# Toward Good Elimination Orders for Symbolic SAT Solving

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

## Abstract

*Fundamentally different from DPLL, a new approach to SAT has recently emerged that abandons search and enlists BDDs to symbolically represent clauses of the CNF. These BDDs are conjoined according to a schedule where some variables may be eliminated by quantification at each step to reduce the size of the intermediate BDDs. SAT solving then reduces to checking whether the final BDD is the zero constant. For this approach to be practical, finding a good quantification schedule is critical. We study the use of a variable elimination algorithm for this purpose, as well as two specific methods for the generation of good elimination orders based on CNF structure. While neither method appears to dominate, we show how one can heuristically select the better using the notion of width. We implement a symbolic SAT solver based on these techniques and evaluate its efficiency and robustness on a set of benchmarks against five other solvers, each having unique characteristics, including winners of the most recent SAT competition.*

## 1. Introduction

Given a propositional theory in Conjunctive Normal Form (CNF), the problem of Propositional Satisfiability (SAT) is to determine whether there exists a truth assignment to its variables under which it evaluates to true. Owing to the significance of SAT in complexity theory as well as its widespread applications in various fields of computer science, substantial effort has been put into the engineering of efficient SAT solvers. The majority of state-of-the-art SAT solvers [4] are based on the DPLL algorithm [12], where a search is conducted in the space of variable assignments for one that satisfies the target propositional theory. After decades of research the efficiency of these solvers has reached unprecedented heights thanks to a combination of modern techniques such as new variable ordering heuristics, watched literals, conflict-directed backtracking, nogood learning, and restarts [3, 25, 28, 39, 16].

A fundamentally different approach to SAT has recently emerged [1, 23, 30], which eliminates search by symbolically representing clauses of the CNF with Binary Decision Diagrams (BDDs) [6]. When these BDDs are conjoined, SAT reduces to checking in constant time whether the result is identical to zero. On the theoretical side, it has been proved that this approach, which we refer to as *Symbolic SAT Solving*, is incomparable to those based on resolution, including the DPLL search (whose underlying proof system is resolution) [17]. Specifically, there are known classes of problems which are easy for the former but exponentially hard for the latter, and vice versa. This analysis is matched by empirical studies where the comparative performance of the two methods has been observed to be problem-dependent [36, 21, 30]. Given the overwhelming popularity enjoyed by DPLL, these results justify more extended research on techniques suited for the BDD-based alternative.

For symbolic SAT solving to be practical, variables are eliminated by existential quantification as early as possible to reduce the size of the intermediate BDDs constructed, and a good quantification schedule is critical. We show in this paper that the *variable elimination* procedure, which has been studied in the field of probabilistic inference [14, 40], can be used as an approach to quantification scheduling. Furthermore, we describe two specific methods, recursive decomposition [11] and min-cut linear arrangement [3], that can be used to generate good elimination orders—which are crucial for efficient variable elimination in general—for symbolic SAT solving in particular. While neither method appears to dominate, we show that one can heuristically select the better based on the *width* of the elimination order generated.

This selection heuristic is theoretically justified in that variable elimination algorithms are known to have a complexity that is exponential only in the width of the elimination order used [14]. In the context of symbolic SAT solving, the width plus 1 translates into the maximum number of variables any intermediate BDD can have. Choosing an order with a smaller width therefore helps minimize the size of these intermediate BDDs.

On the empirical side, we implement a symbolic SAT solver based on these techniques and compare it on a set of benchmarks against five other programs, including zChaff [28] and March_eq [18], winners of the 2004 SAT Competition in the industrial and handmade category respectively [32]. While some of these solvers exhibit a highly optimized performance on certain benchmarks, we observe that our new symbolic solver is the only one to have successfully solved all benchmarks on which the comparison is made.

Interestingly, the variable elimination algorithm we use corresponds exactly to the original Davis-Putnam (DP) procedure [13] for SAT, except that resolution is now replaced with BDD conjunction and quantification (or the *AndExists* operation which combines the two [26]). Our successful implementation of the algorithm indicates that DP can be a very practical algorithm, after all, as long as one chooses an appropriate representation for CNF clauses so that existential quantification can be efficiently carried out.

A major contribution of the present paper is therefore a theoretical study of an example of such representation, namely BDDs, and the demonstration of its practical efficiency in conjunction with good elimination orders. In related work, the ZRes SAT solver, which shall be part of our empirical study, provides another example, where zero-suppressed BDDs [22] are used to encode sets of clauses (as opposed to sets of models in the case of BDDs), and a special algorithm is designed to perform resolution on the encoded clauses [9].

The remainder of the paper is organized as follows. We start with the basics of symbolic SAT solving in Section 2. In Section 3 we describe the use of variable elimination for symbolic SAT solving as well as two specific methods that can be used to generate good elimination orders. We provide an empirical evaluation of these methods in Section 4 and conclude in Section 5.

## 2. Symbolic SAT Solving

We review in this section the definition of SAT and the fundamentals of the symbolic approach to SAT solving. As is customary, we consider propositional formulas in CNF. A CNF formula is defined as a conjunction of clauses, where each clause is a disjunction of literals. A literal is an instance of a Boolean variable or its negation. The following is an example of a four-clause CNF formula over six variables: $(u \vee x \vee y) \wedge (x \vee \neg z) \wedge (\neg u \vee w \vee z) \wedge (v \vee \neg w \vee z)$.

An instance of the SAT problem takes as input a CNF $\Delta$, and asks whether variables of $\Delta$ can be each assigned one of two Boolean values $true$ and $false$ such that $\Delta$ evaluates to $true$. If such a *satisfying assignment*, also known as a *model*, exists, the formula is declared *satisfiable*; otherwise it is declared *unsatisfiable*.

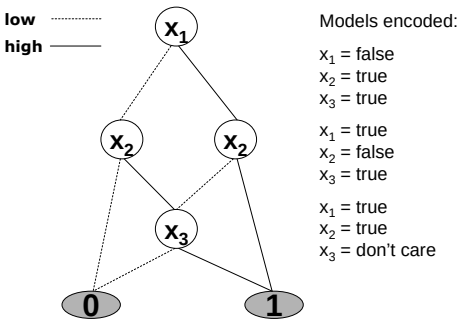


**Figure 1. A BDD and the models it encodes.**

Algorithms based on DPLL [12] constitute the predominant approach adopted by modern SAT solvers that are designed to be *complete*, i.e., guaranteed to solve the problem given sufficient resources. This approach formulates SAT as a systematic search in the space of variable assignments. To find a satisfying assignment it instantiates variables one at a time and backtracks whenever a contradiction is reached. See [39] for a detailed description of this algorithm along with additional techniques commonly employed.

Unlike DPLL that searches for a model, the symbolic approach bases itself on the fact that the set of all models can be encoded as a BDD which can be obtained without search. Figure 1 depicts an example BDD over three variables together with the models it encodes. Formally, a BDD is a rooted directed acyclic graph where there are at most two sinks, labeled with 0 and 1 respectively, and every other node is labeled with a Boolean variable and has exactly two children, distinguished as $low$ and $high$ [6]. A BDD represents a propositional theory whose models can be enumerated by taking all paths from the root to the 1-sink: taking the low (high) child of a node labeled with $x_i$ corresponds to assigning $false$ ($true$) to $x_i$; in case $x_i$ does not appear on the path, it is marked as "don't care" which means either value can be assigned.

In practice, BDDs are usually maintained so that variables appear in the same order on any path from the root to a sink, that there is no node whose two children are identical, and that there are no isomorphic sub-graphs. Under these conditions BDDs are known to be a canonical form, meaning that there is a unique BDD for any propositional theory under a given variable order [6]. Moreover, any binary operation on two BDDs with the same variable order can be carried out using the *Apply* algorithm [6], whose complexity is linear in the product of the operand sizes.

To solve a CNF $\Delta = c_1 \wedge c_2 \wedge \ldots \wedge c_m$ for SAT in the symbolic approach, therefore, one can convert each of the $m$ clauses into a BDD and conjoin them one by one using *Apply* to produce a BDD for $\Delta$: $BDD(\Delta) = \bigwedge_{i=1}^{m} BDD(c_i)$.

Owing to the canonicity of BDDs, the result is then identical to the 0-sink if and only if $\Delta$ is unsatisfiable.

A potential hurdle to this incremental BDD construction is that the intermediate results may be too large for the conjunction to be efficient. Besides, it would be an overkill in any case to actually produce the final BDD that encodes all models, because to test satisfiability all one need know is whether there *is* a model. In other words, one need only check if $\exists X \Delta = true$, where $X = \{x_1, \ldots, x_n\}$ are the variables of $\Delta$, which amounts to the computation of

$$\exists X BDD(\Delta) = \exists X \bigwedge_{i=1}^{m} BDD(c_i). \quad (1)$$

Recall that according to the rules of quantified Boolean logic, an existential quantifier can be restricted to a conjunct if the variable being quantified does not appear in other conjuncts. That is,

$$\exists x[f(x, Y) \wedge g(Z)] \equiv [\exists x f(x, Y)] \wedge g(Z) \quad (2)$$

where $Y$ and $Z$ are sets of variables not including $x$. This means that when BDDs are conjoined (two at a time) in Equation 1, $\exists x_1, \ldots, x_n$ need not all be performed at the end, but can be moved to the individual conjunction steps by virtue of Equation 2. As variables are quantified early on, the intermediate BDDs are expected to have reduced sizes because they now have fewer variables.

This technique is known as *early quantification* [35, 7, 19], and has been extensively used in such areas as symbolic model checking where current state variables are quantified in image computation [7]. The *quantification scheduling* problem is then to order the conjuncts so as to minimize the size of intermediate BDDs. Finding optimal quantification schedules is known to be NP-hard [19], and various heuristics have been proposed and studied in the context of image computation [31, 38, 27]. In the next section we describe *variable elimination* [14, 40] as an approach to quantification scheduling for symbolic SAT solving, as well as two specific methods for the generation of good elimination orders based on the structure of the CNF formula.

## 3. Toward Good Elimination Orders

We describe here the variable elimination algorithm as pertains to our specific task of symbolic SAT solving, and refer the reader to [14] for a detailed presentation of the technique in general. See Algorithm 1 for the pseudocode.

To start with, each clause $c_i$ of CNF $\Delta = c_1 \wedge c_2 \wedge \ldots \wedge c_m$ is converted into a BDD $BDD(c_i)$ (implicit on Line 5). Given a total order $\pi$ on the variables of $\Delta$—known as an *elimination order*, a *bucket* $B_v$ is created for each variable $v$ (Lines 1 and 2) and each $BDD(c_i)$ is assigned to the bucket of its first variable according to order $\pi$ (Lines 3–5). These

**Algorithm 1** VE (CNF $\Delta$, order $\pi$)

```
1:  for each variable v of Δ do
2:      create empty bucket B_v
3:  for each clause c of Δ do
4:      v = first variable of c according to order π
5:      B_v = B_v ∪ {BDD(c)}
6:  for each variable v of Δ in order π do
7:      if B_v is not empty then
8:          BDD_r = conjunction of all elements of B_v
9:          if BDD_r = zero then
10:             return UNSATISFIABLE
11:         BDD_r = ∃v BDD_r
12:         u = first variable of BDD_r according to order π
13:         B_u = B_u ∪ {BDD_r}
14: return SATISFIABLE
```

buckets are then processed one at a time, using order $\pi$ and skipping empty buckets (Lines 6–13). When a bucket $B_v$ is processed, all BDDs in it are conjoined (Line 8) in arbitrary order and variable $v$ is existentially quantified (Line 11) so that it is *eliminated* from the result, which is then thrown into the bucket of its first variable (Lines 12 and 13). During this iteration process if any BDD constructed ends up being the zero constant, the CNF is declared unsatisfiable (Line 10); otherwise it is declared satisfiable (Lines 14).

As we have pointed out, it is not hard to ascertain that this algorithm corresponds exactly to the original DP procedure [13] for satisfiability, except that resolution is now replaced with BDD conjunction and quantification, attaining the same purpose of variable elimination. The compactness of the BDD representation then affords an opportunity for efficiency to be attained where the original DP may suffer from the well-known problem of space explosion.

The complexity of Algorithm 1 will greatly depend on the elimination order used, and the quality of the latter is typically measured by its *width* [14]. The width of an elimination order can be defined with respect to the *connectivity graph* of the CNF formula, where each CNF variable becomes a node of the graph and there is an edge between two nodes if the corresponding variables appear together in some clause of the CNF. When each node is eliminated (deleted together with edges incident on it) from the graph according to some elimination order, we pairwise connect all neighbors of the node before its deletion. The maximum number of neighbors any node has right before its deletion is then defined as the width of the elimination order.

It is important to note that in the context of the above algorithm for SAT solving, the width of an elimination order plus 1 represents the maximum number of variables any intermediate BDD can have. Width minimization will hence be a step toward minimizing the size of the intermediate BDDs and the complexity of the overall elimination algorithm. We describe next two methods that can be

used to generate low-width elimination orders in general, and demonstrate in Section 4 their effectiveness for symbolic SAT solving in particular.

### 3.1. Recursive Decomposition

Recursive decomposition is one of the more competitive methods available for generating elimination orders of low width; on a set of belief networks it has been reported to outperform the *min-fill* heuristic, which was among the best techniques previously used for width minimization [11]. To use this method, one first converts the CNF into a hypergraph where each clause becomes a vertex and each variable a hyperedge connecting all the clauses in which it appears. Using an existing software tool called hMeTis [24], the vertices of this hypergraph are partitioned into two balanced (roughly equal-sized) parts while minimizing the number of hyperedges across; these two parts are then each recursively partitioned in the same fashion, until they contain a single vertex. The result of this process is referred to as a *dtree*, short for decomposition tree, and it is shown that the hypergraph partitioning process heuristically minimizes the *width* of the dtree—which we define next—and a low-width dtree can be used to induce an elimination order also of a low width [11].

The notion of dtree originated in the context of belief networks [10]. When applied to a CNF formula $\Delta$, a dtree can be formally defined as a full binary tree whose leaves correspond to the clauses of $\Delta$. See Figure 2 for an example. One can define the *variables* of a dtree node as the set of variables of the clauses under it, and the *cutset* of a dtree node as the intersection of its two children's variables, excluding all those variables that already appear in the cutsets of its ancestor nodes; for a leaf node the cutset is simply its variables, again excluding those in the cutsets of its ancestors. The *cluster* of a dtree node is then defined as its cutset, plus any of its variables that also appear in its ancestors' cutsets. The size of the largest cluster of a dtree minus 1 is defined as the *width* of the dtree [10].

Figure 2 (top) depicts a dtree for a four-clause CNF, where the cutset is shown inside each node. By traversing such a dtree in post-order one can enumerate all its (nonempty) cutsets in a sequence. For example, $\{v\}, \{w\}, \{y\}, \{x\}, \{u, z\}$ is a possible outcome of such a traversal on the dtree of Figure 2. A sequence of variable groups can be considered as a constraint on the variable order. It is shown in [11] that any elimination order consistent with such a cutset sequence is guaranteed to have no greater width than the dtree. This effectively provides a method for the generation of low-width elimination orders through low-width dtrees constructed by recursive decomposition.

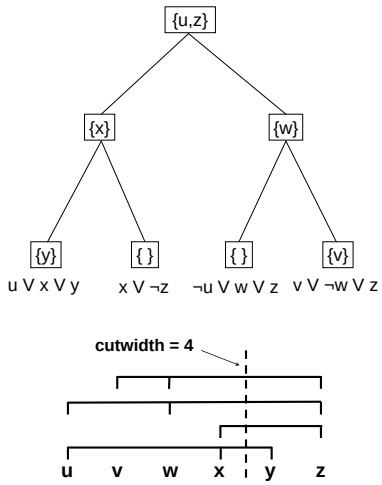


**Figure 2. A dtree and the cutwidth of a variable order for a four-clause CNF.**

### 3.2. Min-Cut Linear Arrangement

Min-Cut linear arrangement has been proposed as a method to generate variable orders that reduce the BDD size as well as the complexity of DPLL-based SAT solving [3]. Similar to recursive decomposition, the technique of hypergraph partitioning [8] is used, but this time each variable of the CNF becomes a vertex and each clause a hyperedge connecting its variables. Unlike in recursive decomposition where every partitioning step is independent of the others, a procedure known as *terminal propagation* [15] is used so that the result is not a dtree, but a vertex order that is expected to have a small *cutwidth*, which is the maximum number of hyperedges one can cut when the vertices of the hypergraph are arranged in a linear order.

Figure 2 (bottom) illustrates the cutwidth of variable order $u, v, w, x, y, z$ for the same four-clause CNF: the largest cut, of size $4$, occurs between variables $x$ and $y$. The use of min-cut linear arrangement to generate CNF variable orders of low cutwidth is referred to as the MINCE heuristic in [3]. Although this heuristic is not designed with variable elimination in mind, we point out here that a variable order of bounded cutwidth is immediately an elimination order of bounded width, because it is known that for a given variable order, $width \leq k \cdot cutwidth$ [5, 34], where $k$ is the maximum clause length of the CNF minus 1. This observation seems in line with our experimental results in Section 4, where MINCE elimination orders appear to be competitive with those produced by recursive decomposition.

Recursive decomposition and min-cut linear arrangement have been implemented as the HGR2BDT program of [11] and the MINCE program of [3]. In Section 4 we will refer to them as DTREE and MINCE, respectively.

| Instance | Variables | Clauses | VE Using DTREE | | | VE Using MINCE | | |
|---|---|---|---|---|---|---|---|---|
| | | | Width | Time | Peak Nodes | Width | Time | Peak Nodes |
| marg3x6 | 106 | 1232 | 72 | 5.62 | 1630090 | 39 | 0.47 | 15330 |
| urquhart-s3-b6 | 54 | 688 | 52 | 1311.82 | 27538812 | 26 | 0.23 | 9198 |
| urquhart-s4-b2 | 70 | 594 | 39 | 0.51 | 131838 | 25 | 0.23 | 7154 |
| am_5_5 | 1076 | 3677 | 55 | 5.22 | 977032 | 639 | $\geq 2000$ | – |
| homer17 | 286 | 1742 | 129 | 133.27 | 5039482 | 241 | $\geq 2000$ | – |
| fpga_10_9_sat_rcr | 135 | 549 | 92 | 1.28 | 140014 | 101 | 2.56 | 794094 |

**Table 1. Recursive decomposition vs. min-cut linear arrangement**

## 4. Experimental Results

The purpose of our experiments is twofold. First, we compare the quality of elimination orders generated by DTREE and MINCE in terms of the corresponding running times of the variable elimination algorithm on BDDs. Second, we show that the combined use of both methods through a selection heuristic can be the basis for an efficient and robust symbolic SAT solver, which provides a quality alternative for situations where the standard DPLL-based solvers may suffer.

To this end, we implemented a SAT solver based on Algorithm 1, using the CUDD package from the University of Colorado [33] for all BDD manipulations. Note that in using CUDD one needs a BDD variable order, in addition to an elimination order required by Algorithm 1. For the former purpose we have used the MINCE orders, with dynamic reordering turned off, for all benchmarks except the pigeonhole instances (for reasons we are yet to identify, the original variable orders work out much better for pigeonhole). In other words, the BDD variable order and the elimination order will coincide when MINCE is used also for the latter. Our selection between DTREE and MINCE hereafter will apply only to the elimination orders.

In our first set of experiments, we ran our solver twice on the same set of benchmarks—we describe our selection of these benchmarks later—with elimination orders generated by DTREE and MINCE respectively. We observe that neither method dominates across the board. While on many instances the two running times are close, occasionally the difference is substantial—Table 1 includes some of these instances. The running times shown are in seconds and include preprocessing (generation of DTREE and MINCE orders) times; the Peak Nodes column gives the maximum number of BDD nodes present in memory during the variable elimination process—an indication of its complexity. All our experiments were run on a 2.4GHz processor with 3.7GB of RAM under Red Hat Linux.

Table 1 clearly suggests a strong correlation between width, running time, and number of peak nodes. Specifically, where the two methods lead to significantly different running times for a given instance, the better performance is usually associated with the elimination order of smaller width. For our second set of experiments, therefore, we set our SAT solver to try both methods as a preprocessing step, and automatically select the elimination order with smaller width to be used in the main algorithm.

We now compare the performance of this program, which we will refer to as VE, with that of five other SAT solvers: ZRes, zChaff, Cassatt, CirCUs, and March_eq. Our choice of these solvers is due to their unique characteristics as well as evidenced efficiency. Performing resolution on zero-suppressed BDDs, ZRes [9] is a solver very similar to ours in spirit, although the actual algorithms are quite different. Based on DPLL, zChaff [28] has been one of the most competitive SAT solvers published: it won the SAT Competition in 2001 and 2002, and again in 2004 in the industrial category [32]. Cassatt implements a novel breadth-first search also using zero-suppressed BDDs, and is reported to surpass zChaff on the pigeonhole and urq-3–8 benchmarks [29]. CirCUs is a hybrid program combining symbolic and DPLL-based SAT solving, and is also reported to outperform zChaff on a number of benchmark families [23]. Finally, March_eq has been the winner of the 2004 SAT Competition in the handmade category [32].

We have been able to download or obtain by email copies of these SAT solvers except for CirCUs (whose release, as part of the VIS package, is slated for the end of summer 2004 according to its authors). In order to compare with CirCUs, we make part of our benchmarks the same as used in their paper [23] so that we may directly quote their results. Other benchmarks are selected so that we include classical problems from the literature as well as those from the 2003 SAT Competition, all downloadable from the Satisfiabiliy Library [20] and Fadi Aloul's SAT Benchmarks [2]: of the pigeonhole family only five instances are available from [20], but additional instances can be easily generated using the provided description; the urq-3–8 families are based on the biconditional formulas described in [37]; the simon and bevan families are from the handmade category, and the kukula and sat02 families from the industrial category, of the 2003 SAT Competition; the fpga-sat family is provided by Fadi Aloul [2].

| Benchmark | Ins | Variables | Clauses | VE | | | | ZRes | zChaff | Cassatt | CirCUs | March_eq |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | MINCE | DTREE | Main | Total | | | | | |
| phole6 | 1 | 42 | 133 | 0.07 | 0.04 | 0.01 | 0.12 | 0.06 | 0.01 | 0.01 | – | 0.01 |
| phole7 | 1 | 56 | 204 | 0.12 | 0.06 | 0.02 | 0.20 | 0.11 | 0.03 | 0.02 | – | 0.14 |
| phole8 | 1 | 72 | 297 | 0.14 | 0.08 | 0.02 | 0.24 | 0.18 | 0.11 | 0.02 | – | 1.35 |
| phole9 | 1 | 90 | 415 | 0.13 | 0.12 | 0.03 | 0.28 | 0.31 | 0.80 | 0.03 | – | 16.20 |
| phole10 | 1 | 110 | 561 | 0.16 | 0.15 | 0.05 | 0.36 | 0.50 | 13.09 | 0.03 | – | 291.64 |
| phole11 | 1 | 132 | 738 | 0.16 | 0.19 | 0.07 | 0.42 | 0.82 | 128.59 | 0.04 | – | (0) |
| phole12 | 1 | 156 | 949 | 0.20 | 0.26 | 0.10 | 0.56 | 1.33 | 1204.99 | 0.08 | – | (0) |
| phole13 | 1 | 182 | 1197 | 0.20 | 0.33 | 0.17 | 0.70 | 1.94 | (0) | 0.07 | – | (0) |
| phole14 | 1 | 210 | 1485 | 0.26 | 0.43 | 0.64 | 1.33 | 3.03 | (0) | 0.09 | – | (0) |
| phole15 | 1 | 240 | 1816 | 0.30 | 0.55 | 4.28 | 5.13 | 4.38 | (0) | 0.11 | – | (0) |
| urq-3 | 10 | 36∼46 | 220∼470 | 1.15 | 0.65 | 0.22 | 2.02 | 1.73 | 454.35 (6) | 0.96 | – | 0.00 |
| urq-4 | 10 | 64∼87 | 356∼1030 | 1.92 | 1.50 | 0.41 | 3.83 | 5.33 | (0) | 15.40 | – | 0.00 |
| urq-5 | 10 | 119∼127 | 978∼1336 | 2.89 | 2.69 | 0.94 | 6.52 | 12.53 | (0) | 120.63 | – | 0.00 |
| urq-6 | 10 | 166∼180 | 1324∼1756 | 4.28 | 4.25 | 1.45 | 9.98 | 26.95 | (0) | 1241.57 (6) | – | 0.04 |
| urq-7 | 10 | 229∼250 | 1880∼2420 | 6.91 | 6.21 | 2.53 | 15.65 | 50.41 | (0) | 6174.53 (7) | – | 0.08 |
| urq-8 | 10 | 304∼327 | 2486∼3252 | 10.27 | 9.86 | 5.22 | 25.35 | 94.31 | (0) | (0) | – | 0.10 |
| bevan/dodecahedron | 1 | 30 | 80 | 0.14 | 0.02 | 0.01 | 0.17 | 0.09 | 0.01 | 0.01 | – | 0.00 |
| bevan/hcb2 | 1 | 12 | 32 | 0.07 | 0.01 | 0.01 | 0.09 | 0.04 | 0.00 | 0.01 | – | 0.00 |
| bevan/hcb3 | 1 | 45 | 288 | 0.22 | 0.06 | 0.03 | 0.31 | 0.19 | 853.56 | 0.13 | – | 0.00 |
| bevan/hcb4 | 1 | 112 | 2048 | 2.06 | 0.48 | 0.17 | 2.71 | 1.12 | (0) | (0) | – | 0.01 |
| bevan/hcb5 | 1 | 225 | 12800 | 6.39 | 5.06 | 1.87 | 13.32 | 5.38 | (0) | (0) | – | 0.09 |
| bevan/hypercube4 | 1 | 32 | 128 | 0.20 | 0.02 | 0.01 | 0.23 | 0.10 | 2.06 | 0.02 | – | 0.00 |
| bevan/hypercube5 | 1 | 80 | 512 | 0.19 | 0.12 | 0.03 | 0.34 | 0.57 | (0) | 3.27 | – | 0.00 |
| bevan/hypercube6 | 1 | 192 | 2048 | 0.84 | 0.63 | 0.24 | 1.71 | 3.31 | (0) | (0) | – | 0.01 |
| bevan/hypercube7 | 1 | 448 | 8192 | 7.17 | 4.55 | 1.89 | 13.61 | 19.32 | (0) | (0) | – | 0.04 |
| bevan/icosahedron | 1 | 30 | 192 | 0.18 | 0.03 | 0.02 | 0.23 | 0.09 | 6.74 | 0.03 | – | 0.00 |
| bevan/icos_stretch | 1 | 45 | 352 | 0.15 | 0.07 | 0.02 | 0.24 | 0.19 | 16.31 | 0.08 | – | 0.00 |
| bevan/marg* | 17 | 12∼156 | 32∼1232 | 2.07 | 1.14 | 0.42 | 3.63 | 6.38 | 229.88 (12) | 62.31 | 1.33 (14) | 0.00 |
| bevan/urqh* | 13 | 18∼226 | 96∼3168 | 3.57 | 3.05 | 0.99 | 7.61 | 11.31 | 395.36 (4) | 155.02 (12) | 4.81 (12) | 0.02 |
| bevan/urqh1c* | 13 | 15∼191 | 64∼1984 | 2.21 | 1.79 | 0.54 | 4.54 | 7.91 | 4094.83 (6) | 411.57 | 6.77 | 0.01 |
| simon/urquhart* | 13 | 45∼288 | 264∼1116 | 2.49 | 1.86 | 0.45 | 4.80 | 19.14 | 423.65 (1) | 12.01 | 4.59 (10) | 0.00 |
| simon/x1* | 19 | 106∼382 | 282∼1018 | 4.44 | 3.27 | 0.52 | 8.23 | 81.30 | (0) | 21.2 | 6.97 | 0.00 |
| simon/x2* | 9 | 118∼382 | 314∼1018 | 2.18 | 1.57 | 0.25 | 4.00 | 40.08 | (0) | 6.52 | 3.38 | 0.00 |
| kulula/am_4_4 | 1 | 433 | 1458 | 0.60 | 0.44 | 0.22 | 1.26 | (0) | 0.34 | (0) | – | 0.88 |
| kulula/am_5_5 | 1 | 1076 | 3677 | 1.29 | 1.29 | 2.64 | 5.22 | (0) | 66.24 | (0) | – | 201.55 |
| kulula/am_6_6 | 1 | 2269 | 7814 | 3.10 | 3.36 | 486.22 | 492.68 | (0) | (0) | (0) | – | (0) |
| sat02/homer17 | 1 | 286 | 1742 | 0.38 | 0.47 | 132.42 | 133.27 | (0) | (0) | 0.18 | – | (0) |
| sat02/homer18 | 1 | 308 | 2030 | 0.38 | 0.57 | 939.87 | 940.82 | (0) | (0) | 0.19 | – | (0) |
| fpga10_8_sat_rcr | 1 | 120 | 448 | 0.93 | 0.12 | 0.44 | 1.49 | (0) | 0.02 | 0.07 | – | 0.01 |
| fpga10_9_sat_rcr | 1 | 135 | 549 | 0.93 | 0.16 | 0.19 | 1.28 | (0) | 1.92 | 0.10 | – | 0.01 |
| fpga12_8_sat_rcr | 1 | 144 | 560 | 0.68 | 0.17 | 0.91 | 1.76 | (0) | 65.52 | 0.11 | – | 0.01 |
| fpga12_9_sat_rcr | 1 | 162 | 684 | 1.08 | 0.20 | 1.60 | 2.88 | (0) | 3.05 | 0.17 | – | 0.01 |
| fpga12_10_sat_rcr | 1 | 180 | 820 | 0.94 | 0.25 | 8.03 | 9.22 | (0) | 813.06 | 0.12 | – | 0.02 |
| fpga12_11_sat_rcr | 1 | 198 | 968 | 1.94 | 0.28 | 15.28 | 17.50 | (0) | 553.65 | 0.23 | – | 0.02 |
| fpga12_12_sat_rcr | 1 | 216 | 1128 | 0.56 | 0.34 | 39.06 | 39.96 | (0) | (0) | 0.21 | – | 0.02 |
| fpga13_9_sat_rcr | 1 | 176 | 759 | 1.35 | 0.22 | 2.28 | 3.85 | (0) | (0) | 0.29 | – | 0.02 |
| fpga13_10_sat_rcr | 1 | 195 | 905 | 1.29 | 0.27 | 10.17 | 11.73 | (0) | 346.45 | 0.26 | – | 0.02 |
| fpga13_11_sat_rcr | 1 | 215 | 1070 | 1.7 | 0.32 | 24.57 | 26.59 | (0) | 347.87 | 0.20 | – | 0.03 |
| fpga13_12_sat_rcr | 1 | 234 | 1242 | 0.55 | 0.38 | 67.93 | 68.86 | (0) | (0) | 0.18 | – | 0.03 |

**Table 2. Running times of six SAT solvers (shown in parentheses is the number of instances solved)**

The results of this comparison are shown in Table 2, where running times are in seconds and represent the total for each group, whose size is given in the second column. The number of variables and the number of clauses are shown in the third and fourth column, as a range in case of multiple instances in a group. The time limit was 2000 seconds per instance for all solvers. In case a solver did not solve all instances of a group, either because it ran out of time or memory, the number of instances solved is shown in parentheses. To offer a clearer picture of the overhead involved in generating both orders while using only one of them, the running time of VE is broken down into three parts: time to generate the elimination order by MINCE and by DTREE is shown respectively in columns 5 and 6; the actual SAT solving time is shown in column 7; column 8 gives the total. As we have alluded to, for CirCUs we only include results as reported in [23]; where data is unavailable a dash is shown instead. Running times reported in [23] for CirCUs are based on the same CPU speed as ours (and 500 MB of RAM); hence we include them as is.

Our main observation here is that VE is the only solver to have successfully solved all instances of every benchmark family. Although the cost of generating the elimination orders (columns 5 and 6) is often nontrivial compared with that of the actual SAT solving (column 7), it appears to be reasonable insurance against potential pitfalls caused by bad orders, as indicated by Table 1.

While the efficiency of VE versus zChaff on these benchmarks is apparent—for many groups zChaff failed to solve any instance at all given the time limit and available memory, we look more closely at the comparison between VE and the other four solvers. The performance of ZRes was comparable to that of VE on many of the benchmarks, but significantly worse on some (e.g., simon/x1* and simon/x2*), especially the industrial and fpga-sat instances, none of which were solved by ZRes. The pigeonhole instances are one benchmark family for which it has been proved that both resolution and symbolic solving will have an exponential complexity [17]. On these instances Cassatt has been shown to exhibit only a polynomial complexity, approximately $O(n^4)$ [29]. This seems consistent with our results where Cassatt was decidedly faster than VE, although VE still gained a remarkable speedup over zChaff and March_eq. On the urq families, it is clear that VE scales much better than Cassatt. Comparing with Cassatt on other instances, and also with CirCUs on the applicable instances, the robustness of VE is notable. All instances were successfully solved by VE, while quite a few of them were not solved by Cassatt and CirCUs given the time limit and available memory. Finally, March_eq exhibited a highly optimized performance on the urq, simon, bevan, and fpga-sat families, but failed to solve some of the pigeonhole, kukula, and sat02 instances, again speaking to the robustness of VE.

## 5. Conclusion

We have shown that symbolic SAT solving based on variable elimination can be a very efficient alternative on problems where standard SAT algorithms may suffer. We have studied two specific methods for the generation of good elimination orders based on CNF structure, and shown how one can heuristically select the better using the notion of width. In comparison with five other programs including winners of the most recent SAT Competition, our implementation of this symbolic SAT solver is the only one to have successfully solved all benchmarks we have used given the time limit and available memory.

We have also pointed out that the original DP procedure for SAT, though long in disuse, can in fact be very practical as long as one uses an appropriate representation for CNF clauses where existential quantification can be efficiently carried out. The previous work on ZRes and the present paper provide two examples of such representation, namely zero-suppressed BDDs and regular BDDs, both leading to promising results on many benchmarks where the traditional DPLL-based solvers tend to suffer. This motivates further research on alternative DP implementations in particular, and alternative SAT solving methods in general.

## Acknowledgments

Thanks to the anonymous reviewers, particularly for pointing out additional references on recent SAT algorithms. This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

## References

[1] A. S. M. Aguirre and M. Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. *Principles and Practice of Constraint Programming*, pages 121–136, 2001.

[2] SAT Benchmarks from Fadi Aloul, http://www.eecs.umich.edu/~faloul/benchmarks.html.

[3] F. Aloul, I. Markov, and K. Sakallah. Faster SAT and smaller BDDs via common function structure. In *International Conference on Computer Aided Design (ICCAD), University of Michigan*, 2001. Tool available for download at http://www.eecs.umich.edu/~faloul/Tools/mince/.

[4] D. L. Berre and L. Simon. The essentials of the SAT 2003 competition. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[5] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. *Journal of Algorithms*, 18:238–255, 1995.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[7] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on Very Large Scale Integration*, 1991.

[8] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Can recursive bisection produce routable placements? In *Proceedings of the Design Automation Conference*, 2000.

[9] P. Chatalic and L. Simon. ZRes: The old Davis-Putnam procedure meets ZBDDs. In D. McAllester, editor, *17th International Conference on Automated Deduction (CADE)*, number 1831 in Lecture Notes in Artificial Intelligence (LNAI), pages 449–454, June 2000.

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

[11] A. Darwiche and M. 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-Verlag, 2001.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, (5)7:394–397, 1962.

[13] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[14] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence*, pages 211–219, 1996.

[15] A. Dunlop and B. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 1(4):92–98, 1985.

[16] E. Goldberg and Y. Novikov. Berkmin: a fast and robust SAT-solver. In *Design Automation and Test in Europe*, pages 142–149, 2002.

[17] J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003.

[18] M. Heule and H. van Maaren. Aligning CNF- and equivalence reasoning. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[19] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *Proceedings of the International Conference on Computer Design*, pages 12–19, 1996.

[20] H. H. Hoos and T. Sttzle. SATLIB: An Online Resource for Research on SAT. In *I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pages 283–292. IOS Press, 2000. SATLIB is available online at *www.satlib.org*.

[21] J. Huang and A. Darwiche. Using DPLL for efficient OBDD construction. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[22] S. ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th international conference on Design automation*, pages 272–277. ACM Press, 1993.

[23] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[24] G. Karypis and V. Kumar. *hMeTiS: A Hypergraph Partitioning Package*, 1998. Available at http://www.cs.umn.edu/~karypis.

[25] J. Marques-Silva and K. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, 1996.

[26] K. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[27] I. Moon and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In *Proceedings of the Formal Methods in Computer Aided Design*, pages 73–90. Springer Verlag, Lecture Notes in Computer Science, Volume 1954, 2000.

[28] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, 2001.

[29] D. B. Motter and I. L. Markov. A compressed breadth-first search for satisfiability. In *Proceedings of ACM Workshop on Algorithm Engineering and Experimentation*, pages 29–42. Lecture Notes in Computer Science, Volume 2409, 2002.

[30] G. Pan and M. Y. Vardi. Search vs symbolic techniques in satisfiability solving. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[31] R. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM International Workshop on Logic Synthesis*, 1995.

[32] The Annual SAT Competitions: http://www.satlive.org/SATCompetition/.

[33] F. Somenzi. CUDD: CU Decision Diagram Package. Release 2.4.0.

[34] D. Thilikos, M. Serna, and H. Bodlaender. A polynomial algorithm for the cutwidth of bounded degree graphs with small treewidth. *Lecture Notes in Computer Science*, 2161:380–390, 2001.

[35] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDDs. In *Proceedings of the IEEE International Conferece on Computer Aided Design*, pages 130–133, 1990.

[36] T. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In *Proocedings of the First International Conference on Constraints in Computational Logics*, pages 34–49. Springer Verlag, Lecture Notes in Computer Science, Volume 845, 1994.

[37] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[38] B. Yang. *Optimizing Model Checking Based on BDD Characterization*. PhD thesis, Carnegie Mellon University, Computer Science Department, 1999.

[39] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 295–313. Springer Verlag, Lecture Notes in Computer Science, 2002.

[40] N. L. Zhang and D. Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.