B-Cubing: New Possibilities for Efficient SAT-Solving

Domagoj Babić, *Student Member, IEEE,* Jesse Bingham, *Student Member, IEEE,* and Alan J. Hu *Member, IEEE*

***Abstract*— SAT (Boolean satisfiability) has become the primary Boolean reasoning engine for many EDA (electronic design automation) applications, so the efficiency of SAT-solving is of great practical importance. B-cubing is our extension and generalization of Goldberg et al.'s supercubing, an approach to pruning in SAT-solving completely different from the standard approach used in leading solvers. We have built a B-cubing-based solver that is competitive with, and often outperforms, leading conventional solvers (e.g., ZChaff II) on a wide range of EDA benchmarks. However, B-cubing is hard to understand, and even the correctness of the algorithm is not obvious. This paper presents the theoretical basis for B-cubing, proves our approach correct, details our implementation and experimental results, and maps out other correct possibilities for further improving SAT-solving.**

***Index Terms*— SAT, Boolean Satisfiability, Search Space Pruning**

I. Background

SAT (Boolean satisfiability) has become the primary Boolean reasoning engine for many EDA (electronic design automation) applications, e.g., SAT-based model checking (bounded [1] and unbounded [2]), FPGA routing [3], ATPG [4], and simulation testcase generation [5]. Such industrial applications typically require complete SAT solvers, meaning that the solver must be capable of proving the problem satisfiable or unsatisfiable. In practice, the SAT solver is usually the capacity limiter of the tool, so the efficiency of SAT solving is of great practical importance.

The DPLL algorithm [6], [7] is the core of most modern, complete SAT solvers. It is essentially a depth-first-search with some search-space pruning techniques. The original paper [6] introduced two simple techniques - the pure literal rule[1] and boolean constraint propagation[2].

Subsequent research on improving the performance of SAT solvers has been mostly focused on improving decision heuristics [8], [9], [10], search-space pruning techniques [11], [12], [13], and efficient implementation [14], [15]. Our focus is on the search-space pruning aspect.

A number of pruning techniques have been proposed so far. Many have proven to be too expensive to be used during the search phase, but can be efficient during preprocessing. The pure literal rule is one example. Other examples include hyper-resolution [16] and equivalence reasoning [17], [18], [19].

All SAT-solvers, in one way or another, detect what are known as *conflicts*. A conflict occurs when it is deduced that the current assignment cannot be extended into a satisfying one. When a conflict is detected, the solver finds the reason for the conflict and tries to resolve it. The simplest method is to backtrack to the last case-split (decision) and try an alternative assignment. A more elaborate method, named *conflict-directed backtracking* (CDB) [11], is to analyze the conflict and backtrack to the variable that is actually responsible for the conflict. *Learning* is closely coupled with CDB. Different solvers feature different learning strategies. One thing in common is that learned clauses correspond to different cuts in the implication graph – a directed graph describing logical dependencies among assigned literals. For a more extensive introduction into CDB and learning, and an exhaustive list of references, the reader is referred to [20].

Learning exploits only a fraction of information inferable from conflicts. Learned clauses in general can be quite long; it is not unusual for them to

This work was supported by an NSERC Discovery Grant and graduate fellowships from the University of British Columbia.

The authors are with the Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, Canada. (babic,jbingham,ajh)@cs.ubc.ca

[1] Also known as "Affirmative-negative rule".

[2] "Rule for the elimination of one-literal clauses" in the original paper.

contain a couple hundred literals. The average size depends on the specifics of the problem that is being solved and the overall implementation and dynamics of the solver. Once the conflict is found and a new clause is learned there are no guarantees that the clause will actually be useful later. According to our experiments, a small percentage of clauses ends up being used frequently, while the others just increase the memory requirements and slow down the core of the solver.

Recently, Goldberg, Prasad, and Brayton introduced a theory that unifies many pruning techniques [21], [22]. The proposed theoretical framework can be used as a basis for the development of new pruning techniques. In the same paper, the authors proposed supercubing, as an example of the application of the theory. Their solver was a proof-of-concept, and although supercubing reduced the number of decisions, no actual speedup was reported. Our follow-on work [23] pointed out that supercubing is not readily compatible with 1-UIP learning and proposed an alternative backtracking scheme to integrate the two techniques, thereby demonstrating the first supercubing-based solver to be performance-competitive with standard SAT solvers.

Nadel [24] observed that valuable pruning information can be obtained from analyzing the set of decision literals that participated in previous conflicts. His solver, Jerusat, keeps what we call *certificates* (clauses that correspond to decision cuts in implication graph) from previous conflicts and analyzes them when a new decision is needed. As with supercubing, the goal is to extract and exploit pruning information that is valid only locally in the search tree. The advantage of Jerusat's approach is that much more pruning information is retained. However, such an approach requires too much memory, so Jerusat keeps certificates only for a certain number of decision levels. When it backtracks out of that window, the certificates are discarded. This approach has several drawbacks. First, certificates suffer from the same problems as 1-UIP learned clauses as they have relatively low pruning information content. Second, pruning can be much more effective close to the root of the search tree and therefore discarding certificates that are "out of the window" can miss the best pruning opportunities.

In a recent conference paper [25], we proposed a new pruning technique, B-cubing (so named because the method goes "beyond cubing"), that generalizes supercubing. The preliminary implementation produced excellent results on numerous benchmark suites, and we provided an informal correctness argument (as well as running extended regression tests). We did not, however, have a solid formal understanding of the approach, nor a proof of correctness. A subsequent workshop paper [26] presented a theoretical justification for B-cubing.

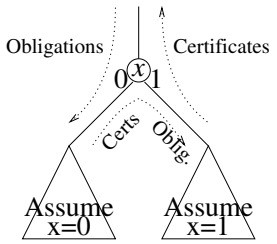


Fig. 1. A SAT-solver works by case-splitting. Information from the left subtree can be used in the right subtree.

This paper is a unified presentation of B-cubing, including theory, implementation, and experimental results. We start with a theoretical foundation for B-cubing, called *obligation-certification trees* (OCT), which provides a very general way to understand different pruning techniques. Using this theory, we can explain B-cubing more clearly and prove the approach correct. We then explain our implementation of an efficient solver based on the B-cubing theory. Experimental results on a wide range of practical benchmark suites show the effectiveness of B-cubing: competitive performance with state-of-the-art solvers, with different strengths and weaknesses, and based on a very different approach. Finally, we note that our implementation is only one of many correct ways to exploit the pruning information derived from B-cubing; exploring other ways to use this information is likely a fertile area for future work.

II. Theoretical Framework

A. Intuitive Overview

A SAT solver works by case-splitting: it assigns a value to a variable, checks to see if the resulting subproblem is satisfiable, and if not, tries the other assignment. This case-splitting naturally corresponds to a search tree that considers all possible assignments to the variables (Fig. 1).

Key to efficient SAT solving is to derive information about the problem that can be used to prune the search tree. For example, conflict-directed learning entails finding a set of literals that guarantees unsatisfiability. This set can be used throughout the search tree, so that the solver never tries to make the exact same assignment again.

Supercubing [21], [22], JeruSAT [24], and B-Cubing [25] all seek to exploit additional pruning information that is valid only in a local part of the search tree. The advantage of keeping some pruning information local to a node is that the solver can perform pruning that is only applicable locally (or, alternatively, that the solver need not store the context information to determine exactly when the pruning information is usable, since the context is implicit in the search tree). Furthermore, the pruning information can be discarded when it is no longer needed.

To formalize this concept, we develop a framework where a node in the search tree inherits from its parent the obligation to prove a part of the search space unsatisfiable. When it is done, it will return to its parent a certificate that a (possibly larger) part of the search space was indeed unsatisfiable. New pruning opportunities arise at the node, because the certificates returned from exploring one branch can be combined with the obligations inherited from above to be used in pruning the other branch.

B. Preliminary Definitions

Let $\mathcal{V}$ be a finite set of boolean variables (also called *positive literals*) and let $\mathcal{V}^\neg$ be the set of *negative literals* $\{\neg v \mid v \in \mathcal{V}\}$. A *literal* is an element of $\mathcal{V} \cup \mathcal{V}^\neg$. We let $\mathsf{bf}(\mathcal{V})$ denote the set of all boolean functions over $\mathcal{V}$.

A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals in which each variable appears at most once. The empty cube (resp. clause) is identified with the boolean constant $\mathbf{1}$ (resp. $\mathbf{0}$). A *minterm* is a cube in which each variable appears exactly once. *Conjunctive Normal Form* (CNF) is the standard way to represent boolean formulae for SAT problems. A formula is in CNF if it is a conjunction of clauses.

Given a boolean formula f over $\mathcal{V}$, a SAT-solving algorithm explores the space of variable assignments to $\mathcal{V}$, and terminates if it finds an assignment that satisfies f, or determines that no such assignment exists. The set of partial assignments (i.e. cubes) checked during this process will form a binary search tree. We formalize this notion as follows.

Definition 1 (search tree): Given variables $\mathcal{V}$, a *search tree* (over $\mathcal{V}$) is a rooted binary tree T with the following properties. Each non-leaf node u of T is labeled with a variable $\mathsf{var}(u) \in \mathcal{V}$, and on any path from the root to a leaf, each $x \in \mathcal{V}$ may appear at most once. Each non-leaf has exactly two children, and the outgoing edges are labeled with 0 and 1.

The 0-child (resp. 1-child) of a node is the node found by following the outgoing edges labeled 0 (resp. 1). The root node of search tree T is denoted $\mathsf{root}(T)$. It follows from Def. 1 that any subtree of a search tree is also a search tree. For a node u in a search tree, the *u-subtree* is the maximal subtree rooted at u. With each node u, we associate a cube $\mathsf{cube}(u)$ defined recursively as follows. If $u = \mathsf{root}(T)$, then $\mathsf{cube}(u) = \mathbf{1}$. Otherwise, let u' be the parent of u. If u is the 0-child of u', then $\mathsf{cube}(u) = \mathsf{cube}(u') \wedge \neg\mathsf{var}(u')$, else $\mathsf{cube}(u) = \mathsf{cube}(u') \wedge \mathsf{var}(u')$. Without loss of generality, we assume that SAT algorithms always visit the 0-child of a node before the 1-child; obviously, an actual implementation could choose to visit the children in either order, with the corresponding modifications to the theory.

C. Obligation-Certification Trees

Let f be an unsatisfiable boolean formula over $\mathcal{V}$ and consider a SAT algorithm $\mathcal{A}$ running against f. $\mathcal{A}$ constructs a search tree T over $\mathcal{V}$, and typically, for each node u of T, the u-subtree is somehow responsible for proving that $\mathsf{cube}(u) \rightarrow \neg f$. However, pruning techniques exploit information acquired previously to visiting the u-subtree to reduce the obligations of the u-subtree. Thus, this subtree is "given" an obligation ϕ and is only required to verify that $(\mathsf{cube}(u) \wedge \phi) \rightarrow \neg f$. In the course of exploring the u-subtree, $\mathcal{A}$ might actually certify that $\psi \rightarrow \neg f$ for some ψ such that $(\mathsf{cube}(u) \wedge \phi) \rightarrow \psi$, i.e. it *at least* confirms that $\mathsf{cube}(u) \wedge \phi$ has no satisfying assignments, but might actually certify the unsatisfiability of a greater space ψ. The fact that ψ is, in general, "too big" can in turn be used to prune in the future. Now, we formally define a search tree in which all nodes are labeled with these obligations and certifications.

Definition 2 (obligation-certification tree): An *obligation-certification tree* (OCT) for boolean formula f over variables $\mathcal{V}$ is a triple (T, ϕ, ψ) where T is a search tree over $\mathcal{V}$, and ϕ and ψ are mappings, respectively called the *obligations* and the *certifications*, that map the nodes of T to $\mathsf{bf}(\mathcal{V})$, such that for all nodes u we have both

$$\mathsf{cube}(u) \rightarrow (\phi(u) \rightarrow \psi(u)) \tag{1a}$$

$$\psi(u) \rightarrow \neg f \tag{1b}$$

Theorem 1: There exists an obligation-certification tree (T, ϕ, ψ) for boolean formula f such that $\phi(\mathsf{root}(T)) = \mathbf{1}$ iff f is unsatisfiable.

Proof: If f is unsatisfiable, it is trivial to construct a suitable OCT, e.g. a single node u with $\phi(u) = \psi(u) = \mathbf{1}$. Conversely, since $\mathsf{cube}(\mathsf{root}(T)) = \mathbf{1}$, (1a) implies that $\psi(\mathsf{root}(T)) = \mathbf{1}$, which along with (1b) imply that f is unsatisfiable. ■

Theorem 1 gives no insight about *how* to construct the obligations and certifications. The value of Theorem 1 is that any algorithm that constructs, explicitly or implicitly, an OCT T with $\phi(\mathsf{root}(T)) = \mathbf{1}$, for all unsatisfiable input formulas, is correct.

Theorem 2, below, formalizes the intuition from Section II-A to suggest a *localized* (in the search tree) approach to constructing obligations and certifications. The theorem states that any search tree satisfying constraints (2a), (2b), and (2c) on ϕ and ψ is indeed an OCT. Intuitively, (2a) states that the obligations of the 0-child must cover those obligations of the parent involving the negative literal, and (2b) requires that the obligations of the 1-child must cover at least the obligations of the parent involving the positive literal that were not covered by the certifications of the 0-child. (2c) simply states that a node certifies exactly the union of the certifications of its two children.

Theorem 2: Let T be a search tree and f a formula over $\mathcal{V}$, and let ϕ and ψ be mappings that take the nodes of T to $\mathsf{bf}(\mathcal{V})$, such that for all nodes u, if u is a leaf we have (1a) and (1b), otherwise, letting u_0 and u_1 respectively be the 0-child and 1-child of u, we have

$$(\phi(u) \wedge \neg \mathsf{var}(u)) \rightarrow \phi(u_0) \tag{2a}$$

$$(\neg\psi(u_0) \wedge \phi(u) \wedge \mathsf{var}(u)) \rightarrow \phi(u_1) \tag{2b}$$

$$\psi(u) \leftrightarrow (\psi(u_0) \vee \psi(u_1)) \tag{2c}$$

Then (T, ϕ, ψ) is an OCT.

Proof: Let u be any node of T, and let T' be the u-subtree of T. We prove by induction on the height of T' that (1a) and (1b) both hold of u. In the base case, T' is a single leaf node u, in which case (1a) and (1b) hold of u by the premise of the theorem statement.

Now let T' be an OCT with root u and 0-child u_0 and 1-child u_1. Letting $x = \mathsf{var}(u)$, by the inductive hypothesis, we have all of:

$$(\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg\phi(u_0) \vee \psi(u_0)) \tag{3a}$$

$$\psi(u_0) \rightarrow \neg f \tag{3b}$$

$$(\mathsf{cube}(u) \wedge x) \rightarrow (\neg\phi(u_1) \vee \psi(u_1)) \tag{3c}$$

$$\psi(u_1) \rightarrow \neg f \tag{3d}$$

(1b) follows easily from (3b), (3d), and (2c). To see that (1a) holds, observe that (3a) implies (by using (2a) and monotonicity)

$$\begin{aligned}
&(\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg(\phi(u) \wedge \neg x) \vee \psi(u_0)) \\
\equiv\ &(\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg\phi(u) \vee x \vee \psi(u_0)) \\
\equiv\ &(\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg\phi(u) \vee \psi(u_0)) \\
\Rightarrow\ &(\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg\phi(u) \vee \psi(u_0) \vee \psi(u_1))
\end{aligned}$$

Similarly, (3c) implies (by using (2b))

$$(\mathsf{cube}(u) \wedge x) \rightarrow (\neg\phi(u) \vee \psi(u_0) \vee \psi(u_1))$$

And thus combining the two cases gives

$$\begin{aligned}
&\mathsf{cube}(u) \rightarrow (\neg\phi(u) \vee \psi(u_0) \vee \psi(u_1)) \\
\equiv\ &\mathsf{cube}(u) \rightarrow (\neg\phi(u) \vee \psi(u)) \qquad \text{by (2c)} \\
\equiv\ &\text{(1a)}
\end{aligned}$$

■

D. B-Cubing

We can now present our B-cubing-based SAT-solving algorithm and prove its correctness.

Intuitively, for a nonleaf node u in the search tree, a constraint $\mathsf{B}(u)$ is constructed while visiting the 0-branch of u, and $\mathsf{B}(u)$ is subsequently used as the obligations while exploring the 1-branch. At a leaf ℓ of the search tree followed by our algorithm, a *certification cube* (CC) $\psi_{\mathsf{cc}}(\ell)$ is constructed such that $\mathsf{cube}(\ell) \rightarrow \psi_{\mathsf{cc}}(\ell)$. For any nonleaf u, let $\mathsf{leaves}_0(u)$ be the set of descendant leaves under the 0-branch of u. We construct $\mathsf{B}(u)$ using some of the CCs found in $\mathsf{leaves}_0(u)$. Note that each CC $\psi_{\mathsf{cc}}(\ell)$ that doesn't contain the literal $\neg\mathsf{var}(u)$ also certifies the corresponding space under the 1-branch of u. Thus, $\mathsf{B}(u)$ involves the CCs in $\mathsf{leaves}_0(u)$ that *do*

contain $\neg\mathsf{var}(u)$, since the disjunction of these over-approximates the subspace under the 1-branch that cannot be proven unsatisfiable using the CCs found in $\psi_{\mathsf{cc}}(\ell)$.

Formally, we have the following:

Definition 3 (B-cube): Let u be a nonleaf node in a search tree that has leaves labelled with certification cubes, let $x_1, \ldots, x_k$ be the list of all variables found in $\mathsf{cube}(u)$, and let $x = \mathsf{var}(u)$. Then the *B-cube* of u is defined[3]

$$\mathsf{B}(u) = \exists x_1, \ldots, x_k, x \left[\bigvee_{\substack{w \in \mathsf{leaves}_0(u) \text{ where} \\ \psi_{\mathsf{cc}}(w) \to \neg x}} \psi_{\mathsf{cc}}(w) \right]$$

$\mathsf{B}(u)$ is the disjunction of all CCs found under the 0-branch of u that contain $\neg x$, with x and also all variables above x in the search tree quantified out. Note that it is possible that $\mathsf{B}(u)$ is the empty disjunction, which is, as usual, defined to be $\mathbf{0}$. Using $\mathsf{B}(u)$ to reduce the obligations under the 1-branch is a sound pruning technique as long as a certain subsumption exists in the inherited obligations.

We abstract optimizations such as learned clauses and boolean constraint propagation, as well as what is inferred about obligations and construction of CCs, through a mechanism we call a *deduction oracle*, denoted $\vdash$. We write $\vdash \theta$, where θ is any formula, to indicate that the solver is somehow able to determine that θ is a tautology (i.e., θ is logically equivalent to $\mathbf{1}$). We require only that $\vdash$ satisfy the following two properties:

Property 1 (Soundness): If $\vdash \theta$, then θ is a tautology.

Property 2 (Minterm evaluation): For any minterm m and any formula f, either $\vdash m \to f$ or $\vdash m \to \neg f$ must hold. In other words, the deduction oracle must at least have the ability to determine if a total assignment (i.e. minterm) satisfies or falsifies f.

Algorithm 1 presents the recursive procedure SAT-SOLVE, which takes a formula f, a cube cube, and an obligation ϕ, and either returns a certification ψ, or exits. The parameter cube is the current partial assignment; the invocation SAT-SOLVE(f, cube, ϕ) is responsible for determining if $f \wedge \mathsf{cube} \wedge \phi$ is satisfiable. We will see that the execution trace of any invocation of SAT-SOLVE (that doesn't exit) is an OCT; the obligation at an invocation is the actual parameter ϕ passed in, and the certification is the return value ψ. It is important to note that our implementation *does not* build these certifications explicitly; they are made explicit in Algorithm 1 to facilitate the proof of correctness, i.e. the proof that the algorithm implicitly builds an OCT (see Theorem 3).

[3]Note that the B-cube is in general not a cube. Supercubing is an instance of our theory in which B(u) is further over-approximated by a single cube.

Algorithm 1 An abstract rendition of our SAT-solver, which utilizes B-cubes to prune the search tree.

```
1:  procedure SAT-SOLVE(f, cube, φ)
2:    if ⊢ cube → f then
3:      exit(SAT)
4:    else if ⊢ ¬φ then
5:      return(0)
6:    else if ⊢ ψ_cc → ¬f for some cube ψ_cc such
        that cube → ψ_cc then
7:      return(ψ_cc)
8:    else
9:      let x be some variable not in cube
10:     ψ_0 := SAT-SOLVE(f, cube ∧ ¬x, φ ∧ ¬x)
11:     if ⊢ [φ]_{x=1} → [φ]_{x=0} then
12:       ψ_1 := SAT-SOLVE(f, cube∧x, φ∧x∧B)
13:     else
14:       ψ_1 := SAT-SOLVE(f, cube ∧ x, φ ∧ x)
15:     end if
16:     return(ψ_0 ∨ ψ_1)
17:   end if
18: end procedure
```

SAT-SOLVE only exits (line 3) if the deduction oracle can determine that cube satisfies f. Otherwise, lines 4–7 correspond to leaf nodes, which return some sort of CC. Lines 9–16 are the recursive case. Note that Algorithm 1 adheres to our simplifying assumption that the 0-branch is always explored first; our implementation is of course more sophisticated and chooses the phase heuristically. Line 9 abstracts any decision heuristic. Because of Property 2 of the deduction oracle, if all variables are present in cube, either line 3 or 7 would have been reached, hence an unassigned variable will always exists at line 9. Line 11 uses the notation $[\phi]_{x=b}$, where $b \in \{0, 1\}$, to denote ϕ restricted to $x = b$. Line 11 tests if $[\phi]_{x=0}$ subsumes $[\phi]_{x=1}$. If the deduction oracle can ascertain that this implication holds, the B-cube of the current invocation may be

used to prune when exploring the 1-branch.[4] On line 12, B denotes $\mathsf{B}(u)$, where u is the current node of the search tree.

We obtain a search tree T from the call tree resulting in an invocation SAT-SOLVE$(f, \mathbf{1}, \mathbf{1})$ by labeling each nonleaf call with the x selected on line 9, and decreeing that the 0-child (resp. 1-child) of a call is the recursive call of line 10 (resp. whichever of line 12 or line 14 is reached).

Theorem 3: If the call SAT-SOLVE$(f,\mathbf{1},\mathbf{1})$ does not result in exit(SAT), then the call tree T represents an OCT (T, ϕ, ψ).

Proof: We show that for each node u of T, the u-subtree, along with the restrictions of ϕ and ψ to the nodes of the subtree, is an OCT for $f \wedge \mathsf{cube}(u)$. To this end, we show by induction on the height of the u-subtree that u satisfies the conditions of Theorem 2. For the base case, u is a leaf and thus corresponds to an invocation in which either lines 5 or 7 is reached. In either case, (1b) and (1a) are both easily seen to hold.

For the inductive step, suppose u is a non-leaf with 0-child u_0 and 1-child u_1, and let $x = \mathsf{var}(u)$. From line 10 we see that $\phi(u_0)$ is precisely $\phi(u) \wedge \neg x$, thus (2a) holds. Also, (2c) obviously holds thanks to line 16.

To prove (2b) requires a case split on whether line 12 or 14 is the invocation of u_1. For the latter case, we note $\phi(u_1) \equiv \phi(u) \wedge x$, hence (2b) trivially holds. Now suppose that line 12 is the invocation of u_1. Then

$$\phi(u_1) \;\equiv\; \phi(u) \wedge x \wedge \mathsf{B}(u) \tag{4}$$

Let C be the set of CCs in the u_0-subtree that contain $\neg x$, and let C' be the set of those that do not contain $\neg x$; in a slight abuse, we will identify these sets with the disjunctions of their constituent cubes. We may hence write

$$\mathsf{B}(u) \;\equiv\; \exists x_1, \ldots, x_k, x : C \tag{5}$$

where $x_1, \ldots, x_k$ is the list of variables in $\mathsf{cube}(u)$. Clearly, because of lines 5, 7, and 16, we have

$$\psi(u_0) \equiv (C \vee C') \tag{6}$$

and therefore

$$(\neg C' \wedge \psi(u_0)) \rightarrow C \tag{7}$$

[4]This subsumption is a technical requirement for correctness of B-cube pruning and is employed in the proof of Theorem 3 .

We also note that for any node w we have $\phi(w) \rightarrow \mathsf{cube}(w)$, since this trivially holds for $w = \mathsf{root}(T)$, and if it holds at a node, then it will hold for any children also, regardless of which of lines 10, 12, or 14 were invoked for the recursive call. Thus we may write $\phi(u_0) \rightarrow \mathsf{cube}(u_0)$. Now, by our inductive hypothesis, the tree rooted at u_0 is an OCT, thus $\mathsf{cube}(u_0) \rightarrow (\phi(u_0) \rightarrow \psi(u_0))$ and hence we have

$$\phi(u_0) \rightarrow \psi(u_0) \tag{8}$$

Now we show $(\neg\psi(u_0) \wedge \phi(u) \wedge x) \rightarrow \mathsf{B}(u)$ through the following logical derivation.

	$\neg\psi(u_0) \wedge \phi(u) \wedge x$	assumption
$\equiv$	$\neg C \wedge \neg C' \wedge \phi(u) \wedge x$	by (6)
$\equiv$	$\neg C' \wedge \phi(u) \wedge x$	since $x \rightarrow \neg C$
$\Rightarrow$	$\neg C' \wedge [\phi(u)]_{x=1}$	logical rule
$\Rightarrow$	$\neg C' \wedge [\phi(u)]_{x=0}$	by line 11 condition
$\equiv$	$\neg C' \wedge [\phi(u_0)]_{x=0}$	since $\phi(u_0) \equiv \phi(u) \wedge \neg x$
$\Rightarrow$	$\neg C' \wedge [\psi(u_0)]_{x=0}$	by (8)
$\equiv$	$[\neg C' \wedge \psi(u_0)]_{x=0}$	C' independent of x
$\Rightarrow$	$[C]_{x=0}$	by (7)
$\Rightarrow$	$\exists x : C$	logical rule
$\Rightarrow$	$\exists x_1, \ldots, x_k, x : C$	logical rule
$\equiv$	$\mathsf{B}(u)$	by (5)

Thus we have $(\neg\psi(u_0) \wedge \phi(u) \wedge x) \rightarrow \mathsf{B}(u)$, which, along with (4), yields (2b). ■

Corollary 1: The call SAT-SOLVE$(f,\mathbf{1},\mathbf{1})$ gives the result exit(SAT) if and only if f is satisfiable.

Proof: If exit(SAT) occurs, then the condition of line 2 held for some cube, and thus cube represents a satisfying subspace for f. Conversely, if exit(SAT) does not occur, then by Theorem 3, the call tree T represents an OCT (T, ϕ, ψ). Since $\phi(\mathsf{root}(T)) = \mathbf{1}$, by Theorem 1 f is unsatisfiable. ■

III. EFFICIENT IMPLEMENTATION

We now turn to the problem of implementing an efficient solver based on our B-cubing theory. The main questions are

- how to represent B-cubes, or approximations thereof, in a memory-efficient manner,
- how to efficiently perform the operations required by Algorithm 1 on the representation, and
- what other optimizations to integrate into the solver.

These questions are addressed in Sections III-A, III-B, and III-C, respectively

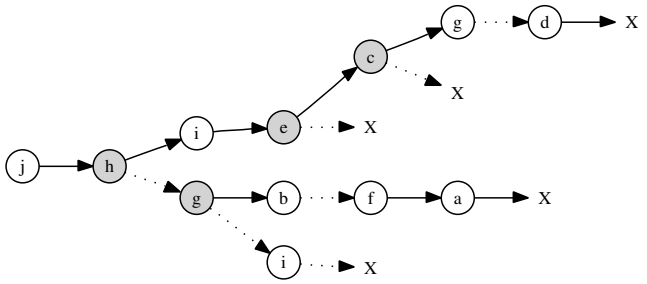


Fig. 2. BCT example

A. Representing and Approximating B-Cubes

In practice, representing B-cubes precisely requires too much time and memory. To solve this problem, we employ a data structure called a *boolean constraint tree* (BCT), based on decision trees [27], to overapproximate B-cubes. The BCTs should be viewed as the saved B-cube information that is used by the deduction oracle to soundly infer $\neg\phi$ (see line 4 of Algorithm 1) and hence prune further recursive calls.

Definition 4 (Boolean Constraint Tree): A BCT is a rooted binary tree T with the following properties:

1) all non-leaves x are labelled with a variable of $\mathcal{V}$,
2) leaf nodes are labelled with the termination symbol X,
3) each non-leaf node has at most two outgoing edges, at most one of which is marked as 0, and at most one of which is marked as 1, and
4) on any path of T from the root to a leaf, each variable appears at most once.

Fig. 2 gives an example of a BCT. 0-edges are represented with dotted lines. We call a node with exactly two children a *branching node*, while a node with exactly one child is called a *literal node*. We identify a literal node with its variable if it has the 1-edge, or with its variable negated if it has the 0-edge. A *stem* of a BCT is a maximal list of literal nodes starting at the root node.

A BCT T defines a Boolean function as follows. For any leaf ℓ of T, we define $\mathsf{cube}(\ell)$ to be same as the cube function on nodes of search tree, i.e. $\mathsf{cube}(\ell)$ is the conjunction of literals found on the unique path from the root to ℓ. Then the Boolean function denoted by a BCT T is defined as:

$$\bigvee_{\ell \text{ is a leaf of } T} \mathsf{cube}(\ell)$$

B. Operations on BCTs

To understand what operations are needed for BCTs, we need to revisit Algorithm 1 and see how it is actually implemented. First, rather than disjoining certifications and passing them up the search tree recursively (line 16), BCTs representing partial B-cubes are created at each decision node in the search tree, and each CC is added (disjoined/unioned) into the relevant BCTs above it in the tree. (Supercubing is implemented similarly [21], [22].) So, we need to implement a routine to union a cube into a BCT, possibly overapproximating it. We also need a way to intersect obligations passed down the search tree, as a BCT, with pruning information derived from the B-cube at a node (i.e. construct a BCT for $\phi \wedge x \wedge \mathsf{B}$ on line 12 of Algorithm 1), so we need to implement a routine to intersect BCTs. The subsumption test on line 11 can also be implemented using our intersection routine.[5] Finally, we will use a BCT to prune the search by assigning all of the literals in the stem of a BCT and passing the rest of the BCT recursively down the search tree. Intuitively, the BCT can be thought of as a blueprint of the search space that needs to be explored. Longer stems mean more pruning, sooner, so longer stems are good for efficiency. Hence, we need a routine to try to move literals into the stem whenever possible. Let us now consider these operations one by one.

Branch BCT nodes do not prune any search space, thus our BCT construction algorithm is carefully designed to maximize the number of literal nodes in every branch and suppress branch nodes closer to leaves. To formalize this property, we define *normalized* BCTs.

Definition 5 (Normalized BCT): A BCT T is a called a *normalized* BCT if, given any branching node n of T, the stems of the two sub-BCTs rooted at 0- and 1-child of n do not have any common literals.

A normalized BCT can be obtained from the non-normalized one by deleting the common literals from adjacent branches and percolating them above

[5] $X \rightarrow Y$ can be determined by computing $X \wedge Y$ and checking if the result is equal to X.

their common branching node. Our normalization algorithm performs the task in time linear in the size of the BCT by simple in-order traversal and some book-keeping. Due to its simplicity, the normalization algorithm is omitted.

The implementation of BCT construction, intersection, and subsumption checking consists of several mutually recursive functions. We now present abstracted and simplified versions of these algorithms.

Letters n, m denote nodes of a BCT, while h, k, l represent literals. Given a branch node n such that $\mathsf{var}(n) = l$, its children will be denoted as $n.child\,(l)$ and $n.child\,(\overline{l})$, while the fields containing pointers to those nodes will be written as $n.l$ and $n.\overline{l}$. Several helper functions are used for more compact representation:

- $Lit\,(n)$ – Returns the set of literals that the node represents. For a literal node n, $Lit\,(n) = l$ (or $Lit\,(n) = \overline{l}$), while for a branch node $Lit\,(n) = \{l, \overline{l}\}$
- $Delnd\,(n)$ – Deletes a node from the BCT and updates all pointers accordingly.
- $Enlist\,(S)$ – Returns a list of literal nodes labelled with literals from the set S.
- $Mknd\,(k)$ – Creates a new node labelled with literal k.

Algorithm 2 computes an overapproximation of the union (disjunction) of a new CC and a BCT (representing an overapproximation of the union of the previously seen CCs). We then normalize the resulting BCT. The union algorithm takes five parameters: a BCT T, a set of literals found in the new CC ψ, and three sets for temporary storage that are initially empty. Literals labeling nodes from T are copied to one of those three sets during the traversal. Candidates for new branch nodes are stored in B, nodes labelled with literals that are conflicting with the new certificate are deleted from T and the literals are stored in the set of eliminated literals E. Finally, the literals present in the new certificate are copied in the set of matching literals M. A deep copy of a set X is written as X'.

The main `while` loop (lines 3-22) finds a path (or paths) in T that are compatible with ψ. There are several cases to consider. If the literal on the path is present in ψ (line 4), the node is skipped over and the literal is added to M. Notice that if n is a branch node such that $\neg\mathsf{var}(n) \in \psi \vee \mathsf{var}(n) \in \psi$,

Algorithm 2 BCT Union

```
1:  function UNION(T, ψ, B, E, M)
2:    n := root(T)
3:    while n ≠ null ∧ n ≠ X do
4:      if ∃l | l ∈ Lit(n) ∧ l ∈ ψ then
5:        M := M ∪ {l}
6:      else if ∃l | l ∈ Lit(n) ∧ l̄ ∈ ψ then
7:        M := M ∪ {l̄}
8:        B := B ∪ {l}
9:        Delnd(n)
10:     else if n is a branch node then
11:       n.l := UNION(n.child(l), ψ, B', E', M')
12:       n.l̄ := UNION(n.child(l̄), ψ, B, E, M)
13:       if n.child(l) = n.child(l̄) = X then
14:         Delnd(n)
15:       end if
16:       return T
17:     else
18:       E := E ∪ {l}
19:       Delnd(n)
20:     end if
21:     n := n.child(l)
22:   end while
23:   if n = null then
24:     T := Enlist(ψ)
25:   else if |B| > 0 ∧ p ≠ null then
26:     n := Mknd(k) | k ∈ B
27:     B := B\k
28:     n.k := Enlist(B ∪ E)
29:     n.k̄ := Enlist({h̄ | h ∈ B} ∪ ψ\M)
30:   end if
31:   return T
32: end function
```

only the branch that is compatible with ψ is taken. If the literal labeling a literal node n directly conflicts with ψ (line 6), n is deleted from T and the literal is added to B as it can be used as a new branch node in place of one of the nodes from $\mathsf{leaves}(n)$. Branch nodes essentially do not prune any search space, and therefore can't conflict with ψ. In that case both paths are followed (line 10). If both children of a branch node are leaf nodes, such a branch is redundant and can be eliminated. Finally, a literal node n such that $\mathsf{var}(n) \notin \psi \wedge \neg\mathsf{var}(n) \notin \psi$ (line 17) prunes the search space, but this pruning cannot be justified with the new certificate. Hence, such a node is actually in conflict with ψ and the corresponding node has to be eliminated.

The postcondition of the `while` loop is that all the literals labeling the nodes on the path starting at the root and ending at a leaf n are compatible with ψ. More formally $\psi \Rightarrow \mathsf{cube}_T(n)$, for a BCT T.

A $null$ node can be found only when the first ψ is added. Since only ψ determines the search space that needs to be explored, T is initialized to a list of nodes labelled with literals from ψ. Alternatively, if a leaf node n is reached, there are three possible cases:

1) All literals from ψ were matched on the path from $\mathsf{root}(T)$ to n.
2) Some literals from ψ were matched on the path and $B = \emptyset$.
3) Some literals from ψ were matched and $|B| > 0$.

In the first case T needs not to be modified, in the second it would be incorrect to append unmatched literals from ψ to T, and in the third a new branch is created (lines 26-29). Literals from $B, E, \{\overline{h} \mid h \in B\}$, and $\psi \backslash M$ are partitioned in two lists, representing the nodes that are and are not compatible with ψ. The newly created branch node points to those two lists, intuitively representing the case split in the search space according to certificates seen so far.

We now consider the algorithm for BCT intersection and subsumption. Subsumption is checked by computing the intersection of two BCTs and checking that the result is equal to one of them. Algorithm 3 outlines the intersection routine. Intuitively, the intersection of two BCTs T_1 and T_2 can be computed by replacing all the leaves of T_1 with a fresh copy of T_2, eliminating all the conflicting branches, and normalizing the resulting BCT. Algorithm 3 performs the first two steps and closely corresponds to our implementation. Out of those three steps, only the second one is somewhat more involved and requires a closer look. When line 21 is reached, the set M will contain all the literals seen on the path π from $\mathsf{root}(T_1)$ to some leaf. BCT T_2 is traversed in-order. As soon as a conflict is found the entire branch is cut off (line 23). Line 25 skips over the nodes that exist on the path π. Finally, line 27 creates a fresh copy of the node that is compatible with π. Nodes with with no leaf descendants appear when there is a conflict in both branches. Such nodes are recursively deleted (line 30). Occasionally, nodes from T_1 also need to be recursively deleted for the same reason. This is a part of postprocessing and is not shown in the code given above.

Algorithm 3 BCT intersection

```
1:  function INTERSECT(T1, T2, M)
2:      assume root(T1) ≠ null ∧ root(T2) ≠ null
3:      n := root(T1)
4:      if n = X then
5:          assume both T1 and T2 are normalized
6:          return APPEND (T2, M)
7:      else
8:          for each l ∈ Lit (n) do
9:              M := M ∪ {l}
10:             n.l := INTERSECT (n.child (l), T2, M)
11:             M := M\l
12:         end for
13:     end if
14:     return n
15: end function
16:
17: function APPEND(T, M)
18:     n := root(T)
19:     if n ≠ X then
20:         m := Mknd (l) | l ∈ Lit (n)
21:         for each l ∈ Lit (n) do
22:             if l̄ ∈ M then
23:                 m.l := null
24:             else if l ∈ M then
25:                 return APPEND (n.child (l), M)
26:             else
27:                 m.l := APPEND (n.child (l), M)
28:             end if
29:         end for
30:         if m.l = m.l̄ = null then
31:             m := null
32:         end if
33:         return m
34:     end if
35:     return n
36: end function
```

C. Integrating B-Cubing and Other Optimizations

Beyond B-cubing, our implemented solver has several characteristics that distinguish it from other contemporary solvers. First, it is based on a different backtracking mechanism, as is used to integrate supercubing and learning [23]. Second, the learned clauses are aggressively deleted in incrementally increasing, but bounded, periods. The clauses to be

deleted are chosen according to their length and participation in the conflicts. Longer clauses that have participated less frequently in the conflicts are more likely to get deleted. Third, our solver features extensive equivalence preprocessing, similar to the March solver [18], [17], [28]. For learning, our solver adds one learned clause that corresponds to the 1-UIP cut [12] per conflict to the clause database. The solver is not randomized and does not do restarts. We believe that adding those two features would increase its robustness.

Given the proof of correctness for the abstract algorithm, it is easy to prove the algorithm in our implementation correct. All of the details of learning and Boolean constraint propagation are handled via the deduction oracle. As long as the optimizations are sound (correct propagation of constraints and correct learning), the proof from the preceding section still holds.

IV. Experimental Results

The data presented here are for the most recent version of our solver HyperSAT against the most recent version of ZChaff II (version 2004.5.13, which is noticeably faster than earlier versions). ZChaff II is one of the best solvers publicly available, and is based on the standard approach to state-space pruning (conflict-directed backtracking and learning). It has been highly tuned and optimized over several years of development. HyperSAT is less mature, although we have tried to optimize it and have benefitted from many techniques in the SAT-solving literature, as described in section III-C.

As test cases, we have chosen nine sets of SAT benchmarks that reflect the types of SAT instances that arise in EDA applications. The PicoJava™ instances result from bounded model checking (BMC) of Sun's PicoJava II™ microprocessor[6]. The second set (IBM BMC) is the encoding of BMC of industrial hardware designs[7]. The third set contains the well-known barrel, longmult, and queueinvar BMC benchmarks from CMU[8]. Integer factorization is the encoding of array and Wallace tree multipliers that read two n-bit prime numbers and generate the $2n$-bit product[9]. The fifth set consists of SAT encodings of constraint satisfaction Problems (CSP)[10]. The next four sets were submitted to SAT competitions by Eugene Goldberg and represent BMC, FPGA routing, equivalence checking, and randomly generated miter circuit instances[11].

TABLE I
Cumulative Runtimes.

Benchmark Set	ZChaff II	HyperSAT
PicoJava™ BMC (76)	9868 (2)	13635 (2)
IBM BMC (13)	58	81
CMU BMC (34)	5872	935
Int Fact (29)	54470 (10)	11723
CSP (40)	107263 (29)	88990 (21)
Goldberg BMC (10)	1602	4599 (1)
FPGA routing (32)	3803 (1)	715
Eq. check (16)	39701 (10)	51377 (13)
Randnet (48)	74978 (19)	66501 (16)

Table I shows cumulative runtimes for each benchmark set. Runtimes are in seconds. The numbers in parentheses are numbers of problem instances: in the first column, this is the total number of instances in the set; in the other columns, this is the number of instances that exceeded the 1 hour timeout. Figure 3 gives scatter plots showing comparative performance on each problem instance. All experiments were on a 3.2 GHz Pentium 4 with 1MB L2 cache and 1GB RAM.

Clearly, HyperSAT is competitive with the latest ZChaff II, with each solver drastically outperforming the other on some problems. The strong performance on the CSP benchmarks is particularly promising, as intelligent simulation testcase generation relies heavily on constraint solving. HyperSAT also reports fewer timeouts overall. In general, it is valuable to have different approaches with different strengths: what really matters is solving more problem instances, not just solving the already-solvable instances faster.

As a more detailed analysis of the effectiveness of B-cubing in pruning the search space, we analyzed the number of decisions made by our solver using B-cubing versus using supercubing. (Table II) These types of comparisons are difficult to interpret, because minor, irrelevant differences between two

[6]http://www-cad.eecs.berkeley.edu/∼kenmcmil/satbench.html

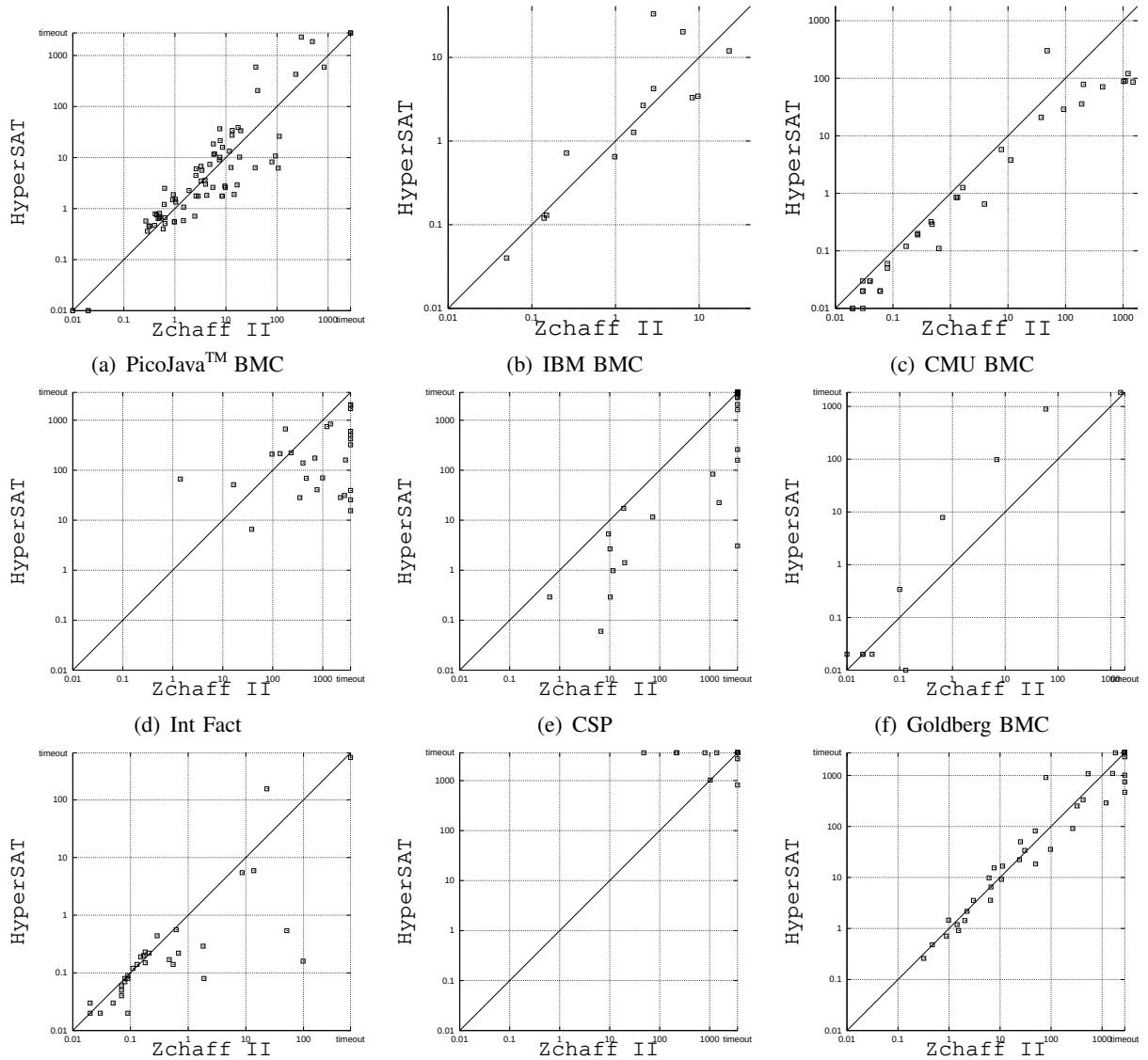
[7]http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/BMC/description.html

[8]http://www-2.cs.cmu.edu/∼modelcheck/bmc/bmc-benchmarks.html

[9]http://www.eecs.umich.edu/∼faloul/benchmarks.html

[10]http://www.nlsde.buaa.edu.cn/∼kexu/benchmarks/benchmarks.htm

[11]http://www.satcompetition.org/2002/submittedbenchs.html



(a) PicoJava™ BMC (b) IBM BMC (c) CMU BMC

(d) Int Fact (e) CSP (f) Goldberg BMC

(g) FPGA routing (h) Eq Check (i) Randnet

Fig. 3. Scatter Plots. Each problem instance plots as a point. Each axis is the runtime of the labeled SAT solver. Timeouts are plotted at 3600 seconds.

solvers can cause the various heuristics to yield radically different results. For two very different solvers, the results would likely be meaningless, hence our comparison of our solver to itself, using two different pruning heuristics. For each benchmark set, we compared only those instances that were successfully solved by both versions of our solver (number indicated in parentheses). As can be seen in the results, the additional pruning information retained and exploited by B-cubing significantly reduced the number of decisions the solver needed to make.[12]

V. Conclusions and Future Work

We have presented a broad theoretical framework for understanding pruning in SAT solvers, as well as an instance of our theory, in the form of B-cubing. Using this theory, we can more simply understand and prove the correctness of our SAT-solving algorithm. We have implemented an efficient B-cubing-

[12]Given that supercubing is simply a less accurate special case of B-cubing, a natural question is how it can be possible for supercubing to beat B-cubing on one benchmark set. The answer is that a slight difference in pruning can result in different decisions being made later by the decision heuristic, which can easily generate enormous differences in the total number of decisions.

TABLE II
AVERAGE NUMBER OF DECISIONS FOR SOLVED BENCHMARKS.

Benchmark Set	Supercubing	B-cubing
PicoJava™ BMC (74)	28911.5	28397.2
IBM BMC (13)	5005.54	5772.77
CMU BMC (34)	32575.1	24564.3
Int Fact (28)	258992	193235
CSP (15)	151210	133026
Goldberg BMC (9)	268962	135823
FPGA routing (32)	43472.5	22655.1
Eq. check (16)	925001	619299
Randnet (22)	375306	267074

based solver, and experimental results show that our solver is competitive with one of the best current solvers, and often outperforms it, on a wide range of benchmarks.

Given the generality of our theory, considerable future work is possible in exploring other pruning techniques permitted by the theory. Obvious directions are better deduction oracles, exploring more general techniques in the gap between Theorem 2 and B-cubing, and finding more efficient ways to implement approximations of B-cubing. For example, HyperSAT currently uses the last cut in the implication graph, which contains only decisions, to construct certification cubes. Conversely, 1-UIP learning [12] makes the cut very close to the conflict. There's a whole range of possible cuts in between those two extremes. In general, the challenge is to find how to exploit the abundance of information that can be learned from the conflicts efficiently. Having a theoretical framework that allows correctly combining techniques that store global information (e.g., conflict-driven learning), via the deduction oracle, and techniques that implicitly store the context (e.g., supercubing, B-cubing), via obligation-certification trees, provides a sound foundation to build on.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *36th ACM/IEEE Design Automation Conference*. ACM Press, 1999, pp. 317–320.

[2] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV 03: Computer-Aided Verification, LNCS 2725*. Springer, 2003, pp. 1–13.

[3] G. Nam, K. Sakallah, and R. Rutenbar, "A Boolean satisfiability-based incremental rerouting approach with application to FPGAs," in *Proceedings of the Conference on Design Automation and Test in Europe*. IEEE Press, 2001, pp. 560–565.

[4] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1167–1176, Sept 1996.

[5] Z. Zeng, M. J. Ciesielski, and B. Rouzeyre, "Functional test generation using constraint logic programming," in *11th International Conference on VLSI/SOC*. IFIP TC10/WG10.5, 2001, pp. 375–387.

[6] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[7] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[8] J. N. Hooker and V. Vinay, "Branching rules for satisfiability," *Journal of Automated Reasoning*, vol. 15, no. 3, pp. 359–383, 1995. [Online]. Available: citeseer.ist.psu.edu/hooker95branching.html

[9] A. Bhalla, I. Lynce, J. de Sousa, and J. Marques-Silva, "Heuristic backtracking algorithms for SAT," *Proceedings. 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, pp. 69– 74, 2003.

[10] C. M. Li and Anbulagan, "Heuristics Based on Unit Propagation for Satisfiability Problems," in *IJCAI (1)*, 1997, pp. 366–371. [Online]. Available: citeseer.ist.psu.edu/li97heuristics.html

[11] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 506–521, 1999.

[12] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Proceedings of the International Conference on Computer-aided Design*. IEEE Press, 2001, pp. 279–285.

[13] F. Bacchus, "Enhancing Davis Putnam with Extended Binary Clause Reasoning," in *Proceedings of National Conference on Artificial Intelligence (AAAI-2002)*, 2002, pp. 613–619.

[14] H. Zhang and M. E. Stickel, "An efficient Algorithm for Unit Propagation," in *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996. [Online]. Available: citeseer.ist.psu.edu/zhang96efficient.html

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of the Design Automation Conference*. ACM Press, 2001, pp. 530–535.

[16] F. Bacchus and J. Winter, "Effective Preprocessing with Hyper-Resolution and Equality Reduction," in *SAT*, 2003, pp. 341–355.

[17] J. P. Warners and H. van Maaren, "A two phase algorithm for solving a class of hard satisfiability problems," *Operations Research Letters*, vol. 23, pp. 81–88, 1998. [Online]. Available: ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R9802.ps.Z

[18] M. Heule and H. van Maaren, "Aligning CNF- and equivalence-reasoning," in *SAT*, 2004, pp. 174–181.

[19] C. M. Li, "Integrating equivalency reasoning into Davis-Putnam procedure," in *AAAI: 17th National Conference on Artificial Intelligence*. AAAI / MIT Press, 2000, pp. 291–296.

[20] L. Zhang and S. Malik, "The quest for efficient boolean satisfiability solvers," in *Proceedings of the 18th International Conference on Automated Deduction*. Springer-Verlag, 2002, pp. 295–313.

[21] M. R. Prasad, "Propositional Satisfiability Algorithms in EDA Applications," Ph.D. dissertation, University of California at Berkeley, 2001.

[22] E. Goldberg, M. R. Prasad, and R. K. Brayton, "Using Problem Symmetry in Search Based Satisfiability Algorithms," in *Proceedings of the Conference on Design, Automation,*

and Test in Europe, 2002, pp. 134–142. [Online]. Available: citeseer.ist.psu.edu/goldberg02using.html

[23] D. Babić and A. J. Hu, "Integration of Supercubing and Learning in a SAT Solver," in *Asia South Pacific Design Automation Conference*. ACM/IEEE, 2005, pp. 438–444.

[24] A. Nadel, "Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and Innovations," Master's thesis, Tel-Aviv University, 2002.

[25] D. Babić, J. Bingham, and A. J. Hu, "Efficient SAT Solving: Beyond Supercubes," in *42nd Design Automation Conference*. ACM/IEEE, 2005, pp. 744–749.

[26] ——, "B-Cubing Theory: New Possibilities for Efficient SAT-Solving," in *International Workshop on High-Level Design Validation and Test*. IEEE Computer Society, 2005, to apear.

[27] D.-Z. Du and K. Ko, *Theory of Computational Complexity*. John Wiley and Sons, 2000.

[28] J. P. Warners and H. van Maaren, "Recognition of tractable satisfiability problems through balanced polynomial representations," in *Proceedings of the 5th Twente Workshop on Graphs and Combinatorial Optimization*. Elsevier Science Publishers B. V., 2000, pp. 229–244.

Domagoj Babić (S'97) received the Dipl.Ing. degree in electrical engineering and M.Sc. in computer science from the Faculty of Electrical Engineering and Computing of Zagreb University, Croatia. He is a Microsoft Fellow and a Ph.D. student in computer science at the University of British Columbia, Vancouver, Canada. His research interests include formal verification of complex software and hardware systems, SAT algorithms, constraint solving in general, and automated theorem proving.

Jesse Bingham (S'99) received the B.Sc. and M.Sc. degrees from the University of Victoria, Canada, in 1998 and 2001, and the Ph.D. degree from the University of British Columbia, Canada, in 2005, all in computer science. His research interests center around formal verification: verification of multiprocessor shared memory models, parameterized model checking, abstraction, and logic. He has recently joined Intel Corporation, in Hillsboro, Oregon.

Alan J. Hu (S'86–M'97) received the B.S. and Ph.D. degrees from Stanford University, in 1989 and 1996. He is an associate professor in the Department of Computer Science at the University of British Columbia. For the past 15 years, his main research focus has been automated, practical techniques for formal verification. He has served on the program committees of most major CAD and formal verification conferences, and chaired or co-chaired CAV (1998), HLDVT (2003), and FMCAD (2004). He was also a Technical Working Group Key Contributor on the 2001 International Technology Roadmap for Semiconductors, and is a member of the Technical Advisory Board of Jasper Design Automation.