Counting Models using Connected Components

Roberto J. Bayardo Jr.
IBM Almaden Research Center
bayardo@alum.mit.edu
http://www.almaden.ibm.com/cs/people/bayardo

J. D. Pehoushek
M.U.S.T. Centre
danpeh@yahoo.com

Abstract

Recent work by Birnbaum & Lozinskii [1999] demonstrated that a clever yet simple extension of the well-known Davis-Putnam procedure for solving instances of propositional satisfiability yields an efficient scheme for counting the number of satisfying assignments (models). We present a new extension, based on recursively identifying connected constraint-graph components, that substantially improves counting performance on random 3-SAT instances as well as benchmark instances from the SATLIB and Beijing suites. In addition, from a structure-based perspective of worst-case complexity, while polynomial time satisfiability checking is known to require only a backtrack search algorithm enhanced with nogood learning, we show that polynomial time counting using backtrack search requires an additional enhancement: good learning.

Introduction

Many practical problems from a variety of domains, most notably planning [Kautz & Selman 1996], have been efficiently solved by formulating them as instances of propositional satisfiability (SAT) and applying any of a number of freely available SAT algorithms. The problem of counting the number of models of a propositional formula (#SAT) has also been shown to have numerous applications [Roth 1996], though fast algorithms for this problem are not yet widely available. A recent paper by Birnbaum and Lozinskii [1999] may help change this situation, since it demonstrates that the Davis-Putnam (DP) algorithm [Davis et al. 1962] (for which freely available implementations are commonplace) can be straightforwardly extended to count models by identifying when subproblems contain no unsatisfied clauses; they call the resulting algorithm CDP.

In this paper, we describe an alternative modification of Davis-Putnam for more efficient model counting. We show that our approach yields the most significant improvements over CDP (orders of magnitude) on real world instances with many solutions, though we also witness large improvements on artificial instances such as those from the random 3-SAT problem space.

The basic idea behind our approach is as follows: by identifying connected components in the constraint graph of a SAT instance, the number of models can be determined by multiplying together the number of models of each subproblem corresponding to a connected component. This is a straightforward consequence of the fact that each subproblem corresponding to a connected component is completely independent of the others. We apply this idea recursively as Davis-Putnam builds a partial satisfying assignment.

The idea of recursively exploiting connected components in solving instances of SAT or the more general constraint satisfaction problem is not entirely new [Freuder & Quinn 1985; Bayardo & Miranker 1995]. In these formulations, however, connected components are identified using the full constraint graph, prior to attempting any solution of the instance. While this method is well-suited for obtaining worst-case complexity bounds for determining satisfiability given structure restricted instances, it offers few opportunities for good average-case performance when the initial constraint graph is dense, as is often the case with SAT (due to the typical abundance of non-binary constraints).

In contrast, our algorithm exploits components dynamically within a Davis-Putnam procedure. The Davis-Putnam procedure attempts to extend a partial solution of the input instance into a full solution. With each new extension, several clauses may be satisfied, and the constraint graph simplifies dynamically in a manner dependent upon the current variable assignments. By applying component identification dynamically, our algorithm is able to fully exploit this simplified structure. The advantage of this dynamic decomposition technique over static decomposition schemes is analogous to that of dynamic over static variable ordering.

The idea of dynamic component detection and exploitation has been proposed before by Rymon [1994], though within a set-enumeration tree search algorithm for identifying prime implicants of a propositional formula. To our knowledge, the technique has not been previously applied within a Davis-Putnam algorithm for the purpose of model counting. We note, however, that non-chronological backtracking schemes such as CBJ [Prosser 1993] and graph-based backjumping [Dechter 1987] effectively exploit component structure while backing up from a contradiction/ dead-end.

Our primary contribution, then, is the demonstration that the dynamic detection and exploitation of connected components within a Davis-Putnam procedure is an efficient technique for model counting across a wide range of SAT instances. We also discuss how to optimize the technique for instances containing a mixture of both over and under-constrained subproblems (such instances are common in the real world). Our implementation is an extension of the relsat algorithm from Bayardo & Schrag [1997], and it appears in the relsat v2.00 release available from the web page of the first author.

In addition to the experimental contributions above, we look at the complexity of counting models in the presence of structure-restricted instances of the constraint-satisfaction problem (of which SAT is a simple restriction). Even though we find that model counting with a backtrack search algorithm can be significantly harder than satisfiability checking in practice, we show that with proper learning enhancements of the backtracking algorithm, satisfiability and model counting have equivalent worst-case runtime. However, while determining satisfiability in polynomial time requires only that a backtrack algorithm be enhanced with the ability to record nogoods, we show this is not sufficient for polynomial-time counting. In addition to the ability to record nogoods, polynomial time counting requires that *goods* be recorded as well. We discuss implementation difficulties which must be addressed before good learning can be applied efficiently in practice.

Definitions

A propositional logic *variable* ranges over the domain $\{\text{true}, \text{false}\}$. An *assignment* is a mapping of these values to variables. A *literal* is the occurrence of a variable, e.g. x, or its negation, e.g. $\neg x$; a positive literal x is satisfied when the variable x is assigned true, and a negative literal $\neg x$ is satisfied when x is assigned false. A *clause* is a simple disjunction of literals, e.g. $(x \vee y \vee \neg z)$; a clause is satisfied when one or more of its literals is satisfied. A *unit clause* contains exactly one variable, and a *binary clause* contains exactly two. The *empty clause* $(\ )$ signals a contradiction (seen in the interpretation, “choose one or more literals to be true from among none”). A *conjunctive normal formula* (CNF) is a conjunction of clauses (e.g. $(a \vee b) \wedge (x \vee y \vee \neg z)$); a CNF is satisfied if all of its clauses are satisfied.

A *model* of a CNF is an assignment mentioning every variable in the CNF that satisfies every clause. For a given CNF, we consider the problems of determining satisfiability (determining if a model exists) and counting the number of models of the instance (SAT and #SAT respectively).

The *constraint graph* of a SAT instance is obtained by representing each variable with a node, and imposing an edge between any pair of variables appearing in the same clause. A connected component of a graph is a maximal subgraph such that for every pair of nodes u, v in the subgraph, there is a path from u to v. All connected components of a graph can be easily identified in linear time in the size of the graph ($O(n + m)$) using a simple depth-first traversal [Melhorn 1984].

Basic Algorithm Description

Before describing our algorithm, we first introduce the counting Davis-Putnam proof procedure (CDP) of Birnbaum & Lozinskii, which appears in the following figure. The algorithm maintains a satisfying truth assignment σ, which is empty upon initial top-level entry to the recursive,

```
CDP($F$, $\sigma$, $n$)
  UNIT-PROPAGATE($F$, $\sigma$)
  **if** $(\ )$ in $F$ **then return** 0
  **if** $F = \varnothing$ **then return** $2^{n - |\sigma|}$
  $\alpha \leftarrow$ SELECT-BRANCH-VARIABLE($F$)
  **return** CDP($F \cup \{(\alpha)\}, \sigma \cup \{\alpha\}, n$) +
         CDP($F \cup \{(\neg\alpha)\}, \sigma \cup \{\neg\alpha\}, n$)
```

call-by-value procedure. It also accepts the number of variables present in the initial formula, n.

CDP exploits the fact that when all clauses in the formula are satisfied by the partial assignment, then any variables remaining unassigned can be assigned a value arbitrarily in order to obtain a solution. Thus, if there are i remaining unassigned variables once all clauses become satisfied, then there are 2^i unique models that include the given partial assignment, so CDP immediately returns 2^i. This step and the omission of the pure literal rule are the only differences between CPD and the classic Davis-Putnam proof procedure (DP). The pure literal rule is omitted since it cannot be used when counting models or enumerating all solutions without potentially compromising completeness.

Like any good implementation of DP, CDP employs a heuristic driven branch-selection function, SELECT-BRANCH-VARIABLE. It also performs unit-propagation (UNIT-PROPAGATE) to immediately assign variables that are mentioned in some unary clause. The CNF F and the truth assignment σ are modified in calls to UNIT-PROPAGATE. This function adds the single literal λ from a unit clause ω to the literal set σ, then it simplifies the CNF by removing any clauses in which λ occurs, and shortens any clauses in which $\neg\lambda$ occurs through resolution. After unit propagation, if F contains a contradiction, then the current subproblem has no models and backtracking is necessary.

```
UNIT-PROPAGATE($F$, $\sigma$)
  **while** (exists $\omega$ in $F$ where $\omega = (\lambda)$)
      $\sigma \leftarrow \sigma \cup \{\lambda\}$
      $F \leftarrow$ SIMPLIFY($F$)
```

Our procedure, which we call DDP for Decomposing-Davis-Putnam, appears in the pseudo-code on the following page. Like CDP, DDP is based on DP, employing the same unit propagation and branch selection functions. Following each full unit propagation step, the constraint graph of the resulting simplified formula is constructed and its connected components identified.[1] Each subproblem corresponding to a component is then attempted recursively. Their solution counts are finally multiplied together to obtain the solution count for the given formula.

1. Our implementation actually determines the component structure lazily while backtracking instead of eagerly before branch selection. Though the effect in terms of search space explored is the same, this implementation detail simplifies some of the remaining optimizations, and also prevents component detection overhead from reducing performance on unsatisfiable instances.

```
DDP($F$, $\sigma$)
  UNIT-PROPAGATE($F$, $\sigma$)
  if $(\ )$ in $F$ then return 0
  if all variables are assigned a value then return 1
  Identify independent subproblems $F_1 \ldots F_j$
     corresponding to connected components of $F$.
  for each subproblem $F_i$, $i = 1 \ldots j$ do
     $\alpha \leftarrow$ SELECT-BRANCH-VARIABLE($F_i$)
     $c_i \leftarrow$ DDP($F_i \cup \{(\alpha)\}, \sigma \cup \{\alpha\}$) +
           DDP($F_i \cup \{(\neg\alpha)\}, \sigma \cup \{\neg\alpha\}$)
  return $\prod_{i = 1 \ldots j} c_i$
```

Optimizations

One modification of “plain vanilla” DDP which can provide substantial performance improvements, particularly for instances with a mixture of both under and over-constrained subproblems, is to attempt the component subproblems of F in a carefully determined order. The idea here exploits the fact that if any subproblem of F turns out to have no solutions, then the overall model count for F is inevitably zero. Thus, by attempting the most-constrained subproblems first, we often avoid futile counting within the under-constrained subproblems in the presence of an unsatisfiable subproblem. Note, however, that such ordering of subproblems is heuristic; there is always a chance we may inadvertently place an unsatisfiable subproblem last in the ordering. To reduce the impact of such a mistake, our implementation of DDP first attempts to determine satisfiability of each subproblem. Only when each subproblem is determined satisfiable does model counting begin. To avoid redundant searching, model counting begins from the state where satisfiability checking left off. Note that this modification also allows DDP to return a single satisfying assignment should one exist, in addition to the solution count.

To further avoid futile search, our implementation solves subproblems in an interleaved fashion, dynamically jumping to another subproblem if the current one turns out to be less constrained than initially estimated. This modification is not describable in the recursive pseudo-code framework used to define DDP above, which implements a strict depth-first search of the developing tree of connected components. This enhancement instead results in a best-first search of the developing component tree. The scores used in this best-first search can be obtained by simply reusing the scores typically provided by the branch-variable selection heuristic -- in our implementation, a subproblem’s score is given by the score assigned to its best branch variable.

Another approach for improving performance is to fuse the counting methods of both CDP and DDP into a single algorithm: the resulting algorithm is nearly identical to DDP, except like CDP, it accepts the number of variables n and returns $2^{n-|\sigma|}$ when the formula is empty in place of simply returning 1 when all variables have been assigned values. Comparing this enhanced algorithm to DDP, every time the formula turns up empty, this enhancement saves $2(n-|\sigma|)$ recursive calls to the top-level procedure since there are $n-|\sigma|$ connected components, one corresponding to each unassigned variable.

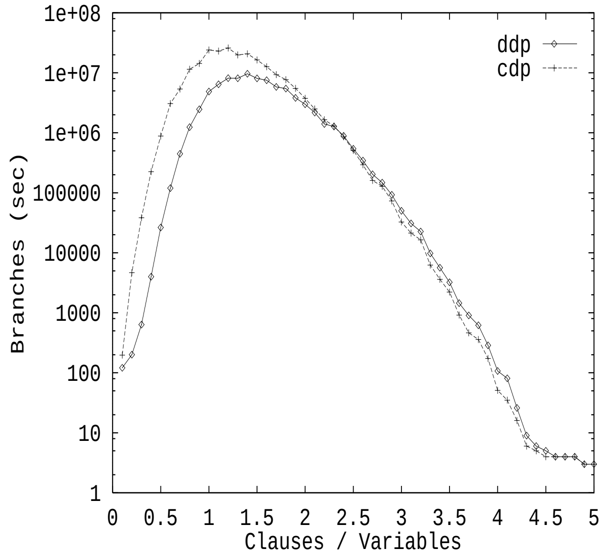
Similar to the above idea, we can invoke an arbitrary algorithm in place of CDP/DDP when F is of a suitable size or structure for obtaining a more efficient model count. This technique was used by Birnbaum & Lozinskii in their implementation of CDP: when F contains fewer than 6 clauses, they invoke an algorithm of Lozinskii [1992] that is more efficient at handling these small formulas. Because these “small-model” enhancements apply equally to both algorithms, we did not exploit them in the experimental comparison described in the following section.

Experimental Comparison

In this section, we experimentally compare CDP and DDP on a wide variety of instances. We implemented these algorithms by modifying and extending the relsat algorithm of Bayardo & Schrag [1997]. Relsat is a Davis-Putnam satisfiability checker that also infers additional clauses (*nogoods*) as search progresses and performs *conflict-directed backjumping* [Prosser 1993] to better recover from contradiction. The clauses added by relsat explain when a certain sub-assignment of the instance results in an unsatisfiable subproblem. The additional clauses are redundant in that they do not affect the solution count of the instance. This implies that by ignoring them during constraint-graph decomposition, they do not compromise correctness of DDP.

Relsat allows one to specify the “order” of learning to be applied, corresponding to the degree of the polynomial in its space complexity (resulting from clause inference). Unless otherwise noted, we used a setting of 3 which is known to be useful for many instances when determining satisfiability [Bayardo & Schrag 1997]. We discuss the effect of this setting on the various instances later in this section.

Due to the high overhead of scoring branch variables (relsat uses expensive unit-propagation based heuristics [Freeman 1995]), code profiling revealed that dynamic component detection was an almost negligible additional expense for the algorithm. For this reason, we sometimes report performance only in terms of the CPU-independent metric of branch selections performed by the algorithm, rather than seconds elapsed. The number of branch selections corresponds to the number of recursive calls made to each procedure, and correlates highly with runtime on instances from the same class. When we do report runtimes, they are for a 400 Mhz Pentium-II IBM IntelliStation M-Pro running Windows NT 4.0.



For our first experiment, we ran CDP and DDP on the same set of 50-variable instances from the uniform random 3-SAT problem space. To generate an instance from this problem space, three distinct variables are uniform-randomly selected out of the pool of n possible variables. Each variable is negated with probability $1/2$. These literals are combined to form a clause. m clauses are created in this manner and conjoined to form the 3-CNF Boolean expression.

We generated 100 instances at every C/V (clause/variable) point plotted along the x axis of the figure above, for a total of 5000 instances. The y axis plots the median number of branch selections, corresponding to the number of recursive calls of the procedure. Corroborating the results of Birnbaum and Lozinskii, we found that the hardness peak for CDP is around 1.2 C/V. We found the hardness peak resulted in a median of ~26 million branch selections, versus ~16 million determined by Birnbaum and Lozinskii. This discrepancy is most likely explained by the lack of small-model enhancements in our implementation.

Surprisingly, DDP does not appear to have an identical hardness peak. DDP instead peaks at approximately 1.5 C/V, with a median of ~8 million branch selections. Unlike the hardness peak for satisfiability, which consistently arises at approximately 4.26 C/V [Crawford & Auton 1996], the hardness peak for model counting may well be more algorithm dependent.

Though somewhat obscured by the logarithmic scale of the y axis, at CDP's hardness peak, DDP is over 3 times faster. Note that as the C/V ratio increases beyond 2.5, CDP gains a slight edge over DDP. We found that the fused DDP/CDP algorithm described in the previous section always performs better than either one alone, but never by a substantial amount. We omit its curve from the plot since it obscures the detail of the others.

DDP appears to be much more advantageous on larger instances we obtained from the SATLIB repository (http://aida.intellektik.informatik.tu-darmstadt.de/~hoos). We ran DDP and CDP on all 100 instances in the "flat200-*" suite

TABLE 1. Performance of and number of models computed by DDP on several "real world" benchmark instances for which model counts could be determined within 1 hour.

instance	branch	sec	count
2bitmax_6	6.2 mil	247	2.1e29
2bitcomp_5	237,379	9	9.8e15
logistics.a	151,265	24	3.8e14
logistics.b	5.3 mil	923	2.4e23
ssa7552-038	105,030	123	2.8e37
ssa7552-158	20,107	21	2.6e31
ssa7552-159	39,980	42	7.7e33
ssa7552-160	34,146	40	7.5e32

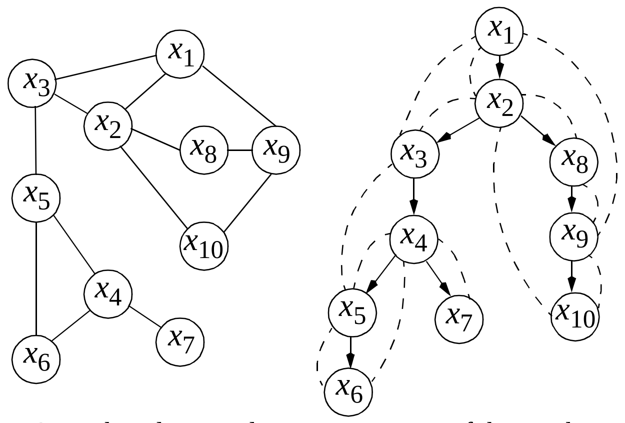
of graph coloring problems, each consisting of 600 variables and 2237 clauses. These instances are artificial graph-coloring instances which are hard for the Brelaz heuristic [Hogg 1996]. For DDP, an instance from this class required 27 seconds and 11,645 branch points on average. CDP was unable to determine a model count for *any* of these instances within the cutoff time of 15 minutes per instance.

We systematically went through the remainder of the SATLIB and beijing suite (available at http://www.cirl.edu/crawford/beijing) of benchmark instances, picking out for experimentation those obtained from "real world" sources such as planning and circuit analysis. Several, such as the ais-* class of instances, the blocksworld instances, hanoi4, and the 4blocks* and 3blocks instances from beijing suite, were not significantly easier for CDP or DDP than a complete enumeration of the solution space by DP with the pure literal rule disabled (called *enumerating* DP). This was because the number of solutions was relatively small -- at most a few hundred or thousand -- making efficient enumeration possible.

Many of the remaining satisfiable instances, though known to be easy when determining satisfiability instead of counting models, were too hard for either algorithm. These instances include logistics.c, logistics.d, 2bitadd_11 and 2bitadd_12. All resulted in timeout after the one hour per instance execution time limit was reached.

We found several instances for which the model count was efficiently determined by DDP even though the count was much too large for enumerating DP (see Table 1).[2] CDP failed to determine a solution count within 1 hour on any of them. Indeed, we were unable to find any instances among the benchmark suites for which CDP significantly outperforms enumerating DP. We note that given more computational power, it is possible some would be revealed by examining more of the randomly generated instances and/or increasing cutoff time substantially beyond one hour.

2. Though counts are rounded in the table, our implementation computes them exactly using a bignum package in order to avoid rounding errors which could result from using a floating point representation.



A graph and a rooted-tree arrangement of the graph.

Effects of Learning

We repeated the experiments from Table 1 with a learn order of 0 to disable the recording of additional clauses by the algorithms during search. In almost every case, this did not substantially alter performance. The only exceptions were logistics.a which went from 24 seconds of CPU time to over 15 minutes, and logistics.b, which resulted in time-out after one hour. The fact that learning often has no effect is not unexpected, since it is activated only in the presence of unsatisfiable subproblems. Instances which have a large number of solutions, like those in Table 1, tend to have only few such subproblems, and these are identified early on by the ordering optimizations that prevent futile counting when unsatisfiable subproblems are present.

Complexity Related Issues

In [Bayardo & Miranker 1996], the constraint-graph of a constraint satisfaction problem (CSP -- of which SAT is a restriction) is recursively decomposed to form a *rooted-tree arrangement* (see figure above) on which a backtracking algorithm similar to DDP is applied. By definition of a rooted-tree arrangement, two variables in different branches belong to different connected components when their common ancestors are assigned values.

The important differences between this algorithm and DDP are the decomposition is done statically before backtrack search instead of dynamically, and the algorithm attempts only to determine satisfiability instead of the number of models. They use this framework to define several graph-based parameters that can be used to bound the number of times each subproblem is attempted by the algorithm when different degrees and styles of learning are applied. These parameters lead to overall bounds on runtime which match the best-known structure-derived runtime bounds.

The key idea to the bounding technique is the realization that the assignment of variables within the *defining set* of a subproblem (the set of ancestors that are connected to at least one subproblem variable in the constraint graph) determines whether the subproblem is satisfiable. For example, the defining set of the subproblem rooted at x_4 in the example arrangement consists of x_3 alone. By recording the defining set assignment as a nogood or a good each time a subproblem is found to be unsatisfiable or satisfiable respectively, the number of times each subproblem is attempted can be bounded by the number of unique defining set assignments.

This algorithm can be easily modified to count solutions with identical bounds on runtime since equivalent assignments of the defining set must clearly lead to an equivalent number of solutions to the subproblem. Instead of recording whether a defining set assignment is “good” or “nogood”, the idea is to pair the solution count of the given subproblem with its defining set assignment. With such a modification, subproblem counts can then be combined and propagated exactly as done by DDP, allowing the algorithms to determine a model count instead of satisfiability with equivalent space and runtime complexity.

From a structure-based complexity perspective, then, counting models with learning-enhanced backtrack search is no more difficult than determining satisfiability. In practice, however, our experiments from the previous sections reveal that counting models is often much more difficult. This is possibly due to the fact that structure-based bounds on runtime are rather conservative; they are based on static constraint-graph analysis, unable to account for efficiencies resulting from dynamic variable ordering and dynamically simplifying constraint graph structure.

Another cause of the difficulty we witnessed in counting may be due to the fact that relsat does not record goods (defining set assignments for which the solution count of the subproblem is greater than zero), the importance of which is indicated by complexity results. Previous work has shown that recording only nogoods during backtrack search leads to effective structure based bounds on runtime for determining satisfiability [Frost & Dechter 1994]. Bayardo & Miranker [1996] demonstrated that while good recording improves runtime complexity of satisfiability checking, it does so by reducing only the base of the exponent.

When counting models, good learning is in fact a necessary enhancement of backtrack search for achieving polynomial time complexity given certain variable arrangements. One simple example is a CSP whose constraint graph is a chain. If we arrange the variables in the order they appear along the chain, then determining satisfiability using a nogood learning backtrack algorithm is of quadratic complexity [Bayardo & Miranker 1994]. Model counting, however, would be exponential in the number of variables if the constraints are sufficiently loose. Recording of goods in addition to nogoods brings the complexity of model counting down to quadratic regardless of constraint looseness, so long as they have the chain (or in fact any tree) structure.

Towards Practical Good Learning

Though we have empirically explored the effects of nogood learning on model counting through relsat’s clause-recording functionality, the effect of good learning remains unknown in practice. An effective implementation is an open problem, and a solution must address at least two significant complications: the identification of small defining sets, and ensuring correctness of the technique in conjunction with dynamic variable ordering.

The defining sets obtained from static constraint graph analysis are too large to use in practice on all but the most structure restricted instances. To overcome this, practical nogood learning algorithms perform conflict analysis to obtain a much smaller set of culprit variables. Conflict analysis techniques work backwards from contradiction, and are therefore inapplicable when attempting to identify the set of assignments responsible for a positive solution count. Techniques for minimizing the defining set in the case of a satisfiable (good) subproblem need to be developed.

Nogood learning is not complicated by dynamic variable ordering; if the defining set assignment of some nogood subproblem reappears, then the solution count of the current subproblem is immediately known to be zero, even if the current subproblem does not exactly match the subproblem from which the defining set assignment was derived. With good learning, in addition to recording the defining set assignment, it seems we must explicitly keep track of all the variables appearing within the subproblem. This is because some of the subproblem's variables may be assigned values once the defining set assignment reappears, potentially affecting its solution count. In order to apply a good, in addition to the defining set assignment matching the partial assignment, all subproblem variables must be in an unassigned state. Good learning with dynamic variable ordering therefore requires additional overhead to check and record the subproblem variables, and perhaps more significantly, it reduces the probability with which a recorded good might be exploited in the future.

On the other hand, since hard instances for model counting typically contain comparatively few unsatisfiable subproblems, there are plenty more opportunities to learn goods than nogoods. This suggests good learning may still prove beneficial on sufficiently difficult instances even given the above limitations.

Conclusions and Future Work

We have demonstrated that recursive identification of connected components within a dynamically simplified constraint graph leads to an efficient DP-based algorithm for counting models of a propositional logic formula. In addition, from a structure-based perspective of complexity, we have shown that model counting using a learning-enhanced backtrack search algorithm is no more difficult than determining satisfiability.

One avenue remaining to be explored is whether good learning will lead to substantial performance gains when counting models in practice, as suggested by complexity results showing that good learning is mandatory for polynomial time counting. We have yet to solve the problem of providing an efficient and effective implementation of good learning in conjunction with dynamic variable ordering. Though this problem is non-trivial, we feel it is not insurmountable, and continue to contemplate solutions.

References

Bayardo, R. J. and Miranker, D. P. 1994. An Optimal Backtrack Algorithm for Tree-Structured Constraint Satisfaction Problems. *Artificial Intelligence,* 71(1):159-181.

Bayardo, R. J. and Miranker, D. P. 1995. On the space-time trade-off in solving constraint satisfaction problems. In *Proc. of the 14th Int'l Joint Conf. on Artificial Intelligence*, 558-562.

Bayardo, R. J. and Miranker, D. P. 1996. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem. In *Proc. 13th Nat'l Conf. on Artificial Intelligence*, 558-562.

Bayardo, R. J. and Schrag, R. 1997. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proc. of the 14th National Conf. on Artificial Intelligence,* 203-208.

Birnbaum, E. and Lozinskii, E. L. 1999. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research* 10:457-477.

Crawford, J. M. and Auton, L. D. 1996. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence* 81(1-2), 31-57.

Davis, M., Logemann, G. and Loveland, D. 1962. A Machine Program for Theorem Proving, *CACM* 5, 394-397.

Dechter R., and Pearl, J., 1987. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34(1):1-38.

Freeman, J. W. 1995. *Improvements to Propositional Satisfiability Search Algorithms.* Ph.D. Dissertation, U. Pennsylvania Dept. of Computer and Information Science.

Freuder, E.C. and Quinn, M.J., 1985. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of IJCAI-85,* 1076-1078.

Frost, D. and Dechter, R. 1994. Dead-End Driven Learning. In *Proc. of the Twelfth Nat'l Conf. on Artificial Intelligence*, 294-300.

Hogg, T. 1996. Refining the Phase Transition in Combinatorial Search. *Artificial Intelligence*, 81:127-154.

Kautz, H. and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proc. 13th Nat'l Conf. on Artificial Intelligence*, 558-562.

Lozinskii, E. 1992. Counting Propositional Models. *Information Processing Letters*, 41(6):327-332.

Melhorn, K. 1984. *Data Structures and Algorithms*, vol 1-3, Springer.

Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3):268-299.

Roth, D. 1996. On the Hardness of Approximate Reasoning. *Artificial Intelligence* 82, 273-302.

Rymon, R. 1994. An SE-tree-based Prime Implicant Generation Algorithm. *Annals of Mathematics and Artificial Intelligence* 11, 1994.