Using Problem Symmetry in Search Based Satisfiability Algorithms*

Evgueni I. Goldberg
Cadence Berkeley Laboratories
Berkeley, CA
egold@cadence.com

Mukul R. Prasad
Fujitsu Labs. of America
Sunnyvale, CA
mukul@fla.fujitsu.com

Robert K. Brayton
Dept. of EECS
University of California, Berkeley
brayton@eecs.berkeley.edu

Abstract

We introduce the notion of problem symmetry in search-based SAT algorithms. We develop a theory of essential points to formally characterize the potential search-space pruning that can be realized by exploiting problem symmetry. We unify several search-pruning techniques used in modern SAT solvers under a single framework, by showing them to be special cases of the general theory of essential points. We also propose a new pruning rule exploiting problem symmetry. Preliminary experimental results validate the efficacy of this rule in providing additional search-space pruning beyond the pruning realized by techniques implemented in leading-edge SAT solvers.

1 Introduction

The *Boolean Satisfiability (SAT)* problem is a core problem in mathematical logic and computing theory. The last decade has seen significant improvements in SAT solver technology [6, 7, 11]. Spurred by these developments SAT solvers have been actively used in a number of EDA applications including ATPG [9], formal verification [1, 2],logic optimization [5] and physical design [10] among others. Almost all leading-edge SAT solvers use a backtracking algorithm based on the classical *Davis-Putnam-Logemann-Loveland procedure (DPLL)* [3] enhanced with some form of non-chronological backtracking and conflict based learning [6, 7]. This work develops the notion of *problem symmetry* to formally characterize and enhance the search space pruning of a SAT solver operating in such a setting.

The notion of *problem symmetry* stems from the simple observation that in certain regions of the Boolean space the unsatisfiability of the CNF under check can be established without using a certain variable, say x. In other words, in this sub-space the CNF is symmetric with respect to x (or this is a *symmetric subspace with respect to* x)[1]. In the context of a backtracking-based SAT algorithm this can be used as follows. Consider the backtracking search tree shown in Figure 1. When exploring the left branch of branching variable x ($x = 0$) the algorithm computes an (under) approximation of the symmetric sub-space (out of the space explored under the branch $x = 0$) with respect to x (subspace R1 in Figure 1) and in the right branch of x ($x = 1$) the counter-part of this symmetric sub-space is pruned (subspace R2 in Figure 1).

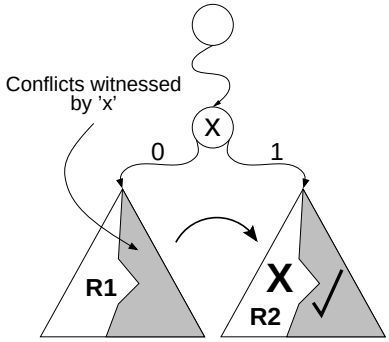


Figure 1. Illustration of Symmetry in Search

Our main contributions in this work are as follows:

- We introduce the notion of *problem symmetry* and formally characterize the potential search-space pruning afforded by it through the theory of *essential points*.
- We show that many popular search pruning techniques (such as the pure-literal rule, non-chronological backtracking and conflict based learning) employed in leading-edge SAT solvers are in fact special cases of pruning under the general theory of essential points. Thereby this work unifies these apparently disparate techniques under a single framework and paves the way for discovering several new pruning techniques.

*This work was supported in part by the California MICRO program and industrial sponsors Cadence, Synopsys and Fujitsu.

[1] Note that this notion of symmetry is distinct from the often used notion of a Boolean function being symmetric with respect to certain variables.

- We propose a new, simple and efficient pruning technique called *supercubing based pruning*, based on problem symmetry. Preliminary experimental results demonstrate this to be effective in providing search-space pruning over and above the pruning afforded by existing techniques in SAT solvers.

The rest of the paper is organized as follows. Section 2 presents the notational framework used in the exposition. In Section 3 we illustrate the notion of problem symmetry with a few examples. The theory of *essential points* and a formal characterization of problem symmetry is developed in Section 4. Section 5 presents theoretical results showing several popular pruning techniques used in SAT solvers to be special cases of the general theory of essential points. In Section 6 we present a new pruning rule called the *supercubing rule*. This is also a special case of problem symmetry but subsumes some existing pruning techniques and is orthogonal to others. Section 7 presents preliminary experimental results validating the efficacy of this rule. Conclusions and suggestions for future research are presented in Section 8.

2 Definitions & Notation

The following discussion will be with respect to SAT instances expressed as *conjunctive normal form (CNF)* formulas. A CNF formula f on n Boolean variables $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$ is a conjunction of m clauses $C_1, C_2, \ldots C_m$. Each clause is a disjunction of literals over the variables $\mathbf{X}$. Let l denote a literal of one of the variables $\mathbf{X}$. *lit*(x) refers to a literal of variable x *i.e.* *lit*(x) is either x or $\overline{x}$. ψ refers to a minterm or point in the 2^n Boolean space of variables $x_1, x_2, \ldots, x_n$. Note that a minterm ψ is a complete Boolean assignment to the variables $\mathbf{X}$. Further, formula f can be evaluated under this assignment. In the sequel we will occasionally use a literal of a variable to refer to a particular value assignment to the variable (*e.g.* $x = 0 \equiv \overline{x}$) and a cube (minterm) to refer to a partial(complete) value assignment to variables of $\mathbf{X}$. $\mathcal{A}(x)$ refers to the current assignment of variable x or alternatively the literal corresponding to that assignment.

The underlying SAT algorithm used for the discussion is the basic DPLL [3] algorithm, augmented with some form of *conflict analysis, non-chronological backtracking* and *conflict clause recording* [6]. This is representative of the SAT methods implemented in most leading-edge SAT solvers [6, 7, 11].

As in [6, 7] a variable that is consciously chosen and assigned a value by the branching procedure is referred to as a *decision variable (assignment)* and is distinguished from a *deduced variable (assignment)* whose value has been implied through *Boolean constraint propagation (BCP)*. A *conflict condition* is denoted by $\mathcal{X}$. A conflict condition occurs when the current partial assignment (during branching) unsatisfies one or more clauses of the CNF. The conflict is identified by one of these clauses, which is referred to as the *conflict clause* of conflict $\mathcal{X}$ and denoted by $C(\mathcal{X})$[2].

$\mathcal{I}(l)$ refers to the clause that was used to imply or deduce the literal l. Although, there can be many such clauses, $\mathcal{I}(l)$ is one of them, which is held responsible for the deduction. $\mathcal{I}(C)$ refers to the set of deduced literals of clause C *i.e.* the set of literals assigned through BCP implications from other clauses. $\mathcal{D}(C)$ refers to the set of literals of C assigned through decision assignments.

Given a conflict condition $\mathcal{X}$, conflict analysis performs the task of identifying a subset of assignments, denoted $\mathcal{A}_{\mathcal{R}}(\mathcal{X})$ (out of the current set of decision and deduced assignments) which can be held responsible for $\mathcal{X}$. As noted in [6, 12] there can be multiple ways at arriving at such a subset (*i.e.* there can be multiple possible $\mathcal{A}_{\mathcal{R}}(\mathcal{X})$ for a given $\mathcal{X}$). For the sake of concreteness we will use the following definition of $\mathcal{A}_{\mathcal{R}}(\mathcal{X})$ in the sequel.

Consider the following recursive marking function $\mathcal{M}(C)$, which operates on a clause C and is defined as

$$\mathcal{M}(C) = \mathcal{D}(C) \cup \mathcal{I}(C) \bigcup_{l \in \mathcal{I}(C)} \mathcal{M}(\mathcal{I}(l)) \qquad (1)$$

Then $\mathcal{A}(\mathcal{X}) = \mathcal{M}(C(\mathcal{X}))$. Further $\mathcal{A}(\mathcal{X})$ can be split into disjoint subsets $\mathcal{A}_{\mathcal{D}}(\mathcal{X})$ and $\mathcal{A}_{\mathcal{I}}(\mathcal{X})$ which are respectively the decision and implied assignments comprising $\mathcal{A}(\mathcal{X})$. The clause $C_{\mathcal{R}}(\mathcal{X})$ recorded on conflict $\mathcal{X}$ is defined to be:

$$\begin{aligned} \mathcal{A}_{\mathcal{R}}(\mathcal{X}) &= \mathcal{A}_{\mathcal{D}}(\mathcal{X}) \\ C_{\mathcal{R}}(\mathcal{X}) &= \bigvee_{l \in \mathcal{A}_{\mathcal{R}}(\mathcal{X})} \bar{l} \end{aligned} \qquad (2)$$

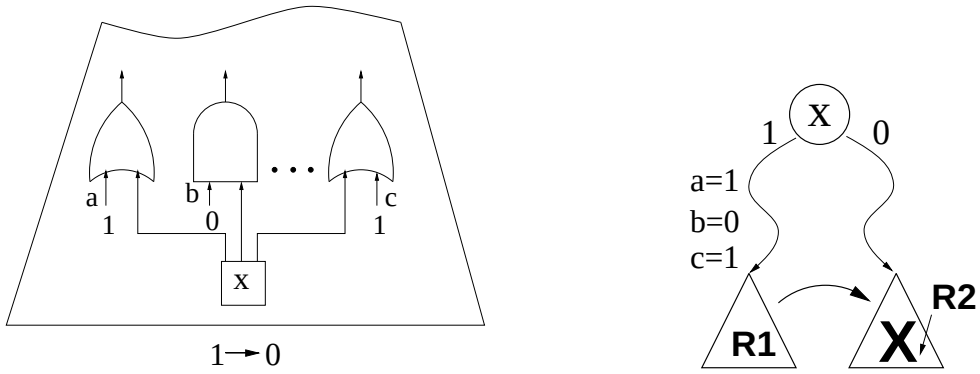
Definition 2.1 *Given a clause C denote by $\mathcal{U}(C)$ the unsatisfiability cube of C which is the set of minterms (assignments) which unsatisfy C, e.g. given $\mathbf{X} = \{x_1, x_2, x_3\}$ and $C = (x_1 + \overline{x_2}), \mathcal{U}(C) = \{\overline{x_1}x_2x_3, \overline{x_1}x_2\overline{x_3}\}$.*

Note that $\mathcal{U}(C)$ can also be interpreted as a cube of literals. For the above example $\mathcal{U}(C) = \overline{x_1}x_2$. In the following we use the two interpretations interchangeably.

3 Problem Symmetry in Search

The notion of problem symmetry has been introduced and its potential in search space pruning motivated briefly in Section 1. In this Section we provide two examples to buttress this understanding and illustrate that 1.) instances of problem symmetry are plentiful in typical SAT instances arising from EDA applications, and 2.) current

[2] This should be distinguished from the new clause $C_{\mathcal{R}}(\mathcal{X})$ which is recorded or deduced on a conflict $\mathcal{X}$.



(a) Example Sub-circuit (b) Backtracking Tree

Figure 2. Example of Symmetry in SAT on circuits

pruning techniques harness only a fraction (albeit inadvertently) of the potential search space pruning afforded by problem symmetry.

Consider the sub-circuit shown in Figure 2(a). Assume that this is part of a larger circuit on which some SAT problem is being solved[3]. Here x is a primary input of the circuit and the three gates shown are the only fanouts of x. Suppose the backtrack tree explored by the SAT algorithm is of the form shown in Figure 2(b). Consider the left branch ($x = 1$) of branching variable x. Suppose that under this branch the algorithm subsequently makes the assignments $a = 1, b = 0$ and $c = 1$ (and potentially other assignments as well) and reaches a sub-space $R1$ (shown in Figure 2(b)). Note that in sub-space $R1$ the value of x is no longer relevant *i.e.* the formula is *symmetric with respect to* x *in* $R1$. Thus, if the algorithm finds sub-space $R1$ unsatisfiable then it need not explore the sub-space $R2$, the counterpart of $R1$ under the branch $x = 0$, as that too will be unsatisfiable. This is a simple and classical case of problem symmetry in SAT instances derived from logic circuits, which *may not* be effectively covered by existing search pruning techniques.

The next example is designed to illustrate that current implementations of conflict clause recording exploit only a fraction of the search space pruning potentially afforded by problem symmetry. Consider the following CNF formula.

$$\begin{aligned} f \quad = \quad & (\overline{b} + c + d)(\overline{b} + \overline{c} + d)(b + c + d)(b + \overline{c}) \\ & (\overline{b} + \overline{c} + \overline{d})(b + c + \overline{d})(a + \overline{b} + c + \overline{d}) \\ & (\overline{a} + \overline{b} + c + \overline{d}) \end{aligned} \tag{3}$$

A typical backtracking tree for solving this CNF is shown in Figure 3. The backtracking algorithm employs conflict analysis, clause recording *etc*. The recorded clauses (as per the specific scheme described in Section 2) are shown below

[3]This means that an appropriate CNF formula is extracted from the circuit and solved by a SAT solver.

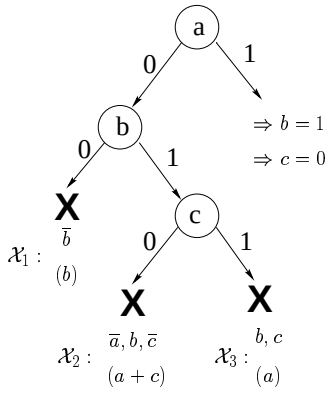


Figure 3. Symmetry in backtrack search

each conflict. Also noted are the set of decision assignments relevant to the conflict. An analysis of the conflicts in the branch $a = 0$ reveals that a was only relevant to a conflict when $b = 1$ and $c = 0$. The rest of the sub-space under $a = 0$ represents the symmetric space with respect to a. Thus, when exploring the right branch of a, *i.e.* $a = 1$ we do not need to explore the sub-space $b = 0 \wedge c = 1$. In other words, on taking the branch $a = 1$ we can immediately assert $b = 1$ and $c = 0$. Note, that the assertion $b = 1$ is also deduced by means of the recorded conflict clause (b) but the additional assignment $c = 0$ comes only through exploiting problem symmetry more fully. Note that this observation is not an artifact of the specific conflict clause recording mechanism used in this work and in this example. Rather it is a fundamental limitation of conflict-based learning in that on a given conflict the recorded clause(s) represents only a fraction of the implicates that can be learned from that conflict. It is neither feasible nor practical to learn all possible implicates. However, it may be possible to extract additional pruning using another, complementary technique

based on problem symmetry. The *Supercubing technique* presented later in Section 6 is a simple example of one such option.

Note that during the search, certain variables, initially picked as decision variables, become deduced variables due to BCP implications from newly added conflict clauses, *e.g.* in Figure 3, $b = 1$ can be treated as a deduced assignment implied from the clause (b), recorded on conflict $\mathcal{X}_1$. Such assignments are called failure-driven assertions (FDA) [6]. However, $b = 1$ may as well be treated as decision assignment. In our treatment, FDAs are treated as deduced assignments for the purpose of generating the recorded conflict clauses $C_{\mathcal{R}}(\mathcal{X})$. However, for generating the responsible assignments shown in Figure 3 (and for the supercubing rule presented in Section 6) FDAs are treated as decision assignments. Both versions of the analysis still use Equations 1 and 2 but generate different sets $\mathcal{A}_{\mathcal{D}}(\mathcal{X})$.

4 The Theory of Essential Points

In this section we develop the notion of *essential points* to formally characterize the search space pruning that can be realized by exploiting problem symmetry.

Definition 4.1 *A point ψ is called $l-$essential if all clauses of f unsatisfied by ψ (must be at least one[4]) contain literal l, e.g. given $f = (\overline{a} + \overline{b})(c)(a + \overline{b} + \overline{c})(\overline{a} + \overline{c})(a + b + \overline{c})$ the minterm abc is an $\overline{a}$-essential point.*

Definition 4.2 *Let ψ and ψ^* be two points in the 2^n Boolean space. ψ^* is said to be x-symmetric to ψ if it is obtained from ψ by flipping the value of variable x in ψ. For example, minterms $\psi = \overline{a}bcd$ and $\psi^* = \overline{a}b\overline{c}d$ are c-symmetric with respect to each other.*

Proposition 4.1 *Let ψ be a complete assignment to variables $x_1, x_2, \ldots, x_n$ (i.e. a minterm of 2^n) which satisfies f. Then assignment ψ^* which is x-symmetric to ψ is either lit(x)-essential (lit$(x) \in \psi$) or satisfies f i.e. is a solution.*

Proof: Suppose ψ^* is neither a solution nor lit(x)-essential (where *lit*$(x) \in \psi$). Then there exists a clause C of f such that ψ^* unsatisfies C and C does not contain **any** x literal. But then C is unsatisfied by ψ as well. Therefore ψ is not a solution of f. Contradiction ! ∎

Proposition 4.2 *If assignment ψ is lit(x)-essential then assignment ψ^*, x-symmetric to ψ, is either $\overline{lit(x)}$-essential or is a solution.*

Proof: Suppose ψ^* is neither a solution nor $\overline{lit(x)}$-essential. Then there exists a clause C of f which does not contain **any** x literal, such that ψ^* unsatisfies C. But then C is unsatisfied by ψ as well. Therefore ψ is not lit(x)-essential. Contradiction ! ∎

For a literal l, the set of l-essential points with respect to the current CNF is denoted by $\mathcal{E}(l)$. The subset of $\mathcal{E}(l)$ lying in a sub-space ϕ is denoted by $\mathcal{E}_{\phi}(l)$ and by $\mathcal{E}_{sub}(l)$ when the sub-space being referred to is clear from the context.

The search space pruning that can be achieved using the notion of essential points can be operationally defined by the following theorem.

Theorem 4.1 *Suppose the algorithm has explored the left branch of variable x (without loss of generality $x = 0$) and found no solution. Moreover, suppose the algorithm has computed $\mathcal{E}_{\overline{x}}(x)$ (the subset of $\mathcal{E}(x)$ in the Boolean sub-space spanned by the $x = 0$ branch). Then under the branch $x = 1$, solutions of f must lie in the set of points x-symmetric to points in $\mathcal{E}_{\overline{x}}(x)$ (denoted by $\mathcal{E}^*_{\overline{x}}(x)$).*

Proof: For correctness, the algorithm only needs to ensure that it does not skip any solutions of the CNF in the branch $x = 1$ (it can prune everything else). By Proposition 4.1 solutions can only be points x-symmetric to points in $\mathcal{E}_{\overline{x}}(x)$. ∎

Theorem 4.1 implies that for testing satisfiability of f, when exploring the branch $x = 1$, the algorithm only needs to explore the set of points $\mathcal{E}^*_{\overline{x}}(x)$. It is also easy to see that it is not necessary to compute the set $\mathcal{E}_{\overline{x}}(x)$ exactly. Any over-approximation of it would work as well, though the amount of pruning would be reduced proportionally.

Under a clause recording scenario, *i.e.* when the algorithm progressively adds implicates of the CNF to the clause database (for example through conflict clause recording) the set of essential points $\mathcal{E}(l)$ for each literal l either remains unchanged or shrinks.

Theorem 4.2 *Let CNF f^+ be obtained from f by adding clause C^+ to f where C^+ is an implicate of f. Then, for any literal l, the set of essential points of l in f^+, denoted $\mathcal{E}^+(l)$ must satisfy $\mathcal{E}^+(l) \subseteq \mathcal{E}(l)$.*

Proof: Consider any minterm $\psi \notin \mathcal{E}(l)$. Then, there must exist a clause C of f such that $l \notin C$ and $\psi \in \mathcal{U}(C)$. But, since $f^+ = f \wedge C^+$, C is also a clause of f^+. Thus, $\psi \notin \mathcal{E}^+(l)$. Therefore, $\psi \notin \mathcal{E}(l) \Rightarrow \psi \notin \mathcal{E}^+(l)$. ∎

The relevance of Theorem 4.2 is that under a clause recording scenario, when a new clause is added, all partial sets of essential points computed up to that point continue to be valid with respect to the new CNF[5].

[4] Thus, satisfying assignments of f are not essential points.

[5] However they can potentially be over-estimates of the essential points with respect to the new CNF.

5 Popular Pruning Techniques: Special Cases of Essential Point Pruning

In the following we show that several popular search pruning techniques such as the *pure-literal rule* [4], *non-chronological backtracking (NCB) and conflict clause recording (or conflict-based learning)* [6] are special cases of the pruning afforded by the theory of essential points. This unifies these techniques under a single framework and paves the way for developing potentially more powerful variants of problem symmetry based pruning.

5.1 The Pure-Literal Rule

The Pure-Literal rule [4] can be used to effect pruning in branching by looking for variables that appear in only one polarity (the *pure polarity*) in open (undecided) clauses, at the current point in the search, and then asserting the variable to the pure polarity. In effect this means pruning the other branch of the variable. If no solution is found in the explored pure-branch, the pruning effected by the pure-literal rule can be explained by the theory of essential points as follows.

The pure-polarity branch of the variable (say $x = 0$) can be considered the left branch of x, which the algorithm explored and found no solution. The other polarity branch $x = 1$ which was pruned by the pure-literal rule is the potential right branch. Thus, if we can prove that the sub-space under the pure-branch $x = 0$ does not contain any x-essential points then the pruning done by the pure-literal rule is explained by Theorem 4.1.

It is sufficient to only consider the case when the pure-literal branch of the pure-literal variable is unsatisfiable, because if there is a solution under the pure-literal branch the algorithm terminates. In such a case the claim of pruning the other branch has no meaning.

Theorem 5.1 *The sub-space under the pure-polarity branch (say $x = 0$) of a pure-literal variable x cannot contain any x-essential points.*

Proof: Consider exploring the pure polarity branch (say $x = 0$) of the pure-literal variable x. By assumption there is no solution under this branch. Now consider the following algorithm which just explores the sub-space under this branch using a stripped-down DPLL procedure (*i.e.* no BCP or pure-literal rule).

Such an algorithm would explore the entire sub-space under the $x = 0$ branch, stopping and *chronologically* backtracking every time the current assignment unsatisfes a clause of the CNF. Let the set of such conflict clauses encountered while exploring this branch be $C_1, C_2, \ldots, C_p$. It is easily seen that $\mathcal{U}(C_1) \cup \mathcal{U}(C_2) \cup \ldots \cup \mathcal{U}(C_p)$ subsumes the entire sub-space under the $x = 0$ branch. Additionally, none of these clauses contain variable x since a conflict clause has all literals unsatisfied by the current assignment and the pure-literal assignment $x = 0$ merely satisfies some clauses and restricts[6] none. The result follows. ■

5.2 Non-Chronological Backtracking (NCB)

The notion of *non-chronological backtracking (NCB)* [6] is used to prune areas of the search space by backtracking to the last variable responsible for the current conflict, rather than the last variable in the current assignment stack. This method effects pruning by skipping the right branch of some of the stack variables. Operationally, this is accomplished by deducing an implicate (through conflict analysis) whose unsatisfiability cube subsumes the regions to be pruned.

Another way of looking at this pruning is that NCB prunes the right branch of a variable x, if and only if all conflicts in the left branch of x were independent of (symmetric in) x. This is obviously a special case of symmetry (described by the theory of essential points) which targets pruning sub-spaces symmetric in a particular variable. Before proving this we state a few simple facts, without proof, to formalize the operational definition of NCB. The interested reader is referred to [6] for details.

- **Fact 1:** NCB pruning is done in a setting where conflict analysis is used to produce conflict clauses (implicates) responsible for the conflict[7].
- **Fact 2:** The deduction procedure for a conflict clause may be simulated by a tree of resolution steps where the leaf clauses are clauses of the original CNF (or previously added conflict clauses) and the variable being resolved out at a node is a deduced variable.
- **Fact 3:** NCB to prune the right branch of variable x happens only on deducing a conflict clause which does not contain any literal of x and whose unsatisfiability cube subsumes the subspace being pruned under the right branch of x.

Proposition 5.1 *If clause C is the resolvent produced by resolving clauses C_1 and C_2 in some common variable (say x) then $\mathcal{U}(C) \subseteq \mathcal{U}(C_1) \cup \mathcal{U}(C_2)$.*

Proof: Without loss of generality, let $C_1 = C_3 \vee x$ and $C_2 = C_4 \vee \overline{x}$, where C_3 and C_4 are some disjunctions of literals. Then $C = C_3 \vee C_4$, $\mathcal{U}(C_1) = \overline{x}\overline{C_3}$ and $\mathcal{U}(C_2) = x\overline{C_4}$. Thus $\mathcal{U}(C) = \overline{C_3} \cdot \overline{C_4} \subseteq (\overline{x}\overline{C_3}) \cup (x\overline{C_4})$. ■

[6] An assignment which sets one more literals in a clause to 0 is said to restrict that clause.

[7] The deduced conflict clauses may or may not be added to the CNF.

Theorem 5.2 *If the right branch of a variable x is eligible for pruning under NCB, then the subspace under the left branch of x (without loss of generality $x = 0$) cannot contain any x-essential points.*

Proof: From Fact 3, there must exist an implicate C, deduced through conflict analysis which does not contain literals x or $\overline{x}$ and which subsumes the subspace under the unexplored right branch, $x = 1$. Since C does not contain literals of x it must also subsume the sub-space under the left branch $x = 0$. Moreover, from Fact 2 there must exist clauses $C_1, C_2, \ldots C_k$ of the current CNF which form the leaves of the *resolution tree* simulating the deduction of C. From the recursive application of Proposition 5.1 it follows that $\mathcal{U}(C) \subseteq \mathcal{U}(C_1) \cup \mathcal{U}(C_2) \cup \ldots \cup \mathcal{U}(C_k)$. Thus, clauses $C_1, C_2, \ldots C_k$ collectively cover the subspace under the left branch of x. Also since the resolution could only be done on deduced variables clauses $C_1, C_2, \ldots C_k$ cannot have variable x. Therefore none of the points covered by them can be x-essential. ∎

5.3 Conflict Clause Recording

Conflict clause recording [6] is a powerful pruning technique that is employed in several successful SAT solvers [6, 7, 11]. The basic idea is to deduce an implicate (through conflict analysis) responsible for the current conflict and add it to the clause database with the aim of avoiding future occurances of the same conflict.

Although not apparent from the above statement of the notion, the recorded conflict clauses do in fact effect symmetry based pruning. Consider the following situation. In the left branch of variable x, say $x = 0$, a conflict $\mathcal{X}$ occurs on which a conflict clause $C_{\mathcal{R}}(\mathcal{X})$ is learned. Now, suppose $C_{\mathcal{R}}(\mathcal{X})$ does not contain literal x (it cannot contain $\overline{x}$). Let the set of assignments, preceding x be given by cube $\mathcal{A}$. Let $\mathcal{A}_1 = \mathcal{A} \wedge \overline{x} \wedge \mathcal{U}(C_{\mathcal{R}}(\mathcal{X}))$ and $\mathcal{A}_2 = \mathcal{A} \wedge x \wedge \mathcal{U}(C_{\mathcal{R}}(\mathcal{X}))$. Note that $\mathcal{A}_2$ is precisely the sub-space potentially prunable by $C_{\mathcal{R}}(\mathcal{X})$ in the right branch $x = 1$ of x.

As shown below, the pruning of sub-space $\mathcal{A}_2$ by clause $C_{\mathcal{R}}(\mathcal{X})$ can be accounted for by the theory of essential points. Due to space limitations we state the results without proof. The interested reader is referred to [8] for the proofs.

Theorem 5.3 *The symmetry based pruning afforded by a recorded conflict clause $C_{\mathcal{R}}(\mathcal{X})$ with respect to a variable x is subsumed by the pruning potentially realizable using essential point based pruning (Theorem 4.1).*

Interestingly, the entire pruning potentially accomplished by a recorded clause, subsequent to its recording, can be broken down into a series of right-branch prunings like the above situation[8].

[8] provided the search is organized as a single tree *i.e.* without restarts.

Theorem 5.4 *The search space pruning provided by a recorded clause $C_{\mathcal{R}}$, can be divided into a set of sub-spaces such that each sub-space lies under the right branch of a variable y, which does not appear in $C_{\mathcal{R}}$, where $C_{\mathcal{R}}$ was recorded in the left branch of y.*

From Theorems 5.3 and 5.4 it follows that conflict clause recording is a special case of essential point pruning.

6 Supercubing-Based Pruning

In this section we develop a simple new pruning rule based on exploiting problem symmetry. This rule is called the *supercubing rule* after the supercube operator defined below, which is the core operation used in implementing it.

Definition 6.1 *Supercubing Operator ($\mathcal{S}$): Given two cubes c_1 and c_2 over the 2^n Boolean space, $\mathcal{S}(c_1, c_2)$ computes the smallest cube containing both c_1 and c_2, i.e. the supercube of c_1 and c_2.*

6.1 Supercubing Procedure & Pruning

The algorithm maintains a cube called the *supercube* for each decision variable currently on the decision stack. The supercube of variable x (denoted $\mathcal{S}_x$) is initialized to $\emptyset$ when x is first chosen for branching. In the left branch of x (say $x = 0$) $\mathcal{S}_x$ is updated on each conflict $\mathcal{X}$ where $x \in \mathcal{A}_{\mathcal{R}}(\mathcal{X})$ ($\mathcal{A}_{\mathcal{R}}(\mathcal{X})$ is computed considering FDAs as decision variables) as follows:

$$\mathcal{S}_x = \mathcal{S}(\mathcal{S}_x, c_{\mathcal{S}}) \qquad \text{where} \quad c_{\mathcal{S}} = \bigwedge_{l \in \mathcal{A}_{\mathcal{R}}(\mathcal{X})} l \qquad (4)$$

After the algorithm has explored the left branch $x = 0$ and found no solution, it would have computed some supercube for x, denoted $\mathcal{S}_x^{final}$. Say $\mathcal{S}_x^{final} = \overline{x} \wedge l_1 \wedge l_2 \wedge \ldots \wedge l_k$ Then in the right branch, $x = 1$ we immediately assert $l_1 = \mathit{TRUE}, l_2 = \mathit{TRUE}, \ldots l_k = \mathit{TRUE}$ *i.e.* the region $x \wedge (\overline{l_1} \vee \overline{l_2} \vee \ldots \vee \overline{l_k})$ is pruned.

Note that the asserted assignments are treated as conscious assignments for the purpose of future conflict analysis and supercubing *i.e.* it is as though these variables $v_{l_1}, v_{l_2}, \ldots, v_{l_k}$ were consciously branched on and the branches $\overline{l_1}, \overline{l_2}, \ldots, \overline{l_k}$ were pruned, while the other branches were explored.

6.2 Proof of Correctness

The proof of correctness of the algorithm requires proving two propositions:

1. Every supercube-based pruning is legal, *i.e.* the pruned space cannot contain a solution.

2. At any point in the algorithm the following property hold for each point (minterm) in the Boolean sub-space that the algorithm has already explored (and found unsatisfiable).

Definition 6.2 *A point ψ satisfies* ***Property A*** *if there exist a cube c_ψ such that $\psi \subseteq c_\psi$ and c_ψ was processed by supercubing (Equation 4) under some previous conflict.*

Proof: The algorithm prunes off (explores) regions of the Boolean space through two kinds of *pruning events*, namely 1.) regular conflicts and 2.) supercube based pruning.

We prove the above two propositions simultaneously by induction on the sequence of pruning events. The overall idea is to prove that if all the points pruned by all previous pruning events satisfy Property A then :
a.) Points pruned by the current pruning event satisfy Property A, and b.) supercube-based pruning is legal.
Base Case : Since pruning occurs only in the right branch of a variable, the first pruning event must be a conflict and by definition, the algorithm would generate a conflict clause covering the pruned region and do supercubing on it. So all pruned points satisfy Property A.
Induction hypothesis : Suppose points pruned by the first k pruning events satisfy property A and are legal prunings.
Induction proof : Consider the $k + 1^{th}$ pruning event. If this is a regular conflict the proof trivially follows as per the base case. So consider the case when it is supercube based pruning performed in the right branch $x = 1$ of some variable x. The region pruned by supercubing $\mathcal{S}_x^{prune} = x \wedge \overline{\exists x \cdot \mathcal{S}_x^{final}}$ Consider any point $\psi^* \in \mathcal{S}_x^{prune}$ and point ψ, which is x-symmetric to ψ^*. Obviously ψ was examined by the algorithm in the left branch of x. Further, $\psi \notin \mathcal{S}_x^{final}$. Also, by the induction hypothesis there exists cube $c_\mathcal{S}$ such that $\psi \in c_\mathcal{S}$ and $c_\mathcal{S}$ was processed by supercubing. Thus, since $c_\mathcal{S} \not\subseteq \mathcal{S}_x^{final}$ cube $c_\mathcal{S}$ must not have variable x which means that it covers point ψ^* as well. Hence all points in $\mathcal{S}_x^{prune}$ are covered by conflict clauses that have already been discovered and processed by the algorithm. This also means that the current pruning is a legal one (since the pruned space is obviously unsatisfiable).

Note that in reality there is a third kind of pruning event, namely BCP deductions. However, the sub-space pruned by them is **completely** accounted for by the conflict clauses of the conflicts lying below this deduction. A simple way to prove this is to take the current branching tree and "push" all BCP deductions to the leaves of the tree *i.e.* after all the conscious assignments in each branch. Since in our procedure all conflict clauses are composed entirely of conscious assignments the same conflicts will still occur, but there will be no BCP-pruned areas this time. Here, the conflict clauses can be trivially seen to cover the entire pruned areas. Also we have not considered pure-literal rule based pruning in this proof since this rule is a special case of Supercubing (see Proposition 6.1). ■

Benchmark	Best Order # Nodes		Worst Order # Nodes	
	Orig.	With SC	Orig.	With SC
SSA-0432-003	1371	1050	3316	1074
SSA-2670-130	44039	38812	109766	66142
BF-0432-007	11487	10811	27298	9099
Queueinvar8	3211	2983	5842	5842
Aim-50-1_6-no-2	27	26	150	84
Aim-100-1_6-no-1	120	64	881	455
Aim-200-1_6-y-1-4	291	193	1155	354
Aim-200-1_6-no-3	457	559	6671	1252
Par-16-1-c	6543	6543	6543	6543
Hole 6	719	719	817	817

Table 1. Supercubing: Experimental results

6.3 Supercubing and Other Pruning Techniques

Proposition 6.1 *The pure-literal rule is a special case of supercubing based pruning.*

The reader is referred to [8] for the proof. The essential idea is that in some of the instances where a null supercube is computed for a decision variable x, supercubing based pruning of the right branch of x is synonymous with an application of the pure-literal rule on x. In other such cases the behavior of the algorithm is identical to NCB. Thus, supercubing overlaps with some instances of NCB. In fact, we conjecture that supercubing subsumes NCB. All our experiments thus far have not yielded a single case where NCB, implemented in the conventional fashion, could prune a sub-space that supercubing could not. However, the operational definition of NCB given in the literature is not precise enough to prove or challenge our conjecture. This could be an interesting problem for future research.

7 Experimental Results

This section presents preliminary experimental results validating the efficacy of the supercubing pruning rule. The pruning rule has been implemented in a prototype SAT solver modeled on the lines of the GRASP SAT solver [6]. The prototype solver implements all the algorithmic features of GRASP including conflict analysis, NCB, conflict based learning and various ordering heuristics. However, the solver has not yet been software engineered for efficiency since its purpose is simply to evaluate the first order efficacy of some pruning techniques. Therefore the reported results are in terms of number of nodes in the SAT search tree, rather than CPU runtimes since reporting the latter would be unfair and not particularly informative.

Preliminary results on selected SAT benchmarks from the DIMACS suite and bounded model checking [1] are reported in Table 1. The benchmark examples have been chosen to be representative of the examples that we ran, ranging from the ones where supercubing gave the maximum improvement to ones where it was not so effective.

For each benchmark the solver was run in two configurations with four possible orderings, DLCS, DLIS, MSTS, MSOS[9] (*i.e.* eight configurations) 1.) **ORIG:** without supercubing but with NCB and clause recording, and 2.) **With SC:** same as ORIG except supercubing is also used. For each benchmark the best and the worst ORIG results (in terms of number of nodes in the search tree) were chosen and are reported in columns 2 and 4 respectively. The corresponding results **with SC** (*i.e.* with the same ordering heuristic as the ORIG result) are reported in columns 3 and 5 respectively.

As shown in Table 1 the search tree size decreases in most cases, sometimes quite significantly. In the odd case (in our experience less than 1% of the cases) *e.g.* Aim-200-1_6-no-3 there is a slight increase. This is because supercubing disturbs the number of recorded clauses and hence the variable order slightly. However, overall supercubing proved beneficial for both the best order and the worst order. The improvements in the case of the worst ordering were more significant suggesting that this pruning technique can partially correct a poor ordering. The supercubing itself added virtually nothing to the runtimes since most of the book-keeping required for it was being done by conflict analysis. The additional supercubing operations were efficiently implemented by bit-vector operations. Thus gains in number of search tree nodes translate directly to runtime gains. Also, since supercubing based pruning partly overlaps with the pruning provided by conflict-based learning using supercubing frequently led to fewer recorded clauses. This feature of supercubing can be used to partly alleviate the clause database memory problems that are becoming an issue in current SAT solvers [7].

8 Conclusions & Future Directions

In this paper we have introduced and formalized the notion of *problem symmetry* in search-based SAT algorithms. We have developed the *theory of essential points* to formally characterize the potential search-space pruning that can be realized by exploiting problem symmetry. We have unified several powerful search pruning techniques used in modern SAT solvers under a single framework, by showing them to be special cases of the theory of essential points. We have also proposed a new pruning rule exploiting problem symmetry and shown it to provide additional search space pruning over the pruning realized by current techniques.

[9] Refer to the GRASP user manual for details on these heuristics.

Current SAT solvers integrate fairly sophisticated search pruning techniques in a very tightly and efficiently engineered software framework. However, there is very little fundamental understanding of how these techniques interact, what search space they prune and what the margin for improvement is. Our current work is a step towards answering these questions. We believe that it is possible to derive a whole family of search pruning techniques with varying cost-power tradeoffs, under the general purview of the theory of essential points. The supercubing rule presented in Section 6 is a simple case in point. It is quite obviously a very weak and cheap realization of essential point pruning. However, it still improves over the state-of-the-art, demonstrating the immense potential for improvement. Our current and future research efforts are aimed at realizing some of this potential.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of TACAS'99*, pages 193–207, March 1999.

[2] R. E. Bryant, S. German, and M. N. Velev. Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic. *ACM Trans. on Computational Logic*, 2(1):1–41, January 2001.

[3] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394–397, July 1962.

[4] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[5] L. A. Entrena and K.-T. Cheng. Combinational and Sequential Logic Optimization by Redundancy Addition and Removal. *IEEE Trans. on CAD*, 14(7):909–916, July 1995.

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

[7] M. H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the* 38^{th} *Design Automation Conference*, pages 530–535, June 2001.

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

[9] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational Test Generation Using Satisfiability. *IEEE Trans. on CAD*, 15(9):1167–1176, Sept. 1996.

[10] R. G. Wood and R. A. Rutenbar. FPGA Routing and Routability Estimation via Boolean Satisfiability. *IEEE Trans. on VLSI*, 6(2):222–231, June 1998.

[11] H. Zhang. SATO: An Efficient Propositional Prover. In *Proc. of the International Conference on Automated Deduction*, pages 272–275, July 1997.

[12] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. of the International Conference on Computer-Aided Design*, pages 279–285, Nov. 2001.