A Structure-Based Variable Ordering Heuristic for SAT

Jinbo Huang and Adnan Darwiche
Computer Science Department
University of California, Los Angeles, CA 90095
{jinbo, darwiche} @cs. ucla. edu

Abstract

We propose a variable ordering heuristic for SAT, which is based on a structural analysis of the SAT problem. We show that when the heuristic is used by a Davis-Putnam SAT solver that employs conflict-directed backtracking, it produces a divide-and-conquer behavior in which the SAT problem is recursively decomposed into smaller problems that are solved independently. We discuss the implications of this divide-and-conquer behavior on our ability to provide structure-based guarantees on the complexity of Davis-Putnam SAT solvers. We also report on the integration of this heuristic with ZChaff— a state-of-the-art SAT solver—showing experimentally that it significantly improves performance on a range of benchmark problems that exhibit structure.

1 Introduction

The class of Boolean satisfiability (SAT) problems has been of perpetual interest to researchers in many areas of computer science. Although these problems have a potentially exponential complexity, fresh techniques have continued to be proposed and implemented, allowing an increasing number of (previously) intractable problems to be solved in reasonable amounts of time.

Most existing complete SAT solvers are based on the Davis-Putnam (DP) procedure [Davis and Putnam, 1960; Davis ct a l , 1962], which formulates the SAT problem as a systematic search problem in the space of variable instantiations, and uses depth-first search to find a solution. Various techniques are employed by DP solvers in an attempt to prune the search space, or to focus it on regions that promise earlier discovery of a solution. It has been noted that, among other things, the order in which variables are instantiated has a great effect on the resulting complexity of the SAT algorithm; see [Li and Anbulagan, 1997] for example.

We propose a new variable ordering heuristic for SAT, which is based on a structural analysis of the SAT problem. We show that when the suggested ordering heuristic is used by a DP solver that employs conflict-directed backtracking [Silva and Sakallah, 1996], it produces a divide-and-conquer behavior in which the SAT problem is recursively decomposed into smaller problems that are solved independently. The general concept is that while the initial SAT problem may not be decomposable, it can be broken into independent sub problems after instantiating a certain number of variables that can be determined structurally. Hence, putting such a decomposing set of variables first in the ordering leads to an early decomposition of the problem. The key, however, is that this process can be repeated recursively, allowing one to recursively decompose the SAT problem down to single clauses. The other key point which we establish analytically is that it is critical for the DP solver to employ conflict-directed backtracking for the suggested order to have this recursive decompositional effect.[1]

By way of experimentation, we incorporated this technique in the publicly available ZChaff program [ZChaff, URL] and tested both the original and modified programs on a range of SAT benchmarks. Our results indicate a significant improvement in speed on the majority of the instances.

The rest of the paper is structured as follows. We start with a brief review of DP solvers in Section 2. We then turn in Section 3 to the proposed variable ordering heuristic, where we define it formally in terms of a graphical model, known as a dtree (decomposition tree). We then describe in Section 4 our integration of the new heuristic with ZChaff [ZChaff, URL], and provide experimental results that illustrate its effectiveness. We then consider a theoretical analysis of the proposed ordering heuristic in Section 5, where we show that when coupled with conflict-directed backtracking, the proposed ordering generates a behavior which is equivalent to a divide-and-conquer search algorithm that recursively decomposes the given SAT problem all the way to single clauses. This correspondence allows us to provide some guarantees on the complexity of the resulting search, depending on the structure of the given SAT problem. We finally close in Section 6 with some concluding remarks.

2 Davis-Putnam (DP) Search

We start by reviewing the basic DP search for the satisfiability problem. As is customary, a propositional theory is expressed in Conjunctive Normal Form (CNF). Recall that to satisfy a

'Sec [Bayardo and Pehoushek, 2000] for an example where (dynamic) decomposition is used, but in the context of model counting.

```
Algorithm 1 sat(Thcory : C)
1: if there is an inconsistent clause in C then
2:     return false (unsatisfiablc)
3: if there is no uninstantiated variable in C then
4:     return true (satisfiable)
5: select an uninstantiated variable v in C
6: return sat{C\_{v=true}) V sat(C\_{v=false})
```

propositional theory in CNF each of its clauses simultaneously has to be satisfied. We will use $C = \{c_1, c_2, \ldots, c_m\}$ to denote our target theory having m clauses where C{ denotes its zth clause. We will write $C|_{v=true}$ $(C|_{v=false})$ to represent the theory obtained by replacing the occurrences of variable v to true (false) in the theory C, Algorithm 1 provides a recursive description of the basic DP algorithm except that it omits the use of unit propagation which we discuss later.

This algorithm is often implemented iteratively instead of recursively, where it explores the search space by instantiating variables one at a time (we will call these instantiations decisions and the corresponding variables decision variables). Each of these decisions is pushed onto the decision stack and the theory is updated to reflect the new variable assignments. When inconsistency is discovered in the theory it goes back on the stack to the last decision whose variable has not been tried both ways, flips it, and proceeds therefrom. In case all previous decision variables have been tried both ways, it declares the theory as unsatisfiablc. The theory is declared satisfiable if all variables have been successfully instantiated, or if all clauses have been subsumed (satisfied by the current, possibly partial, variable assignment).

Note that there is no need to attempt both recursive calls on Line 6 in case one of the calls succeeds. Hence, one opportunity for optimization is to first try the most promising disjunct in Line 6. Note also that we did not specify how to choose the next variable on Line 5. In fact, a different chain of decisions can often lead to a remarkable difference in complexity [Li and Anbulagan, 1997].

We describe in the next section a variable ordering heuristic based on a structural analysis of the SAT problem. We then provide an experimental and a theoretical analysis of the heuristic in later sections.

3 A Variable Group Ordering

Recall that $C = \{c_1, c_2, \ldots, c_m\}$ is our target propositional theory in CNF. We have alluded to our desire of decomposing C into disconnected components. However, if we simply split C into two arbitrary subsets C_L and C_R, these two subtheories will in general have some variables in common and hence not qualify as disconnected components.

To overcome this difficulty we propose the following method. Let V_L and V_R denote the sets of variables mentioned in Ci and C R, respectively. We start the DP search on CNF C, but insist that it only make decisions on variables in $V_L \cap V_R$ at this stage. If unsatisfiable is declared during this process, we are done; otherwise all variables in $V_L \cap V_R$ will at some point have been instantiated.

Observe that the resulting sub-theories C'_L and C'_R now

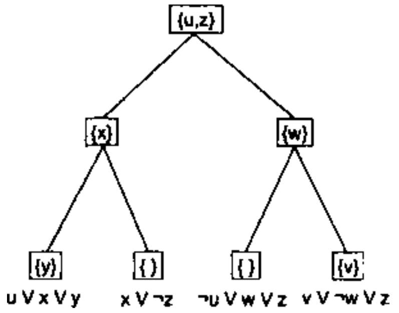

Figure 1: A dtree for a four-clause CNF.

have disjoint sets of (uninstantiated) variables: $V'_L = V_L - V_L \cap V_R$ and $V'_R = V_R - V_L \cap V_R$; they are hence completely independent sub-problems. Ordering one set of variables before the other therefore seems a good strategy from this point on. We have thus obtained a constraint on the order of variables: variables $V_L \cap V_R$ are processed first, followed by V'_L, and then V'_R.[2] Note that V'_L and V'_R can each be recursively broken up in a similar fashion, until all variables are partitioned into a sequence of groups, which serves as our proposed variable ordering. Within each of these groups, the algorithm is free to use any variable order.

To generate such a variable group sequence we will need to specify how the theory C is to be partitioned into sub-theories C_L and C_R, how the sub-theories in turn are to be partitioned, and so on. A graphical model known as a dtree serves nicely for this purpose, as it allows us to offer guarantees on the complexity of the resulting search [Darwiche, 2002].

A dtree (decomposition tree) for a CNF C is a full binary tree whose leaves correspond to the clauses of C\ see Figure 1. Each internal node represents a subset of C corresponding to all the leaves under it; the root in particular represents the original theory C. Such a tree naturally induces a recursive decomposition scheme, partitioning a theory into two parts represented by the node's two children.

A number of properties can be defined for the nodes of a dtree [Darwiche and Hopkins, 2001].

Definition 3.1 The variables of an internal dtree node is the set of variables mentioned in the clauses represented by the leaves under that node; the variables of a leaf node is the set of variables mentioned in the clause it represents.

Definition 3.2 The cutset of an internal dtree node is the intersection of its children's variables, minus all its ancestors' cutsets; the cutset of a leaf node is its variables minus all its ancestors' cutsets.[3]

Figure 1 depicts a dtree for a four-clause CNF theory. The clauses are listed at the bottom below their corresponding leaves. The cutset is shown inside of each node.

We assume the left-right order. Choice between the two orders affords another opportunity for applying heuristics, but is beyond the scope of this paper.

[3] A cutset is usually not defined for a leaf node in a dtree, but we extend the definition here for convenience.

Recall that each dtree node t corresponds to a part of the CNF theory; hence the variables of a dtree node are simply the variables of that part of the CNF theory; and the cutset represents all variables that need to be instantiated before the two sub-theories become disconnected (assuming that all cutsets of t's ancestors have been instantiated). The following two concepts will help formally state our proposed heuristic for ordering variables.

Definition 3.3 A variable group ordering of set V is a partition of variables V into an ordered sequence of subsets.

With respect to the set of variables in Figure 1, the following is a possible variable group ordering: $\{u, v, w\}$, $\{x, y\}$, $\{z\}$; which dictates that variables $\{u, v, w\}$ should be considered before $\{x, y\}$, and $\{x, y\}$ b e f o r $\{z\}$ Note that a strict variable ordering is a special case of variable group ordering in which all groups have size one.

Definition 3.4 The variable group ordering (v.g.o.) induced by a dtree is one obtained recursively as follows: output the cutset of the root; output the v.g.o. of its left child; output the v.g.o. of its right child; discard empty sets.

Our proposed variable ordering scheme can now be stated as simply the variable group ordering induced by the dtree. The variable group ordering induced by the dtree in Figure 1, for example, is $\{u, z\}, \{x\}, \{y\}, \{w\}, \{v\}$.

So far we have not discussed how dtrees are to be constructed. There are obviously many distinct dtrees for any nontrivial CNF theory. To reduce the complexity of the SAT algorithm we have adopted a heuristic that calls for relatively balanced dtrees with small cutsets. A good balance ensures a near-logarithmic tree height, which allows the recursive decomposition to finish faster. And smaller cutsets lead to minimizing the number of cases that may need to be considered for each decomposition. Hence, both of these strategies will help reduce the overall complexity.

To generate dtrees with the above properties, we employ the same hypergraph partitioning technique as described in [Darwiche and Hopkins, 2001]. Our hypergraph for a CNF theory C is constructed by having a node for each clause in C and having a hyperedge for each variable in C connecting all nodes (clauses) that mention the variable. The hypergraph partitioning tool recursively partitions the set of nodes into two (balanced) parts while attempting to minimize the number of edges across. A small number of crossing edges translates into a small number of variables shared between two sets of clauses. Hence the resulting dtree is expected to have relatively small cutsets. The degree of balance is controlled by specifying what is called the balance factor. We have used a balance factor of 35 in our experiments, which tells the program not to let the ratio of the two partitions in size (large over small) exceed $\frac{50+35}{50-35} = \frac{85}{15}$. The reader is refered to [Darwiche and Hopkins, 2001] for a more detailed description of this technique.

4 Experimental Results

To evaluate the effectiveness of our proposed ordering technique, we decided to integrate it with an existing SAT solver to see whether it would improve performance on a variety of benchmarks. Our choice of SAT solver is ZChafT [ZChafT, URL] as it has ranked first in the recent (2001 and 2002) SAT Competitions [SATEX, URL].

Our additions and modifications to ZChafT consist of 1) the package that implements dtree generation; 2) a separate chunk of code that extracts a variable group ordering from the dtree; and 3) change to the ZChafT code such that the program is given the variable group ordering as a second input (the first being the CNF) and forced to select variables from the same group, one group at a time following the specified group order. Within the same variable group, ZChaff is left to use its own heuristic, known as the Variable State Independent Decaying Sum decision heuristic [Moskewicz et a i , June 2001]. The new code is in the form of add-on and replacement files that compile with the original ZChafT package to produce the modified program; it is available for download at http://reasoning.cs.ucla.edu/dtree.sat/.

Our experiments were carried out on a Redhat system with a 930MHz CPU and 700MB of memory. The original ZChafT program and its modified version, which we call Dtree-ZChafT, were run on the same sets of CNFs. We report in Table 1 the comparative performance of these two programs. Note that only those instances that were solved by both programs are included. Instances on which Dtree-ZChafT succeeded but ZChafT ran out of memory are reported in Table 3; the reverse case did not happen for the given test suite. Some of our benchmarks are from [Aloul, URL]; others are from the SATL1B website [SATLIB, URL].

The times shown are in seconds and each represent the total time for all instances in the group. We generated two dtrees per CNF and chose the one with smaller width[4]. During the generation of each dtree, we repeated each hypergraph partitioning step twice and chose the smaller cut. The reported "Dtree Time" includes the time to thus obtain a dtree and that to compute a variable group ordering from it. Adding "Dtree Time" and "Dtree-ZChafT Time" gives the actual running time of Dtree-ZChafT, which is compared with that of the original ZChafT. Figures in bold highlight the benchmarks on which the proposed ordering lead to an improvment. Scores on a single-instance basis are shown in the last column. Considering whole groups, we observe that Dtree-ZChafT leads to improvement on seven out of the ten groups by a factor of between 1.1 and 7.8. Considering individual instances, it also leads to improvement on the majority of them. For a more detailed picture, we have included in Table 2 individual results on a select number of typical instances from each group.

We would like to point out that most of these CNFs are relatively "hard" instances for ZChafT—they require more than a few seconds. On instances where ZChafT finishes in a flash—such as til6, parl6, many of those in UF250, and some of those in Pigeonhole—our proposed ordering was not helpful since generating the dtree alone could take more time than that needed by the plain ZChafT to solve the SAT problem. Finally, to complete the picture, Table 3 reports on instances (not included in the first two tables) on which ZChafT ran out of memory but Dtree-ZChafT succeeded. We also note that

[4]Thc notion of width captures both the cutset sizes and the degree of balance; a formal definition can be found in [Darwiche, 2001].

Table 1: Overall results on all benchmarks

Benchmark	#Instances	SAT/UNSAT	Dtrce Time	Dtree-ZChaff Time	ZChaff Time	Improved Instances
Pigeonhole	8	UNSAT	16	944	7368	5
FPGA-UNSAT	9	UNSAT	64	3379	26849	9
URQ	8	UNSAT	6	236	927	8
UUF250	100	UNSAT	533	32040	41292	84
UF250	100	SAT	538	3515	4378	22
FPGA-SAT	11	SAT	38	21251	18207	6
DIFP_A	14	SAT	447	54070	142755	13
DIFP.W	14	SAT	764	103726	228653	7
ii16	10	SAT	223	73	63	0
par 16	10	SAT	33	23	9	0

Table 3: Instances on which ZChaff failed

Instance	#Vars/#Clauses	Dtree Time	Dtrce-ZChaff Time
hole 14	210/1485	6	3636
hole 15	240/1816	8	14977
fpgal L20_uns_rcr	440/4220	17	7041
urq3_5	46/470	1	589
urq3_6	46/470	1	181
urq3_7	46/470	1	577
urq3_8	46/470	1	1366

Algorithm 2 *sat(Theory* : C, *Dtreenode : T)*

```
1: if there is an inconsistent clause in $C$ then
2:     return $false$
3: if T is $null$ then
4:     return $true$
5: $V = T.cutset$
6: for all instantiations $\alpha$ of $V$ do
7:     if $sat(C|_\alpha, T.left) \wedge sat(C|_\alpha, T.right)$ then
8:         return $true$
9: return $false$
```

the total running time of all experiments was about 2 weeks.

5 Decompositional Semantics of the Heuristic

We show in this section that when the proposed variable ordering heuristic is integrated with a DP solver that employs conflict-directed backtracking, such as ZChaff, it produces a divide-and-conquer behavior in which the solver recursively breaks down the SAT problem into smaller sub-problems that are then solved independently. Specifically, even though the solver is using a DP search method, we will show that it emulates Algorithm 2, which explicitly uses the dtree to realize a divide-and-conquer search. One significance of this correspondence is that we can now offer structure-based guarantees on the complexity of DP solvers that use such orders. Such guarantees have usually been restricted to DP solvers based on resolution [Davis and Putnam, 1960; Dechter and Rish, 1994], which have not been as influential for SAT given their intractable space complexity [Davis and Putnam, 1960; Davis *et al,* 1962].

Algorithm 3 *sat(Thcory* : C, *Dtreenode : T)*

```
1: if there is an inconsistent clause in $C$ then
2:     return $false$
3: if T is $null$ then
4:     return $true$
5: if there is no uninstantiated variable in $T.cutset$ then
6:     return $sat(C, T.left) \wedge sat(C, T.right)$
7: select an uninstantiated variable $v$ in $T.cutset$
8: return $sat(C|_{v=true}, T) \vee sat(C|_{v=false}, T)$
```

Given a CNF *C* and a corresponding dtree, recall that each dtree node *T* corresponds to a subset of the CNF *C.* From here on, we will use *C* : *T* to refer to the set of clauses in CNF *C* corresponding to node *T.* Algorithm 2 takes two inputs: a CNF *C* and a corresponding dtree rooted at node *T.* The following invariant is maintained by the algorithm: clauses *C* : *T* are disconnected from other clauses. This is true trivially for the very first call to Algorithm 2, and remains true for recursive calls since cutsets associated with ancestors of node *T* must be instantiated by the time the recursive call is made. Algorithm 2 decomposes the clauses in *C* : *T* by setting variables in the cutset of node *T* to some value α. Specifically, clauses $C|_\alpha : T$ can now be decomposed into two smaller sets, $C|_\alpha : T.left$ a $C|_\alpha : T.right$ c h are solved independently. If both of these sets of clauses are solved successfully for some instantiation a, the original CNF *C* is declared satisfiable. Otherwise it is declared unsatisfiable.

Note that cutset variables do not need to be instantiated simultaneously as given on Line 6 of Alogirthm 2—instantiating these variables one at a time leads to the variant Algorithm 3. The order in which variables are instantiated by this algorithm is indeed consistent with the group ordering induced by the given dtree, as formalized by Definition 3.4. In fact, the only key difference between Algorithm 3 and a DP solver that uses the dtree group ordering is that when all cutset variables of dtree node *T* are instantiated, Algorithm 3 will spawn two independent computations on $T.left$ and $T.right$. However, the DP solver will sequence these two computations, potentially creating a dependence between them. That is, if a contradiction is reached while solving the second problem $T.right$ in the sequence, the search may backtrack to variables in the first problem $T.left$, therefore

Table 2: Results on select instances

Instance	#Vars/#Clauses	SAT/UNSAT	Dtree Time (s)	Dtree-ZChaff Time (s)	ZChaff Time (s)
hole 10	110/561	UNSAT	2	6	19
hole 11	132/738	UNSAT	3	22	160
hole 12	156/949	UNSAT	3	213	1580
fpga10_15_uns_rcr	300/2130	UNSAT	8	388	2279
fpga11_14_uns_rcr	308/2030	UNSAT	7	234	3968
fpga10_20_uns_rcr	400/3840	UNSAT	15	1859	8912
urq3_9	37/236	UNSAT	1	1	5
urq3_3	43/334	UNSAT	1	48	58
urq4_4	64/356	UNSAT	1	44	465
uuf250-010	250/1065	UNSAT	5	266	436
uu250-030	250/1065	UNSAT	5	65	119
uuf250-050	250/1065	UNSAT	5	286	301
uuf250-070	250/1065	UNSAT	5	913	1360
uuf250-090	250/1065	UNSAT	5	29	37
uf250-010	250/1065	SAT	5	4	0
uf250-030	250/1065	SAT	5	95	55
uf250-050	250/1065	SAT	5	5	3
uf250-070	250/1065	SAT	5	1	0
uf250-090	250/1065	SAT	5	70	229
fpgal0_9_sat_rcr	135/549	SAT	2	1	3
fpgal2-9_sat_rcr	162/684	SAT	3	119	145
fpgal3_9_sat_rcr	176/759	SAT	3	848	265
difp_19_99.arr_rcr	1201/6563	SAT	29	209	401
difp_20.99_arr_rcr	1201/6563	SAT	29	9	220
difp_21.99_arrj-cr	1453/7967	SAT	37	1393	17968
difp_19.3_wal_rcr	1755/10446	SAT	50	294	1041
difp_20_3_wal_rcr	1755/10446	SAT	50	49	838
difp_21_3_waLrcr	2125/12677	SAT	66	4206	28495
iil6al	1650/19368	SAT	34	0	0
par 16-1	1015/3310	SAT	6	1	0

eliminating the benefit of decomposition. As we discuss next though, this is impossible to happen if the solver employs conflict-directed backtracking, as in ZChaff. Specifically, we will show that in such a case, the solver will effectively conduct two independent searches on the sub-problems corresponding to $T.left$ and $T.right$. Hence, the combination of a DP solver, with our ordering heuristic, and conflict-directed backtracking corresponds to an iterative implementation of the divide-and-conquer search conducted by Algorithm 3.

We start by offering a brief description of conflict-directed (nonchronological) backtracking and refer the reader to [Silva and Sakallah, 1996] for more details. In a DP solver, a variable is instantiated either as a *decision* or as an implication through *unit propagation.* In the latter case, the assignments previously made and causing the clause to become unit are recorded as the *causes* of the implication. These recordings enable a backward construction of an *implication graph* when a conflict (contradiction) occurs. With careful analysis of the implication graph, conflict-directed backtracking chooses a decision variable such that, by backtracking to it, at least one of the causes of the conflict is eliminated. One key property of this form of backtracking is this. Suppose that after instantiating variables V, we split the SAT problem into two independent pieces: one involving variables V_L and the other involving variables *VR.* We solve the first problem by instantiating variables V_L, and then start solving the second problem involving variables V_R, only to realize that this problem is not satisfiable under the current settings of variables $V \cup V_L$. Conflict-directed backtracking is guaranteed *not* to backtrack to any of the variables in V_L as it is clever enough to realize that none of these variables are contributing to the contradiction. Instead, conflict-directed backtracking is guaranteed to immediately backtrack to some variable in *V.* Hence, even though the two problems are sequenced, the resulting behavior is similar to that of Line 6 of Algorithm 3.

We will now present an important implication of this correspondence we just established . Suppose that our CNF has a *connectivity graph G,* which is known to have a *treewidth* w.[5] It is known that a dtree must then exist whose height is $\log n$, where n is the number of clauses, and in which every cutset has size $\leq w$ [Darwiche, 2001]. It is also known that running a divide-and-conquer algorithm, such as Algorithm 3, on such a dtree will lead to a time complexity of $O(n \exp(w \log n))$. This complexity will then apply to a DP solver that uses the group ordering derived from the men-

[5]The connectivity graph *G* is obtained by including a vertex to *G* for each CNF variable, and then adding an edge between two vertices iff their variables appear in the same clause. Treewidth is a positive integer that measures graph connectivity: the less connected the graph, the smaller the treewidth. Trees have treewidth 1.

tioned dtree, and employs conflict-directed backtracking.[6]

The time complexity of divide-and-conquer algorithms, such as Algorithm 3, can be reduced by caching of results. Specifically, each time a call is made to *sat(C,T),* the cutsets associated with ancestors of dtree node *T* are guaranteed to be instantiated to some *a.* Moreover, *C* is a CNF that results from applying the instantiation a to the original CNF we started with. It is hence possible that the call *sat(C, T)* will be made multiple times for the same *C* since the original CNF may lead to the same CNF *C* under different variable instantiations. In fact, if all results of *sat(C, T)* are cached and their rc-computation is avoided, the complexity of the divide-and-conquer algorithm given above drops to 0(nexp(w)) [Darwiche, 2001]. This is a linear complexity for any CNF whose connectivity graph has a bounded treewidth.

It is interesting to note, however, that for unsatisfiable sub-problems, caching is already in place if the DP solver implements a conflict-based learning mechanism, as is the case with ZChaff. Hence, not only does the use of our ordering heuristic on top of ZChaff correspond to the divide-and-conquer Algorithm 3, but also to a version of it where unsatisfiable sub-problems are cached so they are not conquered again. There is still an opportunity though to improve performance by caching solutions of satisfiable sub-problems, but that would require a more substantial modification to a DP solver, which is beyond the scope of this paper.

We close this section by a note on static versus dynamic variable ordering heuristics. A static heuristic is one which fixes the variable order before search has started. A dynamic heuristic computes the order during search and is more promising from a pruning viewpoint since the ordering of variables can be done in the context of existing variable settings. However, dynamic heuristics can incur substantial overhead. Our proposed ordering heuristic is neither static nor dynamic, since the order of variables within a group is decided dynamically, while the order of groups is determined statically. A related static variable ordering heuristic is MINCE [Aloul *et al*, Nov 2001], which is mostly for OBDDs but applies to SAT problems as well. MINCE is also a structural heuristic, but has a different semantics than that of our heuristic. Moreover, although the integration of MINCE with ZChaff has been attempted, no experimental results are provided to this effect in [Aloul *et al,* Nov 2001] except for saying "Our preliminary experiments with the recently published Chaff SAT solver indicate that MINCE is not helpful on most standard benchmarks." MINCE was shown to be quite helpful in conjunction with GRASP [Silva and Sakallah, 1996] though, but the benchmarks used appear too easy for ZChaff. In particular, MINCE did best *on pigeonhole, iil 6* and *par 16* which are among the more difficult problems tried, but they become easy problems for ZChaff; see Table 1.

[6]We are assuming here that time used by conflict-directed backtracking is dominated by the mentioned complexity, which appears to be a reasonable assumption given the low overhead recorded for this type of backtracking [Silva and Sakallah, 1996].

6 Conclusion

We have proposed a structure-based variable ordering heuristic for use by SAT solvers based on Davis-Putnam search. We have shown that when the ordering heuristic is used by a solver that employs conflict-directed backtracking, it leads to a divide-and-conquer behavior in which the SAT problem is divided recursively into smaller problems that are solved independently. Based on this decompositional behavior, one can then provide formal guarantees on the complexity of DP solvers in terms of the structure of given SAT problems. We have implemented this heuristic on top of the fastest SAT solver published—ZChaff- -where we demonstrated its effectiveness in significantly boosting performance on a range of benchmarks. The proposed ordering works particularly well on the Pigeonhole, FPGA-UNSAT, and URQ instances, all of which are unsatisfiable theories with structure.

Acknowledgment

This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

References

[Aloul *et al,* Nov 2001] Fadi Aloul, Igor Markov, and Karem Sakallah. Faster sat and smaller bdds via common function structure. In *ICCAD'O1*

[Aloul, URL] http://www.eecs.umich.cdurfaloul/benchmarks.html.

[Bayardo and Pehoushek, 2000] Roberto Bayardo and Joseph Pehoushek. Counting models using connected components. In *AAAI'00.*

[Darwiche and Hopkins, 2001] Adnan Darwichc and Mark Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143,* pages 180 191. Springer-Vcrlag, 2001.

[Darwiche, 2001] Adnan Darwiche. Recursive conditioning. *Artificial Intelligence,* 126(1-2):5-41, 2001.

[Darwiche, 2002] Adnan Darwiche. A compiler for deterministic decomposable negation normal form. In *AAA! '02.*

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *JACM,* 7:201-215, 1960.

[Davis *et al,* 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *JACM,* (5)7:394-397, 1962.

[Dechter and Rish, 1994] Rina Dechter and Irina Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *KR '94.*

[Li and Anbulagan, 1997] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI'97,*

[Moskewicz et al., June 2001] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC'01.*

[SATEX, URL] http://www.lri.fr/`simon/satex.

[SATLIB, URL] http://www.intellektik.informatik.tu-darmstadt.de/SATLIB.

[Silva and Sakallah, 1996] Joao Silva and Karem Sakallah. Grasp a new search algorithm for satisfiability. In *ICCAD'96.*

[ZChaff, URL] http://www.ee.princeton.edu/chaff/zchaff.php.