Using More Reasoning to Improve #SAT Solving

Jessica Davies and **Fahiem Bacchus**
Department of Computer Science
University of Toronto
Toronto, Canada
[jdavies|fbacchus]@cs.toronto.edu

Abstract

Many real-world problems, including inference in Bayes Nets, can be reduced to #SAT, the problem of counting the number of models of a propositional theory. This has motivated the need for efficient #SAT solvers. Currently, such solvers utilize a modified version of DPLL that employs decomposition and caching, techniques that significantly increase the time it takes to process each node in the search space. In addition, the search space is significantly larger than when solving SAT since we must continue searching even after the first solution has been found. It has previously been demonstrated that the size of a DPLL search tree can be significantly reduced by doing more reasoning at each node. However, for SAT the reductions gained are often not worth the extra time required. In this paper we verify the hypothesis that for #SAT this balance changes. In particular, we show that additional reasoning can reduce the size of a #SAT solver's search space, that this reduction cannot always be achieved by the already utilized technique of clause learning, and that this additional reasoning can be cost effective.

Introduction

The goal of model counting, #SAT, is to determine the number of models of a propositional theory. Like SAT, the canonical NP-complete problem, #SAT is a fundamental problem that is complete for the complexity class #P (Valiant 1979). This means that many real-world problems can be reduced to #SAT, motivating the need for efficient #SAT solvers. Currently, inference in Bayesian networks is the dominant application area of #SAT solving (Li, van Beek, & Poupart 2006; Chavira & Darwiche 2006; Sang, Beame, & Kautz 2005b), but #SAT has also been successfully applied to combinatorial problems (Gomes, Sabharwal, & Selman 2006), probabilistic planning (Domshlak & Hoffmann 2006) and diagnosis (Kumar 2002).

State-of-the-art #SAT solvers utilize modified versions of DPLL, the classical backtracking search algorithm for solving SAT. The key modification that makes DPLL efficient for #SAT is the dynamic decomposition of the residual theory into components, solving those components independently, and then caching their solutions so that a component need not be solved repeatedly. It can be shown that this method, called *component caching DPLL*, can achieve a run time that is at least as good as traditional Bayesian network algorithms (e.g., variable elimination or jointrees), and can on some instances be super-polynomial faster (Bacchus, Dalmao, & Pitassi 2003). In practice, it has been demonstrated that component caching DPLL is very effective on many types of Bayes nets including those arising from linkage analysis and relational probabilistic models (Chavira & Darwiche 2005; Chavira, Darwiche, & Jaeger 2004).

Solving #SAT poses some different challenges from ordinary SAT solving. In particular, in SAT we need only search in the space of non-solutions: once a solution is found we can stop. In this non-solution space every leaf of the search tree is a conflict where some clause is falsified. Hence, the technique of clause learning can be very effective in exploiting these conflicts to minimize the time spent traversing the non-solution space. #SAT solvers, however, must traverse the solution space as well as the non-solution space. Conflicts are not encountered in the solution space so clause learning is not effective in speeding up search in this part of the space.

Another key factor is that component caching involves a significant overhead at each node of the search tree: the algorithm must examine the theory for components, compute cache keys and perform cache lookups at each node. These operations are significantly more expensive than the simple operation of unit propagation performed by SAT solvers.

Finally, #SAT solvers are often used for knowledge compilation (Huang & Darwiche 2005). This involves saving the search tree explored while solving the #SAT instance (a trace of DPLL's execution). A number of different queries can then be answered in time linear in the size of the saved search tree. Hence for this important application it can be beneficial to reduce the size of the explored search tree even if more time is consumed. This is analogous to the benefits of using higher levels of optimization in a compiler even when compilation takes more time.

In SAT it has been shown that performing more extensive inference at each node of the search tree, i.e., going beyond unit propagation, can significantly decrease the size of the explored search tree even when clause learning is not employed (Bacchus 2002). For SAT however, since nodes can be processed so quickly, the time more extensive inference requires to reduce the number of nodes searched is often

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

not worthwhile. Our hypothesis is that this balance shifts for #SAT where processing each node is already much more costly and we might want to minimize the size of the search tree in the context of compilation.

In this paper we verify this hypothesis by applying the techniques of hyper-binary resolution and equality reduction originally used for SAT in (Bacchus 2002), and show that this usually reduces the size of the explored search tree and can can often reduce the time taken when solving a #SAT instance. We observe that more reasoning at each node reduces the size of the residual theory making both component detection and caching more efficient. Furthermore, by uncovering more forced literals than unit propagation, the technique can also encourage the partitioning of the theory into disjoint components which can significantly reduce the size of the search tree. These two factors help to improve efficiency in both the solution and non-solution parts of the search space.

The paper is organized as follows. In the next section we present some necessary background. Then we present some formal results that allow us to efficiently integrate hyper-binary resolution and equality reduction with component caching. We have constructed a new #SAT solver called #2clseq by building on the 2clseq solver of (Bacchus 2002), and the next section describes some important features of the implementation. Experimental results demonstrating the effectiveness of our approach are presented next followed by some final conclusions.

Background

In this section we present DPLL with component caching for solving the #SAT problem, and discuss the precise form of extra reasoning that we propose to employ at each node of the search tree.

Model Counting Using DPLL

We assume our input is a propositional logic formula F in Conjunctive Normal Form (CNF). That is, it is a conjunction of clauses, each of which is a disjunction of literals, each of which is a propositional variable or its negation. A *truth assignment* ρ to the variables of F is a *solution* to F if ρ satisfies F (i.e., makes at least one literal in every clause true). The #SAT problem is to determine the number of truth assignments that satisfy F, $\#(F)$. The component caching DPLL algorithm shown in Algorithm 1 computes $\#(F)$.

This algorithm successively chooses an unassigned variable v to branch on (line 3), and then counts the number of solutions for $F|_v$ and $F|_{\neg v}$, where $F|_\ell$ is the simplification of F by setting ℓ to be true (line 14). The sum of these counts is $\#(F)$ (line 17). Each subproblem, $\#(F|_v)$ and $\#(F|_{\neg v})$, is solved using component caching. At line 10 we break the subproblem into components. A component C of $F|_\ell$ is a subset of $F|_\ell$'s clauses such that the clauses in C and $F|_\ell - C$ have no variable in common. This means that any truth assignment for C can be paired with any truth assignment for $F|_\ell - C$ to yield a truth assignment for $F|_\ell$: these truth assignments assign disjoint sets of variables. Hence $\#(F|_\ell) = \#(C) \times \#(F|_\ell - C)$, and

Algorithm 1: Component Caching DPLL

```
1  #DPLL ($F$)
2  begin
3      choose (a variable $v \in F$)
4      foreach $\ell \in \{v, \neg v\}$ do
5          $F|_\ell =$ simplify($F$,$\ell$)
6          if $F|_\ell$ contains an empty clause then
7              count[$\ell$] = 0 % clause learning can be done here.
8          else
9              count[$\ell$] = 1
10             $\mathcal{C} =$ findComponents($F|_\ell$)
11             foreach $C_i \in \mathcal{C}$ while count[$\ell$] $\neq$ 0 do
12                 cN = getCachedValue($C_i$)
13                 if cN == NOT FOUND then
14                     cN = #DPLL ($C_i$)
15                 count[$\ell$] = count[$\ell$] $\times$ cN
16     addCachedValue($F$, count[$v$] + count[$\neg v$])
17     return (count[$v$] + count[$\neg v$])
18 end
```

we can solve each component independently (lines 11–15). For each component we first examine the cache to see if this component has been encountered and solved earlier in the search (line 12). If the component value is not in the cache we solve it with a recursive call, keeping a running product of the solved values (line 15). If any component C_i has $\#(C_i) = 0$ we can stop: $\#F|_\ell$ must also be equal to zero. Finally, $\#(F) = \#(F|_v) + \#(F|_{\neg v})$, so we return that value at line 17 after first inserting it into the cache.

Hyper-Binary Resolution

Typically the simplification employed at line 5 is unit propagation (UP), which involves assigning all literals appearing in clauses of length one the value true, simplifying the resulting theory, and continuing until the theory contains no more unit clauses. In this paper we suggest employing a more powerful form of simplification based on the approach utilized in (Bacchus 2002).

Definition 1 *The* ***Hyper-Binary Resolution*** *(HBR) rule of inference takes as input a $k+1$-ary clause $(l_1, l_2, \ldots, l_k, x)$ and k binary clauses each of the form $(\neg l_i, y)$ $(1 \leq i \leq k)$ and deduces the clause (x, y).*

It can be noted that HBR is simply a sequence of resolution steps compressed into a single rule of inference. Its main advantage is that instead of generating lengthy clauses it only generates binary or unary clauses (unary when $x = y$). When $k = 1$ HBR is simply the ordinary resolution of binary clauses.

Definition 2 *The* ***equality reduction*** *of a CNF F by a pair of equivalent literals $\ell_1 \equiv \ell_2$ is a rule of inference that takes as input the pair of binary clauses ($\neg\ell_1$, ℓ_2) and (ℓ_1, $\neg\ell_2$) and a CNF F and infers the simplified CNF F' formed by (a) replacing in each clause all instances of ℓ_2 by ℓ_1 and $\neg\ell_2$ by $\neg\ell_1$, (b) deleting all duplicate copies of ℓ_1 and $\neg\ell_1$ from the resulting clauses, and (c) removing all of the resulting clauses that contain ℓ_1 and $\neg\ell_1$.*

HBR is a powerful rule of inference that can discover many forced literals. To understand its power consider the *failed-literal* test. This is a test that assigns some literal l the value true and then uses unit propagation to determine if the CNF F yields a contradiction under this assignment. If a contradiction is found then F forces $\neg l$, and we can reduce F by making $\neg l$ true.

We say that a CNF F has been HBR-closed if we have applied HBR and unit propagation until the theory contains no more units and HBR cannot infer any new clauses. It has been shown (Bacchus 2002) that achieving HBR-closure uncovers the same set of forced literals as the exhaustive application of the failed literal test. That is, we must apply the failed-literal test to all literals, and if any literal is found to be forced, we must *reapply* the failed literal test to all remaining literals, continuing until the theory contains no more failed literals. The advantage of HBR-closure is that it can be computed much more efficiently than such an exhaustive application of the failed-literal test.

This means that HBR has the power to considerably reduce the size of the CNF—every forced literal reduces the number or size of the remaining clauses. Equality reduction can further reduce the size of the theory. We use HBR$^=$ to denote the application of HBR, equality reduction, and unit propagation; we say that the CNF F has been **HBR$^=$ closed** if we have applied these inference rules until no more changes can be made to the formula.

Model Counting with HBR$^=$

Critical Overheads in #DPLL

Solving #SAT requires much more computation than SAT solving. First, we must find components (line 10) at each node. This can be done in time linear in the size of the simplified theory $F|_\ell$. Hence a smaller theory reduces the overhead of this step.

Second, components must be looked up and stored in a cache. This involves both space to store the cached components and time to do look up in the cache. Relatively succinct representations of the component can be achieved by exploiting the fact that every component consists of (a) a set of variables V and (b) a subset C of the clauses in the original CNF. In particular, the subset of clauses C has been reduced so that the only literals that remain in them are literals over the variables V. Thus if each clause in the original CNF is assigned a unique index, a component can be uniquely represented by its set of clause indices and its set of variables. We can further reduce the set of clause indices by excluding the index of any binary clause (Thurley 2006) without affecting the soundness of the representation. The resulting sequence of clause indices and variables indices can then be stored in the cache and common hashing techniques can be used for cache lookup. Again, the components of a smaller theory will have a shorter representation, that occupies less space in the cache and is more efficient to calculate.[1]

Finally, the explored search tree is much larger in #SAT since we must solve both subproblems $F|_v$ and $F|_{\neg v}$, whereas a SAT solver can stop as soon as it determines that one of these is satisfiable. Nevertheless, efforts to reduce the search space's size can can pay even larger dividends in #SAT than with SAT solving. For example, current #SAT solvers employ clause learning and heuristics that tend to promote the splitting of the theory into components.

Simplification by HBR$^=$closure

In this paper we propose the use of additional reasoning during the simplification step (line 5). Specifically, our simplification step achieves HBR$^=$ closure. As noted above this can uncover more forced literals than ordinary unit propagation, and each forced literal reduces the size of the theory which lowers the overhead of finding components and caching. Of course, achieving HBR$^=$ closure takes time so the reduction in component detection and caching overheads might or might not result in a net decrease in computation. More importantly, HBR$^=$ closure can generate additional partitions that might reduce the size of the explored search space.

Example 1 Consider the theory $F = C_z \cup C_y \cup C_{a,b,c} \cup \{(z,x), (y,x), (a,x), (b,x), (c,x), (\neg a, \neg b, \neg c)\}$, where C_z, C_y and $C_{a,b,c}$ are clauses connected to z, y and $\{a,b,c\}$ respectively. HBR is able to detect that x is forced in this theory and can thus reduce F to $C_z \cup C_y \cup \big(C_{a,b,c} \cup (\neg a, \neg b, \neg c)\big)$. These are all disjoint components that can be solved independently. Note that unit propagation is not able to detect that x is forced. Of course if the search was to branch on x these components would also be generated. However, heuristics for making branching decisions are never perfect and will often miss opportunities like this.

As mentioned above current #SAT solvers utilize clause learning. Clause learning is often able to discover forced literals. However, it is only capable of finding forced literals when a conflict is generated. In fact, HBR$^=$ can sometimes find forced literals that clause learning might not discover, as demonstrated by the following example.

Example 2 Consider the theory $F = \{\, (a,x), (b,x), (c,x), (\neg a, \neg b, \neg c)\}$. Once again HBR$^=$ can discover that x is forced by F. However, if the search branches on the variables in the sequence a, b, c then x will be forced by unit propagation along every branch and no conflicts will be generated. That is, clause learning will not be able to learn that x is forced under this sequence of branch decisions. As in the previous example, x can be discovered to be forced if the search chooses to branch on it first. However, the heuristics might not make such a choice.

Integrating HBR$^=$ and Component Caching

Having justified the potential of HBR$^=$ simplification for #SAT we now turn to the mechanics of its integration with component caching.

First we examine its interaction with the dynamic generation of components during search.

[1] In practice, the size of the cache is limited. Hence, some policy has to be employed to prune less useful entries when the cache becomes too large during the search. However, such pruning only affects efficiency not correctness. In particular, if a component is no longer in the cache the search simply has to recompute its value.

Theorem 1 *Let F be any CNF, HBR$^=(F)$ be the simplification of F by achieving HBR$^=$ closure, ℓ be any literal, and $F|_\ell$ be the simplification of F by setting ℓ to be true and performing at least unit propagation.*

1. *If variables x and y are in different components of $F|_\ell$, then they are in different components of HBR$^=(F)|_\ell$ if neither has had its value forced.*
2. *Let c be a clause derivable by resolution from F. It can be the case that x and y are in the same component of $(F \cup c)|_\ell$, even though they are in different components of $F|_\ell$.*

This theorem can be proved by considering the new binary clauses (l_1, l_2) that can be inferred by HBR, and then by case analysis showing that these clauses cannot increase connectivity after ℓ is made true and unit propagation is performed. For equality reduction a semantic argument can be used. If $F \models x \equiv y$ then x and y can never be in different components as their truth assignments are not independent.

This theorem says that applying HBR$^=$ inference to F cannot adversely affect how F decomposes as we make decisions during the search. That is, if F without HBR$^=$ inference would break into components $C_1, \ldots, C_k$ after the decision to make ℓ true, then it would break into at least as many components if we first applied HBR$^=$ inference. As Example 1 shows it might break up into even smaller components due to the detection of forced literals. In contrast, clause learning (a resolution process) can learn clauses that block dynamic decomposition.

The issue of learnt clauses blocking dynamic decomposition is dealt with in clause learning #SAT solvers by ignoring the learnt clauses when finding components (a technique shown to be sound in (Sang *et al.* 2004)). However, this option is not available when we apply HBR$^=$ since equality reduction changes the original clauses. Hence without the theorem we would have no guarantee that HBR$^=$ cannot prevent decomposition.

Second we examine the integration of HBR$^=$ with caching. As mentioned above a component can be uniquely represented by its set of variables (all of which are unassigned at the time the component is created) and the indices of the original clauses it contains. Furthermore, as shown in (Thurley 2006) we can ignore the original *binary* clauses it contains (as long as at least unit propagation is being used after every decision). Unfortunately, under equality reduction this representation is no longer sufficient to identify a component. In particular, we need to know which variables are equivalent to the component's variables. Hence, we add an equality mapping to the component representation. With this addition it can be proved that the representation is sound. That is, if two components C_1 and C_2 have the same representation then they must be equivalent.[2] Since the equality mapping must now also match for us to obtain a cache hit it becomes important to do equality reduction under a fixed ordering. That is, it becomes important to always replace x by y (or vice-versa) whenever we detect that $x \equiv y$, rather than sometimes replacing y by x and sometimes x by y. In our implementation we always replace the higher numbered literal by the lower numbered literal when doing equality reduction.

[2]Note that irrespective of equality reduction this representation does not have the guarantee that two equivalent components always have the same representation. That is, this is not an if and only if condition. This arises from the fact that two different original clauses can become equivalent under different partial assignments while their clause indices always remain different.

Empirical Results

We have implemented the approach described above in a new #SAT solver that we call #2clseq. The implementation was accomplished by extending the SAT solver 2clseq (Bacchus 2002) which computes HBR$^=$ closure at each node. We added component detection and caching, changed the search to find all solutions and the heuristics to be more suitable for #SAT. However, we have not yet implemented clause learning, which would probably increase the solver's performance. This also impacted the heuristics we could implement. In particular, the current best performing heuristic for dynamic decomposition (VSADS (Sang, Beame, & Kautz 2005a)) relies on a VSIDS score computed during clause learning. Hence VSADS was unavailable to us. Nevertheless, as our results demonstrate, the approach can still obtain good results even with these current deficiencies in the implementation.

On line 11, #2clseq processes the components in order from largest to smallest clause/variable ratio. This encourages unsatisfiable components to be found early on, minimizing the amount of work that must be abandoned. For components with few variables, the overhead of partitioning and caching is not worthwhile so these operations are turned off for components with fewer than 30 variables. Similarly, theories with a high clause/variable ratio are unlikely to split into disjoint components, so components with a ratio higher than 10 are solved without trying to partition.

We compared our approach with the #SAT solvers SharpSat (Thurley 2006), Cachet (Sang *et al.* 2004), and C2D (Darwiche 2002). Cachet implements component caching DPLL with clause learning and the VSADS heuristic and is built on top of the Zchaff (Moskewicz *et al.* 2001) SAT solver. SharpSat is very similar to Cachet but also employs selective failed literal tests at each node. Finally, C2D is quite different in that it specifically tries to compile the #SAT problem rather than simply solve it. Thus it incurs some additional overhead as compared with the other solvers. Additionally, C2D uses a static decomposition of the theory. When run with a single default mechanism for generating its decomposition, as we did here, it is typically outperformed by the other solvers.

We experimented with a total of 94 satisfiable instances from 10 benchmarks families—**logistics** (3 instances), **bmc** (7), **aim** (32), **ais** (4), **blocksworld** (7), **parity** (10), from (Hoos & Stützle 2000); **ISCAS85** (5), **Grid-Pebbling** (5), **circuits** (7), and **Plan Recognition** (12). These benchmarks have previously been used to evaluate #SAT solvers. We discarded instances that could be solved by all four solvers within 3 seconds, including all of the **aim** and **parity** families, and those that could not be solved by any solver within

Problem	vars	clauses	solns	Cachet (s)	Cachet	C2D (s)	sharpSAT (s)	sharpSAT	#2csleq (s)	#2clseq (# dec.)
					(# dec.)			(# dec.)	(no HBR=)	(no HBR=)
logistics.a	828	6718	3.78E+14	5.29	16248	X	0.33	3527	**1.48** (5.62)	40956 (346344)
logistics.b	843	7301	4.53E+23	17.22	65413	1223.02	2.43	9207	**13.01** (38.64)	683983 (7450661)
logistics.c	1141	10719	3.98E+24	1023.02	3843352	X	395.57	2872878	**524.74** (1565.72)	13378935 (93910938)
bmc-ibm-1	9685	55870	7.33E+300	50.24	49493	337.7	10.4	10197	104.64 (1017.44)	35136 (165393)
bmc-ibm-2	2810	11683	1.33E+19	0.09	289	X	0.08	141	**0.05** (0.06)	626 (1672)
bmc-ibm-3	14930	72106	2.47E+19	71.04	5055	1218.94	13.68	961	**64.87** (X)	4264 (12288)
bmc-ibm-4	28161	139716	9.73E+79	X	1949295	1904.81	28.63	15020	**7.2** (9.47)	**9464** (5276)
bmc-ibm-5	9396	41207	2.46E+171	370.55	324382	766.19	760.04	1304747	**115.24** (260.02)	**277374** (87314)
bmc-ibm-11	32109	150027	3.53E+74	X	562830	X	3570.48	442839	X (X)	138240 (9984)
bmc-ibm-12	39598	194778	2.10E+112	1045.89	56266	X	214.82	9309	2485.68 (X)	65561 (1280)
ais10	181	3151	296	42.75	80220	20.11	7.46	32980	**15.96** (37.68)	**20549** (89206)
ais12	265	5666	1328	3737.96	1965013	X	298.94	1016201	**826.22** (1821.55)	**584693** (2359645)
bw_large.a	459	4675	1	0.04	19	10.88	0.04	0	**0** (0)	0 (0)
bw_large.b	1087	13772	2	0.47	192	62.8	0.17	9	**0.12** (0.12)	**0** (0)
bw_large.c	3016	50457	6	23.92	3151	411.34	2.22	56	**2.59** (7.94)	**4** (90)
bw_large.d	6325	131973	106	556.02	27118	1333.88	175.12	2337	**167.26** (1874.11)	**247** (4756)
c1355	555	1546	2.20E+12	X	19803730	15	X		X (X)	(6569472)
c1908	751	2053	8.59E+9	2390.61	8303298	136.13	2078.25	10300457	**560.48** (1411.19)	26588756 (3156866)
c432	196	514	6.87E+10	0.06	812	2.07	0.04	812	2.04 (0.34)	27092 (10410)
c499	243	714	2.20E+12	X	29624647	7.29	X		**3310.01** (1100.84)	**721826987** (14971828)
c880	417	1060	1.15E+18	X	32464441	353.23	3875.51	49910160	X (X)	(76135680)
grid-pbl-8	72	121	4.46E+14	0.08	1049	0.86	0.05	1113	17.31 (15.4)	6803134 (6450688)
grid-pbl-9	90	154	6.95E+18	0.36	6183	1.17	0.12	4001	13.54 (13.76)	5142882 (5396986)
grid-pbl-10	110	191	5.94E+23	0.41	5000	1.32	0.22	6779	14.47 (9.56)	3433105 (2053175)
grid-pbl15	240	436	3.01E+54	X	27732140	12.11	X		X (X)	357943040 (483652864)
grid-pbl-20	420	781	5.06E+95	X	23280930	1267.02	X		X (X)	1228682752 (59395840)
2bitcomp_6	150	370	9.41E+20	12.85	515568	6.08	7.73	442447	172.95 (169.68)	105986956 (104262748)
2bitmax_6	252	766	2.07E+29	1.76	53856	11.36	1.7	65798	10.58 (10.63)	6197516 (6197668)
ra	1236	11416	1.87E+286	2.43	30402	68.4	1.4	34154	72.19 (69.02)	39348464 (39348464)
rand1	304	578	1.86E+54	20.07	491075	8.98	83.96	2056985	31.24 (36.37)	9957940 (11385334)
rb	1854	11324	5.39E+371	4.99	58776	801.71	9.53	77751	345.51 (316.55)	84953614 (83478044)
rc	2472	17942	7.71E+393	104.1	788331	966.05	88.69	849671	X (X)	233359616 (244355072)
ri	4170	32106	1.30E+719	84.7	10486	188.6	19.84	9326	X (X)	15616 (31744)
4step	165	418	8.64E+4	0.03	55	2.26	0.02	43	**0.01** (0)	44 (76)
prob001	939	3785	5.64E+20	0.07	589	13.47	0.05	418	**0.04** (0.06)	1484 (4422)
prob002	1337	24777	3.23E+10	5.39	3289	115.54	0.6	1954	**1.58** (5.39)	6944 (5962)
prob003	1413	29487	2.80E+11	6.47	2415	59.26	0.39	854	**2.15** (8.83)	8744 (8943)
prob004	2303	20963	2.34E+28	23.89	70003	264.29	6.17	65934	73.67 (3803.63)	834548 (741683)
prob005	2701	29534	7.24E+38	190.58	654126	X	243.76	1915292	X (X)	2067200 (436480)
prob0012	2324	31857	8.29E+36	150.64	389174	X	46.57	539999	441.58 (X)	4496239 (1190144)
tire-1	352	1038	7.26E+8	0.08	1005	4.14	0.1	1187	**0.05** (0.05)	3796 (4368)
tire-2	550	2001	7.39E+11	0.06	486	6.79	0.05	374	**0.03** (0.03)	1008 (964)
tire-3	577	2004	2.23E+11	0.16	1854	5.98	0.06	1347	0.21 (0.22)	30686 (30330)
tire-4	812	3222	1.03E+14	0.41	3451	9.9	0.1	1668	**0.22** (0.26)	19144 (10556)

Figure 1: Comparisons of run times and number of decisions for Cachet, C2D, sharpSAT and #2clseq on structured problems. An X indicates where the run time exceeded the timeout.

5000 seconds, leaving 44 instances whose results are presented in the table.

All tests were run on 2.20GHz Opteron machines having 1GB of RAM (SHARCNET 2007). A timeout of 5000 seconds was set for every solver run, and all solvers were run with their cache size set to 512MB. The run times of Cachet, C2D, sharpSAT and #2clseq are presented in the table, along with the number of decisions for each solver other than C2D (C2D does not report this statistic). Results for #2clseq with dynamic HBR$^=$ reasoning turned off (HBR$^=$ is still performed statically as a preprocessing step) are also shown in brackets. Instances on which #2clseq made fewer decisions than both Cachet and sharpSAT are highlighted in bold text. Cases where #2clseq had a faster run time than at least two of the other solvers are also highlighted in the table.

The results show that there is a great variability in the performance of all of these solvers—each solver being fastest on some instance. This is indicative of the sensitivity of #SAT solving to heuristics. #2clseq's performance often suffered because of its poor heuristic choice. In particular, on most of the problems where it was slower (e.g., the Grid-Pebbling problems) it explored significantly more nodes. This was probably due to the fact that clause learning has not yet been implemented in #2clseq, and as mentioned above the VSADS heuristic was not available to it. In particular, if #2clseq was to make precisely the same variable ordering choices, it must generate a smaller search tree. As shown above the HBR$^=$ reasoning #2clseq performs can only force more variables, increase partitioning, and further reduce the size of the search space. However, the potential of the approach is demonstrated in some of the problems (shown in bold) where #2clseq is able to solve the problem exploring a significantly smaller search tree, e.g., bmc-ibm-4, bmc-ibm-5, and the ais and blocksworld families. In these cases, #2clseq is also often able to solve the problem faster.

Comparing the results of #2clseq with and without HBR$^=$, we see that HBR$^=$ reduces either the number of decisions or the runtime on 25 of the 44 instances. The number of decisions is sometimes slightly lower without HBR$^=$, which can happen if the variable ordering heuristic makes different choices. However, #2clseq without HBR$^=$ never makes significantly fewer decisions except in one case (c499). Furthermore, #2clseq frequently searches more nodes than SharpSat or Cachet in the same amount of time (e.g. c1908, the logistics family), demonstrating that HBR$^=$ reasoning requires little additional overhead.

Overall, #2clseq's performance is comparable with that of Cachet, better than C2D, but not as good as sharpSAT. However, sharpSAT has the advantage of being a well optimized implementation, where as our implementation of #2clseq is far from optimized.

Conclusions

We have presented a new technique for improving #SAT solvers. The technique involves additional reasoning to simplify the theory after each decision is made. We presented intuitive arguments as to why such simplification was useful, and theoretical results indicating that (a) it can never hurt and (b) it integrates well with component caching. Our empirical results are mixed, but they do indicate that the approach has potential. Empirically, our implementation is currently missing clause learning and the resulting heuristics that this technique can provide. With that addition we are optimistic that the approach would demonstrate better performance in practice. Nevertheless, it already achieves performance comparable to some of the existing #SAT solvers.

References

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *Proceedings of the 44th Annual IEEE Symposium on the Foundations of Computer Science*, 340–351.

Bacchus, F. 2002. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *Proceedings of the 18th National Conference on Artificial Intelligence*, 613–619.

Chavira, M., and Darwiche, A. 2005. Compiling Bayesian Networks with Local Structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 1306–1312.

Chavira, M., and Darwiche, A. 2006. Encoding CNFs to Empower Component Analysis. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, 61–74.

Chavira, M.; Darwiche, A.; and Jaeger, M. 2004. Compiling Relational Bayesian Networks for Exact Inference. In *Proceedings of the 2nd European Workshop on Probabilistic Graphical Models*, 49–56.

Darwiche, A. 2002. A Compiler for Deterministic Decomposable Negation Normal Form. In *Proceedings of the 18th National Conference on Artificial Intelligence*, 627–634.

Domshlak, C., and Hoffmann, J. 2006. Fast Probabilistic Planning Through Weighted Model Counting. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*, 243–252.

Gomes, C.; Sabharwal, A.; and Selman, B. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proceedings of the 21st National Conference on Artificial Intelligence*, 54–61.

Hoos, H., and Stützle, T. 2000. SATLIB: An Online Resource for Research on SAT. *SAT* 283–292.

Huang, J., and Darwiche, A. 2005. DPLL with a Trace: From SAT to Knowledge Compilation. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 156–162.

Kumar, T. K. S. 2002. A Model Counting Characterization of Diagnoses. In *Proceedings of the 13th International Workshop on Principles of Diagnosis*, 70–76.

Li, W.; van Beek, P.; and Poupart, P. 2006. Performing Incremental Bayesian Inference by Dynamic Model Counting. In *Proceedings of the 21st National Conference on Artificial Intelligence*, 1173–1179.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Conference on Design Automation*, 530–535.

Sang, T.; Beame, P.; and Kautz, H. 2005a. Heuristics for Fast Exact Model Counting. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 226–240.

Sang, T.; Beame, P.; and Kautz, H. 2005b. Performing Bayesian Inference by Weighted Model Counting. In *Proceedings of the 20th National Conference on Artificial Intelligence*, 475–482.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H.; and Pitassi, T. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, 20–28.

SHARCNET. 2007. Shared Hierarchical Academic Research Computing Network. http://www.sharcnet.ca.

Thurley, M. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCPs. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, 424–429.

Valiant, L. G. 1979. The Complexity of Computing the Permanent. *Theoretical Computer Science* 8:189–201.