# Using DPLL for Efficient OBDD Construction

Jinbo Huang and Adnan Darwiche

Computer Science Department
University of California, Los Angeles
{*jinbo,darwiche*}*@cs.ucla.edu*

**Abstract.** The DPLL procedure has found great success in SAT, where search terminates on the first solution discovered. We show that this procedure is equally promising in a problem where exhaustive search is used, given that it is augmented with appropriate caching. Specifically, we propose two DPLL-based algorithms that construct OBDDs for CNF formulas. These algorithms have a worst-case complexity that is linear in the number of variables and size of the CNF, and exponential only in the *cutwidth* or *pathwidth* of the variable ordering. We show how modern SAT techniques can be harnessed by implementing the algorithms on top of an existing SAT solver. We discuss the advantage of this new construction method over the traditional approach, where OBDDs for subsets of the CNF formula are built and conjoined. Our experiments indicate that on many CNF benchmarks, the new method runs orders of magnitude faster than a comparable implementation of the traditional method.

## 1 Introduction

The DPLL procedure [1] has found great success in the Propositional Satisfiability problem (SAT), attested by a series of SAT solvers that have excelled in the annual SAT competitions [2]. These solvers are, by nature, geared toward finding the first solution quickly, and not particularly concerned with any search space beyond that (solvers for *Quantified Boolean Formulas* are an exception). There has been evidence, however, that DPLL can also be useful in problems requiring exhaustive search, such as model counting [3, 4]. In this paper we explore another such problem, and show that DPLL, coupled with appropriate caching, can be the basis for an efficient program that compiles propositional theories into Ordered Binary Decision Diagrams (OBDDs) [5]. Once theories are expressed as OBDDs, many important queries can be answered in constant or polynomial time, including satisfiability, equivalence, model counting, model enumeration, and clausal entailment [5, 6].

Compiling propositional theories into OBDD has remained a nontrivial task. Traditionally, one enlists software packages which build OBDDs in a bottom-up fashion. For theories in Conjunctive Normal Form (CNF), this means that OBDDs are constructed for individual clauses, and conjoined to produce the OBDD for the whole theory. Although the complexity of OBDD conjunction is

only quadratic in the sizes of the conjuncts [5], this operation has to be repeatedly carried out until the final OBDD is produced. Moreover, experience has shown that the conjuncts involved in the operation—intermediate OBDDs—are often much larger than the OBDD to be finally built, leading to an accumulation of intermediate OBDD nodes that unduly exacerbates the time and space complexities of the construction.

Consider for example uf100-08.cnf, one of the standard benchmarks from the Satisfiability Library [7]. This CNF has 100 variables and its (reduced) OBDD has 176 nodes under the MINCE variable ordering [8]. Yet, to build this OBDD using the popular CUDD package and same variable ordering, a total of $30,640,582$ intermediate nodes are generated, taking 25 minutes on our 2.4GHz processor.

In this paper we propose two DPLL-based algorithms that, unlike the traditional method, build OBDDs for CNFs in a top-down fashion. By using a novel caching scheme, these algorithms have a complexity that is linear in the number of variables and size of the CNF, and exponential only in the *cutwidth* and *pathwidth*, respectively, of the variable ordering with respect to a hypergraph abstraction of the CNF. As a bonus of this theoretical analysis, we provide an upper bound on the OBDD size for arbitrary CNFs. We relate these complexity results to those of some previous work that use the notions of cutwidth and pathwidth. Our upper bound on OBDD size also offers a formal explanation for the effectiveness of a recent class of variable ordering heuristics, which has hiterto been explained only intuitively.

We show next how these algorithms can be implemented on top of a SAT engine, thus harnessing the power of modern techniques, including carefully implemented *Unit Propagation* and *Nonchronological Backtracking*, that underly the success of many SAT solvers. Using multiple sets of experiments, we demonstrate the efficiency of this program and discuss a few related issues.

The rest of the paper is organized as follows. In Section 2 we describe our proposed algorithms for compilation of CNFs into OBDD, followed by a theoretical analysis of their complexities in Section 3. Section 4 is a description of our implementation of these algorithms on top of an existing SAT engine. Section 5 contains experimental results that demonstrate the efficiency of this new program and support the discussion of a few related issues. Section 6 concludes the paper.

## 2 Algorithms

We present in this section two DPLL-based algorithms for compiling CNF formulas into OBDD. Fig. 1 depicts a CNF $\Delta$ and its OBDD under variable order $x, y, z$. Recall that an OBDD is a *Directed Acyclic Graph (DAG)* where there are at most two sinks, labeled with 0 and 1 respectively, and every internal node is labeled with a variable and has exactly two children *low* and *high*; it is further required that variables appear in the same order on all paths from the root to a sink. The semantics of this graph is as follows. Given an instantiation

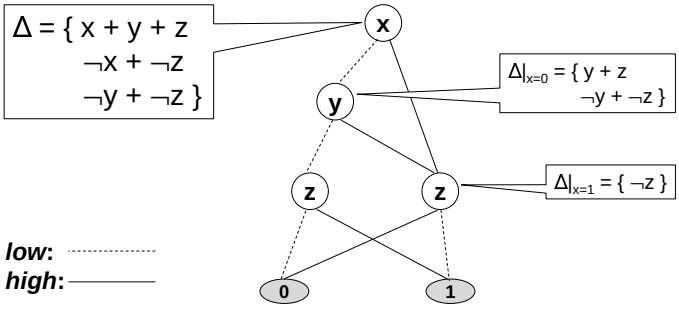



**Fig. 1.** A CNF and its OBDD.

$I$ of the variables, one picks a path from the root to a sink while always choosing the low (high) child of a node if the variable associated with that node is set to 0 (1) by $I$. If the path ends with the 0-sink (1-sink), the theory evaluates to 0 (1) for this variable instantiation.

In this work we consider *reduced* OBDDs, where there is no node whose two children are identical, and no isomorphic sub-graphs exist. It is known that there is a unique reduced OBDD for any propositional formula under a given variable order [5]. As in SAT, the variable order plays an important role in complexity. In the rest of the paper we assume that a variable order $v_1, \ldots, v_n$ has been identified in a preprocessing step to be used for the OBDD. As we point out later, efficient tools exist that generate good variable orders.

Algorithm 1 describes a naive DPLL-style procedure that converts a CNF $\Delta$ into an OBDD by recursively converting its two restrictions, $\Delta|_{v_i=0}$ and $\Delta|_{v_i=1}$, and combining the results using *get_node* (Line 5). This is also illustrated in Fig. 1, where $\Delta|_{x=0}$ and $\Delta|_{x=1}$ are obtained by setting $x$ to 0 and 1, respectively, in CNF $\Delta$. Note that a common technique known as *unique nodes* is used so that the final result will be a DAG—a reduced OBDD, not a tree. Specifically, *get_node* will not construct a new node in these two cases: 1) if its last two arguments are identical, either one of them is returned immediately; 2) if there already exists a node that is labeled with the first argument and has the last two arguments as children (in the right order), that node is returned.

Note that this algorithm can have an exponential complexity even when the final OBDD has a tractable size. The reason is that when different settings of a subset of the variables lead to sub-theories that are logically equivalent, they

**Algorithm 1** *obdd*(CNF $\Delta$, int $i$): should be initially called with $i = 1$.

1: **if** there is an inconsistent clause in $\Delta$ **then**
2: return 0-sink
3: **if** there is no uninstantiated variable in $\Delta$ **then**
4: return 1-sink
5: return $get\_node(i, obdd(\Delta|_{v_i=0}, i+1), obdd(\Delta|_{v_i=1}, i+1))$

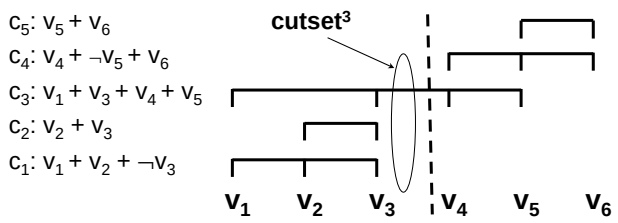



**Fig. 2.** Cutset-based caching.

will be represented by the same OBDD node, while Algorithm 1 will convert each of these sub-theories into OBDDs only to realize that they are all the same.

Consider, for example, variable order $\pi = v_1, v_2, v_3, v_4, v_5, v_6$ for CNF $\Delta = \{c_1, c_2, c_3, c_4, c_5\}$ shown in Fig. 2. When Algorithm 1 is run on this CNF, it will spawn two recursive calls on $i = 2$ (Line 5), because there are two instantiations for variable $v_1$. Note that the number of recursive calls on $i = 3$ will be three, not four, because one of the four instantiations for variables $v_1, v_2$ results in an empty clause, terminating the recursion (Lines 1 and 2). By the same token, five recursive calls on $i = 4$ will be generated.

We will now show that three of these five recursive calls on $i = 4$ are in fact redundant and could have been avoided by caching. Specifically, let $S$ be the set of nontrivial CNFs $\Delta'$ that can be obtained by instantiating the first three variables $v_1, v_2, v_3$ in $\Delta$, we will show that $|S| \leq 2$.

Note that with respect to instantiation of variables $v_1, v_2, v_3$, one can think of the CNF as partitioned into three sets of clauses: clauses over $v_1, v_2, v_3$ only, clauses over $v_4, v_5, v_6$ only, and the rest. Denote these three sets by $left^3 = \{c_1, c_2\}$, $right^3 = \{c_4, c_5\}$, and $cutset^3 = \{c_3\}$, respectively. After the instantiation of variables $v_1, v_2, v_3$, clauses $left^3$ will evaluate to a Boolean constant because their variables have all been set. If this constant is 0, we know that $\Delta'$ is a trivial CNF equal to 0 and hence not in $S$. Otherwise $\Delta'$ will consist of clauses $right^3$, which have not been altered because none of their variables have been set, and clauses in $cutset^3$ which must have been altered by setting variables $v_1, v_2, v_3$. This cutset, however, contains only one clause $v_1 + v_3 + v_4 + v_5$. After *any* instantiation of variables $v_1, v_2, v_3$, this clause can only be in one of two states: either satisfied, or simplified to $v_4 + v_5$. Hence, although there are eight

---

*get_node*(int $i$, BDD *low*, BDD *high*)

1: **if** $low == high$ **then**
2: return *low*
3: **if** $(lookup = unique[(i, low, high)]) \neq$ nil **then**
4: return *lookup*
5: $result =$ new BDD$(i, low, high)$
6: $unique[(i, low, high)] = result$
7: return *result*

**Algorithm 2** *obdd*(CNF $\Delta$, int $i$): $value(C)$ returns a bit vector representing the states (satisfied or not) of clauses $C$ in some fixed order.

1: **if** there is an inconsistent clause in $\Delta$ **then**
2: return 0-sink
3: **if** there is no uninstantiated variable in $\Delta$ **then**
4: return 1-sink
5: **if** $(lookup = cache_{i-1}[value(cutset^{i-1})]) \neq$ nil **then**
6: return *lookup*
7: $result = get\_node(i, obdd(\Delta|_{v_i=0}, i+1), obdd(\Delta|_{v_i=1}, i+1))$
8: $cache_{i-1}[value(cutset^{i-1})] = result$
9: return *result*

**Algorithm 3** *obdd*(CNF $\Delta$, int $i$): $value(S)$ returns a bit vector representing the values of variables $S$ in some fixed order.

1: **if** there is an inconsistent clause in $\Delta$ **then**
2: return 0-sink
3: **if** there is no uninstantiated variable in $\Delta$ **then**
4: return 1-sink
5: **if** $(lookup = cache_{i-1}[value(separator^{i-1})]) \neq$ nil **then**
6: return *lookup*
7: $result = get\_node(i, obdd(\Delta|_{\overline{v_i}}, i+1), obdd(\Delta|_{v_i}, i+1))$
8: $cache_{i-1}[value(separator^{i-1})] = result$
9: return *result*

different instantiations of $v_1, v_2, v_3$ and five that result in nontrivial CNFs, we have $|S| \leq 2 = 2^{|cutset^3|}$.

In general, the $i^{th}$ cutset of a variable order for a CNF is all clauses mentioning a variable at position $\leq i$ and one at position $> i$:

**Definition 1.** *The $i^{th}$ cutset of variable order $\pi = v_1, \ldots, v_n$ for CNF $\Delta = \{c_1, \ldots, c_m\}$, denoted $cutset^i_\Delta(\pi)$ or $cutset^i$ for short, is defined as $\{c \in \Delta : \exists j \leq i < k$ such that clause $c$ mentions variables $v_j$ and $v_k\}$.*

As we have seen, after instantiating the first $i$ variables, each clause in $cutset^i$ can only be in one of two states. The states of clauses $cutset^i$ can therefore be represented by some bit vector $value(cutset^i)$, whose evaluation provides us with a sound equivalence test: two sub-theories, which result from two instantiations of the first $i$ variables, must be equivalent if $cutset^i$ evaluates to the same value for both variable instantiations.

This equivalence test is used by Algorithm 2 to index a cache that stores OBDDs for all sub-theories. Specifically, when two or more sub-theories are found to have the same cutset value, only one of them will be compiled, its OBDD cached (Line 8), and others will simply generate a cache hit and have their OBDD immediately returned (Line 6). By virtue of this caching the complexity of the algorithm is only exponential in the size of the largest cutset. We discuss this complexity result in more detail in Section 3.

We now turn to Algorithm 3, which replicates Algorithm 2 except it uses a slightly different caching scheme. For position $i$ in the variable order, let the $i^{th}$ separator be the subset of the first $i$ variables that appear in clauses of the $i^{th}$ cutset:

**Definition 2.** *The $i^{th}$ separator of variable order $\pi = v_1, \ldots, v_n$ for CNF $\Delta = \{c_1, \ldots, c_m\}$, denoted $separator^i_\Delta(\pi)$ or $separator^i$ for short, is defined as $\{j \leq i : \exists c \in cutset^i_\Delta(\pi)$ such that clause $c$ mentions variable $v_j\}$.*

Given an instantiation of $v_1, \ldots, v_i$, it is clear that the values of variables $separator^i$ alone determine the states of clauses $cutset^i$, and hence the sub-theory $\Delta'$. One can represent the values of these variables, again, by some bit vector and use it to index the cache. Similarly, the complexity of this algorithm is only exponential in the size of the largest separator.

It can be seen that the value of $cutset^i$ does not determine that of $separator^i$, although the reverse, as we have just pointed out, is true. Separator caching can thus be regarded as an approximation of cutset caching, in that it may redundantly process some sub-theories that would have generated a cache hit with cutset caching. As we discuss later, though, separators may sometimes be preferable in practice as their evaluation can be less costly.

## 3 Complexity Results

The nature of the caching method used by our algorithms allows us to provide formal guarantees on their complexities. Our results are given in three theorems whose proofs can be found in the technical report version of this paper [9]. In stating these theorems we will refer to the size of the largest cutset as the *cutwidth*, and the size of the largest separator as the *pathwidth*, of the variable ordering with respect to the underlying CNF:[1]

**Definition 3.** *The cutwidth of variable order $\pi$ for CNF $\Delta$, denoted $cw_\Delta(\pi)$, is $\max_i |cutset^i_\Delta(\pi)|$.*

**Definition 4.** *The pathwidth of variable order $\pi$ for CNF $\Delta$, denoted $pw_\Delta(\pi)$, is $\max_i |separator^i_\Delta(\pi)|$.*

We now present the following two bounds on the time and space complexities of Algorithms 2 and 3 respectively. These results assume that *get_node* runs

[1] These definitions of cutwidth and pathwidth correspond precisely to those found in graph theory, given that one considers a hypergraph abstraction of the CNF formula, where each variable becomes a vertex and each clause a hyperedge enclosing its variables, and defines the cutwidth (pathwidth) of the hypergraph as the maximum cutwidth (pathwidth) among all vertex orderings. Specifically, when restricted to graphs, this notion of cutwidth is equivalent to that identified and studied in [10, 11]; this notion of pathwidth is equivalent, as proven in [12], to that originally introduced by Robertson and Seymour [13] based on the notion of *path decompositions*.

in constant time, but hold even when *unique nodes* is not used and *get_node* constructs a new node each time it is called.

**Theorem 1.** *For CNF $\Delta$ and variable order $\pi$, Algorithm 2 takes $O(sn2^w)$ time and space, where $s$ is the size of $\Delta$, $n$ is the number of variables, and $w = cw_\Delta(\pi)$.*

**Theorem 2.** *For CNF $\Delta$ and variable order $\pi$, Algorithm 3 takes $O(sn2^w)$ time and space, where $s$ is the size of $\Delta$, $n$ is the number of variables, and $w = pw_\Delta(\pi)$.*

As we pointed out earlier, for any given position $i$ in the variable ordering $\pi$, the value of $separator^i$ determines that of $cutset^i$. In other words, the number of possible values for $cutset^i$ can never be larger than that for $separator^i$. Therefore, the result of Theorem 1 can in fact be strengthened by defining $w$ to be $\max_i \min(|cutset^i|, |separator^i|)$. Note that this quantity is guaranteed to be a lower bound on both cutwidth and pathwidth. We will now use it in the following theorem that bounds the OBDD size for arbitrary CNF formulas, where we write $OBDD^\pi_\Delta$ to denote the OBDD for CNF $\Delta$ under variable ordering $\pi$.

**Theorem 3.** *For CNF $\Delta$ and variable order $\pi$, $size(OBDD^\pi_\Delta) \leq n2^w + 2$, where $n$ is the number of variables and $w = \max_i \min(|cutset^i|, |separator^i|)$.*

We will now relate these complexity bounds to two previous results that involve similar parameters. The first of these concerns monotone 2-CNFs, which are CNFs where all clauses have length two and contain only positive literals. It has been proved in [14] that the size of any OBDD for a monotone 2-CNF is bounded by $n(2^w + 1)$, where $n$ is the number of variables and $w$ is the pathwidth of the *reverse* of its variable ordering.[2] This bound may look similar to that of Theorem 3, but is in fact a different result, because a variable ordering and its reverse may have quite different pathwidths. Also, the proof [14] of this result hinges on properties specific to monotone 2-CNFs and does not seem to generalize to arbitrary CNFs.

The second related result involves a SAT algorithm presented in [15], based on a static variable ordering $\pi$ for a CNF. The time complexity of this algorithm is claimed to be $O(m2^w)$ where $m$ is the number of clauses and $w$ is the cutwidth of $\pi$. This bound is comparable to our complexity bound for Algorithm 2. However, OBDD construction is much more difficult than, and in fact subsumes, SAT solving for any given CNF. We are hence offering an algorithm that constructs an OBDD for a CNF with roughly the same time complexity as the algorithm of [15] that only solves SAT for the same CNF.

Finally, we point out that Theorem 3 offers a formal explanation for the effectiveness of a class of variable ordering techniques based on *Min-Cut Linear Arrangement* that have been recently proposed [8, 16, 15]. The MINCE variable ordering [8], for example, has been shown to result in relatively small OBDDs for

[2] The word *reverse* is not used in [14] for this result, but their definition of the pathwidth of $\pi$ corresponds to the pathwidth, in our definition, of the reverse of $\pi$.

various benchmarks. According to its authors, MINCE minimizes the "average" cutset size of the ordering. The observed effectiveness of this technique, however, was only explained intuitively by its tendency toward grouping "connected variables together." According to Theorem 3, variable orderings that minimize cutset sizes are directly optimizing the upper bounds on OBDD size—a fundamental explanation for their effectiveness in practice.

## 4 Implementation

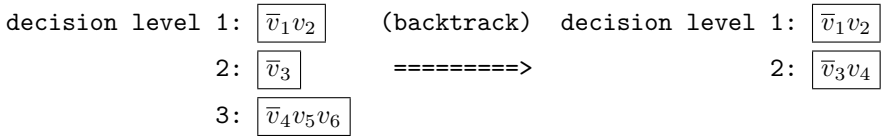
It is possible to implement Algorithms 2 and 3 in their original recursive form. One should then consider adding their own implementation of some efficient mechanism for unit propagation, nonchronological backtracking, and other important components of DPLL search. Since the zChaff SAT solver from Princeton University [17] is known to boast a highly optimized DPLL engine in these respects [2], we have decided to implement our algorithms on top of it instead. Like most modern SAT solvers, however, zChaff is based on an iterative version of DPLL, and thus not immediately adaptable for Algorithms 2 and 3. The following is pseudocode for the DPLL engine of zChaff, reproduced from [17].

```
while(1)
  if(decide_next_branch())                // branching
    while(deduce() == conflict)           // deducing
      blevel = analyze_conflicts();       // learning
      if(blevel == 0) return UNSATISFIABLE;
      else back_track(blevel);            // backtracking
  else return SATISFIABLE;                // all variables have been set
```

Implementation of (an iterative equivalent of) Algorithms 2 and 3 on top of such a SAT solver can generally be achieved in four steps, all of which are done in our case by modifying only the *decide_next_branch* function. First, make sure the program uses the variable order intended for the OBDD. Second, instruct the program to find all solutions instead of one. This can be done by adding a fake conflict clause, also known as a *blocking clause*, whenever a solution is found so that the solver will backtrack and continue to search. Third, maintain a trace of the search in the form of an OBDD (generally incomplete and nonreduced until search terminates; see Fig. 3). That is, keep an OBDD on the side and augment it during search so that it has a root-to-sink path corresponding to each solution found (see Fig. 3); all other paths should end with the zero sink. When search finishes the standard reduction algorithm [5] can be applied to obtain a reduced OBDD, which works by iteratively merging nodes that share the same label and children, and deleting nodes whose two children are identical.

Now that the program constructs OBDDs instead of just finding a solution, the fourth and key step is to put caching in place, which consists of cache insertion and cache lookup. According to Algorithms 2 and 3, every node cached represents the result of compiling some sub-theory of the original CNF into OBDD. Back to our implementation, this implies that we should only cache nodes whose construction is complete, as there are also nodes that are partially

constructed. Consider, for example, the left half of the figure below, which depicts the decision stack of the program when the first solution $\overline{v}_1 v_2 \overline{v}_3 \overline{v}_4 v_5 v_6$ has just been found for the CNF from Fig. 2:




At this point six OBDD nodes (excluding the sinks) are constructed, as shown in the first picture of Fig. 3, to form a path representing the solution. Among these, however, only the last two nodes (labeled with $v_5$ and $v_6$) are complete: their other child, although not yet drawn, must be the zero sink, because instantiations $v_5 = 1$ and $v_6 = 1$ have been implied. The other four nodes all have a child that has not been determined or has not been completely constructed. The nodes labeled with $v_5$ and $v_6$ should therefore be the only nodes to insert into the cache.

In general, whenever a solution is found by the SAT solver and the OBDD is augmented so that it contains a path corresponding to the solution, we may store in the cache all nodes on this path that come after the node labeled with the current decision variable, indexed by their corresponding separator (or cutset) value.

We now continue the example to illustrate the operation of cache lookup. After a blocking clause $v_1 + v_3 + v_4$ is added, the program will backtrack to decision level 2 and insert $v_4 = 1$ as an implication, as shown in the right half of the figure above. Before making the next decision by instantiating $v_5$, the program now has an opportunity to check the cache, both at position 3 (corresponding to partial assignment $\overline{v}_1 v_2 \overline{v}_3$) and position 4 (corresponding to partial assignment $\overline{v}_1 v_2 \overline{v}_3 v_4$). Note that cache lookup at preceding positions have been performed at earlier decision levels, and thus need not be repeated. As it turns out, no cache hits occur at this point.

In general, whenever the SAT solver is about to instantiate variable $v_k$, it may check the cache at every position $i$, where $j \leq i < k$ and $v_j$ is the previous decision variable. The key used in the lookup will then be the value of $separator^i$ (or $cutset^i$). In case of a cache miss the program proceeds as usual by instantiating $v_k$; otherwise the OBDD is augmented so that a partial path corresponding to the current instantiation of variables $v_1, \ldots, v_{k-1}$ exists and is connected directly to the OBDD node returned from the cache (see Cache Hits in Fig. 3); again a blocking clause is added so that the solver will backtrack and continue to search.

To conclude our example with the CNF from Fig. 2, Fig. 3 shows snapshots of the OBDD maintained by the compiler at successive solution findings. Specifically, the first five shots are taken when the program has just found the first, second, third, fourth, and fifth solution, respectively. The next two pictures depict the rest of the solutions found, all through cache hits. The final picture

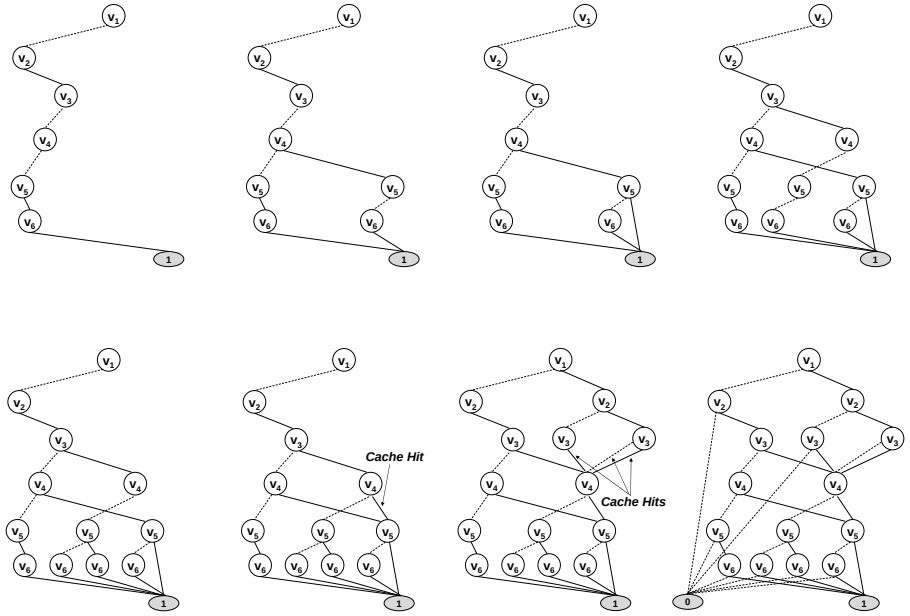



**Fig. 3.** Partially constructed OBDDs at various stages of DPLL, before reduction.

completes the OBDD by supplying the pointers to the zero sink, which have been implicit.

The last component of the compiler is a method to properly compute the values of separators and cutsets. The former is straightforward: the value of $separator^i$ is simply the current instantiation of variables $separator^i$. The latter demands more care. Recall that the correctness of cutset caching hinges on the fact that other variables remain free when variables $v_1, \ldots, v_i$ are instantiated. This does not hold, however, in a real-world SAT solver where unit propagation constantly takes place: an instantiation of variables $v_1, \ldots, v_i$ may well have caused variables at position $> i$ to be set, which in turn alters the states of clauses $cutset^i$, obscuring their true values. To overcome this complication, the states of clauses $cutset^i$ should be determined purely on the instantiation of variables $v_1, \ldots, v_i$, pretending that other variables were all free. This process usually incurs overhead, because one can no longer rely on a quick check of some flag that may have been set by the SAT solver to indicate whether a clause has been satisfied. In our implementation, we simply walk through the literals of each clause in the cutset to determine its state.

Finally, we note that except for dynamic variable ordering, which we have turned off, all features of the original SAT solver remain in effect. In particular, we retain the benefits of unit propagation using watched literals, conflict-directed backtracking, and no-good learning.

**Table 1.** Performance of DPLL vs CUDD

| Benchmark | #CNFs | OBDD Size | DPLL Time (sec) | CUDD Time (sec) | CUDD Nodes |
|---|---|---|---|---|---|
| aim50 | 16 | 52 | 0.00 | 0.19 | 11178 |
| aim100 | 16 | 102 | 0.00 | 21.86 | 2413645 |
| ais | 2 | 1770 | 0.45 | 15.51 | 613200 |
| blocksworld | 5 | 559 | 0.04 | 234.28 | 1759884 |
| flat75 | 10 | 8610 | 0.29 | 1.37 | 99645 |
| flat100 | 10 | 18515 | 1.61 | 15.41 | 639159 |
| parity8 | 10 | 212 | 0.00 | 0.21 | 22280 |
| parity16 | 8 | 674 | 4.20 | 800.29 | 38148066 |
| uf75 | 10 | 1733 | 0.11 | 15.36 | 605228 |
| uf100 | 10 | 1411 | 1.33 | 526.88 | 14154496 |
| iscas89 | 18 | 85462 | 13.22 | 4.66 | 342313 |

**Table 2.** Effect of Caching on Performance of DPLL

| CNF | OBDD Size | #Cache Hits / Entries | Time (sec) | Time without Caching (sec) |
|---|---|---|---|---|
| flat75-1 | 3966 | 387 / 17155 | 0.16 | 1.1 |
| flat75-2 | 2231 | 1281 / 28180 | 0.28 | 320.83 |
| flat75-3 | 14057 | 723 / 24208 | 0.29 | 1.5 |
| flat100-1 | 10385 | 642 / 27232 | 0.78 | 166.13 |
| flat100-2 | 14806 | 1902 / 71336 | 1.57 | 38.83 |
| flat100-3 | 2583 | 464 / 17006 | 0.15 | out of memory |
| iscas89-s208.1 | 1056 | 190 / 2863 | 0.01 | 0.87 |
| iscas89-s344 | 10073 | 1154 / 20225 | 0.07 | out of memory |
| iscas89-s386 | 14078 | 1399 / 180620 | 0.65 | 1.09 |
| iscas89-s510 | 17366 | 991 / 21893 | 0.14 | 64.94 |
| iscas89-s953 | 438246 | 56394 / 2935247 | 38.81 | out of memory |

## 5 Experimental Results

The purpose of our experiments is threefold. First, we demonstrate the efficiency of our program by running it against an implementation of the traditional bottom-up OBDD construction method. Second, we study the effect of caching used by our program by turning it off and observing the change in performance. Third, we investigate the intermediate explosion encountered in bottom-up construction using random CNFs with varying clauses-to-variables ratios. All our compilations use variable orders generated by *MINCE* [18], which implements the heuristic proposed in [8] for minimizing OBDD sizes. Our experiments were run on a 2.4GHz processor with 3.7GB of RAM.

Our first set of experiments are on ten groups of benchmarks taken from the Satisfiability Library [7] plus CNFs based on the first 18 of the ISCAS89 circuits [19]. Our DPLL-based compiler can be set to use either cutset (Algorithm 2) or separator (Algorithm 3) caching. For these experiments the latter has been used,

as it runs slightly faster thanks to the less expensive computation of separator values. For the bottom-up method, we rely on the CUDD package from the University of Colorado [20] to build OBDDs for individual clauses and conjoin them for the final result. Since the order in which these OBDDs are built and conjoined affects the complexity of construction, we have adopted a clause ordering heuristic that was proposed in [21] exactly for use with this method of OBDD construction. This heuristic calls for clauses with higher-indexed variables to be processed first, allowing the OBDD nodes themselves to be constructed in a bottom-up fashion.

The results of these experiments are summarized in Table 1, where the two programs are referred to as DPLL and CUDD, respectively. The second column indicates the number of instances in each group of CNFs. All other figures represent group averages. The time to generate the MINCE variable order is not included as this is a preprocessing step shared by both programs. We observe that DPLL runs faster than CUDD by generally many orders of magnitude, except for the last group which we discuss more toward the end of the section. The last column gives for each group the (average) number of intermediate OBDD nodes generated by CUDD, which, compared with the OBDD size, affords an intuitive explanation for the inefficiency of the bottom-up construction method on these instances. Some instances are not included in this table because CUDD did not successfully compile them. On two of the parity16 instances, for example, DPLL finished in 6.67 and 9.53 seconds, generating 351 and 1017 OBDD nodes, respectively, but CUDD ran out of memory.

To ascertain the effect of caching on the performance of DPLL, we reran it on the same instances with caching turned off. This version of DPLL would then correspond to the original zChaff with the only change being an enforced static variable ordering and the adding of blocking clauses for enumeration of all solutions. We note that on some instances, no cache hits had occurred before and, consequently, disabling caching did not cause any noticeable change in performance. However, on other instances, especially the flat75, flat100, and iscas89 families, performance dropped significantly after caching was turned off. See the results in Table 2. Note that CUDD does better than DPLL on the iscas89 family overall, but not on all the 18 instances. In particular, all of those included in Table 2 are instances where DPLL outperforms CUDD.

We investigate next the performance of DPLL and CUDD on randomly generated 3-CNFs with varying clauses-to-variables ratios. We use *mkcnf* written by Allen van Gelder with the *forced satisfiable* option to generate the CNFs. Our suite of random 3-CNFs consists of those with $n$ variables and $m$ clauses, where $n = 40$, 45, and 50, and $m$ ranges from 10 to $5n$ at intervals of 5. For each $n$-$m$ combination we generate 20 instances. The OBDD sizes as well as running times we report next represent averages over these 20 instances of each ratio. Our first observation is illustrated in Fig. 4, which plots the OBDD size as a function of the clauses-to-variables ratio. It can be seen that for all three groups, the OBDD size peaks around the ratio of 2, and generally decreases toward either direction.

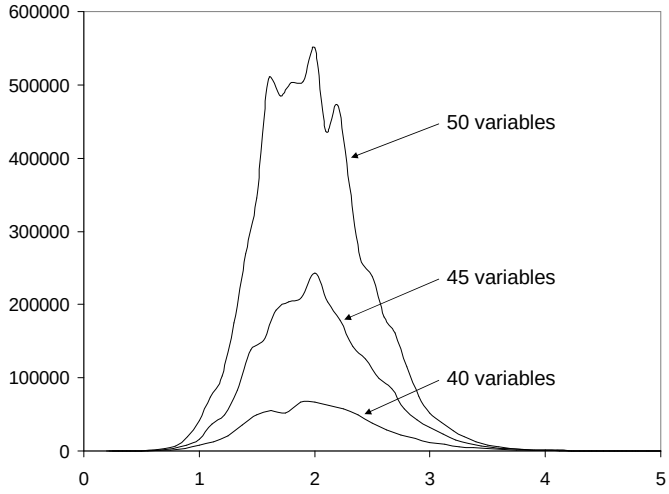



**Fig. 4.** OBDD size as a function of clauses-to-variables ratio.

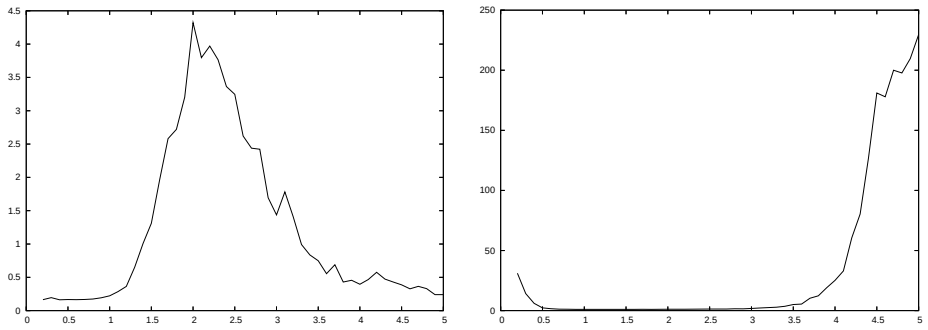


**Fig. 5.** CUDD time (left) and explosion rate (right) as a function of clauses-to-variables ratio, 50 variables.

It is interesting to note, as shown in Fig. 5 (left) for the group of CNFs with 50 variables, that the running time of CUDD also peaks around the ratio of 2, and generally decreases toward either direction.

We now turn to an important issue with the bottom-up construction method used by CUDD—the explosion of intermediate BDD nodes. Fig. 5 (right) shows, for 50 variables, the ratio of the total number of nodes generated by CUDD over the final OBDD size, again as a function of the clauses-to-variables ratio. We observe that over the middle part of the spectrum, between the ratios of 0.6 and 3.6 for example, CUDD produces a low explosion rate. As a result, one may expect CUDD to be generally efficient on CNFs with these ratios, because relatively few dead nodes will be generated.

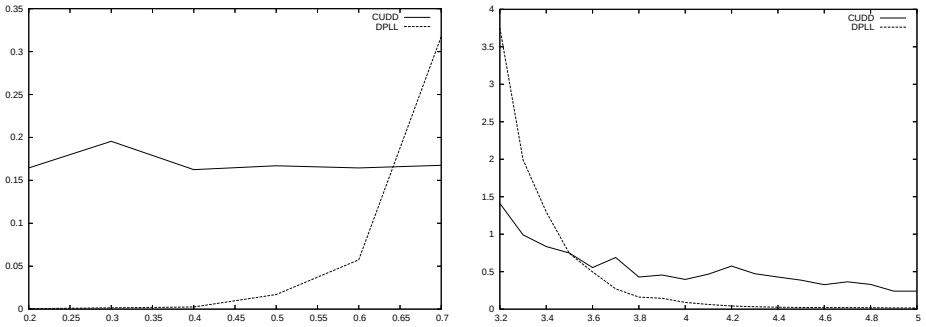



**Fig. 6.** Running time of CUDD vs DPLL on the two extremes of the spectrum.

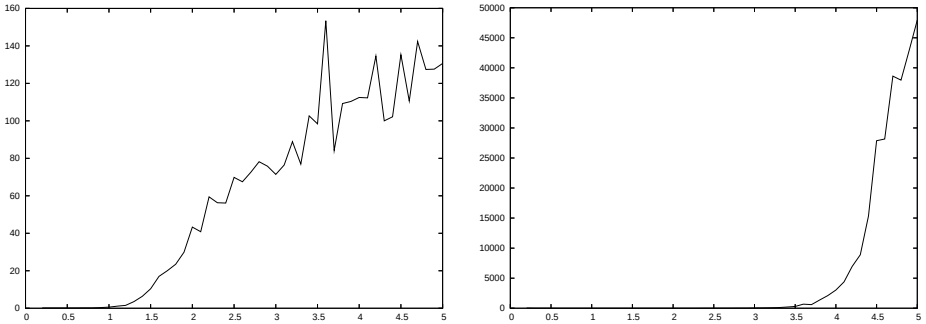


**Fig. 7.** CUDD time (left) and explosion rate (right) without clause ordering.

In fact, our next set of data indicates that it is precisely over this central portion of the gamut that CUDD outperforms DPLL.[3] To view the transition points to a higher precision, we magnify the two end portions of the spectrum and leave out the middle part, as shown in Fig. 6. It can be seen that for ratios $< 0.6$ and those $> 3.6$, DPLL is more efficient than CUDD. This corresponds roughly to what one may have predicted from Fig. 5 (right) based on the rationale that CUDD will tend to be efficient when few dead nodes are generated, which we have alluded to in the previous paragraph. The iscas89 family may be another example to this effect. According to Table 1, the total number of nodes generated by CUDD on this group of CNFs is only about four times the final OBDD size, and CUDD outruns DPLL by about a factor of three.

For these two extremes of the spectrum, we observed the effect of caching on DPLL by turning it off and noting the change in performance. We noticed that for the low ratios the running time increased dramatically (e.g., from 0.06

[3] For these experiments we have used a different implementation of DPLL that does not build on zChaff. Instead, it is written recursively and hence follows more closely the pseudocode of Algorithm 2. For reasons we are yet to identify, this recursive implementation runs faster than the one based on zChaff on this set of random 3-CNFs.

to 1252 seconds for ratio 0.6), and for the high ratios it slighly decreased (e.g., from 0.50 to 0.34 for ratio 3.6). We abscribe this phenomenon to the fact that at the low ratios there are an extremely large number of models for the CNF and hence many opportunities for cache hits, whereas at the high ratios models are sparse and one does not expect many cache hits, if at all, and the overhead of caching can slow the program down.

Finally, we offer a few more words on the two OBDD construction methods that we have been comparing. DPLL represents a top-down approach, where global properties of the CNF formula are exploited throughout the construction. The traditional bottom-up method using CUDD, on the other hand, works locally on subsets of the CNF at any given time. However, the particular implementation we have reported on does not correspond to the pure bottom-up approach, because the clause ordering heuristic we have used effectively gives it also a global view of the CNF structure, and hence some benefits of the top-down approach. In fact, in an additional set of experiments on the random 3-CNFs we turned off clause ordering and noted that the performance of the bottom-up method was now much worse. Fig. 7 (left) plots the running time of CUDD without clause ordering on the 50-variable 3-CNFs, which, instead of having a bell shape, now increases with the number of clauses. Note that the maximum value of the curve has increased from less than 4 seconds (Fig. 5) to over 150 seconds. The explosion rates have also increased significantly; see Fig. 7 (right) compared with Fig. 5 (right).

## 6 Conclusion

We have proposed two DPLL-based algorithms that compile CNF formulas into OBDDs. Theoretical guarantees have been provided on the complexities of these algorithms, and in the process an upper bound has been proved on the OBDD size for arbitrary CNF formulas. We have related these results to some previous complexity bounds that use similar structural parameters. We have described an implementation of these algorithms on top of an existing SAT engine, and demonstrated its efficiency in practice over the traditional bottom-up OBDD construction method on many standard benchmarks. Using randomly generated 3-CNFs, we study the relationships between the OBDD size, CUDD explosion rate, the performance of CUDD versus DPLL, and the effect of caching for varying clauses-to-variables ratios.

## Acknowledgment

We wish to thank the anonymous reviewers for the SAT 2004 conference for commenting on an earlier version of this paper. This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

## References


1. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Journal of the ACM **(5)7** (1962) 394–397
2. SAT Competitions: http://www.satlive.org/SATCompetition/.
3. Birnbaum, E., Lozinskii, E.: The good old Davis-Putnam procedure helps counting models. Journal of Artificial Intelligence Research **10** (1999) 457–477
4. Bayardo, R., Pehoushek, J.: Counting models using connected components. In: AAAI. (2000) 157–162
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
6. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research **17** (2002) 229–264
7. Hoos, H.H., Sttzle, T.: SATLIB: An Online Resource for Research on SAT. In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, IOS Press (2000) 283–292 SATLIB is available online at *www.satlib.org*.
8. Aloul, F., Markov, I., Sakallah, K.: Faster SAT and smaller BDDs via common function structure. In: International Conference on Computer Aided Design (ICCAD), University of Michigan. (2001)
9. Huang, J., Darwiche, A.: Using DPLL for Efficient OBDD Construction. Technical Report D-140, Computer Science Department, UCLA, Los Angeles, CA 90095 (2004)
10. Gavril, F.: Some NP-complete problems on graphs. In: 11th conference on information sciences and systems. (1977) 91–95
11. Thilikos, D., Serna, M., Bodlaender, H.: A Polynomial Algorithm for the cutwidth of bounded degree graphs with small treewidth. Lecture Notes in Computer Science (2001)
12. Kinnersley, N.G.: The vertex separation number of a graph equals its path-width. Information Processing Letters **42** (1992) 345–350
13. Robertson, N., Seymour, P.D.: Graph minors I: Excluding a forest. Journal of Combinatorial Theory, Series B **35** (1983) 39–61
14. Langberg, M., Pnueli, A., Rodeh, Y.: The ROBDD size of simple CNF formulas. In: 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods. (2003)
15. Wang, D., Clarke, E.M., Zhu, Y., Kukula, J.: Using cutwidth to improve symbolic simulation and Boolean satisfiability. In: IEEE International High Level Design Validation and Test Workshop. (2001)
16. Aloul, F., Markov, I., Sakallah, K.: FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic. In: Great Lakes Symposium on VLSI (GLSVLSI), Washington D.C. (2003) 116–119
17. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of ICCAD 2001, San Jose, CA. (2001)
18. MINCE for download: http://www.eecs.umich.edu/~faloul/Tools/mince/.
19. ISCAS89 Benchmark Circuits, http://www.cbl.ncsu.edu/CBL_Docs/iscas89.html.
20. Somenzi, F.: CUDD: CU Decision Diagram Package. (Release 2.4.0)
21. Aloul, F., Markov, I., Sakallah, K.: Faster SAT and smaller BDDs via common function structure. Technical Report CSE-TR-445-01, University of Michigan (2001)