B-Cubing Theory: New Possibilities for Efficient SAT-Solving*

Domagoj Babić, Jesse Bingham, and Alan J. Hu
Department of Computer Science, University of British Columbia
(babic,jbingham,ajh)@cs.ubc.ca

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 clarifies the theoretical basis for B-cubing, proves our approach correct, and maps out other correct possibilities for further improving SAT-solving.

1 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 [6] and unbounded [15]), FPGA routing [18], ATPG [20], and simulation testcase generation [23]. 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 [8, 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 [8] 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 [11, 5, 13], search-space pruning techniques [14, 25, 3], and efficient implementation [24, 16]. 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 [4] and equivalence reasoning [21, 10, 12].

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) [14], 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 [26].

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 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 [19, 9]. The proposed theoretical framework can be used as a basis for

*Supported by an NSERC Discovery Grant and graduate fellowships from the University of British Columbia.

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

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

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 [2] 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 [17] 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 recent work [1], 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.

This paper corrects that deficiency. We introduce a new theory to explain 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. The theoretical understanding also shows that there are many correct ways to exploit the pruning information derived from B-cubing, which is likely a fertile area for future work.

2 Experimental Motivation

We structure this paper backwards (experiments preceding theory) because our research proceeded somewhat backwards: we had developed a SAT solver that performed well and behaved correctly in extensive regression testing before we had fully understood the theoretical foundation of our approach or proven it correct. We have published an informal explanation of our solver along with experimental results elsewhere [1]; here, we briefly review the experimental performance of an implementation of our new approach compared to a leading solver implementing the standard approach to state-space pruning.

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. Beyond B-cubing, HyperSAT has several characteristics that distinguish it from other contemporary solvers. First, it is based on different backtracking mechanism as explained in [2]. 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 [10, 21, 22]. For learning, HyperSAT adds one learned clause that corresponds to the 1-UIP cut [25] per conflict to the clause database. HyperSAT is not randomized and does not do restarts. We believe that adding those two features would increase its robustness.

As test cases, we have chosen nine sets of SAT benchmarks that reflect the types of SAT instances that arise in EDA applications. PicoJava™ instances result from Bounded Model Checking (BMC) of Sun's PicoJava II™ microprocessor[3]. Second set (IBM BMC) is the encoding of BMC of industrial hardware designs[4]. The third set contains the well-known barrel, longmult, and queueinvar BMC benchmarks from CMU[5]. Integer factorization is the encoding of array and Wallace tree multipliers that read two n-bit prime numbers and generate $2n$-bit product[6]. The fifth set is SAT encoding of Constraint Satisfaction Problems (CSP)[7]. 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[8].

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

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

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

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

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

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

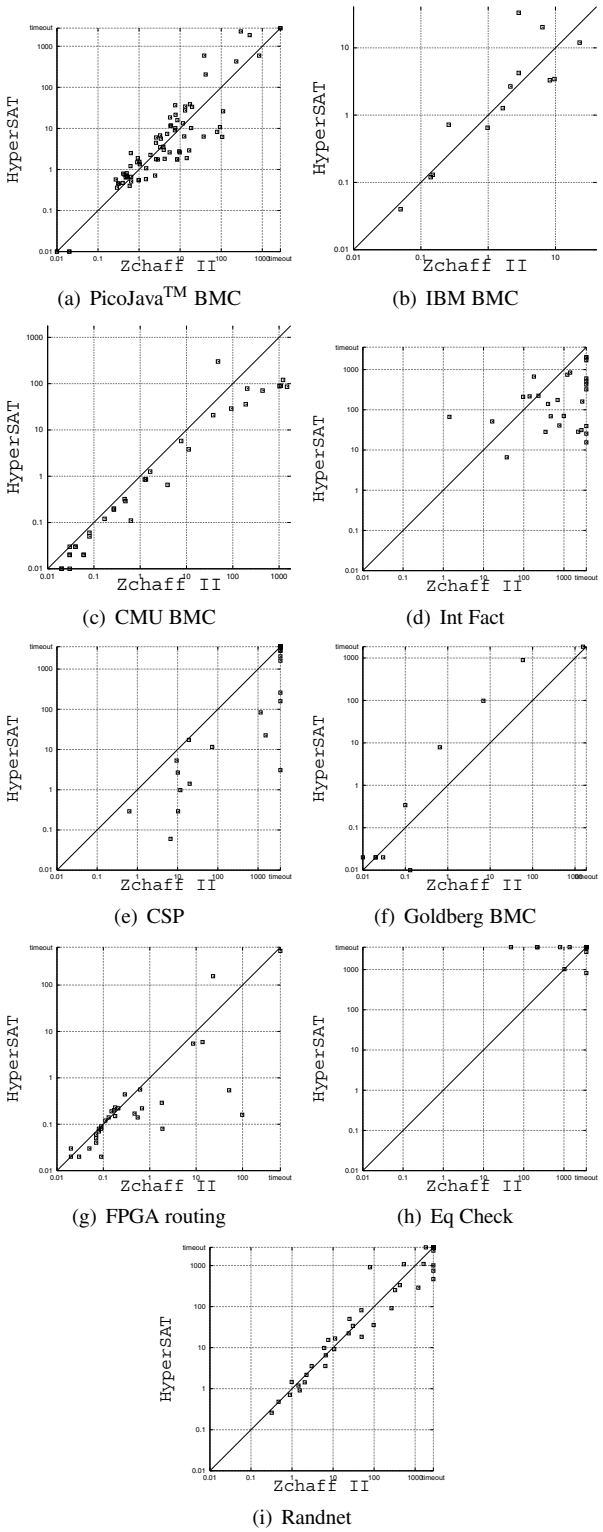
Benchmark Set	ZChaff II	HyperSAT
PicoJava^{TM} BMC (76)	9868 (2)	13635 (2)
IBM BMC (13)	58	81
CMU BMC (34)	5872	935
Int Fact (29)	54470 (10)	11723
CSP (40)	107263 (29)	88990 (21)
Goldberg BMC (10)	1602	4599 (1)
FPGA routing (32)	3803 (1)	715
Eq. check (16)	39701 (10)	51377 (13)
Randnet (48)	74978 (19)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

Table 1. Cumulative Runtimes. 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.

Table 1 shows cumulative runtime for each benchmark set. Figure 1 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.

Given the promising experimental results, we would like develop a solid understanding of the theory behind our method, and prove it correct.



(a) PicoJava^{TM} BMC (b) IBM BMC (c) CMU BMC (d) Int Fact (e) CSP (f) Goldberg BMC (g) FPGA routing (h) Eq Check (i) Randnet

Figure 1. 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.

3 Theoretical Framework

3.1 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. 2).

A 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 [19, 9], JeruSAT [17], and B-Cubing [1] 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.

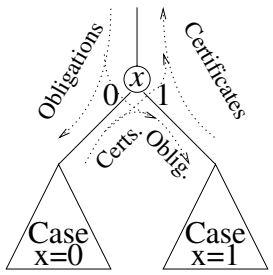


Figure 2. Information from the left subtree is used in the right subtree.

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.

3.2 Preliminary Definitions

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

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 **1** (resp. **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 $\mathscr{V}$, a SAT-solving algorithm explores the space of variable assignments to $\mathscr{V}$, 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 $\mathscr{V}$, a* search tree *(over $\mathscr{V}$) is a rooted binary tree T with the following properties. Each non-leaf node u of T is labeled with a variable $\mathsf{var}(u) \in \mathscr{V}$, and on any path from the root to a leaf, each $x \in \mathscr{V}$ 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 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 $\mathsf{cube}(u) = \mathsf{cube}(u') \wedge \mathsf{var}(u')$. To simplify our presentation, we assume that SAT algorithms always visit the 0-child of a node before the 1-child; it should be clear that generality is not lost.

3.3 Obligation-Certification Trees

Let f be an unsatisfiable boolean formula over $\mathscr{V}$ and consider a SAT algorithm $\mathscr{A}$ running against f. $\mathscr{A}$ constructs a search tree T over $\mathscr{V}$, 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, $\mathscr{A}$ 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.

For example, consider the standard DPLL-style solver with conflict-directed learning. As the decision heuristic makes decisions, the solver is essentially constructing a path in a search tree to a node u, where $\mathsf{cube}(u)$ contains all the decision literals (and their implications) that have been made so far. The solver will proceed to look for satisfying solutions by extending the partial assignment of literals in $\mathsf{cube}(u)$, and it cannot claim unsatisfiability until exploring the entire subspace (thereby proving that $\mathsf{cube}(u) \rightarrow \neg f$). If the solver deduces a conflict at u, the conflict analysis procedure finds a subset of the assigned literals that implies the conflict. The conjunction of these literals constitutes a certification ψ of unsatisfiability for $\mathsf{cube}(u)$, since $\mathsf{cube}(u) \rightarrow \psi$ and $\psi \rightarrow \neg f$. Conflict-directed learning consists of reusing ψ to reduce the obligations ϕ of future

nodes in the search tree: since we know that $\psi \rightarrow \neg f$, we can restrict future obligations to the space where $\neg\psi$. In the standard approach, this is typically implemented by negating ψ to create a conflict clause, and conjoining that into the clause database of f.

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 $\mathscr{V}$ is a triple (T, ϕ, ψ) where T is a search tree over $\mathscr{V}$, and ϕ and ψ are mappings, respectively called the* obligations *and the* certifications*, that map the nodes of T to $\mathsf{bf}(\mathscr{V})$, such that for all nodes u we have both*

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

Condition (1a) states that the certification $\psi(u)$ must cover the entire obligation $\phi(u)$, when restricted to the appropriate part of the search space defined by $\mathsf{cube}(u)$. Condition (1b) guarantees that $\psi(u)$ really is a certification of unsatisfiability.

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 3.1 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 $\mathscr{V}$, and let ϕ and ψ be mappings that take the nodes of T to $\mathsf{bf}(\mathscr{V})$, 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

$$\begin{aligned}(3a) \Rightarrow & (\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg(\phi(u) \wedge \neg x) \vee \psi(u_0)) \text{ by (2a)}\\ \equiv & (\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg\phi(u) \vee x \vee \psi(u_0))\\ \equiv & (\mathsf{cube}(u) \wedge \neg x) \rightarrow (\neg\phi(u) \vee \psi(u_0))\\ \Rightarrow & (\mathsf{cube}(u) \wedge \neg x) \rightarrow (\psi(u_0) \vee \neg\phi(u) \vee \psi(u_1))\end{aligned}$$

Similarly,

$$(3c) \quad \Rightarrow (\mathsf{cube}(u) \wedge x) \rightarrow (\psi(u_0) \vee \neg\phi(u) \vee \psi(u_1))$$

And thus we have

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

□

4 B-Cubing

4.1 Abstract Algorithm

We can now present a very general, abstract version of our algorithm, and prove its correctness.

Roughly, 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 to reduce the obligations (i.e. prune) 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. Now $\mathsf{B}(u)$ is built from the CCs found in $\mathsf{leaves}_0(u)$. 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 these over-approximate 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*[9]

$$\mathsf{B}(u) = \exists x_1,\ldots,x_k,x : \bigvee_{w\in\mathsf{leaves}_0(u) \text{ and } \psi_{\mathsf{cc}}(w)\rightarrow\neg x} \psi_{\mathsf{cc}}(w)$$

$\mathsf{B}(u)$ disjoins all CCs found under the 0-branch of u that contain $\neg x$, and quantifies out x along with all variables above x in the search tree. 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 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 $\vdash m\rightarrow f$ or $\vdash m\rightarrow\neg f$ 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; an 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).

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:         ψ0 := SAT-SOLVE(f, cube∧¬x, φ∧¬x)
11:         if ⊢ [φ]_{x=1} → [φ]_{x=0} then
12:             ψ1 := SAT-SOLVE(f, cube∧x, φ∧x∧B)
13:         else
14:             ψ1 := SAT-SOLVE(f, cube∧x, φ∧x)
15:         end if
16:         return(ψ0 ∨ ψ1)
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.[10] B denotes $\mathsf{B}(u)$, where u is the current node of the search tree.

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

[9] 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.

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

subtree, is an OCT for f. 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 $\phi(u_1) = \phi(u) \wedge x$, hence (2b) trivially holds. Now suppose that line 12 is the invocation of u_1. Then

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

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

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

where $x_1, \ldots, x_k$ is the list of variables in $\mathsf{cube}(u)$. Clearly, because of lines 5, 6, and 16, we have $\psi(u_0) \equiv (C \vee C')$, and therefore

$$(\neg C' \wedge \psi(u_0)) \rightarrow C \tag{6}$$

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

$$\phi(u_0) \rightarrow \psi(u_0) \tag{7}$$

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

□

4.2 Concrete Implementation

We now turn to the correctness of the algorithm we actually implemented. Details of HyperSAT's implementation are available elsewhere [1], but the main difficulties we faced when building HyperSAT were how to integrate learning and other pruning techniques with B-cubing or supercubing, and how to correctly maintain a small and efficient approximation of the B-cube.

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.

Similarly, for practical efficiency, we developed a data structure to represent approximations of B-cubes. We invested considerable effort in making this data structure efficient, which complicated our understanding of the algorithm. With the theoretical framework, however, we simply note that our data structure is an overapproximation of the B-cube, and the proof of Algorithm 1 still holds. Note that since supercubing further overapproximates B-cubes, we have also proven the correctness of supercubing, with or without integration with conflict-directed learning.

Obviously, different implementation choices can differ greatly in performance, but the algorithmic correctness is assured. Freed from worrying about the details of correctness, we can hopefully consider much more subtle performance optimizations.

5 Conclusions and Future Work

We have presented a broad theoretical framework for understanding pruning in SAT solvers, as well as a generalized, abstract version of our B-cubing pruning algorithm, which we have proven correct. Using this theory, we can more simply understand and prove the correctness of the algorithm in our new SAT solver. 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 curently uses the last cut in the implication graph, which contains only decisions, to construct certification cubes. Conversely, 1-UIP learning [25] 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 techiques 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] D. Babić, J. Bingham, and A. J. Hu. Efficient SAT Solving: Beyond Supercubes. In *42nd Design Automation Conference*, pages 744–749. ACM/IEEE, 2005.

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

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

[4] F. Bacchus and J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *SAT*, pages 341–355, 2003.

[5] 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*, pages 69–74, 2003.

[6] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317–320. ACM Press, 1999.

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

[8] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[9] 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*, pages 134–142, 2002.

[10] M. Heule and H. van Maaren. Aligning CNF- and equivalence-reasoning. In *SAT*, pages 174–181, 2004.

[11] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[12] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI: 17th National Conference on Artificial Intelligence*, pages 291–296. AAAI / MIT Press, 2000.

[13] C. M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *IJCAI (1)*, pages 366–371, 1997.

[14] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.

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

[16] 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*, pages 530–535. ACM Press, 2001.

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

[18] 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*, pages 560–565. IEEE Press, 2001.

[19] M. R. Prasad. *Propositional Satisfiability Algorithms in EDA Applications*. PhD thesis, University of California at Berkeley, 2001.

[20] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, Sept 1996.

[21] J. P. Warners and H. van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. *Operations Research letters*, 23:81–88, 1998.

[22] J. P. Warners and H. van Maaren. Recognition of tractable satisfiability problems through balanced polynomial representations. In *Proceedings of the 5th Twente workshop on on Graphs and combinatorial optimization*, pages 229–244. Elsevier Science Publishers B. V., 2000.

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

[24] 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.

[25] 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*, pages 279–285. IEEE Press, 2001.

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