauses: $(\neg d \vee \neg g_1 \vee G)$ and $(\neg d \vee g_1 \vee \neg G)$. For a CPT entry with value 0 or 1, as in rows 1 and 4 of the CPT for have-rest, the value of the node is fully determined by its parents and we encode the implication using one clause without using a chance variable.

### General Bayesian Networks

Now we consider the more general case of encoding multiple-valued nodes. As in Figure 3, suppose that the network has only two nodes: a Boolean node do-work and a 3-valued node get-tired with values Low, Medium, High.




| $p(D)$ |
|---|
| 0.5 |

| $D$ | $p$(Low) | $p$(Medium) | $p$(High) |
|---|---|---|---|
| True | 0.2 | 0.4 | 0.4 |
| False | 0.6 | 0.3 | 0.1 |

Figure 3: A Bayesian network example with a multiple valued node.

To encode the states of node get-tired, we use 3 variables, $G_L$, $G_M$, and $G_H$, and 4 constraint clauses to ensure that exactly one of these variables is True. A chance variable for a CPT entry has a weight equal to the conditional probability that the entry is True given that no prior variable in the row is True. For example, for the first row in the CPT for get-tired, we add two chance variables: $a$ and $b$ with the weight of $a$ set to 0.2 and the weight of $b$ set to $\frac{0.4}{1-0.2} = 0.5$. The last entry in the row does not need a chance variable. For this row we get three clauses: $(\neg D \vee \neg a \vee G_L)$, $(\neg D \vee a \vee \neg b \vee G_M)$, and $(\neg D \vee a \vee b \vee G_H)$.
Turning all such propositions into clauses and with the additional constraints that state variables are exclusive, the encoding for the example with a multiple-valued node is shown in Fig. 4. In general, if a node can take on $k$ values, $k - 1$ chance variables are added for each row in its CPT.

**State variables:** $G_L, G_M, G_H$
**Chance variables (weights in parentheses):**
at do-work: $D$ (0.5)
at get-tired: $a$ (0.2), $b$ (0.5), $c$ (0.6), $d$ (0.75)
**clauses for node get-tired**
$(\neg G_L, \neg G_M)(\neg G_M, \neg G_H)(\neg G_M, \neg G_H)(G_L, G_M, G_H)$
$(\neg D, \neg a, G_L)(\neg D, a, \neg b, G_M)(\neg D, a, b, G_H)$
$(D, \neg c, G_L)(D, c, \neg d, G_M)(D, c, d, G_H)$

Figure 4: Variables and clauses for the example in Fig. 3

## Weighted Model Counting

**Algorithm 1** BasicWeightedModelCounting

```
BWMC(φ)
// returns the weight of the CNF formula φ
  if φ is empty, return 1
  if φ has an empty clause, return 0
  select a variable v in φ to branch
  return BWMC(φ|v=0) × weight(−v)+
         BWMC(φ|v=1) × weight(+v)
```

Basic Weighted Model Counting (BWMC) is a simple recursive DPLL-style algorithm that for our Bayesian network encoding will use two types of variables: chance variables with $weight(+v) + weight(-v) = 1$ and unweighted state variables to which we impute $weight(+v) = weight(-v) = 1$. The *weight* of a (partial) variable assignment is the product of weights of the literals in that assignment. If $s$ is a total assignment satisfying $\phi$ write $s \models \phi$. The weight of a formula $\phi$ is $\sum_{s \models \phi} weight(s)$. The following is immediate.

**Lemma 1.** *The result returned by BWMC($\phi$) for a CNF formula $\phi$ is $weight(\phi)$.*

A *legal instantiation* of a Bayesian network $N$ is a complete value assignment to the Bayesian network nodes that has non-zero probability. Any legal instantiation $I$ of $N$ immediately yields a partial assignment $\pi(I)$ of the state variables of the CNF $\phi$ encoding $N$.

**Lemma 2.** *If $\phi$ is the encoding of Bayesian network $N$ with legal instantiation $I$ then*
$p(I) = \sum_{s \models \phi \text{ and } s \text{ extends } \pi(I)} weight(s)$,
*where $p(I)$ is the likelihood of $I$.*

*Proof.* Fix any legal instantiation $I$ of the Bayes network $N$. The partial assignment $\pi = \pi(I)$ will assign *true* to all state variables corresponding to values assigned by $I$. It remains to assign truth values to the chance variables in the CPTs; We define this part $\pi$ in each such CPT separately. Given instantiation $I$ there is a unique associated entry in each of the CPTs in $N$; the values of the immediate predecessors determines the row, and the value of the node determines the column. If that column is not the last column, there will be an associated chance variable; $\pi$ will assign *true* to that variable and *false* to all prior variables in that row. If that column is the last column, there will not be an associated chance variable but $\pi$ will assign *false* to all variables in that row. The remaining chance variables in the CPT will be unassigned.

By our definition of $\phi$ the weight of the portion of $\pi$ in the CPT is equal to the probability of the associated entry in the CPT. It is also easy to check that all the clauses defined for the node $V$ of $N$ to which the CPT is associated are satisfied by $\pi$. Every variable $v$ that is not assigned a value in $\pi$ is a chance variable of $\phi$ and is therefore a primary variable in the weighted model counting algorithm; this means that $weight(+v) + weight(-v) = 1$ and thus the total weight of all total assignments $s$ that extend $\pi$ is equal to the weight of $\pi$ which is the product of the weights of the portion of $\pi$ in each associated CPT. This is exactly equal to $P(I)$ by definition.

The reverse direction is also easy to check: Any satisfying assignment $s$ for $\phi$ must extend some partial assignment $\pi$ as defined above. Since $s$ satisfies the exclusive clauses of $\pi$, precisely one state variable associated with each node is assigned value true. As above, the values of these state variables determine an associated entry in each CPT. The form of the clauses defined for the CPT in each row will force the assignment to the chance variables in the row to be of the form of $\pi$ above. □

**Theorem 3.** *If $\phi$ is the encoding of a Bayesian network $N$ and $C$ is a constraint on $N$, BWMC ($\phi \wedge C$) returns the likelihood of the network $N$ with constraint $C$.*

*Proof.* By Lemma 1, $BWMC(\phi \wedge C)$ computes the weighted sum of solutions. By Lemma 2, this is equal to the sum of the likelihoods of those instantiations that satisfy $C$, which by enumeration is indeed the likelihood of the constrained Bayes network. □

Therefore, if $\phi$ is the CNF encoding of a Bayesian network, a general query P(Q|E) on that network can be answered by $\frac{BWMC(\phi \wedge Q \wedge E)}{BWMC(\phi \wedge E)}$. We should emphasize that it supports queries and evidence in arbitrary propositional form, not available by any other exact inference methods.

**Weighted Cachet: Optimized Weighted Model Counting** BWMC above is a generalization of exact model counting for #SAT in which the weights are no longer constrained to be $\frac{1}{2}$. To provide an optimized implementation of weighted model counting, we have modified Cachet, the fastest exact model-counting system available, which is built on top of zChaff (Zhang *et al.* 2001). Cachet combines unit propagation, clause learning, non-chronological backtracking and component caching, and can take advantage of a variety of dynamic branching heuristics (Sang *et al.* 2005).

### Weighted Model Counting for All Marginals

On inference we frequently want to calculate marginal probabilities of all variables. The algorithm MarginalizeAll shows how BWMC can be extended to do this in the context of unit propagations. The vector $Marginals$ has an entry for each variable in $\phi$ and is passed by reference, while $LMarginals$ and $RMarginals$ are corresponding local vectors storing the marginals computed by the recursive calls on left and right subtrees. When MarginalizeAll returns, the result $LW + RW$ is $weight(\phi)$, and $Marginals$ contains the weighted marginals — the real marginals multiplied by $weight(\phi)$. The marginals for variables found during the recursive calls must be multiplied by the weight of the unit propagations for those branches. Those variables in $\phi$ that disappear from a branch without having been explicitly set have their marginals for that branch set to their original positive weight (multiplied by the weight of the branch).

**Algorithm 2** MarginalizeAll

$MarginalizeAll(\phi, Marginals)$
// returns weight of formula $\phi$
// all weighted var marginals stored in vector $Marginals$
  if $\phi$ is empty, return 1
  if $\phi$ has an empty clause, return 0
  select a variable $v$ in $\phi$ to branch
  $UP(\phi, -v)$ =unit propagations resulted from $\phi|_{v=0}$
  $UP(\phi, +v)$ =unit propagations resulted from $\phi|_{v=1}$
  $InitializeVector(LMarginals, 0)$
  $InitializeVector(RMarginals, 0)$
  $LW = MarginalizeAll(\phi|_{UP(\phi,-v)}, LMarginals)$
      $\times weight(UP(\phi, -v))$
  $RW = MarginalizeAll(\phi|_{UP(\phi,+v)}, RMarginals)$
      $\times weight(UP(\phi, +v))$
  for each var $x$ in $\phi|_{UP(\phi,-v)}$
    $LMarginals[x] \ \times = weight(UP(\phi, -v))$
  for each var $x$ in $\phi|_{UP(\phi,+v)}$
    $RMarginals[x] \ \times = weight(UP(\phi, +v))$
  for each var $x$ in $UP(\phi, -v)$
    if $x$ is in positive form
      then $LMarginals[x] = LW$
    else $LMarginals[x] = 0$
  for each var $x$ in $UP(\phi, +v)$
    if $x$ is in positive form
      then $RMarginals[x] = RW$
    else $LMarginals[x] = 0$
  for each var $x$ in $\phi$ but not in $UP(\phi, -v) \cup \phi|_{UP(\phi,-v)}$
    $LMarginals[x] = LW \times weight(+x)$
  for each var $x$ in $\phi$ but not in $UP(\phi, +v) \cup \phi|_{UP(\phi,+v)}$
    $RMarginals[x] = RW \times weight(+x)$
  $Marginals = SumVector(LMarginals, RMarginals)$
  return $LW + RW$

Our experiments were performed using an extension of this algorithm that works with component caching, clause learning and non-chronological backtracking as used in Cachet. This requires caching both the weight and the vector of marginals for each component and can use considerably more space than Cachet's weighted model counting. In addition, combining the marginals when the residual formula consists of several components is somewhat more complicated. In our experiments, when the problem fits in memory, computing all marginals is only about 10% — 40% slower than computing only the weight of the formula.

## Experimental Results

We compared Cachet against state-of-the-art algorithms for exact Bayesian inference on benchmark problems from three distinct domains. The competing approaches are (i) the join tree algorithm, as implemented in Netica (Norsys Software Corp., http://www.norsys.com); (ii) recursive conditioning (RC) as implemented in SamIam version 2.2 (http://reasoning.cs.ucla.edu/samiam/); and value elimination as implemented in Valelim (Bacchus *et al.* 2003b).

We deliberately selected benchmark problems that are intrinsically hard because they are highly structured and contain many logical dependencies between variables. We do not claim that Cachet is always, or even usually, superior to others. (In particular, on problems with small tree-width, the join tree approach is likely to be much faster.) We simply claim that these are non-trivial, challenging problems, which contain natural patterns of structure and are of interest on their own to the probabilistic reasoning community.

| Grid networks, deterministic ratio = 0.5 | | | | |
|---|---|---|---|---|
| size | Join Tree | RC | Val. Elim. | Cachet |
| $10 \times 10$ | 0.02 | 0.88 | 2.0 | 7.3 |
| $12 \times 12$ | 0.55 | 1.6 | 15.4 | 38 |
| $14 \times 14$ | 21 | 7.9 | 87 | 419 |
| $16 \times 16$ | X | 104 | 20861 (6) | 890 |
| $18 \times 18$ | X | 2126 | X | 13111 |
| $20 \times 20$ | X | X | X | X |

| Grid networks, deterministic ratio = 0.75 | | | | |
|---|---|---|---|---|
| size | Join Tree | RC | Val. Elim. | Cachet |
| $10 \times 10$ | 0.02 | 0.87 | 0.15 | 0.30 |
| $12 \times 12$ | 0.47 | 1.5 | 1.4 | 1.0 |
| $14 \times 14$ | 20 | 15 | 8.3 | 4.7 |
| $16 \times 16$ | 227 (3) | 93 | 71 | 39 |
| $18 \times 18$ | X | 1751 | 1053 (9) | 81 |
| $20 \times 20$ | X | 24026 (7) | 94997 (5) | 248 |
| $22 \times 22$ | X | X | X | 1300 |
| $24 \times 24$ | X | X | X | 9967 (7) |

| Grid networks, deterministic ratio = 0.9 | | | | |
|---|---|---|---|---|
| size | Join Tree | RC | Val. Elim. | Cachet |
| $10 \times 10$ | 0.02 | 0.87 | 0.02 | 0.06 |
| $12 \times 12$ | 0.61 | 1.5 | 0.06 | 0.13 |
| $14 \times 14$ | 17 | 11 | 0.23 | 0.23 |
| $16 \times 16$ | 259 | 102 | 0.55 | 0.47 |
| $18 \times 18$ | X | 1151 | 1.9 | 1.4 |
| $20 \times 20$ | X | 44675 (6) | 13 | 1.7 |
| $22 \times 22$ | X | X | 31 | 4.9 |
| $24 \times 24$ | X | X | 84 | 4.5 |
| $26 \times 26$ | X | X | 8010 (7) | 14 |
| $30 \times 30$ | X | X | X | 108 |
| $34 \times 34$ | X | X | X | 888 |
| $38 \times 38$ | X | X | X | 4133 |

Figure 5: Median runtimes in seconds of join tree (Netica), recursive conditioning (SamIam), value elimination (Valelim), and model counting (Cachet) on 10 examples of grid networks at each size. A number in parenthesis indicates only that many out of 10 were solved in 48 hours; X indicates that none were solved due to memory out or time out.

We also note that our current implementation of Cachet, unlike the other solvers, does not perform any relevancy reasoning before answering a query, which hurts it when a query can be answered by consulting only a small portion of a network. The grid network domain is in fact deliberately designed so that everything is relevant to the query.

### Grid Networks

Our first problem domain is grid networks. The variables of an $N \times N$ grid network are denoted $X_{i,j}$ for $1 \leq i, j \leq N$. Each node $X_i, j$ has parents $X_{i-1,j}$ and $X_{i,j-1}$, when those indices are greater than zero. Thus $X_{1,1}$ is a source and $X_{n,n}$ is a sink. Given CPTs for nodes, the problem is to compute the marginal probability of the sink $X_{n,n}$. The fraction of the nodes that are assigned deterministic 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 $(0, 1)$.

Problems were generated in DNE(for Netica etc.) and in BIF format, and then converted, as described before, to

| problem | vars | Join Tree | RC | Val. Elim. | Cachet |
|---|---|---|---|---|---|
| 4-step | 165 | 0.16 | 8.3 | 0.03 | 0.03 |
| 5-step | 177 | 56 | 36 | 0.04 | 0.03 |
| tire-1 | 352 | X | X | 0.68 | 0.12 |
| tire-2 | 550 | X | X | 4.1 | 0.09 |
| tire-3 | 577 | X | X | 24 | 0.23 |
| tire-4 | 812 | X | X | 25 | 1.1 |
| log-1 | 939 | X | X | 24 | 0.11 |
| log-2 | 1337 | X | X | X | 7.9 |
| log-3 | 1413 | X | X | X | 9.7 |
| log-4 | 2303 | X | X | X | 65 |
| log-5 | 2701 | X | X | X | 388 |

Figure 6: Running time in seconds on plan recognition problems. The timing for Val. Elim is the average time to query a single marginal; for the other algorithms, the total time to compute all marginals. X indicates the solver halted due to out-of-memory or did not complete with 48 hours.

the CNF encoding for Cachet. Fig. 5 summarizes the results. Experiments were run on Linux servers, each with dual 2.8GHz processors and 4GB of memory.

Not surprisingly, join tree can only solve the smallest instances, because it runs out of space due to large cliques in the triangulated graph. Recursive conditioning provides the best performance on graphs that are 50% deterministic up to size 18, but on larger problems at higher deterministic ratios is outperformed by both value elimination and model counting.[1] At 90% deterministic nodes, Cachet scales to much larger problems than other methods, consistently solving problems with 1,444 variables ($38 \times 38$), while the largest problem solved by the competing methods contains 576 variables ($26 \times 26$).

## Plan Recognition

The second domain consists of strategic plan recognition problems. Suppose we are watching a rational agent, and want to predict what he or she will do in the future. Furthermore, we know the agent's goals, and all the actions the agent can perform. What can we infer about the probability of the agent performing any particular action? Such plan recognition problems commonly arise in strategic situations, such as military operations.

We formalize the problem as follows: We are given a planning domain described in the form of deterministic STRIPS operators, an initial state, and a set of goals to hold at a specified time in the future. The agent can do anything that is consistent with achieving the goals. Our task is to compute the marginal probability that the agent performs each fully-instantiated action at each time slice.

We generated a set of such plan recognition problems of various sizes in several underlying planning domains by modifying the Blackbox planning as satisfiability system (Kautz and Selman 1999). Cachet could compute the marginals directly by counting the models of the CNF encoding of the planning problems. For the other solvers, we modified Blackbox so that it generated DNE format. Nonsymmetric logical constraints were encoded by introducing conflict variables (Pearl 1988). For example, $p \supset q$ can be encoded by adding a variable $c$ with parents $p$ and $q$, where the CPT for $c$ says it is true *iff* $p$ is true and $q$ is false, and finally asserting $\neg c$ in the evidence.

[1] A newer version of SamIam, not yet distributed at the time of this submission, promises to provide improved performance due to a significantly altered implementation of recursive conditioning.

| size = 50+50, ratio = 0.1, 10 instances each entry | | | |
|---|---|---|---|
| prior | Join Tree | RC | Cachet |
| 0.05 | 1.9 | 3.5 | 1.4 |
| 0.1 | 6 | 2.5 | 1.0 |
| 0.2 | 4 | 3.4 | 3.4 |
| size = 60+60, ratio = 0.1, 10 instances each entry | | | |
| prior | Join Tree | RC | Cachet |
| 0.05 | 52 (5) | 5.7 (2) | 1.7 |
| 0.1 | 46 (3) | 33 (3) | 3.9 |
| 0.2 | 45 (5) | 60 (4) | 54 |
| size = 70+70, ratio = 0.1, 10 instances each entry | | | |
| prior | Join Tree | RC | Cachet |
| 0.05 | X | X | 12 |
| 0.1 | X | X | 60 |
| 0.2 | X | X | 136 |
| size = 100+100, 10 instances each entry, Cachet | | | |
| prior | ratio=0.1 | ratio=0.2 | ratio=0.3 |
| 0.05 | 3705 (7) | 7.9 | 0.077 |
| 0.1 | 98617 (6) | 13 | 0.45 |
| 0.2 | 150572 (4) | 6034 (7) | 43 |

Figure 7: Median runtime on DQMR networks in seconds. Numbers in parenthesis is the number of examples solved if less than 10. X indicates memory-out or time-out.

Fig. 6 summarizes the results. We queried for all marginals using join tree, recursive conditioning, and model counting. As noted in the table, because the implementation we used for value elimination can only query a single node at a time, we instead measured the average run time over a selection of 25 non-trivial queries. The "tire" and "log" problems are based instances from the Tireworld and Logistics domains in the Blackbox distribution. The 4-step and 5-step are small Logistics instances created for this paper.

Model counting handily outperforms the other methods on these problems. Join tree quickly runs out of memory, and recursive conditioning's static value ordering only allows it solve the smallest instances. Value elimination is the only alternative that is competitive, which is consistent with the fact that the algorithm is, as described in the related work section, similar in many respects to Cachet. We hypothesize that Cachet's added power in this domain comes from its use of clause learning and more general component caching.

## DQMR Networks

Our final class of test problems is an abstract version of the QMR-DT medical diagnosis Bayesian networks (Shwe *et al.* 1991). Each problem is given by a two layer bipartite network in which the top layer consists of diseases and the bottom layer consists of symptoms. If a disease may result a symptom, there is an edge from the disease to the symptom. In the CPTs for DQMR (unlike those of QMR-DT) a symptom is completely determined by the diseases that cause it; *i.e.*, it is modeled as an OR rather than a noisy OR of its inputs. As in QMR-DT, every disease has an independent prior probability.

For our experiments, we varied the numbers of diseases and symptoms from 50 to 100 and chose the edges of the bipartite graph randomly, with each symptom caused by four

randomly chosen diseases. The problem was to compute the marginal probabilities for all the diseases given a set of consistent observations of symptoms. The size of the observation set varied between 10% to 30% of all symptoms.

Fig. 7 summarizes the results for join tree, recursive conditioning, and model counting with Cachet for computing all marginals. Although all methods were capable of quickly solving problems with 50 symptoms, both join tree and RC failed on more than half the instances of size 60 and every instance of size 70 and above.

## Discussion & Conclusions

We have provided the first evidence that compiling Bayesian networks to CNF model counting problems is not only a theoretical exercise, but in many cases a practical way to solve challenging inference problems. Such compilation approach allows us to immediately leverage techniques used in the state-of-the-art SAT and model counting engines, such as fast constraint propagation, clause learning, dynamic variable branching heuristics, component caching.

We have presented a general translation from Bayesian networks into weighted model counting on CNF, and also noted that many probabilistic problems, such as the plan recognition benchmarks discussed above, can also be directly represented and solved in CNF.

It is important to note that we do not attempt to argue that compilation and model counting replaces proven approaches such as the join tree algorithm. Rather, it is a complementary approach, which is particularly suitable for problems with complex structure that does not decompose into small cliques, but where many of the dependencies between variables are entirely or partially deterministic. In such cases, the efficient logical machine underlying model counting programs like Cachet stands a good chance of quickly reducing the problem into small subproblems.

Finally, our overview of related work argued that other recent algorithms for Bayesian inference, and in particular, recursive conditioning and value elimination, are quite similar to model counting, and differ mainly in the details of caching and variable branching. It would not be surprising if all the techniques in the current version of Cachet were to appear in a future Bayesian network engine, which proved then to be even faster on the benchmarks from this paper. However, we would also expect satisfiability solvers and the associated model-counting algorithms to continue to improve apace, roughly doubling in speed and problem size every two years. It will be an interesting competition for the foreseeable future.

## References


D. Allen and A. Darwiche. New advances in inference by recursive conditioning. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence UAI-2003*, pages 2–10, 2003.

F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proceedings 44th IEEE FOCS 2003*, pages 340–351, 2003.

F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in Artificial Intelligence UAI-2003*, pages 20–28, 2003.

R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings, AAAI-97: 14th National Conference on Artificial Intelligence*, pages 203–208, 1997.

P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proceedings 18th Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.

M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational bayesian networks for exact inference. In *Proceedings of the Second European Workshop on Probabilistic Graphical Models (PGM-2004)*, pages 49–56, 2004.

S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *DIMACS Series in Theoretical Computer Science*, 1997.

A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 125(1-2):5–41, 2001.

A. Darwiche. A logical approach to factoring belief networks. In *Proceedings of International Conference on Knowledge Representation and Reasoning*, pages 409–420, 2002.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

E. Goldberg and Y. Novikov. Berkmin: a fast and robust satsolver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.

H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 318–325. Morgan Kaufmann, 1999.

H. Kautz and B. Selman. Ten challenges redux: Recent progress in propositional reasoning and search. In *Ninth International Conference on Principles and Practice of Constraint Programming CP 2003*, 2003.

M. L. Littman. Initial experiments in stochastic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 667–672, 1999.

S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the 15th AAAI*, pages 954–959, 1998.

J. P. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, pages 220–227, San Jose, CA, November 1996. ACM/IEEE.

J. Pearl. *Probablistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Mateo, CA, 1988.

D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1/2):273–302, 1996.

T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

T. Sang, P. Beame, and H. Kautz. Heuristics for fast exact model counting. To appear in SAT05, 2005.

M. Shwe, B. Middleton, D. Heckerman, M. Henrion, E. Horvitz, H. Lehmann, and G. Cooper. Probabilistic diagnosis using a reformulation of the internist- 1/qmr knowledge base i. the probabilistic model and inference algorithms. *Methods of Information in Medicine*, 30:241–255, 1991.

D. J. Spiegelhalter. Probabilistic reasoning in predictive expert systems. In L. N. Kanal and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence*. Elsevier/North-Holland, 1986.

L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, 2001. ACM/IEEE.