Guiding Real-World SAT Solving with Dynamic Hypergraph Separator Decomposition

Wei Li and Peter van Beek
School of Computer Science
University of Waterloo
200 University Ave. West
Waterloo, Ontario N2L 3G1, Canada
{w22li,vanbeek}@uwaterloo.ca

Abstract

The general solution of satisfiability problems is NP-Complete. Although state-of-the-art SAT solvers can efficiently obtain the solutions of many real-world instances, there are still a large number of real-world SAT families which cannot be solved in reasonable time. Much effort has been spent to take advantage of the internal structure of SAT instances. Existing decomposition techniques are based on preprocessing the static structure of the original problem. We present a dynamic decomposition method based on hypergraph separators. Integrating the separator decomposition into the variable ordering of a modern SAT solver leads to speedups on large real-world satisfiability problems. Compared with a static decomposition based variable ordering, such as Dtree (Huang and Darwiche, 2003), our approach does not need time to construct the full tree decomposition, which sometimes needs more time than the solving process itself. Our primary focus is to achieve speedups on large real-world satisfiability problems. Our results show that the new solver often outperforms both regular zChaff and zChaff integrated with Dtree decomposition. The dynamic separator decomposition shows promise in that it significantly decreases the number of decisions for some real-world problems.

1. Introduction

Methods to solve the satisfiability problem play an important role in several industrial applications of computer-aided design and verification. Such applications include automatic test pattern generation, formal verification and routing of field-programmable gate arrays.

However, the worst-case complexity of solving SAT problems using the original DPLL is $O(m2^n)$, where m is the number of clauses and n is the number of variables. So an efficient general algorithm is not expected to be found. In order to improve this worst-case complexity, a variety of structural decomposition methods have been investigated. The best known tree decomposition leads to a time complexity in $O(n2^{w+1})$, where w is the tree-width of the hypergraph representation of the SAT problem.

Methods to improve the efficiency by exploiting the problem structure have been intensively investigated in both the CSP (Constraint Satisfaction Problems) and SAT communities [5, 6, 13]. The term "structure" means the problem's structural properties, which can be presented as graph theoretic properties of the constraint graph or constraint hypergraph. In real-world applications, the internal structure of the corresponding SAT problems represent the following facts:

1. Real-world applications are built and designed in a modular way. Modularization with minimal interconnectivity is encouraged.
2. Each module uses distinct name spaces and variables.
3. Small sets of variables are designed to control an application.

Problem structure has been used to guide the variable ordering heuristic in backtracking search since Freuder[4], who presents how to construct the variable ordering by finding the biconnected components of a constraint graph. Recent experimental results show that guiding the variable ordering heuristic using tree decompositions can improve the performance of SAT and CSP solvers [1, 3, 9, 10]. Huang and Darwiche's [9] variable ordering heuristic uses a Dtree, a static binary tree decomposition, to compute the variable group ordering. Because the Dtree has to be constructed before search, the pre-established variable groupings never change during the execution of the solving. Bjesse et al. [3] present dynamic variable groupings. However, no ex-

perimental results are shown to demonstrate whether the method is more promising than static variable groupings. Also, there is still no conclusion about how to arrange the order of subproblems induced by tree decomposition. Amir and McIlraith [2] provide the heuristic of solving the most constrained subproblem first. But again, no experimental evaluation is performed. A static global variable ordering based on recursive min-cut bisection of hypergraphs was proposed in [1]. This preprocessing approach does not require modifications to the SAT solver. But most modern SAT solvers use dynamic variable ordering. The pre-established static variable ordering can only be used to break the ties of dynamic variable ordering in some situations.

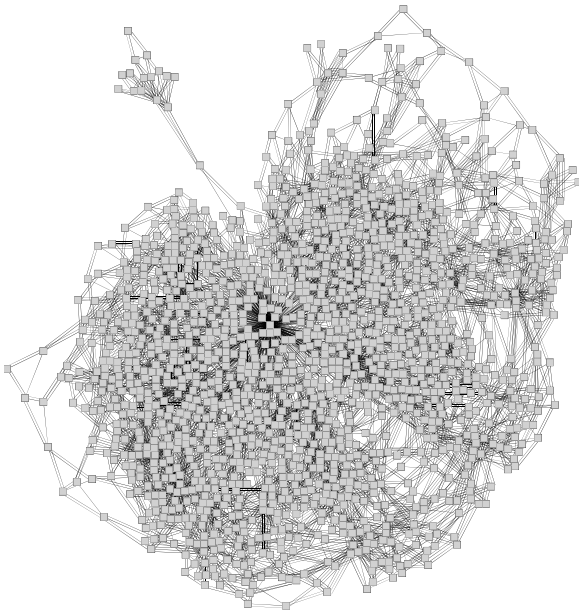

Figure 1. Constraint graph of original dp05s05 problem.

The visualization approach proposed in [13] provides an empirical tool to observe and analyze the structure of real-world SAT problems. It shows that long implication chains exist in those instances. Unit propagation is a look-ahead strategy for all of the cutting-edge SAT solvers. Since most of the variables on the implication chains are instantiated after making a relatively small number of decisions, the internal structure of real-world instances often change dramatically in different parts of the search tree. For example, Figure 1 is the constraint graph of the bounded model checking instance dp05s05, which is from the dining philosophers problem. Figure 2 is obtained by setting proposition 1283 to false, and after subsequent unit propagation.

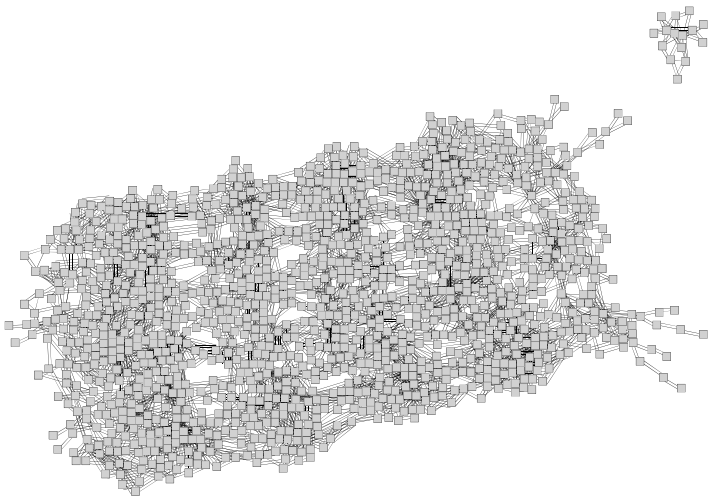

Figure 2. dp05s05 problem at decision level 1, after one variable has been instantiated.

Since the structure of a SAT problem changes dramatically during the running time of DPLL, in this paper, we use a dynamic decomposition method based on hypergraph separators. A separator of a hypergraph H is a set of hyper-edges whose removal chops H into disjoint sub-hypergraphs whose sizes stand in some sought relation. Finding hypergraph separators naturally leads to a divide-and-conquer strategy. The separator becomes the root of the corresponding tree structure, while the subtrees become the subproblems induced by the separator. The most important difference between Dtree decomposition [9] and our hypergraph separator decomposition is that the generation of the separator does not depend on its subproblems, which means that the separator based decomposition can stop at any time during the decomposition process. In contrast, to construct a node of Dtree we need to merge the variables in each subtree under the same parent node, which means that Dtree decomposition needs time to build the whole Dtree before the solving process and the Dtree will never change. We report our effort of using hypergraph separator decomposition to guide the variable ordering of a SAT solver dynamically, which includes (i) highly ranked variables are added to the current separator dynamically; (ii) various subproblem ordering heuristics are tried; and (iii) hypergraph separators are generated dynamically during backtracking

search rather than statically prior to search.

Our primary focus is to achieve speedups on large real-world satisfiability problems. We combined the state-of-the-art SAT solver zChaff [14] with hypergraph separator decomposition and tested it on SAT 2002 competition benchmarks. Our results show that the new solver often outperforms both the regular zChaff and the zChaff integrated with Dtree decomposition in solving real-world problems. Furthermore, the new solver solved more hard instances than the Dtree decomposition within a given cutoff time limit.

2. Hypergraph Separator Decomposition

Given a propositional formula F in CNF, its hypergraph representation $H = (X, E)$ is a hypergraph whose vertex set X consists of the clauses in F, and there is a hyper-edge for each Boolean variable in F connecting all of the clauses (vertices) that contain that variable.

A hypergraph separator decomposition is a triple (H, S, R) where

1. $S \subset E$, and the removal of S separates H into k unconnected components or disjoint subgraphs $H_1, \ldots, H_k$.
2. R is a relation over the size of the disjoint subgraphs $|H_1|, \ldots, |H_k|$.

The relation R is a balance factor. In our experiments (following [9]), we used 15%/85%, meaning a disjoint subgraph must contain at least 15% and at most 85% of the nodes in the whole hypergraph.

Once we have the hypergraph representation of a formula F, the entire formula can be decomposed into smaller subgraphs (subproblems) giving a divide-and-conquer strategy. There is a tradeoff between separator size and the number and size of the subproblems. In order to generate more and smaller subproblems, the separator has to be enlarged. Our hope is that relatively small separators exist in real-world SAT instances and that the instances can be decomposed into a large number of similar sized subproblems by the separator.

3. Dynamic Decomposition and DPLL

The DPLL algorithm is the core of modern SAT solvers. Algorithm1 shows the original DPLL algorithm. We use F to denote a propositional formula. $F \mid v = 0$ ($F \mid v = 1$) represent the new formula obtained by replacing the variable v with $false$ ($true$). The Unit_Propagation(F,C) function returns the simplified formula, where no more unit clauses exist. In this section, we discuss how to integrate hypergraph separator decomposition into the DPLL algorithm. We examine two approaches to improve DPLL: Enhanced Static Decomposition (ESD) and Dynamic Separator Decomposition (DSD).

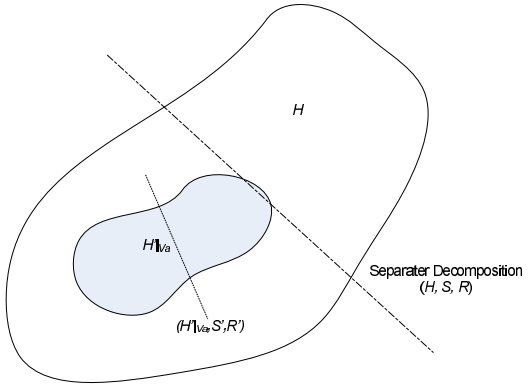

Figure 3. Propagation Synchronization

Algorithm 1 DPLL(F)

```
1: if F is an empty clause set then
2:     return true
3: end if
4: if F contains an inconsistent clause then
5:     return false
6: end if
7: if F contains a unit clause C then
8:     F = Unit_Propagation(F,C)
9: end if
10: choose an uninstantiated variable v in F
11: return DPLL(F | v = 0) or DPLL(F | v = 1)
```

ESD is to decompose the CNF before backtracking search, then dynamically adding variables to the separator. Given the hypergraph separator decomposition, we generate a variable group based on the hyper-edges in the separator. Whenever the next decision variable is needed, an uninstantiated variable is chosen from this group. The separator variable group is determined before starting DPLL.We also dynamically add variables to this variable group. It is well-known that the variable ordering can dramatically influence search performance. As well, according to our experimental results, the first separator plays an important role in improving the searching performance. To guarantee that the top level separator contains those variables which are highly ranked by the variable ordering heuristic, variables that are highly ranked can be added to the top separator if they are not already present.

Another approach is to dynamically generate the separator variable group recursively during DPLL (see [3]). Con-

sider the DPLL procedure. Boolean constraint propagation is the essence of DPLL. When the value of a variable can be determined because of a unit clause, we can remove it from the corresponding subproblem. Figure 3 shows a hypergraph H, and its implied hypergraph $H|Va$. After the variable Va and all the implied variables of Va are removed from H, it is possible that the separator decomposition of H is not a valid decomposition for $H|Va$. Clearly we can expect a better separator decomposition if we update the separator during the search after unit propagation of each variable.

Algorithm 2 DSD_DPLL(F,S,G)

```
1: if F is an empty clause set then
2:    return true
3: end if
4: if F contains an inconsistent clause then
5:    return false
6: end if
7: if F contains a unit clause C then
8:    F = Unit_Propagation(F,C)
9: end if
10: if S is empty then
11:    S=Separator(G)
12: end if
13: if there is no uninstantiated variable in S then
14:    for each constraint graph partition Gi do
15:       if DSD_DPLL(F, φ, Gi)=false then
16:          return false
17:       end if
18:    end for
19:    return true
20: else
21:    choose an uninstantiated variable v from S
22:    return DSD_DPLL(F|v = 0, S, G) or
23:    DSD_DPLL(F|v = 1, S, G)
24: end if
```

Algorithm 2 takes three inputs: the propositional formula F, corresponding constraint graph G, and S, the separator of G, whose initial value is ϕ. After unit propagation, we create and maintain a separator of G. We choose the next variable from S until all the variables are instantiated and F is decomposed into several sub-problems. The separator created after unit propagation is based on the simplified constraint graph. F is $true$ when all its subproblems are $true$, otherwise it is $false$.

We also consider dynamic subproblem ordering heuristics. When we have several subproblems, we need to decide which subproblem to solve first. Most modern SAT solvers have a dynamic variable ordering heuristic. For example, zChaff uses Variable State Independent Decaying Sum (VSIDS) heuristic. To combine VSIDS with subproblem ordering, each hyper-edge is given a weight, which dynamically changes with VSIDS. Four dynamic subproblem ordering heuristics were tested in our experiments:

1. MVSF: Subproblem with maximum VSIDS sum first.
2. MASF: Subproblem with maximum VSIDS average first.
3. SSF: Subproblem with smallest clause number first.
4. MCSF: The most constrained subproblem first[2].

4. Experiment Implementation and Result

In our implementation, a CNF is represented by a dual weighted hypergraph. Because the problem of computing an optimal partition of a hypergraph is NP-complete, a multi-level hypergraph partition algorithm package, hMETIS [7, 11], is used to find separators. The basic idea of multi-level algorithms is to construct a sequence of successively smaller hypergraphs by collapsing appropriate vertices, then find a partition of the small coarsened hypergraph, and finally obtain the approximated separator of the original hypergraph from the coarsest hypergraph step by step.

Generally, there are two different ways to merge vertices together to form single vertices in the next level coarse hypergraph: edge coarsening and hyperedge coarsening. In hyperedge coarsening, vertices are grouped together that correspond to hyperedges. In our experiments, we found that the coarsening scheme was an important factor for producing high quality hypergraph separator decomposition and preference was given to hyperedge coarsening.

We enhanced the separator decomposition with subproblem ordering heuristics and dynamic variable adding. Four subproblem ordering heuristics, MVSF, MASF, SSF and MCSF, were implemented. Before we instantiate the variables in the separator (zChaff uses the VSIDS heuristic), a group of variables with highest scores are added to the current separator. After all the variables in the separator of a hypergraph have been instantiated, the sub-hypergraphs of the current separator are updated to eliminate variables implied by the instantiations of the variables in the separator.

The benchmark instances presented in this paper are from the industry category of SAT Competition 2002. There are 219 typical industrial SAT encoded instances included in this category [8]. Our experiments were performed on a PC with a 2.67GHz Pentium 4 processor and 1Gb of RAM. Each runtime is the average of 10 runs with a 15 minute CPU cut-off limit. All runs that did not complete in the time limit did not contribute to the average. The time limit is longer than the SAT 2002 competition (see [12]).

In Table 1, we compare the runtime of zChaff and enhanced static separator decomposition (ESD). The times shown represent the total time for the instances which were

Table 1: CPU time (sec.) and number of improved instances comparing zChaff and Separator (2-way ESD + MASF + Dynamically Adding Variable).

Benchmark	zChaff	Separator	#Solved/#Inst.	SAT/UNSAT	Improved
bmc1	0	0	4/4	UNSAT	3
bmc2	770	**387**	5/6	UNSAT	4
Bart	**35**	44	3/21	SAT	1
Homer	3164	**586**	9/15	UNSAT	9
Lisa	1782	**996**	9/14	SAT	6
cmpadd	**4**	5	8/8	UNSAT	5
Matrix	31	**23**	2/5	UNSAT	2
fpga_routing	**27**	38	27/32	MIXED	10
Graph_coloring	**11517**	11690	150/300	MIXED	107
onestep_rand_net	443	**125**	15/16	UNSAT	9
multistep_rand_net	**180**	506	2/16	UNSAT	0
ezfact	1367	**1270**	31/41	MIXED	18
qg	191	**162**	10/19	MIXED	7

Table 2: CPU time (sec.) and number of improved instances comparing zChaff+Dtree vs Separator (2-way ESD + MASF + dynamically adding variable).

Benchmark	zChaff+ Dtree	Dtree Time	Separator	#Solved/ #Inst.	SAT/ UNSAT	Improved
bmc1	0.01	3	0.01	4/4	UNSAT	4
bmc2	0.07	2.5	**0.04**	1/6	UNSAT	1
Bart	150	2	**44**	3/21	SAT	3
Homer	**216**	13	586	9/15	UNSAT	0
Lisa	**700**	87	1451	11/14	SAT	6
cmpadd	**0.76**	31	4.57	8/8	UNSAT	8
Matrix	90	6	**23**	2/5	UNSAT	2
fpga_routing	11	2076	**8**	17/32	MIXED	17
Graph_Coloring	40767	230	**19761**	160/300	MIXED	129
one_step_randnet	240	380	**125**	15/16	UNSAT	11
ezfact	759	179	**543**	31/41	MIXED	22
qg	182	503	**162**	10/19	MIXED	7

solved within the time limit. Table 2 reports the comparative performance of zChaff+Dtree and zChaff+ESD. The Dtree Time reports the time of constructing a Dtree and zChaff+Dtree reports the runtime of zChaff with the variable group ordering from the Dtree. In contrast, the runtime of finding the graph separator decomposition is included in the runtime of solving the instances. In Table 1 and Table 2, only those instances which can be solved in 15 minutes by both programs are included. Our experimental results also show that the separator decomposition can solve much harder instances than Dtree decomposition. Among the 11 industrial problems, there is only one case—the multi-step Rand-net problem—in which zChaff is much faster. However, most instances of this problem cannot be solved by any solver we tested.

The dynamic separator decomposition constructs a new separator each time whenever a new decision is made. Because of the overhead of propagation synchronization, the runtime of the dynamic separator decomposition is very slow. 70% of instances cannot be solved in 15 minutes. However, the solver using dynamic separator decomposition often makes many fewer decisions and implications than zChaff and the static separator decomposition (see Table 3).

In the reported experiments, the MASF heuristic was used as the subproblem ordering heuristic. Our experimental results show that it is better than the other subproblem ordering heuristics, but more work is needed to confirm this conclusion.

5. Discussion

Compared with the completely recursive dynamic decomposition, enhanced static decomposition is more practical and easy to implement. After adding high ranking variables into the separator and instantiating them, the long implication chains are "started" at the very beginning. In Figure 2, 4, and 5, independent subproblems are naturally oc-

Table 3: Max decision level, decision number and implication number comparing zChaff vs Separator (2-way DSD + MASF)

	zChaff			Dynamic Separator		
Benchmark	Max Level	Decision#	Implication#	Max Level	Decision#	Implication#
lisa19_3_a	65	181568	36789189	**43**	**61296**	**9359648**
lisa19_99_a	75	262798	55335471	**39**	**31149**	**4925937**
lisa20_99_a	62	99361	19340591	**43**	**85672**	**14808291**
homer06	124	56754	1071432	**103**	**17713**	**212052**
homer07	124	110607	2126296	**107**	**29398**	**420020**
homer08	141	134769269	**8053**	**123**	**73096**	642087
homer09	178	237750	5176136	**150**	**139264**	**1666138**
homer10	**197**	**283545**	7590761	203	387330	**7285721**
homer11	156	121239	2412764	**133**	**46633**	**672700**
homer12	162	242994	4871283	**140**	**123244**	**1871968**
homer13	173	300403	6093351	**154**	**137859**	**246163**
homer14	194	588502	13377128	**170**	**315076**	**5790068**
homer15	258	1127302	31367313	**239**	**629895**	**16626020**
homer16	198	369264	7156381	**156**	**193118**	**2441261**
homer17	203	394490	7980266	**170**	**291448**	**5691210**
Hanoi4	39	4696	309408	**30**	**1508**	**153196**

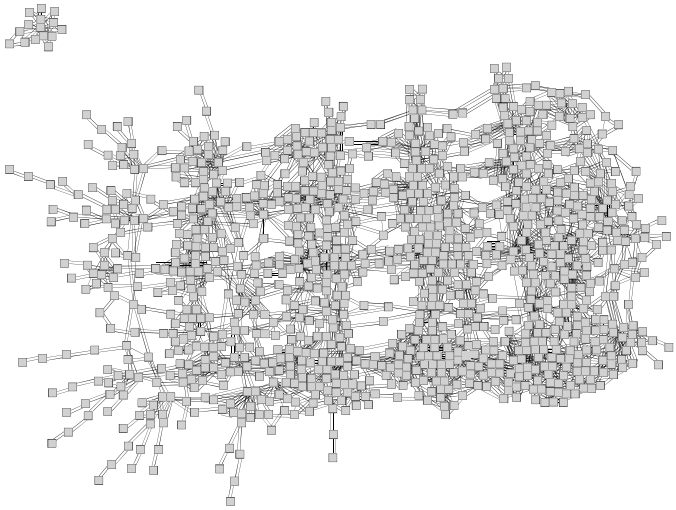

Figure 4. dp05s05 problem at decision level 10.

Figure 5. decomposed dp05s05 problem at decision level 10.

curring. Lead by zChaff's Variable State Independent Decaying Sum Ordering, a small separator is constructed based on the simplified structure. In the process of complete dynamic decomposition, the runtime depends on both the problem size and the number of separators created. At the same decision level, the more decisions we make, the more times we need to update old separators. We use the decision distribution diagram (DDD) to show that the solving process of real-world instances may have an influence on the complexity of dynamic dicomposition. The diagram is generated by recording the number of decision made at every decision level of dynamic decomposition. Figure 6 is the DDD of a random circuit checking problem. This diagram shows that the separator needs to be updated frequently at the root of the search tree. In the contrast, Figure 7 shows that BART11, an instance of circuit model checking, has a

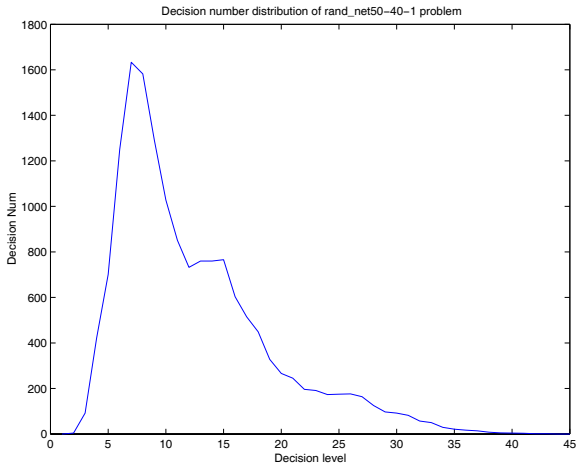

Figure 6. Decision distribution diagram of randnet50401

very easy decision making process at the beginning. Generally, the second case is more welcome since the easily solvable variables simplify the problem right before the dynamic decomposition.

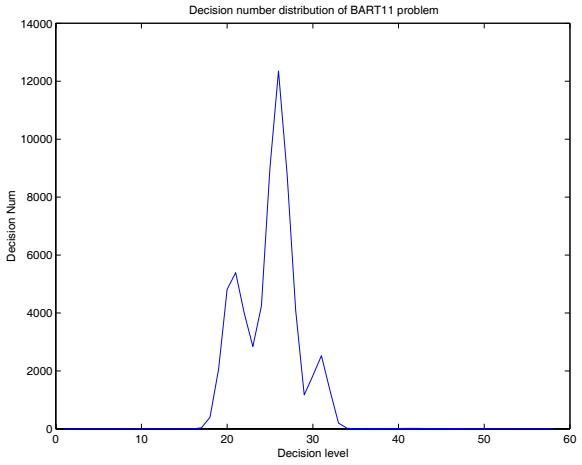

Figure 7. Decision distribution diagram of BART11

6. Conclusions

We presented dynamic decomposition methods based on hypergraph separators. Integrating the hypergraph separator based decomposition into the variable ordering of a modern SAT solver led to speedups on large real-world satisfiability problems. Compared with Dtree, our approach does not need time to construct the full tree decomposition, which sometimes needs more time than the solving process. The dynamic separator decomposition shows promise in that it significantly decreases the number of decisions for some real-world problems. However, the work we have presented here represents a first step and better techniques and implementation are still needed to improve its running time. Finally, experimental results show that for certain problems, a specific subproblem ordering heuristic is required to efficiently solve large problem instances.

For future work, we intend to examine how the techniques discussed in this paper can be applied to the problem of counting the number of satisfying assignments (or models).

References

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. MINCE: A static global variable-ordering for SAT and BDD. In *International Workshop on Logic and Synthesis*. University of Michigan, June 2001.

[2] E. Amir and S. McIlraith. Solving satisfiability using decompostion and the most constrained subproblem. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing*, 2001.

[3] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, and Y. Zhu. Guiding SAT diagnosis with tree decompositions. In *SAT'03*, pages 315–329, 2003.

[4] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32:755–761, 1985.

[5] E. C. Freuder. *Expoiting Structure in Constraint Satisfaction Problems*, volume 131 of *NATO ASI Series F: Computer and System Sciences*. Springer-Verlag, 1994.

[6] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124:282–343, 2000.

[7] hMETIS. www-users.cs.umn.edu/~karypis/metis/hmetis.

[8] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. Kluwer Academic, 2000.

[9] J. Huang and A. Darwiche. A struture-based variable ordering heuristic for SAT. In *IJCAI-03*, pages 1167–1172, 2003.

[10] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.

[11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. Technical report, University of Minnesota, 1997.

[12] L. Simon, D. L. Berre, and E. A. Hirsch. The sat2002 competition. citeseer.ist.psu.edu/simon02sat.html.

[13] C. Sinz. Visualizing the internal structure of SAT instances. In *SAT' 04*, May 2004.

[14] zChaff. www.ee.princeton.edu/~chaff/zchaff.php.