Heuristics for Fast Exact Model Counting

Tian Sang, Paul Beame, and Henry Kautz

Computer Science and Engineering,
University of Washington, Seattle WA 98195-2350
{sang,beame,kautz}@cs.washington.edu

Abstract. An important extension of satisfiability testing is model-counting, a task that corresponds to problems such as probabilistic reasoning and computing the permanent of a Boolean matrix. We recently introduced Cachet, an exact model-counting algorithm that combines formula caching, clause learning, and component analysis. This paper reports on experiments with various techniques for improving the performance of Cachet, including component-selection strategies, variable-selection branching heuristics, randomization, backtracking schemes, and cross-component implications. The result of this work is a highly-tuned version of Cachet, the first (and currently, only) system able to exactly determine the marginal probabilities of variables in random 3-SAT formulas with 150+ variables. We use this to discover an interesting property of random formulas that does not seem to have been previously observed.

1 Introduction

The substantial progress in practical algorithms for satisfiability has opened up the possibility of solving many related, and even more difficult, logical problems. In recent work [10], we introduced Cachet, which applies techniques for practical satisfiability algorithms to the associated counting problem, #SAT, that requires the computation of the number of all satisfying assignments of a given CNF formula.

Cachet is an exact model-counting algorithm which combines formula caching [7, 2, 5], clause learning [8, 13, 14], and dynamic component analysis [4, 2, 3]. Cachet was shown to outperform other model counting algorithms on a wide range of problems.

In [10], the primary focus was on managing the component caching, and integrating it with clause learning since the combination of the two can cause subtle errors if certain cross-component implications are not controlled. Handling these problems involved techniques to flush certain cache entries and to detect and prevent cross-component implications involving learned clauses.

In this paper we examine a wide range of different techniques for improving the performance of Cachet, including component-selection strategies, variable-selection branching heuristics, backtracking schemes, and randomization. Many of these techniques have previously worked well in SAT solvers. In addition to studying these heuristics we also study the impact of using a more liberal but still sound method, suggested in [10], controlling cross-components implications involving learned clauses.

One major goal of this work beyond improving the performance of Cachet itself is to determine which of the heuristics from SAT solvers are well-suited for use in #SAT solvers in general. Our results show that some popular techniques such as randomization and aggressive non-chronological backtracking perform quite poorly when combined with component caching as in Cachet and do not appear to be particularly well-suited to use in #SAT solvers in general.

In the case of variable-selection branching heuristics, we observe that in Cachet the tradeoffs between heuristics are somewhat different than in the case of SAT solvers. Based on previous heuristics, we develop a new hybrid branching heuristic, VSADS, which appears to be a good choice for model counting algorithms involving component caching. We also observe that the right application of variable-selection heuristics is secondary to component selection, and we use a new method for selecting components that reduces the amount of wasted effort when the component cache must be flushed. Our experiments also show that the more liberal method for controlling cross-component implications has only a relatively small impact on almost all problems.

Finally, we show how our tuned version of Cachet can be extended to compute all marginal probabilities of variables in random 3-CNF formulas of 150+ variables (at sufficiently high clause-variable ratios). This allows us to discover a new pattern in these marginals. At a clause-variable ratio of roughly 3.4 the conditional probability that a randomly-chosen variable in a satisfying assignment is true is uniformly distributed between 0 and 1. Moreover we derive curves that allow us to predict these probabilities at other ratios. Such results may have explanatory power in the analysis of simple DPLL algorithms.

In the next section we give an overview of Cachet. In Section 3 we discuss the impact of branching heuristics, followed by randomization in Section 4, cross-component implications in Section 5, and non-chronological backtracking in Section 6. Finally, we discuss our methods and results in computing marginal probabilities in Section 7.

2 Overview of Cachet

Cachet, presented in [10], is a practical tool for solving #SAT. It is implemented as a modification and extension of the state-of-the-art SAT solver zChaff [14]. In addition to the 1UIP clause learning of zChaff, Cachet adds two other features that are critical to its performance: an explicit on-the-fly calculation of the connected components of the residual formula at each node in the search tree, and a cache to store the components' model counts so that they do not need to be recalculated when those components reappear later in the search.

Although SAT solvers typically eschew explicit computation of residual formulas, the higher complexity of #SAT complexity means that the sizes of the residual formulas we can deal with are smaller and the benefits of explicit computation outweigh its costs. For the #SAT problems that we can solve using Cachet, the entire overhead of maintaining the residual formulas and calculating connected components is usually roughly half of the total runtime. The learned clauses are used for unit propagations but not considered in the component computation, because their large number would make the component computation much more expensive, and because they would connect subformulas that would otherwise be disjoint components, reducing the advantage of the component decomposition.

Cached components are represented explicitly together with their values in a hash table. The size of this hash table is fixed via an input parameter, and a lazy deletion scheme based on the age of a cached entry is used to keep the table small.

As shown in [10], there can be a very subtle interaction between the component decomposition and unit propagation involving learned clauses. To avoid this, in the original version of Cachet we disallowed any unit propagation inference between two connected components of the residual formula. However, we also showed that this restriction is not strictly necessary and determined the general conditions under which such cross-component unit propagation is safe.

In the original version of Cachet, we also did not use certain features of zChaff, including non-chronological backtracking and its VSIDS variable selection heuristic. Some aspects of non-chronological backtracking as implemented in zChaff are not suitable for model counting. For example, zChaff uses unit propagation from learned clauses rather than explicitly flipping values of decision variables, which works for SAT because all previously explored branches are unsatisfiable; this is not the case for #SAT. We also happened not to use VSIDS because we were exploring heuristics that took advantage of the explicit connected component computation.

In this paper we study the range of options usually considered for SAT solvers and see how they apply in Cachet. These heuristics include branching heuristics as well as randomization and non-chronological backtracking. We also analyze the importance of cross-component implications in component caching context. Finally, we present an extension to Cachet that computes all marginals for satisfiable CNF formulas.

3 Branching

3.1 The Role of Components in Branching Decisions

At any decision-making point, Cachet explicitly maintains the residual formula determined by the current variable assignment, in the form of disjoint components. Thus, at any such point, Cachet can use this partition of variables as part of its branching decisions, information that is not usually available to SAT solvers. Moreover, because these components are disjoint, each component is largely independent of the others. (There is some cross-component information available in the form of learned clauses but, as we will see in section 5, exploiting this information does not have a major impact on the performance of the algorithm.)

Therefore, we separate branching heuristics into two parts: the choice of component and the choice of decision variable/literal within that component. The component selection strategy that the version of Cachet from [10] applied was a pure DFS strategy; that is, only a child of the most recently branched component can be selected as the next component to branch on.

If a component is satisfiable, then all of its child components are satisfiable and it does not matter which child is chosen first; eventually every child component needs to be analyzed, and cached component values are not helpful to their disjoint siblings. However, if a component is unsatisfiable, then at least one of its child components must be unsatisfiable and the values of the others are irrelevant. Naturally, it is preferable to find such a child component first to avoid unnecessary work on satisfiable children. Moreover, not only is the work done on those satisfiable child components useless, but as shown in [10], the calculated values stored in the cache for these components can actually be corrupted by the existence of their not-yet-discovered unsatisfiable sibling and must be explicitly removed from the cache.

Unfortunately, there is no easy predictor for which component will be unsatisfiable. We tried choosing the component with the largest clause/variable ratio, but that was not particularly effective. The solution we have settled on is simple: select the *smallest component first*, measured by number of variables. Because calculating the value of a smaller component is easier, if we do indeed have to abandon this work later because of an unsatisfiable sibling, the amount of wasted effort will be minimized.

We also modified the pure DFS branching order described in [10] so that Cachet attempts to discover unsatisfiable sibling components as quickly as possible. If there are a number of branchable components available, Cachet selects the smallest component C and applies the variable branching heuristics to begin the exploration of C. However, once the first satisfying assignment is found for C, further search in that component is temporarily halted and search within the next smallest remaining component is initiated from that point in the search tree. Once the last child component is found to be satisfiable its analysis is completed and the algorithm backtracks to complete the next-to-last child component, etc. If one of the child components is found to be unsatisfiable, the algorithm backtracks to the point in the search tree where the components were generated. The amount of work in the satisfiable case is still only the sum of the costs of analyzing each component and substantial work may have been saved in the unsatisfiable case.

3.2 Variable Branching Heuristics

Good variable branching heuristics can be critical to the performance of DPLL-based SAT solvers and, since Cachet is a DPLL-based #SAT solver, it is natural that its performance also depends on a good variable branching heuristic. We explore a number of the different branching heuristics available including dynamic literal count heuristics, conflict driven heuristics, and unit-propagation based heuristics. We also develop a new heuristic, VSADS, that seems to be well-suited for #SAT. All these heuristics are currently implemented in Cachet and can be selected by a command line argument. We first review these heuristics.

Literal Count Heuristics Literal count heuristics [12], make their branching decision based only on the number of occurrences of a variable in the residual formula. If the positive literal $+v$ appears V_p times and the negative literal $-v$ appears V_n times in the residual formula, using a score for variables v as either $V_p + V_n$ or $max(V_p, V_n)$ results, respectively, in the Dynamic Largest Combined Sum (DLCS) and Dynamic Largest Individual Sum (DLIS) heuristics. The highest scored v is selected as the decision variable, and its value is set to true if $V_p > V_n$, false otherwise. The goal is to eliminate as many clauses as possible without considering the impact of unit propagation.

Our original version of Cachet used only these simple heuristics, which are easy to evaluate during component detection. We tried several versions and in our experiments observed that the best was to choose the highest DLCS score with DLIS as a tie-breaker; we refer to this as DLCS-DLIS in our tables of results.

Exact Unit Propagation Count (EUPC) Heuristics Various unit-propagation-based heuristics have been widely used since early SAT solvers. Such heuristics compute the score of a variable by some magic function over the weights of its positive and negative forms, where a literal's weight is obtained by considering the amount of simplification it yields in unit propagations. Setting proper parameters for such a function is a bit of a black art. In Cachet we tested an EUPC procedure similar to that described for `relsat` [4]. To compute the score of variable v, the EUPC heuristic in ideal form will select a literal whose unit propagation will generate a conflict, and otherwise will choose the best variable score given by the following formula:

$$score(v) = |UP(+v)| \times |UP(-v)| + |UP(+v)| + |UP(-v)|$$

where $UP(\ell)$ is the number of unit propagations induced by setting ℓ to true. Evaluating exact unit propagations for many variables is very expensive, so we also use a preprocessing step as described in [4]. That is, for every variable we compute its approximate score with $|UP(+v)|$ approximated by the number of binary clauses containing literal $-v$ and $|UP(-v)|$ approximated by the number of binary clauses containing literal $+v$. Then the unit propagations and exact scores are computed only for the 10 variables with the best approximate scores.

Approximate Unit Propagation Count (AUPC) Heuristics By computing a better estimate of the amount of unit propagation that will take place, the AUPC heuristic, suggested in the paper on Berkmin [6], avoids any explicit unit propagations and can be computed more efficiently. The idea is simple: to estimate the impact of assigning $v = 0$ more correctly, not only should the binary clauses containing literal $+v$ be counted, but the binary clauses touching the literals whose negated forms are in binary clauses with v should also be counted. For example, if there is a binary clause $(-u, v)$, when estimating unit propagations resulting from assigning $v = 0$, all the binary clauses containing literal $+u$ should be counted too. The score of a variable is defined as the sum of the scores of its positive form and negative form, and the variable with highest score is chosen as decision variable.

Variable State Independent Decaying Sum (VSIDS) The VSIDS selection heuristic is one of the major successes of Chaff [9, 14]. It takes the history of the search into account but does not analyze the residual formula directly. (This is the reason for the word 'Independent' in its name.) Initially, all variable scores are their literal counts in the original formula. When a conflict is encountered, the scores of all literals in the learned conflict clause are incremented. All variable scores are divided by a constant factor periodically. The idea is to give a higher priority to the literals satisfying recent conflict clauses, which are believed to be more important and necessarily satisfied first. An advantage of VSIDS is its easy score-computing procedure, because it does not require any information from the current residual formula. In fact zChaff does not need to maintain a residual formula.

After many decaying periods, the influence of initial variable scores and old conflicts decay to negligible values and variable scores only depend on recent conflict clauses. If there are very few recent conflicts, then most variables will have very low or even 0 scores, thus decision-making can be quite random. For SAT-solving purposes, this is not a serious problem, because it probably means the formula is underconstrained and thus easily satisfied. However, in the context of model counting, it is often the case that there are few conflicts in some part of the search tree and in these parts VSIDS will make random decisions.

Variable State Aware Decaying Sum (VSADS) VSADS combines the merits of both VSIDS and DLCS. It is expressly suited for Cachet and can benefit from both conflict-driven learning and dynamic greedy heuristics based on the residual formula. Since all literal counts can be obtained during component detection with little extra overhead, there is no reason for the algorithm not to be "Aware" of this important information for decision-making when most variables have very low VSIDS scores. The VSADS score of a variable is the combined weighted sum of its VSIDS score and its DLCS score:

$$score(VSADS) = p \times score(VSIDS) + q \times score(DLCS)$$

where p and q are some constant factors. Within a component, the variable with the best VSADS score is selected as decision variable. With this derivation, VSADS is expected to be more like VSIDS when there are many conflicts discovered and more like DLCS when there are few conflicts. We report experimental results for $p = 1$ and $q = 0.5$, but the runtime is not particularly sensitive to these precise values.

Experimental Results Figure 1 shows the results of different heuristics on a number of benchmark problems, including logistics problems produced by Blackbox, satisfiable grid-pebbling problems [10], and benchmarks from SATLIB including circuit problems, flat-200 (graph coloring) and uf200 (3-SAT). The last two sets each contain 100 instances, and the average runtime and the median runtime are given respectively.

Despite being a simple combination of VSIDS and DLCS, VSADS frequently outperforms each of them alone by a large margin. The superiority of VSADS over VSIDS is particularly evident in cases in which VSIDS does not even finish. This is likely because of the random decisions that VSIDS makes when there are few conflicts. In most instances VSADS also significantly improves on DLCS alone. Unit-propagation-count based heuristics EUPC and AUPC are often quite good too, especially on flat-200, uf200 and some logistics problems, but VSADS usually outperforms them and seems to be the most stable one overall. For the remainder of our experiments we report on results using the VSADS heuristic.

4 Randomization

Randomization is commonly used in SAT solvers and appears to be helpful on many problems. In fact, every heuristic discussed before can be randomized easily. The randomization can be either on a tie-breaker among variables with the same score or a random selection of variables whose scores are close to the highest.

Problems	variables	clauses	solutions	DLCS	VSIDS	VSADS	EUPC	AUPC
Circuit								
2bitcomp_6	150	370	9.41E+20	121	43	15	92	112
2bitmax_6	252	766	2.07E+29	189	22	2	21	35
rand1	304	578	1.86E+54	9	X	23	10	16
ra	1236	11416	1.87E+286	3.2	X	3.4	8.2	3.9
rb	1854	11324	5.39E+371	7.1	X	7.5	23	7.9
rc	2472	17942	7.71E+393	172	X	189	736	271
ri	4170	32106	1.30E+719	206	78	119	1296	1571
Grid-Pebbling								
grid-pbl-8	72	121	4.46E+14	0.10	0.10	0.06	0.21	0.15
grid-pbl-9	90	154	6.95E+18	1.1	0.44	0.35	0.35	0.27
grid-pbl-10	110	191	5.94E+23	4.6	0.58	0.37	12	3.2
Logistics								
prob001	939	3785	5.64E+20	0.19	0.13	0.06	0.11	0.06
prob002	1337	24777	3.23E+10	30	10	8	16	21
prob003	1413	29487	2.80E+11	63	15	9	20	35
prob004	2303	20963	2.34E+28	116	68	44	32	133
prob005	2701	29534	7.24E+38	464	11924	331	456	215
prob012	2324	31857	8.29E+36	341	X	304	231	145
flat-200(100)	600	2337						
average	-	-	2.22E+13	102	4.5	4.8	4.6	6.8
median	-	-	4.83E+11	63	2.8	2.9	2.7	3.9
uf200(100)	200	860						
average	-	-	1.57E+9	21	7.1	7.2	3.0	4.8
median	-	-	3074825	18	6.6	6.5	2.5	3.7

Fig. 1. Runtime in seconds of Cachet on a 2.8 GHz Pentium 4 processor with 2 GB memory using various dynamic branching heuristics (X=time out after 12 hours).

We expected that randomization would be somewhat less of a help to #SAT search than for SAT search. One of the major advantages that randomization gives SAT solvers is the ability to find easily searchable portions of the space and avoid getting stuck in hard parts of the space. #SAT solvers, however, cannot avoid searching the entire space and thus might not benefit from this ability. However, it was a little surprising to us that randomization almost always hurts model counting. Figure 2 shows the impact of using randomized VSADS on some problems. VSADS-rand computes variable scores the same way as VSADS does, but it selects a decision variable randomly from the variables whose scores are within 25% of the best score. On the first three problems, we ran VSADS-rand 100 times and used statistical results. Since there are already 100 instances of randomly chosen problems in flat-200 and uf200, we just run VSADS-rand on each problem once. We tried a number of other experiments using other fractions and other heuristics such as EUPC but the results were similar. It is very clear that the effect of randomization is uniformly quite negative. In fact the minimum runtime found in 100 trials is often worse than the deterministic algorithm.

While we do not have a complete explanation for this negative effect, a major reason for this is the impact of randomization on component caching. Using randomiza-

Problems	grid-pbl-10	prob004	2bitcomp_6	flat-200(100)	uf200(100)
VSADS	0.37	44	15	4.8 / 2.9	7.2 / 6.5
VSADS-rand					
average	6.8	433	54	7.0	12
median	3.7	402	54	3.7	11
maximum	29	1073	81	N/A	N/A
minimum	0.24	145	38	N/A	N/A
STDEV	7.7	200	12	N/A	N/A

Fig. 2. Runtimes in seconds of VSADS versus randomized VSADS on selected problems.

tion would seem to lower the likelihood of getting repeated components. Consider the search tree on formula F in which there are two large residual formulas A and B that have many variables and clauses in common but are not exactly the same. Since A and B have similar structure, a deterministic branching heuristic is likely to have similar variable scores and make similar choices that could easily lead to cache hits involving subproblems of A and B. A randomized heuristic is more likely to create subproblems that diverge from each other and only leads to cache hits on smaller subformulas. Although our experiments showed similar total numbers of cache hits in using randomization, there seemed to be fewer cache hits at high levels in the search tree.

5 Cross-Component Implications

As discussed in our overview, in combining clause learning and component caching we only determine components on the residual formula, not on the the learned clauses. Learned clauses that cross between components can become unit clauses by instantiations of variables within the current component. Unit implications generated in this way are called cross-component implications.

In [10], it was shown that cross-component implications can lead to incorrect values for other components. To guarantee correctness cross-component implications were prohibited; each unit propagation from learned clauses was generated but if the variable was not in the current component the unit propagation was ignored.

However, it was also shown that any implications of literals within the current component that result from further propagations of the literals found in cross-component implications are indeed sound. In prohibiting all cross-component implications, these sound inferences that could help simplify the formula were lost.

In the current version of Cachet such sound implications of cross-component unit propagation are optionally allowed by maintaining a list of cross-component implications. Cross-component implications are detected at the unit-propagation stage, and stored in the list. When branching on a component, it is checked to see if it contains any variable in the list. If the current component has been changed by previous cross-component implications, before branching on it, a new component detection is performed over it, which will update the related data structures correctly. This solution is easy to implement but the overhead can be high, for every element of the cross-component implication list needs to be checked at every decision-making point. For-

Problems	cross-component implications	total implications	time without cross-implications	time with cross-implications
2bitcomp_6	13	3454480	15	15
rand1	6195	5131281	23	23
grid-pbl-8	25	5713	0.06	0.06
prob001	49	16671	0.06	0.06
prob002	225	474770	8	8
prob003	133	426881	9	9
prob004	8988	6575820	50	44
prob005	20607	39391726	482	331
prob012	3674	31862649	313	304
flat-200(100)				
average	277	1010086	4.9	4.8
median	239	767136	2.9	2.9

Fig. 3. Runtime in seconds of VSADS with and without cross-component implications.

tunately, the ratio of cross-component implications is very small, at most 0.14% of all implications in our tested formulas, so the overhead is negligible.

Figure 3 shows the impact of cross-component implications. We ran experiments on a much larger suite of problems than are listed; all those not shown have fewer than 5 cross-component implications, mostly none. There was one instance in which the speedup using cross-component implications was 46%, but most others were negligible. We conclude that cross-component implication is not an important factor in general.

6 Chronological vs. Non-chronological Backtracking

Non-chronological backtracking is one of the most successful techniques used in SAT solvers. When a clause is learned, the search backtracks to the highest level in the search tree at which the learned clause yields a unit propagation. All decisions made between the level at which the clause is learned and the backtrack level are abandoned since they are in some sense irrelevant to the cause of the conflict found. Since no satisfying assignments have yet been found in that subtree, the only information lost by this abandonment is the path from the backtracking destination to the conflict point, which does not take much time to recover.

However, in the model counting scenario, a direct implementation of the above non-chronological backtracking scheme would abandon work on subtrees of the search tree in which satisfying assignments have already been found and tabulated, and this tabulation would have to be re-computed. This is even worse in the component caching context in which the conflict found may be in a completely separate component from the work being abandoned. Moreover, redoing dynamic component detection and cache checking has an even more significant cost.

As discussed earlier, the basic version of Cachet does have some form of non-chronological backtracking in that finding unsatisfiable components can cause a backtrack that abandons work on sibling components. In contrast to this we use the term far-backtracking to refer to the form of non-chronological backtracking described above.

Problems	# of far backtracks	total #conflicts with far-back	total #conflicts w/o far-back	time with far-back	time w/o far-back
Circuit					
2bitcomp_6	930	1212	1393	15	15
2bitmax_6	1031	1501	1222	6	2
rand1	1270	2114	4095	18	23
ra	0	0	0	3.4	3.4
rb	101	176	220	9	7.5
rc	353	502	511	199	189
ri	13010	13070	10212	164	119
Grid-Pebbling					
grid-pbl-8	75	122	286	0.15	0.06
grid-pbl-9	275	388	773	0.22	0.35
grid-pbl-10	355	506	730	1.8	0.37
Logistics					
prob001	355	506	730	0.07	0.06
prob002	1968	2042	2416	10	8
prob003	2117	2176	2022	14	9
prob004	5657	6694	5492	1062	44
prob005	26233	31392	21951	10121	331
prob012	12020	13563	15677	2860	304
flat-200(100)					
average	6884	6958	7119	4.9	4.8
median	5150	5188	5105	3.0	2.9
uf200(100)					
average	24101	24144	26469	7.5	7.2
median	23014	23067	25778	6.9	6.5

Fig. 4. Runtimes in seconds and number of backtracks using VSADS in Cachet with and without far-backtracking.

We considered two forms of far-backtracking, the full original far-backtracking and one in which the backtrack moves up to the highest level below the far backtrack level at which the subtree does not already have a satisfying assignment found. This latter approach eliminates the problem of abandoned satisfying assignments but it does not backtrack very far and creates additional overhead. It did not make a significant difference so we do not report the numbers for it.

Figure 4 shows the comparison of Cachet with and without far-backtracking. Even with far-backtracking enabled, some of the backtracks are the same as they would be without it, and we report the number of far backtracks separately from the total number of backtracks that are due to conflicts in the case that far-backtracking is turned on. (In SAT algorithms all backtracks are due to conflicts but in model counting most backtracks involve satisfiable left subtrees.)

While far-backtracking occasionally provides a significant improvement in runtime when the input formula is indeed unsatisfiable, overall it typically performed worse than without far-backtracking. As a result, we do not use far-backtracking as the default but we allow it as an option.

Algorithm marginalizeAll

```
$marginalizeAll(\Phi, Marginals)$
// returns satisfying probability of formula $\phi$
// marginals of all variables are returned in vector $Marginals$ as well
  if $\Phi$ is empty, return 1
  if $\Phi$ has an empty clause, return 0
  $LeftValue = RightValue = 1/2$                    // initializing
  $initializeVector(LeftMarginals, 0)$
  $initializeVector(RightMarginals, 0)$
  select a variable $v$ in $\Phi$ to branch          // branching
  $extractComponents(\Phi|_{v=0})$
  for each component $\phi$ of $\Phi|_{v=0}$
    $LeftValue\ \times = marginalizeAll(\phi, LeftMarginals)$
  for each variable $x \in \Phi$
    if $x \in \Phi|_{v=0}$
      $LeftMarginals[x]\ \times = LeftValue$         // adjusting
    else
      $LeftMarginals[x] = LeftValue/2$
  $LeftMarginals[v] = 0$
  $extractComponents(\Phi|_{v=1})$
  for each component $\phi$ of $\Phi|_{v=1}$
    $RightValue\ \times = marginalizeAll(\phi, RightMarginals)$
  for each variable $x \in \Phi$
    if $x \in \Phi|_{v=1}$
      $RightMarginals[x]\ \times = RightValue$       // adjusting
    else
      $RightMarginals[x] = RightValue/2$
  $RightMarginals[v] = RightValue$
  $Marginals = sumVector(LeftMarginals, RightMarginals)$
  $Marginals\ / = (LeftValue + RightValue)$          // normalizing
  return $LeftValue + RightValue$
```

Fig. 5. Simplified version of algorithm to compute marginal probabilities of all variables.

7 Computing All Marginals

We now show how our basic exact model counting algorithm Cachet can be modified to compute marginal probabilities for all variables, that is, the fraction of satisfying assignments in which each variable is true. Although Cachet does not maintain explicit information about the satisfying assignments found, we can maintain enough statistics as we analyze each component to determine the overall marginal probabilities. The basic idea requires that each component is associated not only with a weight representing the fraction of all assignments that satisfy it but also with a vector of marginal probabilities for each of its variables.

In Figure 5 we show a simplified recursive algorithm $marginalizeAll$ that returns the count and passes up the marginal probabilities as well as the count. In this simplified version, the left branch is assumed to correspond to the assignment $v = 0$ and the parameter $Marginals$ is passed by reference. Variables in different components are disjoint, so their marginals can be calculated separately but finally need to be adjusted by the overall satisfying probability. The marginal of any variable that has disappeared in the simplified formula is just equal to half of the satisfying probability by definition. For the decision variable, only its positive branch should be counted. $LeftValue + RightValue$ is the satisfying probability of Φ and the normalizing factor for all marginals of Φ.

Though described in a recursive fashion, the real implementation of this algorithm works with component caching in the context of non-recursive backtracking that it inherits from zChaff. Moreover, in this simplified version we have ignored the issue of unit propagations. Variables following via unit propagation do not appear in the formula on which a recursive call is made so their marginals are not computed recursively but must be set based on the fraction of satisfying assignments found in the recursive call. The details of this calculation and extension to using arbitrary weights is addressed in [11] where Cachet is extended to handle Bayesian inference.

The overhead of computing all marginals is proportional to the number of variables in the components, rather than the total number of variables, because at a node in the search where the model count is returned, only those relevant variables need to be examined. But it may need a significant amount of memory for caching the marginal vectors. In this way, we are able to compute all marginals quite efficiently, usually with only 10% to 30% extra overhead if the problem fits in the memory.

7.1 Marginals of Random 3-CNF Formulas

In this section we show how the extension of Cachet for computing all marginal probabilities allows us to study new features of random 3-SAT problems. This problem has received a great deal of interest both algorithmically and theoretically. It is known experimentally that there is a sharp satisfiability threshold for random 3-SAT at a ratio of roughly 4.3 clauses per variable. However, the largest proved lower bound on the satisfiability threshold for such formulas is at ratio 3.42 [1] using a very restricted version of DPLL that does not backtrack but makes irrevocable choices as it proceeds. (In fact, almost all analyses of the lower bounds on the satisfiability thresholds for random 3-SAT are based on such restricted DPLL algorithms.)

In Figures 6 and 7 we show the the experimental cumulative distribution of marginals of random 3-CNF formulas of 75 and 150 variables respectively at different ratios. The plots are the result of running experiments with 100 random formulas at each ratio, sorting the variables by their marginals, and taking a subsample of 150 equally-spaced points in this sorted list for ease of plotting the results. Thus the X-axis represents the fraction of all variables considered and the Y-axis represents the marginal probability that a variable is true in satisfying assignments of the formula in which it appears. (Although this is plotted in aggregate, individual formulas have similar plots to these aggregate plots.)

For a cumulative distribution derived in this manner (sometimes called a QQ or quantile-quantile plot), a uniform distribution would be represented as a straight line from $(0, 0)$ to $(1, 1)$. Moreover, we can read off simple properties from these plots. For example, for 75 variables at ratio 4.1, more than 20% of variables were virtually always false in all satisfying assignments and a similar fraction were virtually always true. Since randomly chosen formulas are chosen symmetrically with respect to the signs of their literals, we should expect that the cumulative distribution functions will be symmetric about the point $(0.5, 0.5)$ as is borne out in our experiments.

Our experiments show, not surprisingly, that at low ratios the biases of variables are rarely extreme and that variables become significantly more biased as the ratios increase. At the lowest ratio, 0.6, in Figure 6, a constant fraction of variables do not appear in the formula so the flat section of the curve shows that a constant fraction of variables is completely unbiased at marginal probability 0.5.

In other plots of curves for fixed ratios and varying numbers of variables, we observed that the shape of the curves of the cumulative distribution function seemed to be nearly the same at a given ratio, independent of the number of variables. For example, at ratio 3.9, the lack of smoothness in the plots due to experimental noise almost compensated for any differences in the shapes of the curves.

One interesting property of these cumulative distribution functions is the precise ratio at which the marginal probabilities are uniform. As can be seen from both the 75 variable and 150 variable plots, this point appears to be somewhere around ratio 3.4, although it is a bit difficult to pinpoint precisely. Above this ratio, the marginal probabilities of variables are skewed more towards being biased than unbiased. It seems plausible that the distribution of marginal probabilities is particularly significant for the behavior of non-backtracking DPLL algorithms like the one analyzed in [1]. Is it merely a coincidence that the best ratio at which that algorithm succeeds is very close to the ratio at which the distribution of variable biases becomes skewed towards biased variables?

Fig. 6. Cumulative distribution (QQ plot) of marginal probabilities of variables in random 3-CNF formulas of 75 variables at various ratios

Fig. 7. Cumulative distribution (QQ plot) of marginal probabilities of variables in random 3-CNF formulas of 150 variables at ratios ≥ 3.4

8 Conclusion

Many of the techniques that apply to SAT solvers have natural counterparts in exact #SAT solvers such as Cachet but their utility in SAT solvers may not be indicative of their utility in #SAT solvers.

We have shown that popular techniques for SAT solvers such as randomization and aggressive non-chronological backtracking are often detrimental to the performance of Cachet. We have developed a new hybrid branching heuristic, VSADS, that in conjunction with a careful component selection scheme seems to be the best overall choice for Cachet. Furthermore, based on experiments, more sophisticated methods for cross-component implications appear to be of only very marginal utility.

Finally, we observed that #SAT solutions are merely the start of what can be obtained easily using Cachet by demonstrating the ability to obtain interesting results on the marginal probabilities of random 3-CNF formulas.

References

1. G. Lalas A. Kaporis, L. Kirousis. The probabilistics analysis of a greedy satisfiability algorithm. In *European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 574–585, 2002.

2. F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and Complexity Results for #SAT and Bayesian inference. In *Proceedings 44th Annual Symposium on Foundations of Computer Science*, Boston, MA, October 2003. IEEE.
3. 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.
4. Roberto J. Bayardo Jr. and Joseph D. Pehoushek. Counting models using connected components. In *Proceedings, AAAI-00: 17th National Conference on Artificial Intelligence*, pages 157–162, 2000.
5. Paul Beame, Russell Impagliazzo, Toniann Pitassi, and Nathan Segerlind. Memoization and DPLL: Formula Caching proof systems. In *Proceedings Eighteenth Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.
6. E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.
7. S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the 14th AAAI*, pages 954–959, 1998.
8. 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.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001. ACM/IEEE.
10. Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.
11. Tian Sang, Paul Beame, and Henry Kautz. Solving bayesian networks by weighted model counting. Submitted, 2005.
12. Joo P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, pages 62–74, 1999.
13. Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction, LNAI*, volume 1249, pages 272–275, July 1997.
14. 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, San Jose, CA, November 2001. ACM/IEEE.