Tel Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
The Blavatnik School of Computer Science

Understanding and Improving a Modern SAT Solver

Alexander Nadel

Supervised by Professor Nachum Dershowitz

A Thesis Submitted for the Degree of Doctor of Philosophy

Submitted to the Senate of Tel Aviv University
August 2009

Contents

List of Figures

List of Tables

To my wife, Miriam

Acknowledgments

First, I would like to thank Prof. Nachum Dershowitz, my research advisor, for his infinite support throughout my graduate studies at Tel Aviv University. Nachum's wisdom and knowledge of the field inspired and guided me over the years. His reviews and suggestions are gratefully acknowledged. I also thank my fellow PhD student, Iddo Tzameret, for valuable advice in the area of his expertise – Proof Complexity. I would like to thank Prof. Orna Grumberg and the anonymous reviewers for reading the thesis and providing useful comments, which helped me to improve the thesis considerably.

During the course of this research, I had the great opportunity to be employed part time in the Formal Technology and Logic Group at Intel, Haifa. I would like to thank all my colleagues at Intel: Ziyad Hanna, Amit Palti, Zurab Khasidashvili, Moran Gordon, Baruch Sterin, Vadim Ryvchin, Yulik Feldman, Jacob Katz, Daher Kaiss, Ranan Fraer, Michael Lifshits, Orly Cohen, Dmitry Korchemny and others, with whom I shared many discussions and thoughts. I am especially grateful to my previous and current managers at Intel – Ziyad Hanna and Amit Palti, whose constant support has been an invaluable contribution to my work, which combines academic research and industrial experience. I would like to thank Zurab Khasidashvili for sharing with me his experience and understanding of the field.

Finally, I would like to thank my parents Lion and Elianora for their commitment to giving me the best education. I am deeply grateful to my children Michael, Nicole and Shani for all the great time we spend together. I am glad to dedicate this thesis to my wife, Miriam, for her sincere love.

Abstract

Propositional satisfiability (SAT) is an NP-complete problem, holding a central place in computer science and engineering. SAT has numerous applications in formal verification, artificial intelligence and other areas. Modern SAT solvers, using an enhanced version of the backtrack search Davis-Logemann-Loveland (DLL) algorithm, are able to successfully cope with instances comprising millions of variables. This work is an attempt to shed new light on the functionality of a modern SAT solver. We also propose a number of enhancements that are empirically useful, especially in the formal verification domain.

We propose a framework for presenting and analyzing a modern DLL-based SAT solver. We provide a basic backtracking algorithm that explicitly shows the process of resolution refutation construction. Our approach is based on the notion of a parent resolution derivation – a resolution proof for validness of a flip operation. We show how to derive the algorithm of a modern SAT solver from basic backtracking step-by-step.

This resolution-based approach allows us to define new criteria for measuring the practical impact of different schemes for conflict-driven learning by making the notion of search pruning more formal. We show that the 1UIP scheme, enhanced by conflict clause minimization, is better than other known schemes in terms of pruning. This explains its empirical advantage over other schemes.

We propose an enhancement to the minimized 1UIP scheme, called local conflict clause recording. This technique improves the performance of a modern SAT solver by recording additional conflict clauses. Local conflict clause recording makes the learning less dependent on the variable polarity selection heuristic.

Assignment stack shrinking is a technique whose goal is to shrink the size of the assignment stack and conflict clauses. We demonstrate the empirical usefulness of assignment stack shrinking and analyze its impact on the

performance of a modern SAT solver, comparing it to the impact of conflict clause minimization and rapid restarts.

Furthermore, a new decision heuristic for SAT, called the clause-based heuristic, is introduced. This heuristic is designed to increase the likelihood that interrelated variables will be chosen in proximity. It maintains a clause list containing both the initial and conflict clauses. The next decision literal is picked from the first unsatisfied clause. We propose various methods for initially organizing the clause list and for moving clauses within it. Our approach results in a significant performance boost over existing heuristics tested on hard real-world industrial benchmarks.

Finally, we present an algorithm for minimal unsatisfiable core extraction that is able to find a minimal unsatisfiable core for large real-world formulas. Benchmark families, arising in formal verification of hardware, are of particular interest to us. Modern SAT solvers are able to produce a resolution refutation of a given unsatisfiable formula, whose sources are the input clauses and whose sink is the empty clause. Our method's basic version removes the input clauses connected to the empty clause one by one from the resolution refutation, preserving the validity of the refutation by adding other clauses and resolution relations until no more input clauses can be removed. In the end, all the input clauses, connected to the empty clause, comprise the minimal unsatisfiable core.

Chapter 1

Introduction

Propositional satisfiability (SAT) is the problem of determining, for a formula in propositional calculus, whether a satisfying assignment for its variables exists. SAT holds a central place in the large family of NP-complete problems (see, e.g., [12], [21]). Therefore, it is unlikely that there is an algorithm that can solve it in reasonable time in all cases. Nonetheless, algorithms exist that are capable of quickly solving many instances resulting from real-world problems. SAT has numerous applications in formal verification (e.g., [54]), as well as in artificial intelligence (e.g., [34]) and many other fields of computer science and engineering.

The basic backtracking algorithm [15] is commonly understood as a search-based algorithm, which checks whether a satisfying assignment for the input formula, provided in Conjunctive Normal Form (CNF), exists. The algorithm works by exploring the assignment space in a depth-first search manner. It maintains a partial assignment and extends it by assigning previously unassigned decision variables until a certain clause is falsified. In this case, the algorithm backtracks and flips the last assigned decision variable. If all the assigned variables have already been flipped, the formula is unsatisfiable. If a model for the input formula is found, the formula is satisfiable.

The plain backtrack algorithm is unable to solve large real-world instances, but it can be enhanced by various algorithms, crucial for practical efficiency.

Modern SAT solvers spend 80–90% of their runtime performing Boolean Constraint Propagation (BCP). BCP, already suggested in the original paper on solving SAT with backtrack search [15], forces the assignment of values to variables appearing in unit clauses (clauses having one unassigned literal). These choices are picked as "decisions" whenever possible. BCP is used by modern SAT solvers to quickly identify (failure) leaves of the search tree, referred to as conflicts. Implication relations between assigned literals can be represented using the so-called "implication graph". Efficient data structures for BCP were proposed in [45, 11].

The paper [15] proposed another modification to the plain backtrack search algorithm, called the pure literal elimination rule. A pure literal is a literal that appears in only one polarity, that is, only positively or negatively, in clauses that are not yet satisfied. The pure literal elimination rule removes all the clauses containing such a literal. However, this modification is not used in modern SAT solvers, due to the high overhead required for detecting pure literals.

One key step for making SAT efficient in practice was the introduction of conflict-driven learning (CDL) by the authors of the GRASP [60] and rel_sat [3] SAT solvers. Conflict-driven learning applies a number of learning and pruning algorithms when a conflict is identified. The origin of conflict-driven learning goes back to the work that was done on the constraint satisfaction problem (CSP) [55].

Modern SAT solvers, such as Minisat [19], Eureka [48], RSAT [53], and PicoSAT [6], inherit their CDL algorithm from the 2001 version of the Chaff SAT solver [45], in which the CDL scheme of rel_sat and GRASP was further improved. Chaff's scheme applies 1UIP-based conflict-directed backjumping, non-chronological backtracking and 1UIP conflict clause recording [45]. In the literature on practical design of SAT solvers, all these techniques and BCP are presented and analyzed together, based on implication graph analysis [60, 45, 27, 56, 69, 62].

Although Chaff's CDL scheme is widely used, it is not fully understood. Compare, for example, the statement by the authors of Chaff, provided in the context of comparison between different CDL schemes: "[T]he effectiveness

of certain searching schemes can only be determined by empirical data" [69].

The Chaff algorithm can be viewed as an algorithm for assignment space or search-tree exploration, yet it can also be seen as an algorithm that constructs a resolution refutation of a given formula. The latter approach to understanding a modern SAT solver is well-known [37, 23, 22, 50], but not usually used in the literature dedicated to practical aspects of SAT solving. From our perspective, the main reason for the lack of clarity was the fact that the Chaff algorithm has not been formulated in a way that shows both processes of search-tree exploration and resolution refutation construction.

Chapter 2 of this work provides a framework for presenting and analyzing the functionality of a modern SAT solver. We provide an implementation of the basic backtrack search algorithm and show how to integrate other algorithms implemented in Chaff into our framework step-by-step. Our algorithm explicitly demonstrates the construction of a resolution refutation for an unsatisfiable formula. It does not force the solver to use BCP. We do not use the notion of implication graph in our analysis, but define all the algorithms related to conflict analysis based on resolution. Our approach associates each flipped variable with a parent resolution derivation – a resolution proof for validness of a flip operation.

A resolution-based approach to understanding the conflict-driven learning algorithm of a SAT solver was used in [50]. The primary goal of the paper [50] was to provide a formalism for a SAT solver in a way that allows one to easily integrate a SAT solver as a DPLL(T) engine into a Satisfiability Modulo Theories (SMT) solver. In contrast to our framework, oriented towards core SAT solving, the paper [50] did not provide an explicit algorithm for a modern SAT solver.

Chapter 3 uses the framework of Chapter 2 to understand and enhance the conflict-driven learning algorithm of a modern SAT solver.

Section 3.1 shows how to integrate various conflict-driven learning techniques, including the UIP-n scheme, proposed in this work, the *All*UIP [69] scheme and conflict clause minimization algorithm [4, 62], into our framework.

Section 3.2 reviews an implication-based approach to conflict-driven

learning.

Section 3.3 makes the commonly used notion of search pruning [41, 58] more formal. We distinguish between backward pruning and forward pruning. Backward pruning is carried out when the algorithm is backtracking. It is characterized by the number of nodes in such parent resolution derivations of unassigned flipped variables, which were not required for deriving a parent resolution derivation for the new flip. Forward pruning relates to the impact of recorded conflict clauses on the subsequent search. We define a new measure for forward pruning, called pre-flip learning uselessness. The idea is to find, for every flipped variable, in what fraction of conflict clauses it participated before the flip. The larger this fraction, the less helpful are the conflict clauses, recorded before the flip, for pruning the search space after the flip.

Section 3.4 demonstrates that the minimized 1UIP scheme, that is the 1UIP scheme of [45], enhanced by conflict clause minimization [4, 62], is superior to other schemes in terms of both backward and forward pruning. This explains the empirical advantage of the 1UIP scheme over other schemes.

Section 3.5 introduces an enhancement to the minimized 1UIP scheme, called local conflict clause recording. This technique records additional conflict clauses, whenever certain conditions hold. The idea behind local conflict clause recording is improving the pruning by making the conflict clause recording less dependent on the heuristic for choosing the polarity of assigned variables. We demonstrate the practical usefulness of local conflict clause recording on industrial benchmarks. Sections 3.3 and 3.5 are based on our paper [18].

Section 3.6 is dedicated to assignment stack shrinking, a technique, proposed by the author of this work in [47], and further enhanced by the authors of the 2004 version of Chaff [40]. This technique tries to dynamically reduce the size of conflict clauses and to unassign irrelevant literals from the assignment stack. If certain conditions hold for a newly learned conflict clause, shrinking unassigns some of the literals of the conflict clause and reassigns them to 0. BCP follows each assignment. Section 3.6 reaffirms the empirical usefulness of assignment stack shrinking and shows that it cannot

be simulated or subsumed by conflict clause minimization or rapid restarts, disproving a supposition of [6].

A crucial factor influencing the performance of a SAT solver is its decision heuristic. The heuristic decides which variable to choose at each decision point during the search and what value to assign it first. Modern decision heuristics are dynamic – that is, they refocus the search on recently derived conflict clauses. VSIDS [45] – the first such dynamic heuristic – maintains a score for each literal. The score is increased when the literal appears in a conflict clause; once in a while, scores are halved. Another well-known decision heuristic, which proved to be even more successful than VSIDS on industrial benchmarks, is that of Berkmin [27]. The Berkmin heuristic [27] is more dynamic than VSIDS. It organizes all conflict clauses in a list and picks the next decision literal from the topmost unsatisfied clause in the list. If no such clause exists, a secondary VSIDS-like choice-heuristic is used.

In Chapter 4, we introduce a new decision heuristic that has been found to be efficient on hard real-world industrial benchmarks. The Berkmin heuristic is indeed more dynamic than VSIDS, but we claim another advantage for the Berkmin heuristic over VSIDS in that it tends to pick interrelated variables, that is, variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch and satisfying and removing "problematic" clauses in satisfiable branches. However, this potential advantage is diluted by the fact that the Berkmin heuristic does not put the initial clauses in the clause list and applies a secondary VSIDS-like heuristic. Our proposal, which we call the clause-based heuristic (CBH), maintains a clause list containing both the initial and the conflict clauses, thus increasing the chances of picking interrelated variables. The next decision literal is picked from the topmost unsatisfied clause. No secondary heuristic is required. Also, whenever a new conflict clause is derived, CBH moves clauses that participated in the resolution derivation of the new conflict clause to the top of the list. In addition, we propose various methods for initially organizing the clause list. Our approach results in a significant performance boost over both VSIDS and the Berkmin heuristic.

The idea of moving the clauses used for a new conflict clause derivation

towards the head of the list was proposed independently of our work in the papers [25, 26] and implemented in the HaifaSat solver. The HaifaSat heuristic is called Clause-Move-To-Front (CMTF). Its usefulness is justified in the framework of an abstraction/refinement model. In contrast to our approach, CMTF maintains only the conflict clauses in the list, hence it should tend to pick less interrelated variables than CBH.

Chapter 4 is based upon our paper [16].

When a formula is unsatisfiable, it is often required to find an unsatisfiable core – that is, a small unsatisfiable subset of the formula's clauses. Example applications include functional verification of hardware [43], field-programmable gate array routing [49], and abstraction refinement [42]. An unsatisfiable core is a minimal, if it becomes satisfiable whenever any of its clauses is removed. It is always desirable to find a minimal unsatisfiable core, but this problem is very hard. (It is D^P-complete; see [52].)

Chapter 5, based upon our paper [17], presents an algorithm that is able to find a minimal unsatisfiable core for large real-world formulas. The only approach for unsatisfiable core extraction that scales well for formal verification benchmarks was independently proposed in [70] and in [28]. We refer to this method as the empty-clause cone (EC) algorithm. EC exploits the ability of modern SAT solvers to produce a resolution refutation, given an unsatisfiable formula. EC takes initial clauses, connected to the empty clause $\square$, as the unsatisfiable core. Invoking EC until a fixed point is reached [70] allows one to reduce the unsatisfiable core even more. However, the resulting cores can be further reduced. The basic flow of the algorithm for minimal unsatisfiable core extraction proposed in Chapter 5 is composed of the following steps. First, produce a resolution refutation of a given formula using a SAT solver. Second, drop from the resolution refutation all clauses not connected to $\square$. At this point, all the initial clauses, connected to $\square$, comprise an unsatsifiable core. Third, try to remove each remaining clause C from the unsatisfiable core by invoking the SAT solver on the resolution refutation, excluding the cone of C. The algorithm terminates when all the initial clauses remaining in the resolution refutation comprise a minimal unsatisfiable core.

The author of this thesis is the main author of the papers [16, 17, 18].

In addition, he participated in works on simultaneous satisfiability in model checking [35] and on bitvector satisfiability [10], not reported herein.

Chapter 2

Understanding a Modern SAT Solver

In this chapter we propose a framework for presenting and understanding the functionality of modern SAT solvers.

Section 2.1 introduces the SAT Solver Skeleton (SSS) in Algorithm 1 – a formulation of the backtrack search algorithm, where the resolution refutation construction is shown explicitly. We provide a correctness proof of the algorithm. Section 2.2 shows how to introduce techniques, which enhance modern SAT solvers, into SSS. These techniques include Boolean Constraint Propagation (BCP), 1UIP-based conflict-directed backjumping, non-chronological backtracking, conflict clause recording, restarts and conflict clause deletion. Each above-mentioned enhancement can be added to our algorithm independently of the other. In particular, using BCP is not a must. This is in contrast to the standard approach to describing the conflict-driven learning engine of a SAT solver (provided in Section 3.2), where 1UIP-based CDB, NCB and 1UIP-based CCR are described together, based on implication graph-analysis which is dependent on BCP.

2.1 SAT Solver Skeleton

We start with basic definitions, related to propositional logic.

Definition 1 (Variable; Literal)**.** *We denote* (propositional) variables *by lowercase Latin letters. A* literal *is a variable v or its negation $\neg v$. The Boolean values are denoted 1 and 0. For variable v and Boolean value κ, v^κ is the corresponding literal; that is, $v^1 = v$ and $v^0 = \neg v$.*

Definition 2 (Clause; Empty clause; CNF formula)**.** *A* clause *is a disjunction (or set) of literals. The empty clause is denoted by $\square$. A* Conjunctive Normal Form (CNF) formula *is a conjunction of clauses $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ or equivalently, a set $\{C_1, \ldots, C_m\}$.*

Definition 3 (Assignment; Complete assignment; Partial assignment)**.** *An* assignment *(or* partial assignment*) σ assigns Boolean values to all (or some) of the variables in a set of formulas. An assignment that assigns values to all the variables is called* complete*. The literal v^1 is assigned 1 or 0 by σ, iff the variable v is assigned 1 or 0 by σ. The literal v^0 is assigned 1 or 0 by σ, iff the variable v is assigned 0 or 1 by σ.*

Definition 4 (Satisfied clause; Falsified clause)**.** *Suppose that σ is an assignment. Then, a clause C is* satisfied *by σ, if one of the literals of C is assigned 1 by σ. A clause C is* falsified *by σ, if all the literals of C are assigned 0 by σ.*

Definition 5 (Satisfied CNF formula; Falsified CNF formula)**.** *A CNF formula F is* satisfied *by σ, if all the clauses of F are satisfied by σ; a CNF formula F is falsified by σ, if one of the clauses of F is falsified by σ.*

Definition 6 (Model)**.** *An assignment σ is a model to CNF formula F if F is satisfied by σ.*

Definition 7 (Satisfiable CNF formula; Unsatisfiable CNF formula)**.** *A CNF formula F is* satisfiable *iff there exists a model to F. Otherwise, the formula F is* unsatisfiable*.*

Resolution is a widely studied simple proof system that can be used to prove the unsatisfiability of CNF formulas. We now provide a number of definitions, related to resolution.

Definition 8 (Resolution rule; Resolvent; Pivot variable)**.** *The* resolution rule *states that given clauses $D_1 = A \vee v$ and $D_2 = B \vee \neg v$, where A and B are also clauses, we can derive the clause $C = A \vee B$ by resolving on v. The clause C is called a* resolvent *of clauses D_1 and D_2 on* pivot variable *v. The resolution rule application is denoted by $C = D_1 \otimes^v D_2$.*

Definition 9 (Resolution derivation; Size of resolution derivation; Target clause)**.** *A* resolution derivation *of a* target clause *C from a CNF formula F is a sequence $\pi = \{C_1, C_2, \ldots, C_p\}$, where $C_p \equiv C$ and each clause C_i is either a clause of F (an initial clause) or derived by applying the resolution rule to C_j and C_k, where $j, k < i$ (a derived clause). The* size *of π is p, the number of clauses occurring in it. The target clause of a resolution derivation π is denoted by π^T.*

Two resolution derivations from F, whose target clauses are resolvable, can be composed to obtain a new resolution derivation of F. In the following definition, we assume that resolution derivations can also be considered sets of clauses, hence the set difference operation $\setminus$ is well defined for resolution derivations.

Definition 10 (Composition of Resolution Derivations)**.** *Let π and ρ be two resolution derivations from F, such that their target clauses π^T and ρ^T are resolvable on v. Then, the following sequence of clauses is a composition of π and ρ: $\tau = \pi, \rho \setminus \pi, \pi^T \otimes^v \rho^T$. We denote $\tau = \pi \otimes^v \rho$.*

It is not hard to check that a composition of two resolution derivations, whose target clauses are resolvable, is a resolution derivation.

Proposition 1 (Composition of Resolution Derivations' Correctness)**.** *Let π and ρ be two resolution derivations from F, such that the clauses π^T and ρ^T are resolvable on v. Then, $\tau = \pi \otimes^v \rho$ is a resolution derivation from F.*

Proof. Consider first all the clauses of τ, except the target clause. Each such clause C belongs to either π or ρ. Hence, C is either an initial clause or is derived from previous clauses in either π or ρ. However, τ contains all the

clauses of π or ρ maintaining the order by construction. Thus, C is either an initial clause or is derived from previous clauses in τ.

The target clause τ^T is derived from two previous clauses of τ by construction.

Hence, τ is a resolution derivation from F. □

For an example of a resolution composition, consider $F = \{a \vee b, a \vee \neg b, \neg a \vee b, \neg a \vee \neg b\}$; $\pi = \{a \vee b, a \vee \neg b, a\}$; $\rho = \{\neg a \vee b, \neg a \vee \neg b, \neg a\}$. The target clauses $\pi^T = a$ and $\rho^T = \neg a$ are resolvable on a. Hence, the composition of π and ρ is $\tau = \pi^T \otimes^a \rho^T = \{a \vee b, a \vee \neg b, a, \neg a \vee b, \neg a \vee \neg b, \neg a, \square\}$. The target clause of τ is the empty clause, hence τ is a refutation of F in the sense provided in the next definition.

Definition 11 (Resolution refutation; Refutation)**.** *Any resolution derivation of the empty clause* □ *from* F *is called a* resolution refutation *or simply a* refutation *of* F*.*

The following proposition is well-known.

Proposition 2 (Soundness and Completeness of Resolution)**.** *A formula* F *is unsatisfiable iff it has a refutation.*

A resolution derivation can be conveniently represented by a rooted binary directed acyclic graph (dag). Vertices of the dag correspond to clauses of the derivation. Leaves of the dag are clauses in F. The root contains the target clause. Internal nodes correspond to resolution rule applications. An internal node contains the resolvent clause of its two children. Each edge is marked with a literal comprising the negation of the pivot variable appearance in the clause in its head. More specifically, an edge from $A \vee v$ to $C = (A \vee v) \otimes^v (B \vee \neg v)$ is marked with $\neg v$ and an edge from $B \vee \neg v$ to C is marked with v.

An example of a refutation, of size 8, appears in Fig. 2.3(a) on page 23.

Now we present an implementation of the backtrack search algorithm, Algorithm 1, which we refer to as the *SAT Solver Skeleton (SSS)*. SSS uses

two subroutines Flip and AnalyzeConfBtAndFlip, depicted in Algorithms 2 and 3, respectively.

First, we provide an informal description of the flow of SSS, including its subroutines. The algorithm checks whether the input formula F is satisfiable or unsatisfiable. In the former case, the algorithm returns a model to the formula and, in the latter case, it returns a resolution refutation of the formula. The algorithm works by exploring the assignment space in a depth-first search manner. It maintains a partial assignment and extends it by assigning previously unassigned variables until a certain clause of F is falsified. In this case, the algorithm identifies the last assigned variable that should be flipped in order to satisfy the falsified clause and flips its value. (If all the assigned variables have already been flipped, the formula is unsatisfiable.) The flip ensures that the algorithm will explore previously unexplored subspaces. Each flip operation is associated with the so-called parent resolution derivation, whose target clause is called a parent clause. A parent clause constitutes an implication of the flip from a subset of previously assigned literals. The parent resolution derivation shows how to derive the parent clause from the formula, thus providing a proof that there are indeed no satisfying assignments in the subspace explored by the algorithm and left with the flip. After the flip, the algorithm may again find a falsified clause, in which case it would build a parent resolution derivation for the upcoming flip; then backtrack and flip. This process continues until either of the following two event occurs:

1. All the clauses are satisfied by the current assignment, in which case the formula is satisfiable and the assignment is the model, returned by the algorithm.

2. The algorithm encounteres a falsified clause and all the relevant assigned variables have already been flipped. In this case, the algorithm returns a resolution refutation of the formula, generated while checking that none of the variables can be flipped.

We now describe the flow of SSS, including its subroutines, in more detail. An explanation of all the symbols used by the algorithm and its subroutines is summarized in Fig. 2.1. We will use the notions of the main loop, conflict analysis loop and backtracking loop in our analysis. The position of these loops in the code of our algorithms is provided in Fig. 2.2. We assume that all the variables and data structures are defined in the global context.

The algorithm starts by initializing the assignment level s to 0 at line 1. The assignment level is defined as follows:

Definition 12 (Assignment level)**.** *The* assignment level s*, maintained by Algorithm 1, is the current depth of the backtrack search.*

Definition 13 (Assigned variable; Assigned literal)**.** *A variable v is* assigned *at level i, if $1 \leq i \leq s$ (s is the assignment level) and $v_i \equiv v$. A literal v^κ is assigned at level i, if v is assigned at level i and $\sigma_i = \kappa$.*

The main loop of the search starts at line 2. Each iteration of the main loop starts by increasing the assignment level s and assigning an unassigned variable v_s a Boolean value σ_s. Then, the algorithm records that the current assignment level is non-flipped.

Definition 14 (Flipped assignment level; Non-flipped assignment level)**.** *Let s be the current assignment level. Then, each assignment level i, $1 \leq i \leq s$, is either* flipped *or* non-flipped*. The flip status is maintained in the array FlipStatus by Algorithm 1.*

Definition 15 (Flipped variable; Non-flipped variable; Flipped literal; Non-flipped literal)**.** *A variable/literal, assigned at assignment level s, is* flipped/non-flipped*, if s is flipped/non-flipped.*

We will see that assignment level s would become flipped if the algorithm concluded that none of the complete assignments that are consistent with $\sigma_{1\ldots s}$ constitutes a model of F; however it could not conclude that there are no models under $\sigma_{1\ldots s-1}$. In this case, the value of s must be flipped and the resulting subspace must be checked for models.

1. s: the current assignment level, the current depth of the search.
2. v_i: for each i, $1 \leq i \leq s$: v_i denotes the variable, assigned at assignment level i.
3. σ_i: for each i, $1 \leq i \leq s$: σ_i denotes the Boolean value, assigned to v_i.
4. **ChooseNewLiteral**: a function that returns a pair $\langle v_s, \sigma_s \rangle$ consisting of an unassigned variable v_s and a Boolean value σ_s, which can be either 1 or 0.
5. **FlipStatus**: an array, indexed by the assignment level i, $1 \leq i \leq s$, specifying if the variable v_i, assigned at assignment level i, was flipped. FlipStatus[i] can either be true or false.
6. $\sigma_{1 \ldots s}$: The partial assignment to variables, assigned between assignment levels $1 \ldots s$.
7. π_i: The parent resolution derivation corresponding to the flipped assignment level i (see Definition 17).

Figure 2.1: Explanation of symbols, used by Algorithm 1 and its subroutines

1. **Main loop**: a loop, starting at line 2 of SSS (Algorithm 1)
2. **Conflict analysis loop**: a while loop, starting at line 10 of SSS (Algorithm 1)
3. **Backtracking loop**: a while loop, starting at line 2 of AnalyzeConfBtAndFlip (Algorithm 3)

Figure 2.2: The names of loops, used while analyzing the functionality of Algorithm 1 and its subroutines

Algorithm 1 SAT Solver Skeleton or SSS (CNF formula $F := \{C_1, C_2, \ldots, C_m\}$)

1: $s := 0$
2: **loop**
3: $s := s + 1$
4: $\langle v_s, \sigma_s \rangle := \mathit{ChooseNewLiteral}()$
5: $\mathit{FlipStatus}[s] := false$
6: **if** F is satisfied by $\sigma_{1\ldots s}$ **then**
7: **return** F is satisfied by $\sigma_{1\ldots s}$
8: **if** $\exists C_l \in F : C_l$ is falsified by $\sigma_{1\ldots s}$ **then**
9: Flip($\{C_l\}$)
10: **while** $\exists C_r \in F : C_r$ is falsified by $\sigma_{1\ldots s}$ **do**
11: $\rho :=$ AnalyzeConfBtAndFlip(C_r)
12: **if** $s = 0$ **then**
13: **return** F is unsatisfiable with refutation ρ

Algorithm 2 Flip (Resolution derivation ρ)

1: $\pi_s := \rho$
2: $\sigma_s := \neg\sigma_s$
3: $\mathit{FlipStatus}[s] := true$

Algorithm 3 AnalyzeConfBtAndFlip (Clause C_r)

1: $\rho := \{C_r\}$
2: **while** $s > 0$ and ($\mathit{FlipStatus}[s] = true$ or $v_s^{\neg\sigma_s} \notin \rho^T$) **do**
3: **if** $v_s^{\neg\sigma_s} \in \rho^T$ **then**
4: $\rho := \pi_s \otimes^{v_s} \rho$
5: $s := s - 1$
6: **if** $s \neq 0$ **then**
7: Flip(ρ)
8: **return** ρ

Line 6 checks if the formula is satisfied with the current assignment, in which case the algorithm returns. Otherwise, the algorithm checks if there exists a clause that is falsified by the current assignment (line 8). If none of the clauses is falsified, the main loop continues. If one of the clauses C_l is falsified, we say that a conflict takes place.

Definition 16 (Conflict; Blocking clause)**.** *We say that a conflict takes place if there exists a clause $C \in F$, falsified by $\sigma_{1\dots s}$. The clause C is called a* blocking clause*.*

If a conflict is detected by the condition of line 8, the algorithm flips the value of the assigned variable v_s by invoking the function Flip and providing it with a resolution derivation, consisting of the single blocking clause. The function Flip receives as input a resolution derivation, which serves as the parent resolution derivation of the assignment level s after the flip.

Definition 17 (Parent resolution derivation; Parent clause)**.** *A resolution derivation of ρ^T from F ρ is a* parent resolution derivation *for a flipped assignment level s, if $\rho^T = \neg A \vee v_s^{\sigma_s}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \dots s-1$. The target clause of the parent resolution derivation is called a* parent clause*.*

The parent clause can be understood as the reason for the flip: an implication of the flip of v_s from a conjunction of a subset of previous assignments $\neg A \vee v_s^{\sigma_s} \equiv A \rightarrow v_s^{\sigma_s}$. The goals of conflict analyses and backtracking are to find a non-flipped assignment level to which to backtrack and to build a resolution derivation that can serve as the parent resolution derivation after the flip. We will see that the parent invariant provided below holds. Intuitively, the invariant ensures that each flip is legitimate.

Invariant 1 (Parent invariant)**.** *For each flipped assignment level i, $1 \leq i \leq s$, π_s is a parent resolution derivation.*

Now, we return to the flow of the algorithm, at line 9, which invokes the function Flip, provided in Algorithm 2. The function Flip records the parent

resolution derivation of s; flips the Boolean value of v_s and marks the assignment level as flipped. If no conflict follows the flip, a new decision is required and the algorithm returns to the main loop. Otherwise, the algorithm enters the conflict analysis loop, starting at line 10. The algorithm exits the loop when a new decision is required or the formula is proved to be unsatisfiable. The conflict analysis loop invokes the function AnalyzeConfBtAndFlip, implemented in Algorithm 3. AnalyzeConfBtAndFlip either backtracks to an assignment level that should be flipped and flips it, or backtracks to assignment level 0 if the formula is unsatisfiable. In the process, it builds a resolution derivation that serves either as a parent resolution derivation for the newly flipped assignment level or as a refutation of the formula.

The resolution derivation, maintained by the function AnalyzeConfBtAndFlip, is called the backtracking resolution derivation.

Definition 18 (Backtracking resolution derivation; Backtracking clause)**.** *Assume s is the current assignment level. A resolution derivation ρ is a* backtracking resolution derivation*, if it either holds that:*

1. *$s > 0$ and $\rho^T = \neg A \vee v_s^{\neg\sigma_s}$, or*
2. *$s \geq 0$ and $\rho^T = \neg A$ (ρ^T is the empty clause $\square$, if A is empty).*

In both cases, A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \ldots s-1$ (A must be empty, if $s \leq 1$). The target clause of the backtracking resolution derivation is called the backtracking clause*.*

We will prove later that ρ, maintained by AnalyzeConfBtAndFlip, is indeed a backtracking resolution derivation. More specifically, we will prove that the following invariant holds in the beginning of each iteration of the backtracking loop.

Invariant 2 (Backtracking invariant)**.** *The sequence of clauses ρ, maintained by AnalyzeConfBtAndFlip, is a backtracking resolution derivation.*

The main component of function AnalyzeConfBtAndFlip is the backtracking loop, starting at line 2. The backtracking loop halts when either:

(1) the backtracking level s is a non-flipped level and the backtracking clause contains the negation of the literal, assigned at this level; or (2) the assignment level becomes 0. In the former case, it follows from the definitions of the backtracking clause and the parent clause that the backtracking clause can serve as the parent clause for the newly flipped assignment level for the flip that occurs at line 7. Lemma 2 shows that in the latter case, the backtracking resolution derivation becomes the refutation of F. Hence the parent invariant holds after the algorithm exits the function AnalyzeConfBtAndFlip and returns to line 12 of SSS. SSS checks if the formula is unsatisfiable, and continues to check the condition of the conflict analysis loop. If the condition holds, the algorithm continues with another iteration of the conflict analysis loop; otherwise, it returns to the main loop.

Now we provide a correctness proof for our algorithm. The flow of the algorithm will be studied and analyzed in detail during the proof. We will also provide two examples, demonstrating the functionality of AnalyzeConfBtAndFlip, after formulating and proving the two lemmas related to AnalyzeConfBtAndFlip.

We start off with a lemma claiming the consistency of the backtracking loop.

Lemma 1 (Backtracking loop consistency)**.** *Suppose the backtracking invariant holds just before the algorithm checks the condition of the backtracking loop (line 2 of Algorithm 3), then one of the following three post-conditions holds:*

1. *The algorithm enters the backtracking loop. It will reach line 2 of Algorithm 3 – that is, the condition of the backtracking loop once more after the current iteration is completed. The backtracking invariant will hold at this point.*

2. *The algorithm does not enter the backtracking loop; $s = 0$ and ρ is a refutation of F.*

3. *The algorithm does not enter the backtracking loop; $s \neq 0$; s is a non-flipped assignment level; ρ is a backtracking resolution derivation and*

$v_s^{\neg\sigma_s} \in \rho^T$.

Proof. We distinguish between the following five cases, one of which must hold when the algorithm checks the condition of the backtracking loop (line 2 of Algorithm 3)). We denote the values of s and ρ in the beginning of the iteration as simply s and ρ; and at the end of the iteration as s' and ρ', respectively.

1. The assignment level s is 0. In this case, the backtracking loop condition does not hold. The backtracking clause must be the empty clause $\square$ by the backtracking invariant. Thus, ρ is a refutation of F. Hence, the algorithm exits the loop when post-condition 2 holds.

2. The assignment level $s > 0$ is non-flipped and the negation of the literal assigned at s belongs to the backtracking clause. In this case, the algorithm exits the loop, when post-condition 3 holds by construction. In this case, the algorithm found a variable to flip and constructed the parent resolution derivation for the flip.

3. The assignment level $s > 0$ is flipped and the negation of the literal assigned at s belongs to the backtracking clause. In this case, the algorithm enters the loop. The backtracking loop resolves the parent derivation of s π_s with the backtracking derivation ρ and updates the backtracking derivation with the result. To verify that $\pi_s \otimes^{v_s} \rho$ is a valid composition of resolutions, we need to check that π_s^T and ρ^T are resolvable. By the parent invariant, it holds that $\pi_s^T = \neg A \vee v_s^{\sigma_s}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \dots s-1$. By the backtracking invariant and our assumption that the negation of the literal assigned at s belongs to the backtracking clause, $\rho^T = \neg B \vee v_s^{\neg\sigma_s}$, where B is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \dots s-1$. Hence, π_s^T and ρ^T are resolvable on v_s; thus $\pi_s \otimes^{v_s} \rho$ is a resolution derivation. Moreover, after the algorithm decrements s and reaches the loop condition once again, ρ' is a backtracking resolution derivation, since its target clause is composed of a conjunction of a subset of zero or

more literals, assigned at assignment levels $1 \ldots s' - 1$ and, optionally, $v_{s'}^{\neg\sigma_{s'}}$. Therefore, post-condition 1 holds.

4. The assignment level s is non-flipped and the negation of the literal assigned at s does not belong to the backtracking clause. In this case, the algorithm enters the loop, but does not change the backtracking resolution derivation. The backtracking invariant still holds after decrementing the assignment level and reaching the loop condition once again, since the backtracking clause is still composed of a negation of a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \ldots s' - 1$ and, optionally, $v_{s'}^{\neg\sigma_{s'}}$. Therefore, post-condition 1 holds. The behavior of our algorithm in this case shows the difference between our resolution-aware algorithm and the original backtracking algorithm DLL [15], which flips every non-flipped variable.

5. The assignment level s is flipped and the negation of the literal assigned at s does not belong to the backtracking clause. Post-condition 1 holds for exactly the same reasons that it held for the previous case (when s was non-flipped). It is interesting, however, that in our case the parent resolution derivation of s is not included in the newly created parent resolution derivation. We say that resolution backward pruning takes place in this case. Resolution backward pruning corresponds to one of the three cases of backward pruning, which will be analyzed in Section 3.3. We relate search pruning to the algorithm's ability to reduce the number of nodes in the final resolution refutation of the formula. In our case, the parent resolution derivation of v_s is not included in the derivation of the new backtracking clause; thus it will not be included in the parent resolution derivation of the newly flipped variable, which in turn means that it will not be included in the final resolution refutation of the formula. We will encounter the other two types of backward pruning when discussing non-chronological backtracking and 1UIP-based conflict-directed backjumping.

□

Lemma 2 (Backtracking consistency)**.** *Suppose that when the function AnalyzeConfBtAndFlip is invoked, the following pre-conditions hold:*

1. *The assignment level s is flipped, $s > 0$ and $C_r = \neg A \vee v_s^{\neg\sigma_s}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \ldots s-1$.*

2. *The parent invariant (invariant 1) holds.*

Denote the values of s and ρ, when AnalyzeConfBtAndFlip finishes by s' and ρ'. AnalyzeConfBtAndFlip exits when either of the following post-conditions hold:

1. *The assignment level s' is 0 and ρ' is a resolution refutation of F.*

2. *The assignment level s' is flipped; $s' < s$; the parent invariant holds.*

Proof. First, we show that the backtracking invariant holds, when the condition of the backtracking loop is reached for the first time. At this point ρ is a backtracking resolution refutation, consisting of the single clause C_r, serving as the target clause. By pre-condition 1 of the lemma, $C_r = \neg A \vee v_s^{\neg\sigma_s}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \ldots s-1$, and $s > 0$. This condition is sufficient for ensuring that C_r is a backtracking clause and ρ is a backtracking resolution derivation. Hence, the backtracking invariant holds.

Observe that the loop must terminate, since s is decreased at each iteration, and the condition $s > 0$ is a terminating condition for the backtracking loop. It also holds that $s' < s$, since the condition of the backtracking loop always holds for the first iteration, hence the assignment level is decreased at least once. The subsequent iterations of the backtracking loop may only decrease the assignment level.

From an iterative application of post-condition 1 of Lemma 1, the backtracking invariant must hold each time before the algorithm checks the condition of the backtracking loop (line 2), including the last iteration. At the

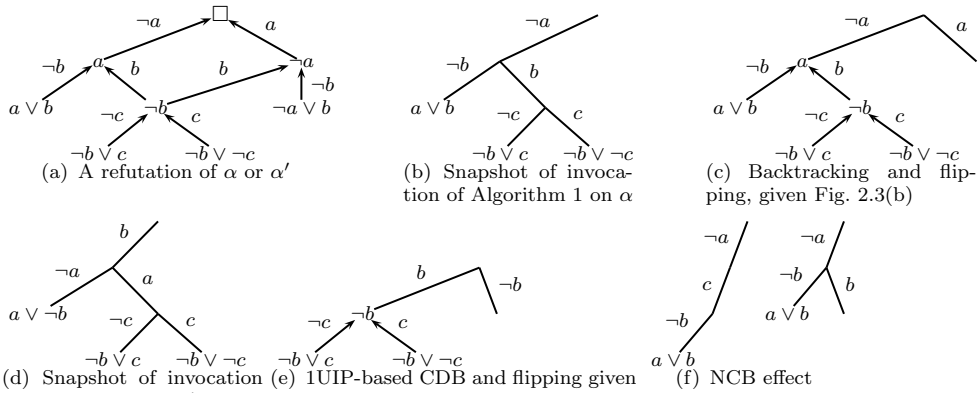
last iteration, the condition of the loop does not hold, hence one of post-conditions 2 or 3 of Lemma 1 must hold. We show that in either case, one of the post-conditions of our lemma holds.

If the algorithm exits the loop, when $s' = 0$ and ρ' is a resolution refutation, then the function AnalyzeConfBtAndFlip exits, returning a refutation, and post-condition 1 of our lemma holds.

Suppose that the algorithm exits the backtracking loop, when $s' \neq 0$, s' is a non-flipped assignment level, ρ' is a backtracking resolution derivation and $v_{s'}^{\neg\sigma_{s'}} \in \rho'^T$. Then, the function AnalyzeConfBtAndFlip flips the value of $v_{s'}$. In this case, the backtracking resolution derivation ρ' can serve as a parent resolution derivation for the new flip. Previously, we proved that ρ is a resolution refutation. The backtracking clause ρ'^T fulfills the requirements for serving the parent clause for the following reason: we assumed that $v_{s'}^{\neg\sigma_{s'}} \in \rho'^T$; the rest of the literals of ρ'^T must be $\neg A$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \ldots s' - 1$ by the definition of a backtracking clause. Note also that the algorithm did not modify the parent resolution derivations of the assignment levels lower than s'. Hence, the parent invariant holds, when the function AnalyzeConfBtAndFlip exits. We have already shown that $s' < s$ and we have demonstrated that s' is a flipped level. Thus, post-condition 2 holds. □

Now we demonstrate the functionality of function AnalyzeConfBtAndFlip on two examples.

First, consider Fig. 2.3(b), which shows a snapshot of an invocation of SSS on input formula α just after the third conflict. The rightmost path comprises the current partial assignment, that is $v_1^{\sigma_1} = \neg a; v_2^{\sigma_2} = b; v_3^{\sigma_3} = c$. Assignment level 1 is non-flipped and assignment levels 2 and 3 are flipped. The parent resolution derivations of levels 2 and 3 are $\{a \vee b\}$ and $\{\neg b \vee c\}$, respectively. The function AnalyzeConfBtAndFlip is provided with the blocking clause $\neg b \vee \neg c$ as a parameter. The algorithm initializes the backtracking resolution derivation with the blocking clause. The first iteration of the backtracking loop starts at assignment level 3. The literal $\neg c$ appears in the backtracking clause; thus the algorithm updates the backtracking resolution derivation.



(a) A refutation of α or α'

(b) Snapshot of invocation of Algorithm 1 on α

(c) Backtracking and flipping, given Fig. 2.3(b)

(d) Snapshot of invocation of Algorithm 1 on α'

(e) 1UIP-based CDB and flipping given Fig. 2.3(b); backtracking and flipping given Fig. 2.3(d)

(f) NCB effect

Figure 2.3: Examples of search trees, resolution refutations and the impact of various algorithms, given the formulas $\alpha = (a \vee b) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee b)$ and $\alpha' = \alpha \wedge (a \vee \neg b)$

The new backtracking clause is $\neg b = \neg b \vee c \otimes^{c} \neg b \vee \neg c$. The algorithm backtracks to assignment level 2 and enters another iteration of the backtracking loop. The literal $\neg b$ appears in the backtracking clause; hence the backtracking resolution derivation is updated. The new backtracking clause is $a = a \vee b \otimes^{b} \neg b$. The algorithm backtracks to the non-flipped assignment level 1. The negation of the assigned literal $\neg a$ appears in the backtracking clause. Hence, the backtracking loop terminates. The algorithm flips the value of $\neg a$ using the newly derived backtracking resolution derivation as the parent resolution derivation. The situation at this point is shown in Fig. 2.3(c). The current assignment level is 1 and the only assigned variable is a. The bottom-non-flipped part of the figure, which included nodes with clauses and arrowed edges, represents the parent resolution derivation, created by the backtracking loop. This parent resolution derivation can be represented non-graphically as $\{\neg b \vee c, \neg b \vee \neg c, \neg b, a \vee b, a\}$.

Consider now Fig. 2.3(d), representing another snapshot of an invocation of SSS after the third conflict. The current assignment level is 3. In the first iteration of the backtracking loop, the backtracking resolution derivation is updated. The new backtracking clause is $\neg b = \neg b \vee c \otimes^{c} \neg b \vee \neg c$. The flipped assignment variable a does not appear in the backtracking clause,

hence backtracking continues and the backtracking resolution derivation is not updated. The backtracking stops at the non-flipped assignment level 1 when the backtracking clause is $\neg b$. The situation that results after the flip appears in Fig. 2.3(e). The bottom-non-flipped part of the figure represents the parent resolution derivation of $\neg b$. Note that the parent resolution of $\neg a$, which consists of the single clause $a \vee \neg b$, does not appear in the new parent resolution derivation.

Next, we provide a proof that, given a CNF formula, Algorithm 1 terminates and returns a resolution refutation if the input formula is unsatisfiable, or a model if the formula is satisfiable. We need to formulate an assignment invariant that states that none of the clauses of F is falsified just before choosing new assignment literal.

Invariant 3 (Assignment invariant)**.** *None of the clauses of the input formula F is falsified with $\sigma_{1\ldots s}$ before invoking line 4 of SSS (Algorithm 1).*

Now we introduce the termination function for SSS. We will prove termination of SSS by demonstrating that the finite termination function must increase.

Definition 19 (Termination function)**.** *Let p be the number of variables in the input CNF formula F. The* termination function*, which can be calculated at each point of SSS execution, is a pair of integer numbers $\langle t, s\rangle$. The second component of the pair s is the assignment level. The first component of the pair t is determined as follows. Suppose bit number 0 is the least significant bit. Then, bit number i of t is 1 iff $i \leq s$ and i is a flipped assignment level, that is $FlipStatus[i] = true$. Bit number 0 of t and bits, whose number is greater than s, are set to 0.*

Proposition 3. *The termination function is finite.*

Proof. For termination function $\langle t, s\rangle$, the assignment level s is bound by the number of variables. Only bits 1 to s of t, inclusively, can be assigned 1, hence t is a finite number as well. □

Definition 20 (Comparison of termination functions)**.** *Let $f_1 = \langle t_1, s_1\rangle$ and $f_2 \langle t_2, s_2\rangle$ be two termination functions. Then,*

- $f_1 > f_2$ *iff* $t_1 > t_2$ *or* $t_1 = t_2$ *and* $s_1 > s_2$.
- $f_2 > f_1$ *iff* $t_2 > t_1$ *or* $t_2 = t_1$ *and* $s_2 > s_1$.
- $f_1 = f_2$ *iff* $f_1 \not> f_2$ *and* $f_1 \not< f_2$.

Lemma 3 (Correctness and termination)**.** *Suppose that SSS (Algorithm 1) is situated just before choosing the new assigned literal at line 4. Suppose that the assignment and the parent invariants hold and that the termination function is* f_1*. Then, one of following post-condition holds:*

1. *The algorithm will reach line 4 once again. The assignment and the parent invariants will still hold. The new termination function* f_2 *will be strictly greater than* f_1*.*
2. *The algorithm will return that the formula is satisfiable with the model* $\sigma_{1...s}$*. In this case,* $\sigma_{1...s}$ *indeed satisfies* F*.*
3. *The algorithm will return that the formula is unsatisfiable with the resolution refutation* ρ*. In this case,* ρ *is indeed a resolution refutation of* F*.*

Proof. We distinguish between the following events:

1. The algorithm chooses the new literal and then exits immediately, since $\sigma_{1...m}$ satisfies F. In this case, post-condition 2 of our lemma holds. The assignment $\sigma_{1...m}$ is a model by construction.
2. The algorithm chooses a new literal and does not encounter a model; the condition for the conflict of line 8 does not hold. In this case, the algorithm continues with another iteration of the main loop and reaches line 4 once again. The assignment invariant still holds, since otherwise the condition of line 8 for the conflict would have held. The parent invariant still holds, since the flipped levels remained as they were; the structure of their parent resolution derivations also did not change. Finally, the new termination function is greater than the previous one, since the first component of the pair did not change, but the second one (the assignment level) increased by 1.

3. The algorithm chooses a new literal and does not encounter a model; the condition for the conflict of line 8 holds; there is no conflict after the flip of line 9. In this case, the algorithm continues to another iteration of the main loop and reaches line 4 once again. The assignment invariant still holds, since otherwise the condition of line 10 for the conflict would have held. Now we show that the parent invariant still holds. The only new flipped assignment level is associated with a parent resolution derivation, consisting of the single clause C_l that used to be the blocking clause of the conflict. We know that C_l was falsified by $\sigma_{1\ldots s}$ before the flip. We also know that the assignment invariant held before the flip; hence the assignment at level s before the flip was necessary to falsify the clause C_l. This means that C_l consists of a negation of a conjunction of a subset of 0 or more literals assigned at levels $1 \ldots s-1$ and $v_s^{\sigma'_s}$, where σ' is the partial assignment after the flip. Hence, $\{C_l\}$ is a parent resolution derivation and the parent invariant holds. Finally, the termination function increased, since the first component of the pair was increased due to the last flip.

4. The algorithm chooses a new literal and does not encounter a model; the condition for the conflict of line 8 holds and there is a conflict after the flip of line 9, discovered by the conflict loop condition (line 10). In this case, the algorithm enters the conflict analysis loop. Note that when analyzing the previous case, we showed that the parent invariant still holds after the flip and that the termination function increased. Consider now the first iteration of the conflict analysis loop. As usual, we denote the values of s and ρ, when AnalyzeConfBtAndFlip finishes, by s' and ρ'. By Lemma 2, the AnalyzeConfBtAndFlip exits when either $s' = 0$ and ρ' is a resolution refutation of F, in which case post-condition 3 of our lemma holds, or when the assignment level s' is flipped; $s' < s$ and the parent invariant holds. Assume the latter. In this case the algorithm reaches the condition of the conflict analysis loop once again. If it holds, then from repeated application of Lemma 2 it follows that whenever (and if) the condition of the con-

flict analysis loop is reached once again, the parent invariant holds. Note that by Lemma 2, the condition of the conflict analysis loop may be reached a limited number of times, since each application of AnalyzeConfBtAndFlip decreases the assignment level by 1, and SSS returns unsatisfiable, if the assignment level is decreased to 0. Suppose we are at the stage when the condition of the conflict analysis loop does not hold and the algorithm exits the loop. Then, the algorithm proceeds to another iteration of the main loop. The assignment invariant must hold at this point, since otherwise the conflict analysis loop would not have terminated. As we have shown, the parent invariant must also hold; the new termination function f_2 must be greater than f_1, since the first component of the pair increased due to the last flip.

□

Theorem 1 (Correctness and termination)**.** *Given a satisfiable formula F, SSS (Algorithm 1) will return that the formula is satisfiable with the model $\sigma_{1...s}$. In this case, $\sigma_{1...s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS will return that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. Note that the first time SSS reaches line 4, both the assignment and the parent invariants trivially hold. Hence, both pre-conditions for Lemma 3 are fulfilled and we can apply the lemma iteratively until one of the post-conditions 2 or 3 of Lemma 3 hold. Indeed, whenever post-condition 1 of the lemma holds, we can use it as a pre-condition for applying the lemma once again. Note that each application of the lemma that satisfies post-condition 1 increases the finite termination function. Hence, in the end, one of the post-conditions 2 or 3 must hold. Therefore, the algorithm will terminate returning either a model, if F is satisfiable, or a refutation, if F is unsatisfiable. The algorithm cannot return a refutation for a satisfiable formula due to the resolution's soundness. It cannot return a model for an unsatisfiable formula, since the algorithm explicitly verifies that the returned assignment is a model at line 6. □

2.2 From the SAT Solver Skeleton to a Modern SAT Solver

In this section, we describe a number of widely used techniques that enhance the basic backtracking algorithm SSS. These techniques include:

- Boolean Constraint Propagation (BCP) [15]
- Non-Chronological Backtracking (NCB) [3, 60]
- 1UIP-based Conflict-Direct Backjumping (CDB) [3, 60, 45]
- Conflict Clause Recording (CCR) [63, 60, 45]
- Restarts [29]
- Conflict Clause Deletion [3]

We show how to augment Algorithm 1 (SSS) with each one for the above-mentioned algorithms and discuss them in the light of our understanding of the SAT solver functionality. Our pseudo-code allows one to choose the decision heuristic, the restart strategy, the clause deletion strategy and the data structures. Integrating all the algorithms into SSS results in a generalization of the algorithm, implemented in the Chaff-2001 SAT solver [45]. Fixing the decision heuristic, the restart strategy, the clause deletion strategy and the data structures to the ones proposed in [45] would make our algorithm identical to Chaff-2001.

This section uses a number of concepts, well-known in the literature, but not yet addressed in our work. We start with a number of definitions.

Definition 21 (Decision)**.** *The operation of choosing a new literal at line 4 of SSS (Algorithm 1) is called a* decision*.*

Each assigned variable is associated not only with the assignment level, but also with a decision level.

Definition 22 (Decision level)**.** *Each variable v, assigned at assignment level i, is associated with a decision level d, equal to the number of non-flipped assignment levels between 1 and i, inclusively. The decision level of an assigned literal v^κ is defined to be the decision level of the variable v.*

For example, the decision level of all the assigned literals appearing on Fig. 2.3(b) on page 23 is equal to 1, since the first assignment level is the only non-flipped assignment level.

Definition 23 (Decision variable; Decision literal)**.** *A variable (literal), assigned at assignment level i, is a* decision variable (literal) *iff i is a non-flipped assignment level.*

Proposition 4 (Decision Variable Consistency)**.** *Each decision level has only one decision variable/literal associated with it.*

Proof. Follows from Definitions 22 and 23. □

The decision variable of decision level 1 on Fig. 2.3(b) is a, whereas its decision literal is $\neg a$.

Definition 24 (Current decision level)**.** *Each stage of the SSS (Algorithm 1) invocation is associated with the* current decision level*: the number of assigned decision variables.*

It is sometimes convenient to reason about the resolution derivation that would turn the parent resolution derivation after the flip.

Definition 25 (Asserting resolution derivation; Asserting clause)**.** *The resolution derivation, supported as a parameter to the function Flip, is called an* asserting resolution derivation*. The target clause of the asserting resolution derivation is called the* asserting clause*.*

Proposition 5 (Asserting Clause Consistency)**.** *The asserting clause must be of the form $\rho^T = \neg A \vee v_s^{\neg\sigma_s}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \ldots s-1$.*

Proof. One can show that the parent invariant must hold after each flip by applying arguments similar to those appearing in the proof of Lemma 3. (We skip the proof, since it would be very similar to that of Lemma 3.) The algorithm is going to use the asserting resolution derivation as the parent resolution derivation after the flip. The value of v_s will be flipped to $\neg\sigma_s$. Denote $\tau_s = \neg\sigma_s$. By the parent invariant, the parent clause must be of the form $\neg A \vee v_s^{\tau_s}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \dots s-1$. The asserting clause is the parent clause before the flip, that is before negating σ_s; hence $\rho^T = \neg A \vee v_s^{\neg\sigma_s}$. $\square$

We provide an intuitive notion of the n^{th} highest assignment level.

Definition 26 (n^{th} highest assignment level)**.** *Assume* $C = v_{i_1}^{\sigma_{i_1}} \vee v_{i_2}^{\sigma_{i_2}} \vee \dots \vee v_{i_k}^{\sigma_{i_k}}$ *be a clause, containing 1 or more literals, assigned by SSS. Suppose that for each* $j : 1 < j \leq k$*, the assignment level of* $v_{i_j}^{\sigma^{i_j}}$ *is greater than that of* $v_{i_{j-1}}^{\sigma^{i_{j-1}}}$*. Then, the* highest assignment level *of* C *is* i_k*. The* n^{th} highest assignment level *of* C *is* i_{k-n+1} *if* $k > (n-1)$ *and 0 otherwise.*

Definition 27 (Asserting literal; Failure-driven assertion)**.** *The literal with the highest assignment level in an asserting clause is called the* asserting literal*. The operation of flipping an asserting literal is called a* failure-driven assertion*.*

Each remaining section of this chapter is dedicated to an algorithmic enhancement of SSS. The correctness proofs show how to update the proof of Theorem 1, which depends on the proofs of Lemmas 1, 2 and 3, when a specific additional algorithm is applied. We show how to update the proofs of relevant lemmas.

2.2.1 Boolean Constraint Propagation (BCP)

Boolean Constraint Propagation (BCP) is the process of repeatedly employing the unit clause rule, proposed in [15], until a fixed-point is reached. First, we define concepts, related to BCP, using our terminology.

Definition 28 (Unit clause)**.** *Suppose that SSS (Algorithm 1) is situated just before making a new decision at line 4. A clause C is a* unit clause *if $C = \neg A \vee v^{\kappa}$, where A is a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \dots s-1$ and v^{κ} is an unassigned literal.*

Now we define the unit clause rule.

Definition 29 (Unit clause rule)**.** *Suppose that SSS (Algorithm 1) is situated just before making a new decision at line 4. If there exists a unit clause $C = \neg A \vee v^{\kappa}$, the algorithm must assign the variable v the value $\neg\kappa$. In this case, one says that the* unit clause rule *was applied in unit clause C. If there are a number of unit clauses, the algorithm chooses one of them.*

Boolean Constraint Propagation (BCP) forces the algorithm to use unit clause rule, whenever possible. To implement BCP within SSS, do the following:

BCP (invoked instead of line 4 of Algorithm 1):
if $\exists C \in F : C = \neg A \vee v^{\kappa}$ is a unit clause **then**
  $\langle v_s, \sigma_s \rangle := \langle v, \neg\kappa \rangle$
else
  $\langle v_s, \sigma_s \rangle := \mathit{ChooseNewLiteral}()$

Our definition of the unit clause rule is different from the original one [15] in that we enforce the choice of the negation of the unassigned literal, appearing in the unit clause, rather than the literal itself. This approach allows one to describe the algorithm implemented in modern SAT solvers in our terminology without referring to implications and implication graphs, thus detaching conflict analysis from BCP. In our framework, a conflict always follows a unit clause rule application; hence the algorithm is forced to flip the value of v_s immediately. Suppose that the unit clause rule is applied to the unit clause $C = \neg A \vee v^{\kappa}$. The literal assigned as a result of the unit clause rule application and flipping is an implied literal in the standard terminology of [60]. In our framework, implied literals are treated as regular flipped literals.

The use of data structure enabling one to implement BCP efficiently is crucial for a SAT solver. The most popular data structure was introduced in

Chaff-2001 [45] and is called the *Two Watched Literals (2WL)* data structure. The idea is that it is sufficient to maintain pointers to only two of the literals of each clause to support BCP. The predecessor of 2WL is SATO's SAT solver Head/Tail [68]. Recent improvements to 2WL can be found in [11].

A interesting research direction that could follow from our approach is to try applying BCP selectively, while all the conflict analysis techniques, such as non-chronological backtracking, 1UIP-based conflict clause recording and others, can still always be applied. It is widely accepted that BCP helps accelerate modern SAT solvers, though it typically consumes 80–90% of a solver's run-time [45]. The added value of BCP is that it allows the solver to quickly propagate information and find conflicts. However, the solver must visit additional clauses in the large clause base of modern SAT solvers, hence exessive BCP application might result in a high cache miss rate. We suppose that it could be advantageous not to apply BCP at least in some cases. To strenthen this hypothesis, we demonstrate below that, in some cases, most of the propagations carried out by BCP are not relevant for the proof. Note that it is sufficient to maintain only one watched literal per clause, where the algorithm is allowed to skip propagation.

Proposition 6. *There is a formula whose shortest resolution refutation by SSS with BCP is linearly longer than in SSS without BCP.*

Proof. Consider a formula consisting of (1) eight clauses, each of size 3, corresponding to all possible disjunctions between literals of variables a, b, c, excluding tautologies, and (2) the following set of k clauses for each literal $p \in D = \{a, b, c, \neg a, \neg b, \neg c\}$: $C^p = (p \vee l_1^p) \wedge (\neg l_1^p \vee l_2^p) \wedge (\neg l_2^p \vee l_3^p) \wedge \ldots \wedge (\neg l_{k-1}^p \vee l_k^p)$. The variables $L^p = \{l_1^p \ldots l_k^p\}$ are fresh variables for each of D's literals.

Clearly, there exists an invocation of SSS generating a refutation of size 7, which ignores clause set (2). BCP, however, forces k additional, useless inferences. More specifically, if p is the first literal of D that is assigned, then all the literals of $L^{\neg p}$ are assigned either before p or as a result of BCP, after p's assignment.

The size of the refutation generated by an invocation of SSS with BCP on this example is $\Omega(3+6k)$, compared to a constant size refutation for plain

SSS. □

As BCP is only a special decision strategy for SSS, the algorithm is still correct.

Theorem 2 (Correctness and termination of SSS with BCP)**.** *Given a satisfiable formula F, SSS with BCP will return that the formula is satisfiable with the model $\sigma_{1...s}$. In this case, $\sigma_{1...s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with BCP will return that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. BCP can be simulated by implementing the function ChooseNewLiteral, so that it would choose literals whose negation appears in unit clauses. The correctness and termination of plain SSS does not depend on the choices made by ChooseNewLiteral; hence SSS with BCP still terminates with a correct result.

□

2.2.2 Non-Chronological Backtracking (NCB)

Non-chronological backtracking (NCB) [3, 60] is a technique, applied before each flip, which tries to find and eliminate unnecessary assignments. The notion of NCB was proposed in [60]. It is also related to the ideas of applying conflict-directed backjumping in constraint satisfaction problem (CSP) [55] to SAT, discussed in [3].

Suppose that SSS is about to flip a certain variable v_s at assignment level s, after building the asserting resolution derivation ρ (the beginning of function Flip in Algorithm 2). Non-chronological backtracking removes assignment levels between the highest assignment level and the second highest assignment level of the asserting clause, non-inclusively, before each flip (if possible). After the above-described operation, the NCB implementation of Chaff also increases s to the closest non-flipped assignment level, that is to the assignment level of the closest decision variable. This step is carried out so as not to redo BCP.

To implement NCB within SSS, do the following:

Non-Chronological Backtracking (NCB) (invoked just before starting the function Flip (Algorithm 2)):
$h :=$ Second highest assignment level in ρ^T
$t :=$ First non-flipped assignment level greater than h
if $t < s$ **then**
  $v_t := v_s$
  $\sigma_t := \sigma_s$
  $s := t$

Note that after applying non-chronological backtracking and flipping, the variable v_s is assigned immediately after all the variables of a certain decision level d; and the decision level of v_s after the flipping becomes d. This decision level is called the backtrack level.

Definition 30 (Backtrack level)**.** *The decision level of a flipped variable after applying non-chronological backtracking and flipping is called the* backtrack level.

Fig. 2.3(f) on page 23 shows the effect of NCB. A snapshot of an SSS invocation after the first conflict is depicted on the left-hand side. The asserting clause for the first flip is $a \vee b$. The algorithm identifies the fact that the assignment level 2 can be deleted. The clause ρ^T still remains an asserting clause. The algorithm deletes assignment level 2 before the flip. The situation that results appears on the right-hand side of Fig. 2.3(f).

As we will see in more detail in Section 3.3, the NCB algorithm induces the second type of backward search pruning, which we call NCB backward pruning. If a flipped assignment level is removed by NCB, the algorithm also "forgets" its parent resolution derivation. Thus, such parent resolution derivations will not be part of the final resolution refutation of the given formula. NCB backward pruning does not occur in the example in Fig. 2.3(f), since the algorithm does not delete flipped assignment levels.

The correctness NCB follows from the fact that after applying NCB, ρ^T is still an asserting clause that becomes a parent clause after the flip.

Theorem 3 (Correctness and termination of SSS with NCB)**.** *Given a satisfiable formula F, SSS with NCB will return that the formula is satisfiable with the model $\sigma_{1\dots s}$. In this case, $\sigma_{1\dots s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with NCB will return that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. After applying NCB, ρ^T is still an asserting clause that would become a parent clause after the flip, since it is still composed of the literal, assigned at the current assignment level s and the negation of a conjunction of a subset of zero or more literals, assigned at assignment levels $1 \dots s-1$. Hence, the parent invariant is not violated by NCB; thus the proof of correctness of SSS, provided in Theorem 1, is not affected by applying NCB and our theorem holds.

□

2.2.3 1UIP-based Conflict-Directed Backjumping (CDB)

The idea of conflict-directed backjumping (CDB) was proposed for the context of SAT in [60, 3] and can be traced back to the work on CSP [55]. Using unique implication points during conflict analysis was proposed in [60]. Backjumping, whenever a unique implication point is discovered, was proposed in Chaff-2001 [45]. We now provide the Chaff-2001 algorithm.

A unique implication point (UIP) [60] is a well-known concept, whose name is rooted in the implication-based approach to conflict analysis. We express this notion in our framework.

Definition 31 (n^{th} unique implication point)**.** *Suppose that SSS is backtracking over a flipped assignment level s in function AnalyzeConfBtAndFlip (Algorithm 3) visiting line 4, when the condition $v_s^{\neg\sigma_s} \in \rho^T$ holds. Then, the variable v_s is a* Unique Implication Point (UIP) *if it is the only variable of ρ_T, assigned at the current decision level. Backtracking may find more than one UIP. UIPs are counted in the order in which they appear during the*

backtracking phase starting with 1. In addition, the decision variable of each decision level is considered to be the last UIP of that decision level.

Let g be the current decision level. Let t be the assignment level of the decision variable of g. The idea behind 1UIP-based conflict-directed backjumping [45] is as follows: once the first UIP variable v_s is discovered during backtracking, continue as if v_s was a decision variable, assigned instead of v_t, whose parent clause is the current backtracking clause. One way to think about 1UIP-based CDB is as substituting the decision v_t by v_s a-posteriori. This is implemented as follows:

1UIP-based CDB (invoked just before line 4 of Algorithm 3):
if there exist non-flipped assignment levels **then**
  $g :=$ The decision level of v_s
  if $v_s^{\neg\sigma_s}$ is the only literal in ρ^T with the decision level g **then**
    $t :=$ The assignment level of the decision variable of g
    $v_t := v_s$
    $\sigma_t := \sigma_s$
    $FlipStatus[t] := false$
    $s := t$
    Continue to the condition of the while loop

See the transformation of Fig. 2.3(b) into Fig. 2.3(e) on page 23 for an example of the effect of 1UIP-based CDB. After the algorithm derives a new backtracking clause $\neg b$ during backtracking, it discovers that it contains only one variable, b, assigned at the last (and only) decision level. Therefore, it deletes the decision a, and simulates a situation, identical to the one that would have been created if the decision b had been taken instead of the decision $\neg a$. This situation is shown in Fig. 2.3(e). The parent clause and parent resolution derivation of the first assignment level are updated to the backtracking clause and its derivation.

1UIP-based CDB induces the third type of backward pruning, analyzed in Section 3.3, which we call UIP backward pruning. Consider a flipped assignment level p that is situated between the assignment level of the last decision and the assignment level of the first UIP variable, inclusively. The

parent resolution derivation corresponding to p is not included in the newly derived parent resolution; thus it will not be included in the final resolution refutation. We say that it is pruned. In our example, the parent resolution derivation corresponding to the second assignment level that consists of a single clause $a \vee b$ is pruned.

The correctness of 1UIP-based CDB follows from the fact that applying the 1UIP-based CDB does not break the backtracking invariant; hence Lemma 1 still holds.

Theorem 4 (Correctness and termination of SSS with 1UIP-based CDB)**.** *Given a satisfiable formula F, SSS with 1UIP-based CDB will return that the formula is satisfiable with the model $\sigma_{1\ldots s}$. In this case, $\sigma_{1\ldots s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with 1UIP-based CDB will return the fact that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. If the first UIP is the decision variable, then the 1UIP-based CDB algorithm, described above, is not invoked at all and the correctness of our theorem trivially follows from Theorem 1.

Suppose that the first UIP is a flipped variable. First, we show that Lemma 1 still holds after updating the backtracking loop with the 1UIP-based CDB algorithm. Suppose that the algorithm enters the iteration of the backtracking loop, where 1UIP-based CDB is applied. The last line of 1UIP-based CDB application states that the algorithm continues to the condition of the while loop. After applying 1UIP-based CDB, the backtracking invariant continues to hold, since the backtracking clause is still composed of the literal assigned at the updated assignment level s' and the negation of a conjunction of a subset of zero or more literals assigned at assignment levels $1 \ldots s' - 1$. Hence, post-condition 1 of Lemma 1 holds.

The proof of Lemma 2 uses the fact that AnalyzeConfBtAndFlip must decrease the assignment level, hence it is also important to show that the assignment level is still decreased by the iteration of AnalyzeConfBtAndFlip that uses 1UIP-based CDB, even though line 5 of AnalyzeConfBtAndFlip, which explicitly decreases the assignment level, is not reached by such it-

eration. This statement is correct, since the assignment level is changed to the assignment level of the last decision, which must be lower than the assignment level of the first UIP by construction.

Other parts of the proof of the correctness and termination of plain SSS, provided in Theorem 1, remain unaffected; hence our theorem holds.

□

We underscore the fact that we do not consider 1UIP-based conflict clause recording in this section, but only 1UIP-based CDB. In our analysis, these two concepts are not necessarily related.

2.2.4 Conflict Clause Recording

Conflict clause recording (CCR) is a powerful algorithm that is used by modern SAT solvers to prune the search space. Conflict clause recording grew out of work in AI on explanation-based learning [63]. It was proposed to be employed in modern SAT solvers in [3, 60]. In our terminology, CCR is an enhancement to SSS, allowing the algorithm to use some or all of the derived clauses for conflict identification and propagation.

Definition 32 (Conflict clause)**.** *A* conflict clause *is a clause, derived from the initial formula F by resolution during SSS invocation.*

It has been shown that CCR as practiced in today's SAT solvers, assuming unlimited restarts (restarts are addressed in Section 2.2.5), corresponds to a proof system exponentially more powerful than that of plain backtracking [4] in a sense defined precisely in [13]. Conflict clause recording can be as powerful as general resolution, while backtracking has been known to correspond to the exponentially weaker tree-like resolution. This is in contrast to BCP, NCB and 1UIP-based CDB, which can be understood as pruning techniques or heuristics, but they do not add inference power to the basic algorithm, which remains as powerful as tree-like resolution. We do not analyze the inference power of algorithms for SAT in this work, concentrating on more practical aspects of SAT solving. The interested reader is referred to [4] for more details.

The most popular scheme for CCR records only the clauses that served as parent clauses for conflict identification and propagation.

Definition 33 (Parent-based conflict clause recording)**.** Parent-based conflict clause recording *is a conflict clause recording scheme that records only the clauses that served as parent clauses for conflict identification and propagation.*

Basically, parent-based conflict clause recording was first employed in Chaff-2001 [45], which used 1UIP-based CDB and recorded only the parent clause for each flip, called the 1UIP conflict clause. Chaff's scheme for conflict clause recording is called *1UIP-based conflict clause recording*. In the literature on practical SAT solver design, CCR is defined via implication graph terminology: when a conflict occurs, a clause, corresponding to a cut in the implication graph, is added to the formula. We provide a simpler and a more flexible definition of CCR. This approach is not new, as it already appears in the literature [50], but it is usually not used in the area of practical SAT solver design.

To implement conflict clause recording, a set of currently used conflict clauses L must be maintained. The algorithm should use $F \cup L$ for conflict identification and propagation. To implement parent-based conflict clause recoding within SSS, the following steps must be carried out:

1. Add the following line just after the line 1 of Algorithm 1:

   $L := \{\}$

2. Replace F by $F \cup L$ in lines 6, 10 and 8 of Algorithm 1 and in the first line of the algorithm for BCP implementation, provided in Section 2.2.1.

3. Insert the following code just after line 7 of function AnalyzeConfBtAndFlip (Algorithm 3):

   $L := L \cup \{\rho^T\}$

To implement any other conflict clause recording scheme, it is sufficient to add some or all of the clauses, derived by resolution by Algorithm 3, to L.

Employing CCR within SSS does not violate algorithm correctness.

Theorem 5 (Correctness and termination of SSS with CCR)**.** *Given a satisfiable formula F, SSS with CCR will return that the formula is satisfiable with the model $\sigma_{1...s}$. In this case, $\sigma_{1...s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with CCR will return the fact that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. All of the arguments used in the proof of Theorem 1 still hold, even if conflict clauses are recorded (independently of the conflict clause recording scheme). □

2.2.5 Conflict Clause Deletion (CCD) and Restarts

Recording and keeping too many conflict clauses may lead to memory explosion and BCP deceleration. *Conflict clause deletion (CCD)* [3], also known as relevance-based learning, deletes unnecessary conflict clauses. The first heuristic for CCD was proposed in [3]: clauses are deleted as soon as the number of unassigned literals becomes greater than some threshold k. A similar idea is used in GRASP [60] and Chaff [45]. The paper on the Berkmin SAT solver [27] proposed removing clauses based, not just on their size, but also on their activity – that is, the number of times the clause was used for conflict derivation, and age – that is, when the clause was recorded. We are unaware of any survey paper on conflict clause deletion strategies.

To implemented conflict clause deletion, it is sufficient to allow the solver to delete any conflict clause at any time:

Conflict Clause Deletion (invoked just before line 3 of SSS (Algorithm 1))
RemoveSomeConflictClausesIfRequired(L)

The function RemoveSomeConflictClausesIfRequired may remove or leave any conflict clause, whenever invoked.

A *restart* stops the backtrack search process, unassigning all the variables and restarting the search. Restarts have been proposed and shown effective for real-world SAT instances [29]. Chaff [45] restarts the search after i conflict clauses are recorded, where i is an integer threshold number, slowly increased during algorithm invocation. A number of more dynamic restart strategies were proposed recently [6, 57]. We refer the reader to some recent papers on this topic for an overview [32, 57].

For implementing a restart strategy within SSS, it is sufficient to allow the algorithm to restart at every point of the search:

Restarts (invoked just before line 3 of SSS (Algorithm 1))
if RestartNow() **then**
  $s := 0$

The function RestartNow can return true or false at each invocation.

Employing CCD and restarts violate the termination arguments for SSS. For example, if the function RestartNow always returns true, the algorithm never terminates. One way to eliminate this problem is to force the algorithm to record at least one conflict clause between two subsequent invocations of restart and never delete it. It should be noticed, though, that conflict clause deletion and restart strategies, as implemented in modern SAT solvers, seem to be efficient enough to cope with this problem, even if special measures to avoid infinite loops are not taken.

Theorem 6 (Correctness and termination of SSS with restarts and conflict clause deletion)**.** *Suppose that at least one new conflict clause that is never deleted is recorded between two subsequent invocation of restarts. Then, given a satisfiable formula F, SSS with restarts and conflict clause deletion will return that the formula is satisfiable with the model $\sigma_{1...s}$. In this case, $\sigma_{1...s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with CCR will return the fact that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. We need to check the impact of restarts and conflict clause deletion on the proof of Lemma 3, since the other lemmas concentrate on proper

functionality of backtracking, which is not related to restarts and conflict clause deletion. It is easy to see that restarts and CCD do not violate the soundness of post-conditions 2 and 3 of Lemma 3. Also, post-condition 1 holds if a restart is not applied during the new iteration of the main loop. However, the termination function becomes 0, when a restart is applied, hence post-condition 2 does not hold, whenever a restart is applied. To prove termination, we update the termination function to contain a triple $\langle c, t, s \rangle$, whose first element is the number of conflict clauses and the other elements are defined as in Definition 19. Note that there is a finite number of clauses, given a finite variable domain. Now the termination function must grow, even if restarts are applied, since it is guaranteed that the algorithm records at least one new conflict clause between restarts. The parent invariant trivially holds after a restart. Hence, post-condition 2 holds. □

Chapter 3

Understanding and Enhancing Conflict-Driven Learning

Conflict-driven learning (CDL) [63, 55, 3, 60, 45] is a series of algorithmic improvements to the backtrack search algorithm, applied upon detection of a conflict. Chaff's CDL algorithm is used as the baseline approach in modern state-of-the-art solvers, such as Siege [56], Minisat [19], Berkmin [27], Eureka [48], and PicoSAT [6]. Chaff's conflict-driven learning algorithm employs non-chronological backtracking, 1UIP-based conflict-directed backjumping and parent-based conflict clause recording. In the literature, Chaff's CDL engine is referred to as the *1UIP scheme* for conflict-driven learning.

Definition 34 (1UIP scheme for conflict-driven learning)**.** *The 1UIP scheme for conflict-driven learning consists of the application of the following algorithms: 1UIP-based conflict-directed backjumping, non-chronological backtracking and parent-based conflict clause recording.*

Reference [69] analyzed the performance of various schemes for conflict-driven learning and reached the conclusion that the 1UIP scheme is the most efficient one empirically. These empirical results were confirmed in [56]. However, the reasons for this scheme's success were never clarified: "[T]he effectiveness of certain searching schemes can only be determined by empirical data" [69]. Here, we hope to better understand this phenomenon.

Section 3.1 shows how to integrate the following variations and improvements to Chaff's conflict-driven learning engine into our framework:

- The *All*UIP scheme for CDL, proposed in [69].
- A family of schemes, which we call the UIP-n schemes, proposed by the author of this work in [18]. These schemes terminate the backtracking process, when UIP number n is encountered.
- Conflict clause minimization [4, 62] is a technique that is applied to newly derived conflict clauses. It removes literals from the conflict clause C if they can be derived from other literals of C. The 1UIP scheme with conflict clause minimization is called the minimized 1UIP scheme for conflict-driven learning. It is known to be efficient and is integrated into several modern SAT solvers, such as Minisat [19], Eureka [48], RSAT [53] and PicoSAT [6].

Section 3.2 describes the 1UIP scheme and the above-mentioned schemes for CDL using the implication graph-based approach to conflict-driven learning used in most of the papers on this subject [60, 45, 27, 56, 69, 4, 62].

Section 3.3 formalizes the concept of search pruning by relating it to the size of resolution derivations, maintained by the algorithm.

Section 3.4 shows that the minimized 1UIP scheme is better than others in terms of search pruning both analytically and empirically. Our analysis justifies the empirical superiority of the minimized 1UIP scheme over other schemes.

Section 3.5 introduces an enhancement to the minimized 1UIP scheme for CDL, called local conflict clause recording, proposed by the author of this work in [18]. Local conflict clause recording enhances the minimized 1UIP scheme by recording additional conflict clauses. We show that local conflict clause recording improves the performance of a modern SAT solver.

Section 3.6 is dedicated to conflict clause-based assignment stack shrinking, a technique, proposed by the author of this work in Jerusat [46, 47], and further fine-tuned in the 2004 version of the Chaff solver, called Zchaff2004 [40]. Shrinking tries to dynamically reduce the size of conflict

clauses and to unassign such assigned variables that are irrelevant to conflicts. We reaffirm the experimental results of [40], showing that shrinking often leads to faster solving times, especially for microprocessor verification benchmarks. We also demonstrate experimentally that assignment stack shrinking leads to faster solving times, even when conflict clause minimization and rapid restarts [6] are used, disproving a supposition of [6] that assignment stack shrinking is subsumed or simulated by conflict clause minimization and rapid restarts.

3.1 Integrating Other Conflict-Driven Learning Schemes into our Framework

In this section, we show how to employ the UIP-n and the *All*UIP schemes as well as conflict clause minimization in our framework.

3.1.1 The UIP-n Scheme

We start with a description of a family of schemes for conflict-directed backjumping, called UIP-n-based CDB. Recall the definition and the discussion of the 1UIP-based CDB, provided in Section 2.2.3 on page 35. UIP-n-based CDB is similar to the 1UIP-based CDB with the exception that it skips the first $n-1$ UIP's for the notion of a UIP). The asserting clause, generated by applying UIP-n-based CDB, is called the *UIP-n conflict clause*.

For an example of a UIP-2-based CDB, consider Figure 3.1(a) on page 55. Suppose the solver is backtracking, following a conflict. Resolving the blocking clause $a \vee \neg b \vee \neg e \vee f$ with the parent clause of f: $\neg e \vee \neg f$ results in a clause $a \vee \neg b \vee \neg e$. This clause contains only one variable assigned at the last decision level, namely e; hence e is the first UIP and the 1UIP-based CDB would stop the backtracking process. The UIP-2-based CDB continues backtracking. The next resolution operation results in a clause $a \vee \neg b \vee \neg c = a \vee \neg b \vee \neg e \otimes^{e} \neg c \vee e$. Here the only variable that is assigned at the last decision level is c, hence we found the second UIP and UIP-2-based CDB stops backtracking.

The UIP-1-based CDB is identical to the 1UIP-based CDB, hence the following algorithm can be seen as a generalization of 1UIP-based CDB:

UIP-n-based CDB (n) (invoked just before line 4 of Algorithm 3):
Require: *UIPCount* is initialized to 1 at the beginning of Algorithm 3
if there exist non-flipped assignment levels **then**
$g :=$ the decision level of v_s
if $v_s^{\neg\sigma_s}$ is the only literal in ρ^T with the decision level g **then**
if $UIPCount = n$ **then**
$t :=$ the assignment level of the decision variable of g
$v_t := v_s$
$\sigma_t := \sigma_s$
$FlipStatus[t] := false$
$s := t$
Continue to the condition of the while loop
else
$UIPCount := UIPCount + 1$

The definition of the UIP-n scheme for conflict-driven learning is very similar to that of the 1UIP scheme for CDL (Definition 34), with the exception that UIP-n-based CDB is used instead of 1UIP-based CDB.

Definition 35 (UIP-n scheme for conflict-driven learning)**.** *The UIP-n scheme for conflict-driven learning comprises the application of the following algorithms: UIP-n-based conflict-directed backjumping, non-chronological backtracking and parent-based conflict clause recording.*

It will be shown in Section 3.4 that the 1UIP-based CDB scheme is advantageous over UIP-n-based CDB for $n > 1$ both theoretically – in terms of search pruning, and empirically – in terms of performance.

Theorem 7 (Correctness and termination of SSS with UIP-n-based CDB)**.** *Given a satisfiable formula F, SSS with UIP-n-based CDB will return that the formula is satisfiable with the model $\sigma_{1...s}$. In this case, $\sigma_{1...s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with UIP-n-based CDB will return*

that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.

Proof. The arguments are very similar to that of the proof of correctness and termination of SSS with 1UIP-based CDB provided for Theorem 4. Therefore, we skip the proof here. □

3.1.2 The *All*UIP Scheme

One scheme for CDL, whose empirical inferiority to the 1UIP scheme remains unexplained, is the *All*UIP scheme [69]. It terminates the backtracking process when the so-called *All UIP conflict clause* is derived. After the algorithm identifies the first UIP, the *All*UIP scheme applies the resolution rule on the backtracking clause using variables of lower decision levels as pivot variables, thereby ensuring that in the end, the conflict clause does not contain more than one literal per decision level. The primary goal is to make the clause shorter, keeping in mind that shorter clauses are more suitable for BCP. The *All*UIP conflict clause is indeed much shorter than the 1UIP conflict clause, yet the performance of the 1UIP scheme is superior to the *All*UIP scheme [69, 56]. We justify this observation analytically in Section 3.4.

Now we describe *All*UIP-based CDB.

Let d be a decision level. It is convenient for the subsequent discussion to define a restriction of a clause to a decision level.

Definition 36 (Clause restriction to a decision level)**.** *Let C be a clause and d be a decision level. Then C's restriction to decision level d, denoted by $C \restriction_d$, is a clause $D \subseteq C$, such that all the literals of D are assigned at decision level d.*

*All*UIP-based CDB leaves only one variable per decision level in the derived clause by performing additional resolution steps:

*All***UIP-based CDB** (invoked just before line 7 of Algorithm 3):
Require: The 1UIP-based CDB algorithm, provided in Section 2.2.3, is applied.

$g :=$ The current decision level
$d := g - 1$
while $d > 0$ **do**
  while $\rho^T \restriction_d > 1$ **do**
    $h :=$ The highest assignment level in $\rho^T \restriction_d$
    $\rho := \pi_h \otimes^{v_h} \rho$
  $d := d - 1$

In our example in Figure 3.1(a), the 1UIP conflict clause is $a \vee \neg b \vee \neg e$. The algorithm resolves it with the parent clause of b at decision level 1 and receives the clause $a \vee \neg e = a \vee \neg b \vee \neg e \otimes^b a \vee b$. This clause contains one literal from each decision level, hence serving as the *All*UIP conflict clause.

As usual, the *All*UIP scheme applies *All*UIP-based CDB, NCB and parent-based CCR.

Definition 37 (*All*UIP scheme for conflict-driven learning)**.** *The All UIP scheme for conflict-driven learning comprises the application of the following algorithms: All UIP-based conflict-directed backjumping, non-chronological backtracking and parent-based conflict clause recording.*

Theorem 8 (Correctness and termination of SSS with *All*UIP-based CDB)**.** *Given a satisfiable formula F, SSS with All UIP-based CDB will return that the formula is satisfiable with the model $\sigma_{1...s}$. In this case, $\sigma_{1...s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with All UIP-based CDB will return that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. Lemma 1 remains valid, since the code of *All*UIP-based CDB does not modify the backtracking loop. The proofs of Lemma 3 and Theorem 1 remain valid as well. It is sufficient to prove that the second post-condition of Lemma 2 still holds, given that its pre-conditions hold. (If the first post-condition holds, the *All*UIP-based CDB algorithm is not invoked by construction).

The new assignment level s' is not changed by *All*UIP-based CDB. Hence, it remains to prove that the parent invariant holds after the flip. By Proposition 5, it is sufficient to show that ρ remains an asserting resolution derivation

after each application of the resolution rule by the *All*UIP-based CDB algorithm. We prove this fact by induction on the number of applications of the resolution rule.

Consider the situation before *All*UIP-based CDB applies the resolution rule. The clause ρ^T is the 1UIP clause in this case. It is shown inside the proof of Theorem 4 on page 37 that Lemma 2 holds for 1UIP-based CDB; hence by Proposition 5, ρ must be an asserting resolution derivation that becomes the parent resolution derivation after the flip.

Now consider an arbitrary application of the resolution rule by *All*UIP-based CDB $\rho' := \pi_h \otimes^{v_h} \rho$. (We denote by ρ' the resolution derivation ρ after the algorithm executes the above statement.) Note that the literal $v_{s'}^{\neg\sigma_{s'}}$ is never used as a pivot variable by the algorithm, since the algorithm visits decision levels lower than g. Consequently, it remains to show that $\rho'^T \setminus \left\{v_{s'}^{\neg\sigma_{s'}}\right\}$ consists of negation of literals, assigned at assignment levels $1 \ldots s' - 1$.

By induction, ρ is an asserting resolution derivation. Hence, ρ^T consists of the literal $v_{s'}^{\neg\sigma_{s'}}$ and negation of literals assigned at assignment levels $1 \ldots s' - 1$. By the pre-condition of Lemma 2, the parent invariant holds, hence π_h consists of the literal $v_h^{\sigma_h}$ and negation of literals assigned at assignment levels $1 \ldots h - 1$, where $h < s'$. Thus, the resolution rule application is correct and ρ' is still a resolution derivation. Moreover, $\rho'^T \setminus \left\{v_{s'}^{\neg\sigma_{s'}}\right\}$ consists of negation of literals, assigned at assignment levels $1 \ldots s' - 1$. □

3.1.3 Conflict Clause Minimization

Conflict clause minimization tries to remove literals from a conflict clause by applying additional resolution steps using parent clauses of assigned literals. Only clauses of decision levels lower than the last decision level are considered for minimization. Conflict clause minimization was discovered independently in [4] and by the first author of [62], who implemented it in version 1.13 of the Minisat SAT solver [20].

Conflict clause minimization may only reduce the size of the conflict clause or leave it unchanged, thus it should always be advantageous (at least, if the

Algorithm 4 *RemoveIfPossible*(ρ, t)

```
1: if t is a non-flipped assignment level then
2:    return ρ
3: else
4:    θ := ρ ⊗^{v_t} π_t
5: q := t − 1
6: while q > 0 do
7:    if v_q^{¬σ_q} ∈ θ^T and v_q^{¬σ_q} ∉ ρ^T then
8:       if t is a non-flipped assignment level then
9:          return ρ
10:      else
11:         θ := θ ⊗^{v_t} π_t
12:   q := q − 1
13: return θ
```

computation time is not too long.) In practice, conflict clause minimization is successfully applied in modern SAT solvers, such as Minisat [19], Eureka [48], RSAT [53] and PicoSAT [6]. According to [6], minimization is able to remove 32% of the literals on average, which means that when minimization is disabled average clause length increases by almost 50%. Experimental results, provided in Sections 3.4.1 and 3.6 of this work, demonstrate the empirical usefulness of minimization. The practical usefulness of minimization was also shown in a recent paper [62].

Algorithm 4 is an auxiliary algorithm that is invoked by the main algorithm for conflict clause minimization to check if a certain literal can be removed from the conflict clause by applying the resolution rule. It receives a resolution derivation ρ and an assignment level t of a literal that belongs to the conflict clause ρ^T. It tries to remove $v_t^{\neg\sigma_t}$ from clause ρ^T by applying resolution using parent clauses of assigned literals. Algorithm 4 maintains a *current resolution derivation* θ, initialized with ρ, resolved with the parent resolution derivation of $v_t^{\neg\sigma_t}$.

The algorithm iterates over the assignment stack. At each assignment level, it tries to resolve out of θ^T all the literals that do not belong to conflict clause ρ^T. Suppose the algorithm visits assignment level q. Assume that the negation of the assigned literal belongs to the target clause of the current

resolution derivation, but not to the target clause of the parent resolution derivation, that is, $v_q^{\neg\sigma_q} \in \theta^T$ and $v_q^{\neg\sigma_q} \notin \rho^T$. If the assignment level is not flipped, then it is impossible to resolve upon v_q, since it has no parent clause. In this case, the algorithm concludes that it cannot remove the literal $v_t^{\neg\sigma_t}$ from the conflict clause without adding other literals; hence the algorithm returns the initial resolution derivation ρ. If the assignment level is flipped, the algorithm resolves θ with the parent clause of v_q. If the algorithm completes iterating over the assignment stack and it does not return, then this means that it succeeded in generating a resolution derivation θ, whose target clause is of the form $\theta^T = \rho^T \setminus \{v_s^{\neg\sigma_s}\}$.

Now we provide an algorithm for conflict clause minimization.

Conflict Clause Minimization (invoked just before line 7 of Algorithm 3):
$d :=$ the current decision level
$D := \rho^T \setminus \rho^T \restriction_d$
for i from 1 to length of D **do**
  $t := i^{th}$ highest assignment level in D
  $\rho :=$ RemoveIfPossible(ρ, t)

The algorithm iterates over the conflict clause and checks whether each literal assigned at lower decision levels can be removed from the clause. If this is the case, the algorithm replaces the resolution derivation with the minimized resolution derivation.

In our example in Figure 3.1(a), the 1UIP conflict clause is $a \vee \neg b \vee \neg e$. The literals a and $\neg b$ belong to decision level 1 – lower than the current decision level 2. The literal $\neg b$ has the highest assignment level of the two literals. The algorithm checks if $\neg b$ can be removed from the conflict clause by invoking RemoveIfPossible. The resolution derivation θ is initialized with the clause $a \vee \neg e = a \vee \neg b \vee \neg e \otimes^b a \vee b$. The only remaining assigned literal a belongs to the conflict clause, hence the minimization succeeds and the algorithm RemoveIfPossible returns the updated resolution derivation, which replaces ρ.

Now, we can define the concept of a minimized scheme for conflict-driven learning.

Definition 38 (Minimized scheme for conflict-driven learning)**.** *A scheme for conflict-driven learning is* minimized *if conflict clause minimization is applied.*

It is possible to implement the conflict clause minimization algorithm more efficiently using a recursive implementation and re-using information about visited literals between different invocations of the function RemoveIf-Possible. We refer the reader to the recent papers [62, 24] for more details.

It is left to prove the correctness of conflict clause minimization.

Proposition 7 (Correctness of removing a minimized literal)**.** *Suppose that Algorithm 4 is invoked. Assume that:*

1. *The parameter ρ is an asserting resolution derivation for s.*
2. *The parameter t is an assignment level of a literal that belongs to ρ^T and that $t < s$.*
3. *The parent invariant holds.*
4. *The algorithm returns at line 13.*

Then, the algorithm returns a resolution derivation θ, such that $\theta^T = \rho^T \setminus \{v_t^{\neg\sigma_t}\}$.

Proof. Denote the resolution derivation, returned by the algorithm, by θ'.

It is sufficient to prove the following claim: at the beginning of each iteration of the while loop just before the termination condition is evaluated, $\theta^T = (\rho^T \setminus \{v_t^{\neg\sigma_t}\}) \cup D$, where D is a disjunction of 0 or more negations of the literals, assigned at assignment levels $1 \dots q$. Indeed, if the claim holds, then $\theta'^T = \rho^T \setminus \{v_t^{\neg\sigma_t}\}$, since $q = 0$ when the termination condition of the while loop holds.

We prove by induction on the number of iterations of the while loop. At the first iteration, $\theta := \rho \otimes^{v_t} \pi_t$. By pre-condition 1, ρ^T consists of $v_t^{\neg\sigma_t}$ and

negation of literals, assigned at assignment levels $0 \dots q$. Note that t must be a flipped assignment level; otherwise the algorithm would have exited. By pre-conditions 2 and 3, π_t consists of $v_t^{\sigma_t}$ and negation of literals, assigned at assignment levels $0 \dots q$. Hence, θ^T consists of ρ^T without the pivot variable v_t, but possibly with negations of other literals, assigned at assignment levels $0 \dots q$. We have now proved the base case.

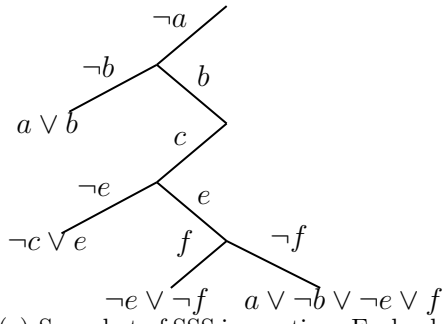
Consider an arbitrary iteration of the while loop. If θ was not changed since the beginning of the previous iteration, we are done. Otherwise, denote by τ the derivation θ at the beginning of the previous iteration. By the induction hypothesis, $\tau^T = (\rho^T \setminus \{v_{q+1}^{\neg\sigma_{q+1}}\}) \cup D$, where D is a disjunction of 0 or more literals, assigned at assignment levels $1 \dots q+1$. We need to show that the last iteration of the while loop removed the variable v_{q+1} from τ and added only negation of literals, assigned at levels $0 \dots q$. By construction of the algorithm, $\theta := \tau \otimes^{v_{q+1}} \pi_{q+1}$. Hence, the variable v_{q+1} is removed from τ. By the parent invariant, which is guaranteed to hold by pre-condition 3, π_{q+1} does not contain any literals, except v_{q+1} and negation of literals assigned at assignment levels $0 \dots q$. □

Theorem 9 (Correctness and termination of SSS with conflict clause minimization.)**.** *Given a satisfiable formula F, SSS with conflict clause minimization will return that the formula is satisfiable with the model $\sigma_{1\dots s}$. In this case, $\sigma_{1\dots s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with conflict clause minimization will return that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

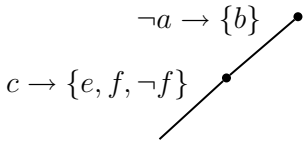
Proof. The proof is very similar to the proof of Theorem 8. Again, it is sufficient to prove that the second post-condition of Lemma 2 still holds, given that its pre-conditions hold.

The new assignment level s' is not changed by conflict clause minimization. Hence, it remains to prove that the parent invariant holds after the flip. The pre-condition 2 of Lemma 2 guarantees that it holds before the flip. By Proposition 5, it is sufficient to show that ρ remains an asserting resolution derivation after minimization. It follows from Proposition 7 that minimiza-

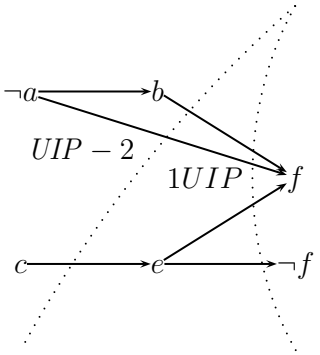
tion may either remove the literal from ρ^T by applying the resolution rule over ρ^T and parent clauses of assigned literals or return ρ untouched. (It is not hard to verify that pre-conditions of Proposition 7 hold, whenever RemoveIfPossible is applied.) In both cases the returned resolution derivation is still an asserting resolution derivation. □



(a) Snapshot of SSS invocation. Each edge corresponds to an assignment level.



(b) BCP-aware notation of the same situation as in Figure 3.1(a). Each edge corresponds to a decision level and is marked with the decision variable on the left and implied variables in brackets.



(c) Implication graph

Figure 3.1: Conflict-driven learning example. Suppose that the input formula contains the following clauses: $(a \vee b); (a \vee \neg b \vee \neg e \vee f); (\neg c \vee e); (\neg e \vee \neg f)$. The conflict clauses, generated by applying the 1UIP scheme, the UIP-2 scheme, the *All*UIP scheme and the minimized 1UIP-based scheme are $a \vee \neg b \vee \neg e$, $a \vee \neg b \vee \neg c$, $a \vee \neg e$ and $a \vee \neg e$, respectively.

3.2 Implication-Based Approach to Conflict-Driven Learning

We now describe Chaff's conflict-driven learning algorithm using the implication-based approach that forces the solver to use BCP whenever possible. Our goal is to create a reference point to the standard approach. Almost all the notions of this section have already been defined in our framework.

At each *decision level*, a Chaff-like solver picks a *decision variable* and assigns it a Boolean value. It propagates the new decision using Boolean Constraints Propagation (BCP): while there exists a unit clause $A \vee v^{\kappa}$, where the literals of A are assigned 0 and v is unassigned, assign v the value κ. The literal v^{κ}, assigned during BCP, is called an *implied literal*. The associated unit clause $A \vee v^{\kappa}$ of an implied literal is called the *parent clause* of literal v^{κ}, denoted by $Par(v^{\kappa})$.

BCP invocation may result in a *conflict* – a situation where BCP finds that all literals in the so-called *blocking clause* are forced to be 0. When this occurs, the solver enters a conflict analysis mode, wherein it records one or more conflict clauses. One of the conflict clauses must be an *asserting conflict clause* – a conflict clause, containing one, and only one literal, called the *asserting literal*, assigned at the last decision level. After the conflict, the solver backtracks to the lowest possible decision level d, such that the asserting conflict clause has only one unassigned literal – the asserting literal. This operation is referred to as *non-chronological backtracking*. The solver flips the value of the asserting literal and propagates the new value in the conflict clause using BCP at decision level d. This operation is referred to as a *failure-driven assertion*. At this point, the negation of an asserting literal is called a *flipped literal* and its variable is called a *flipped variable*. Note that Chaff-like solvers consider the flipped literal to be an implied literal, which does not define a new decision level. This was not the case in GRASP, which treated flipped literals as literals defining a new decision level. Our definition of a decision level (Definition 22 on page 29) sticks to the terminology of Chaff, since it is the standard in the literature. Next we discuss the notions of an implication graph and express the concept of a unique implication point

in the implication graph-based terminology.

Implication relations between assigned literals can be visualized using an *implication graph*. Each vertex in the graph corresponds to an assigned literal. (We will use the notions of vertex and literal interchangeably in the implication graph context.) An edge connects vertices a and b if $\neg a$ appears in the parent clause of b. Upon conflict, we restrict the implication graph to contain only literals connected to the blocking clause. An example of an implication graph appears in Figure 3.1(c). The *implication level* of an assigned variable a, denoted by $il(a)$ is the maximal distance between a and $dvar(a)$, where $dvar(a)$ is the decision variable of the decision level of a. For example, in Figure 3.1 it is the case that $il(c) = 0$ and $il(e) = 1$.

As we have seen, a central notion of conflict analysis is that of the *Unique Implication Point* (UIP) [60]. We now define this notion using implication graph-related terminology. A vertex a in the implication graph *dominates* vertex b if every path from $dvar(a)$ to b passes through a. A UIP with respect to a set of literals A and decision level d, denoted $UIP(A, d)$, is a vertex in the implication graph, which dominates all the literals of $A \restriction_d$, where $A \restriction_d$ is the subset of A's literals, whose decision level is d.[1] A decision level d may have a number of UIPs. The literals $UIP(A, d)$ can be ordered according to their implication level. We denote the UIPs by $UIP_i(A, d)$, where UIP_1 has the maximal implication level.

A $UIP(A, d)$ can be thought of as the unique reason for the implication of literals $A \restriction_d$ at decision level d. If one unassigns all the literals assigned at d, assigns $UIP(A, d)$ and propagates using BCP, all literals of A are implied. To imply $A \restriction_d$ without having any assumptions, one is required to use a subset of literals, assigned at levels lower than d, in addition to $UIP(A, d)$. Let $\Pi_i(A, d)$ to be the set of literals, appearing on a path of length greater than 0 from $UIP_i(A, d)$ to $A \restriction_d$, including $A \restriction_d$. Any $UIP_i(A, d)$ corresponds to an edge cut of the implication graph, called a $UIP_i(A, d)$ cut. The literals of $\Pi_i(A, d)$ are on the right-hand side of the cut. The literals $Res_i(A, d) = \cup_{a \in \Pi_i(A,d)} Par_i(a)$ and $UIP_i(A, d)$ are on the

[1] We provide a definition of UIP that is slightly more extended than the one usually used in the literature. We also suppose that $A \restriction_d$ is non-empty.

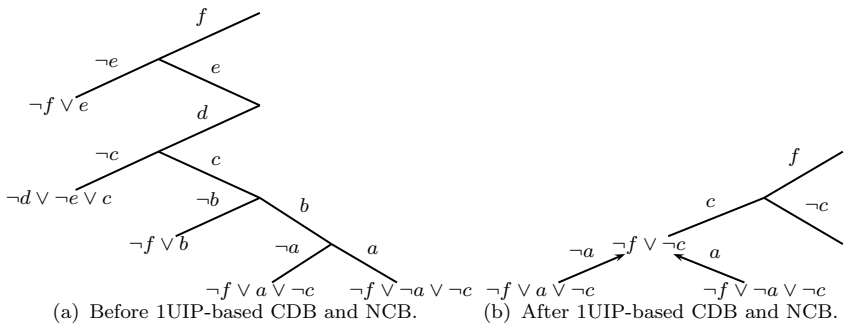
left-hand side of the cut. The structure of the implication graph infers that assigning $Res_i(A,d)$ and $UIP_i(A,d)$ is sufficient in order to imply $A\restriction_d$ using BCP without any assumptions.

Now we are ready to describe various schemes for conflict analysis. We start with the 1IUP scheme. One conflict clause, called the *1UIP conflict clause*, is recorded. The clause consists of the negation of literals of $\beta_1 = Res_1(A,d) \cup UIP_1(A,d)$. Observe that β_1 is sufficient to imply the conflict itself. For example, the 1UIP clause in Figure 3.1(c) is $a \vee \neg b \vee \neg e$. The 1UIP clause is an asserting conflict clause; thus it can be used for a failure-driven assertion.

As we have seen in previous sections, one can think of many other schemes for conflict-driven learning. A comprehensive analysis and evaluation of different schemes for conflict clause recording is [69]. One idea was to use UIPs for decision levels lower than d. Suppose that the previous lowest decision level before d in β_1 is d_2. Then, β_2 is produced by taking β_1, substituting $\beta_1\restriction_{d_2}$ by $UIP_1(\beta_1,d_2)$ and adding $Res_1(\beta_1,d_2)$'s literals to the clause. The negation of β_2 is the 2UIP conflict clause. This operation can be recursively applied at every decision level in β_i, in descending order. The resulting conflict clause is referred to as the *All*UIP conflict clause. In our example, $\neg a \vee \neg e$ is the *All*UIP conflict clause.

Another natural family of conflict analysis schemes would be what we call the *UIP-j schemes*. UIP-j records one conflict clause, consisting of the negation of literals of $Res_j(A,d) \cup UIP_j(A,d)$. The recorded clause corresponds to j's UIP of the last decision level. Observe that in our terminology the UIP-1 scheme is identical to 1UIP. In our example, $a \vee \neg b \vee \neg c$ is the UIP-2 conflict clause.

Conflict clause minimization is a technique for reducing the size of conflict clauses. Given a clause B, a literal $a \in B$ is *B-redundant* if its negation is implied by other literals of B. *Minimizing a clause* means removing from it all B-redundant literals. It is not hard to see that the resulting *minimized* clause is still a valid clause. In our example, b is β_1-redundant, thus $\neg b \vee \neg e$ is the minimized 1UIP conflict clause. Note that an initial clause is always subsumed by the minimized one.



(a) Before 1UIP-based CDB and NCB. (b) After 1UIP-based CDB and NCB.

Figure 3.2: Three kinds of backward pruning. The variables $b/c/e$ are skipped due to resolution/UIP/NCB backward pruning.

3.3 Capturing the Notion of Search Pruning

The goal of this section is making the commonly used notion of search pruning [41, 58] more formal. We show in Section 3.4 that the minimized 1UIP scheme is better than other known schemes in terms of both backward and forward pruning. This serves as an explanation of its empirical advantage over other schemes.

We define *pruning* as the ability of a certain conflict-driven learning scheme to reduce the number of nodes in the resolution refutation generated by the algorithm. We distinguish between backward pruning and forward pruning.

Backward pruning is carried out when backtracking over some flipped assignment levels. Suppose that the algorithm is backtracking and the assignment level is s. Suppose that s is a flipped assignment level. Observe now that if the resolution rule is not applied during backtracking over s, then the parent resolution derivation π_s will not be included in the newly generated asserting resolution derivation.

Definition 39 (Skipped variable/literal/resolution derivation/clause/node)**.** *A flipped assigned variable (literal) is* skipped *when the algorithm is backtracking over it, but its parent resolution is not included into the newly constructed asserting resolution derivation. Similarly, a resolution derivation is* skipped*, when it is a parent resolution derivation of a skipped variable.*

A clause (node) in a resolution derivation is skipped*, when the resolution derivation is skipped.*

If conflict clause recording is not used, clauses that belong to the parent resolution of a skipped variable will not be included in the final refutation of the input formula. Otherwise, a clause from the parent resolution derivation of a skipped variable must be recorded and reused in a new conflict clause derivation to be included in the final refutation. Consequently, it has a lower chance of being included into it.

One can distinguish between three types of backward pruning:

1. *Resolution backward pruning* is carried out by Algorithm 1 without any enhancements when it does not apply the resolution rule for some flipped assignment level s, when $v_s^{\neg\sigma_s} \notin \rho^T$, that is when the condition at line 3 of Algorithm 3 does not hold. Consider the example in Figure 3.2(a). The backtracking clause after the first resolution rule application is $\neg f \vee \neg c$. The variable b will be skipped, since it does not appear in the target clause of the backtracking resolution derivation.

2. *UIP backward pruning* is carried out by the UIP-n scheme (or the 1UIP scheme), after discovering the UIP variable. More specifically, this condition becomes true when a flipped variable v_s is skipped during backtracking by the UIP-n-based (1UIP-based) CDB algorithm, since the UIP-n (1UIP) variable belongs to an assignment level, greater than or equal to s. For example, variable c in Figure 3.2(a) will be skipped due to UIP backward pruning. Indeed, c itself is the 1UIP variable, thus, its own parent resolution derivation is substituted by the newly derived resolution derivation ρ.

3. *NCB backward pruning* is carried out by the NCB algorithm, when it skips flipped assignment levels. For example, the variable e is skipped due to NCB backward pruning in Figure 3.2(a), since the asserting clause does not contain e and c can be flipped at assignment level 2.

Forward pruning is performed by recording conflict clauses, expected to

be reused frequently for conflict identification or propagation, if BCP is used. We define a measure for forward pruning in the next section.

3.4 The Pruning Effect of Different CDL Schemes

We chose to compare the best known 1UIP scheme with UIP-2 and *All*UIP. We believe that the latter two schemes are representative enough to highlight the advantages of 1UIP over other schemes. The comparison with *All*UIP shows why it is not worthwhile resolving newly created clauses with parent clauses of variables that do not belong to the decision level of the 1UIP variable. The comparison with UIP-2 shows why it is advantageous to pick the first UIP of the last decision level, rather than other UIPs. We also discuss the effect of conflict clause minimization. When comparing the contribution of different schemes to backward pruning, we take into consideration their impact on one particular conflict. Analyzing the contribution to backward pruning in additional conflicts is left for future research.

The reason for choosing the first UIP, rather than other UIPs, is because it is optimal for both backward and forward pruning. Proposition 8 analyzes the impact of 1UIP-based CDB and UIP-2-based CDB on backward pruning. We show that the number of the nodes in parent resolution derivations skipped due to backward pruning by the 1UIP scheme is at least the same compared with the UIP-2 scheme. We will see in the experimental results section that the 1UIP scheme allows the algorithm to skip more nodes during backward pruning in practice.

Proposition 8. *Let $N_1(N_2)$ be the number of resolution refutation nodes skipped due to backward pruning in one particular conflict due to 1UIP (UIP-2) backward pruning. Then, it always holds that $N_1 \geq N_2$.*

Proof. Denote by t_1 the assignment level containing the 1UIP variable; and by t_2 the assignment level containing the UIP-2 variable. Let us compare the impact of the 1UIP and UIP-2 schemes on UIP, NCB and resolution backward pruning.

1. Flipped variables assigned between t_2 and t_1 contribute to UIP pruning only for the 1UIP scheme, whereas flipped variables assigned after t_1 contribute to both 1UIP and UIP-2 UIP pruning. Thus, 1UIP-based CDB skips at least the same amount of resolution derivation nodes due to UIP backward pruning as UIP-2-based CDB.

2. The backtrack level for 1UIP-based NCB is never greater than for UIP-2-based NCB, since each additional resolution rule application may only add literals to the asserting clause. Thus, the number of nodes, skipped due to NCB backward pruning for 1UIP-based CDB, is at least the same as UIP-2-based CDB.

3. It seems that the number of nodes skipped due to resolution backward pruning may increase in the UIP-2 scheme, since additional flipped assignment levels may be skipped while backtracking at levels between t_2 and t_1. However, all these variables must be skipped due to UIP backward pruning by the 1UIP scheme. Therefore, any variable skipped due to resolution backward pruning by the UIP-2 scheme must also be skipped by the 1UIP scheme. This may happen as a result of UIP backward pruning, rather than resolution backward pruning.

□

Now we analyze why the 1UIP scheme is better than the *All*UIP scheme, although *All*UIP conflict clauses are typically much shorter than 1UIP conflict clauses [56]. Let the assignment level be s. Resolving at assignment levels l for $l < s$ does not have any impact on backward pruning. We claim that 1UIP conflict clauses tend to be used more for conflict identification and propagation than *All*UIP conflict clauses. Thus, 1UIP conflict clauses are advantageous in terms of forward pruning.

Now we define a more formal measure for forward pruning. First, we need to extend our terminology.

Definition 40 (Pre-flip conflict clause)**.** *Let $v_s^{\sigma_s}$ be a flipped literal. Then, a conflict clause is a* pre-flip conflict clause *for the literal $v_s^{\sigma_s}$ and the variable*

v_s, if it was learned before the variable v_s was flipped – that is, when the variable v_s was used as a non-flipped variable for the last time.

We analyze what fraction of pre-flip conflict clauses, for a flipped literal $v_s^{\sigma_s}$, contains $v_s^{\sigma_s}$. Note that if a pre-flip conflict clause for $v_s^{\sigma_s}$ contain $v_s^{\sigma_s}$, then it becomes satisfied immediately after the flip, hence it cannot help prune the search space. Consequently, we distinguish between useful and useless pre-flip conflict clauses.

Definition 41 (Useful pre-flip conflict clause; Useless pre-flip conflict clause)**.** *Let C be a pre-flip conflict clause for a flipped literal $v_s^{\sigma_s}$. The clause C is* useful *if it does not contain the literal $v_s^{\sigma_s}$; otherwise C is* useless*.*

Now, we can characterize the impact of pre-flip learning on the pruning after the flip.

Definition 42 (Pre-flip learning uselessness)**.** *Let v_s be a flipped variable. The* pre-flip learning uselessness *of $v_s^{\sigma_s}$, denoted by $Fr^+(v_s^{\sigma_s})$, is the fraction of useless pre-flip conflict clauses, out of all pre-flip conflict clauses.*

Note that pre-flip learning usefulness is a real number, varying between 0 to 1. If $Fr^+(v_s^{\sigma_s})$ is 1, then all the conflict clauses recorded before the flip of a variable v_s are useless for pruning after the flip, since they become satisfied. Observe that $Fr^+(v_s^{\sigma_s})$ cannot be 0, since the parent clause of s must contain $v_s^{\sigma_s}$. The lower the pre-flip learning uselessness is, the more helpful are the pre-flip conflict clauses for pruning the search space after flipping the variable v_s.

The key observation for understanding the reasons for the empirical advantage of the 1UIP scheme over the *All*UIP scheme is that, empirically, pre-flip learning uselessness is twice as high for the *All*UIP scheme than for the 1UIP scheme. The explanation for this phenomenon is as follows. Consider a non-flipped decision literal $v_s^{\sigma_s}$, assigned at assignment level s and decision level d. Assume that v_s would become the asserting literal and would be flipped. Suppose that the algorithm is exploring the search space at decision levels higher than d. 1UIP conflict clauses tend to contain literals, implied from v_s as a result of BCP, rather than v_s itself. *All*UIP clauses

tend to contain v_s itself, since the resolution rule is applied on the variable, assigned at decision level d, even when the current decision level is much greater than d. See Figure 3.3 for an example of this phenomenon.

The impact of the 1UIP scheme on forward pruning should also be greater than the UIP-2 scheme for two reasons. First, 1UIP conflict clauses have at most the same size as UIP-2 conflict clauses for one particular conflict. Second, UIP-2 conflict clauses have additional variables assigned at lower decision levels, hence the pre-flip learning uselessness tends to be larger for the UIP-2 scheme.

Conflict clause minimization has no impact on backward pruning. However, it should be useful for forward pruning. Apparently, a shorter clause is better in terms of pruning than a longer clause, subsumed by it. Observe that an *All*UIP conflict clause cannot be minimized. We will see that, empirically, conflict clause minimization is highly beneficial for the 1UIP and UIP-2 schemes.

3.4.1 Empirical Results

We implemented 1UIP, UIP-2 and *All*UIP schemes for conflict analysis within a version of the industrial SAT solver Eureka [48] that did not employ assignment stack shrinking. Conflict clause minimization was used by default, but we also included results for 1UIP and UIP-2 schemes without minimization. Remember that conflict clause minimization has no impact on the *All*UIP scheme. All experiments were carried out on a machine with 4Gb memory and two Intel Xeon CPU 3.06 processors. We used instances from well-known industrial benchmark families, taken from bounded model checking (family *longmult*; instances *longmult10, longmult11*) [7]; microprocessor verification (families *fvp-unsat.2.0, pipe_unsat_1.0*; instances *4pipe, 5pipe, 8pipe_k, 9pipe_k*) [67] and equivalence checking (family *goldberg03-hard_eq_check*; instances *rotmul, term1mul*) [5]. The three schemes are compared in Tables 3.1 and 3.2. These tables show the execution time and the number of conflicts. In addition, they show the average pre-flip learning uselessness Fr^+, as well as the mean number of skipped resolution refutation nodes. Figure 3.4 shows

the relative distribution of the skipped variables according to the categories defined at the end of Section 3.3.

The main conclusions of our experiments are as follows:

1. The minimized 1UIP scheme is indeed more powerful and robust than other schemes. It is always faster than UIP-2, and outperforms *All*UIP by orders of magnitude in four instances, shown in Table 3.1.

2. Pre-flip learning uselessness is double for *All*UIP as compared with 1UIP. This data explains 1UIP's superiority over *All*UIP.

3. Of all the schemes, UIP-2 skips the fewest nodes/flipped variables, mainly due to less powerful UIP pruning. This agrees with our analysis in Section 3.4. In addition, pre-flip learning uselessness for the UIP-2 scheme is slightly higher than for the 1UIP scheme, hinting that the 1UIP scheme is better for forward pruning, though not by an order of magnitude.

4. Surprisingly, in some examples the *All*UIP scheme allows one to skip more nodes and flipped variables than 1UIP. Figure 3.4 shows that this happens mainly due to better resolution pruning by the *All*UIP scheme. According to the analysis in Section 3.4, the number of skipped variables should be about the same for both schemes. This expected behavior is indeed observed in the four instances in Table 3.1, where *All*UIP is outperformed by several orders of magnitude. Studying the reasons for the unexpected behavior in the other four instances (in Table 3.2), where the gap between 1UIP and *All*UIP is not large, is left for future research.

5. Conflict clause minimization is very helpful indeed for both the 1UIP and UIP-2 schemes. The number of skipped nodes is not influenced by minimization, which confirms our observation from Section 3.4 according to which conflict clause minimization does not contribute to backward pruning. Thus, its contribution is to forward pruning. Section 3.6 provides additional evidence for the empirical usefulness of conflict clause minimization.

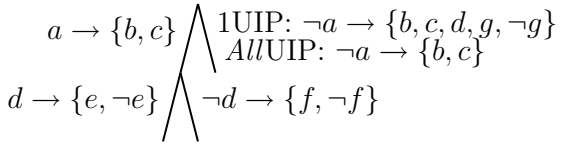


Figure 3.3: One example of the superiority of 1UIP over AllUIP. Suppose the input formula is $(a \vee d \vee g) \wedge (a \vee d \vee \neg g) \wedge (a \vee c) \wedge (a \vee b) \wedge (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee \neg d \vee e) \wedge (\neg b \vee \neg c \vee \neg d \vee \neg e) \wedge (\neg a \vee d \vee f) \wedge (\neg a \vee d \vee \neg f)$, and assume we invoke a modern SAT solver for this formula. For this figure, we suppose that BCP is used. We mark each edge by variables, propagated as a result of BCP, in addition to the decision literal. The solver first picks the literal a, propagates its value, then picks d, propagates and encounters a conflict. The 1UIP clause is $\neg b \vee \neg c \vee \neg d$; the AllUIP clause is $\neg a \vee \neg d$. After flipping d, both the AllUIP and the 1UIP conflict clauses are $\neg a$. After propagating, 1UIP would yield a conflict, meaning that the formula is unsatisfiable. In contrast, AllUIP would not result in a conflict, since all previously recorded conflict clauses have been satisfied.

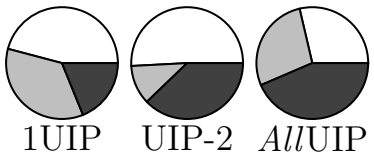


Figure 3.4: Reasons for skipping flipped variables for each of the schemes. The white slice corresponds to the relative number of variables skipped due to NCB pruning. The light gray slice corresponds of UIP pruning. The dark gray slice corresponds to resolution pruning.

Table 3.1: Comparing 1UIP, 1UIP w/o minimization, UIP-2, UIP-2 w/o minimization and *All*UIP on selected instances. The timeout is 14400 sec. The rows display: (Tm) execution time in seconds; (Con) number of conflicts, multiplied by 10^{-3}; (Fr^+) average Fr^+; (NSk) average number of resolution derivation nodes skipped per conflict

Instance	Res	1UIP	1Unm	UIP-2	U-2n	*All*UIP
4pipe	Tm	51	37	148	147	11930
	Con	101	77	309	275	29986
	Fr^+	0.41	0.40	0.38	0.40	0.83
	NSk	0.19	0.19	0.14	0.13	0.24
5pipe	Tm	50	39	347	283	t/o
	Con	85	62	562	420	28186
	Fr^+	0.40	0.37	0.33	0.35	0.84
	NSk	0.18	0.19	0.14	0.13	0.21
8pipe_k	Tm	2426	4035	t/o	t/o	t/o
	Con	1479	1783	10129	8526	13192
	Fr^+	0.37	0.38	0.26	0.26	0.81
	NSk	0.21	0.22	0.13	0.13	0.19
9pipe_k	Tm	1493	3412	t/o	14205	t/o
	Con	641	1098	6040	4793	6548
	Fr^+	0.37	0.38	0.27	0.28	0.85
	NSk	0.20	0.20	0.16	0.15	0.20

Table 3.2: Comparing 1UIP, 1UIP w/o minimization, UIP-2, UIP-2 w/o minimization and *All*UIP on selected instances. The timeout is 14400 sec. The rows display: (Tm) execution time in seconds; (Con) number of conflicts, multiplied by 10^{-3}; (Fr^+) average Fr^+; (NSk) average number of resolution derivation nodes skipped per conflict

Instance	Res	1UIP	1Unm	UIP-2	U-2n	*All*UIP
longmult10	Tm	485	634	513	798	590
	Con	238	297	262	367	380
	Fr^+	0.37	0.37	0.34	0.35	0.84
	NSk	0.13	0.12	0.11	0.10	0.24
longmult11	Tm	559	855	756	1080	690
	Con	273	378	346	462	471
	Fr^+	0.37	0.38	0.35	0.34	0.83
	NSk	0.14	0.12	0.11	0.10	0.25
rotmul	Tm	578	985	1186	1548	992
	Con	615	1001	1371	1790	1576
	Fr^+	0.52	0.51	0.48	0.46	0.84
	NSk	0.16	0.15	0.13	0.12	0.27
term1mul	Tm	2173	3558	5213	9686	2975
	Con	1585	2479	3751	6709	3060
	Fr^+	0.55	0.52	0.54	0.52	0.86
	NSk	0.15	0.15	0.11	0.10	0.26

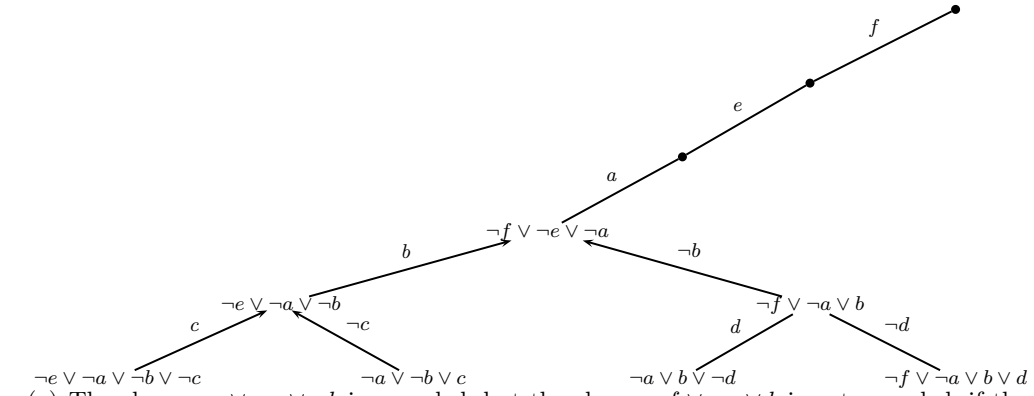
3.5 Local Conflict Clause Recording

In this section, we propose an enhancement to the 1UIP scheme for conflict-driven learning, called *local conflict clause recording*, and demonstrate its positive practical impact. A local conflict clause is a conflict clause, recorded in addition to the 1UIP conflict clause, if certain conditions hold. A local conflict clause is not used as an asserting clause. The algorithm may use it for propagation and conflict identification; thus local conflict clause recording may be found helpful for forward pruning.

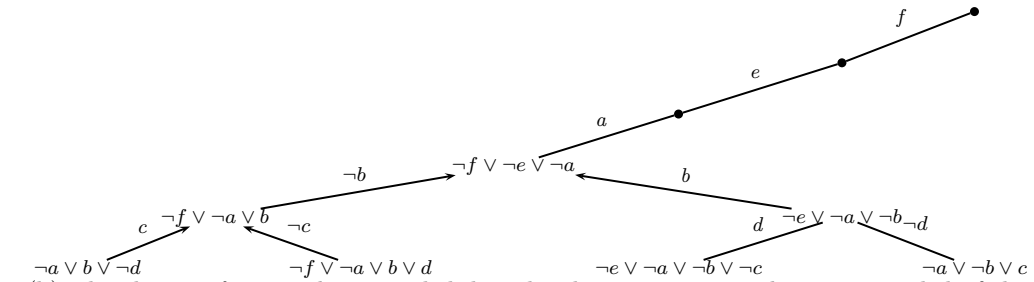
In this section, we assume that the solver uses the basic backtracking algorithm SSS (Algorithm 1), enhanced by BCP (Section 2.2.1) and parent-based conflict clause recording (Section 2.2.4).

The observation behind our proposal is that the set of conflict clauses, recorded by standard conflict clause recording schemes, depends too much on the initial choice of polarity (sign) of assigned variables. This problem is illustrated by Figure 3.5. The two subfigures show an invocation of Algorithm 1 on a given formula. In both cases, the algorithm is about to flip a. The only difference between the two invocations is the choice for the initial polarity for the variable b. Parent-based conflict clause recording (e.g., the 1UIP scheme) records different clauses, depending on the initial polarity of b. When the variable b is assigned 1 first, the clause $\neg e \vee \neg a \vee \neg b$ serves as a parent clause and is recorded, but the clause $\neg f \vee \neg a \vee b$ is an intermediate clause, used during backtracking; hence, it is not recorded. The situation is opposite, when b is assigned 0 first. In this case, the clause $\neg f \vee \neg a \vee b$ is recorded as a parent clause, but $\neg e \vee \neg a \vee \neg b$ is not recorded. The clauses $\neg f \vee \neg a \vee b$ and $\neg e \vee \neg a \vee \neg b$ are different in two literals – that is, in two-thirds of their literals overall; so intuitively, it seems that in this example it would be advantageous to record both clauses.

Now we generalize the above observation. Consider any asserting resolution derivation, derived for flipping a certain variable. A clause in the asserting resolution derivation was recorded as a conflict clause, only if it served as a parent clause for a certain flipped variable. (Recall that the widely used minimized 1UIP scheme uses parent-based conflict clause recording, hence



(a) The clause $\neg e \vee \neg a \vee \neg b$ is recorded, but the clause $\neg f \vee \neg a \vee b$ is not recorded, if the variable b is first assigned 1.



(b) The clause $\neg f \vee \neg a \vee b$ is recorded, but the clause $\neg e \vee \neg a \vee \neg b$ is not recorded, if the variable b is first assigned 0.

Figure 3.5: An example showing the need in local conflict clause recording. Standard conflict clause recording depends too much on literal selection heuristic. Different clauses are recorded (by e.g., 1UIP scheme) while exploring the subspace under the assignment $f = 1; e = 1; a = 1$, depending on the initial polarity selection for the variable b. Suppose that the input formula contains the clauses $\neg e \vee \neg a \vee \neg b \vee \neg c$; $\neg a \vee \neg b \vee c$; $\neg a \vee b \vee \neg d$; $\neg f \vee \neg a \vee b \vee d$.

this problem occurs in modern SAT solvers.) Nonetheless, the structure of the parent resolution derivation suggests that it is sufficient to change the polarities of the assigned variables to change the set of conflict clauses. This is shown in Figure 3.6. The problem is that the algorithm might miss conflict clauses, important for forward pruning.

One solution would be to record all the clauses, generated while backtracking (intermediate clauses in the terminology of [56]), as conflict clauses. However, this solution would not be useful in practice, since it would mean recording many similar clauses. This is expected to slow down the BCP pro-

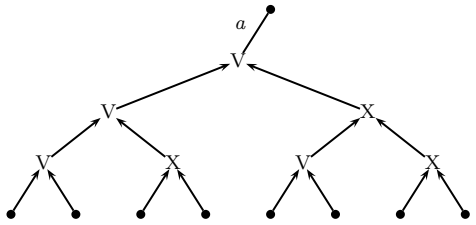


Figure 3.6: A generic example showing the need for local conflict clause recording. An asserting resolution derivation for the upcoming flip of the variable a is shown. Nodes, marked by V correspond to clauses, recorded by parent-based conflict clause recording (e.g., the 1UIP scheme). Nodes, marked by X, correspond to clauses, appearing in parent resolution derivations, but are not recorded as conflict clauses. Note that a node is marked with V iff an outgoing edge from the node goes to the right. This occurs, since an outgoing edge goes to the right iff the corresponding clause is a parent clause, responsible for a flip.

cess without providing a real benefit in forward pruning. It is preferable to add only some of the clauses, according to a certain heuristic.

In the analysis below, we need to distinguish between two kinds of flipped variables, depending on the circumstances of the flip. (In our framework, a variable is flipped when the function Flip changes the flip status FlipStatus of its assignment level to true.) A variable may be flipped as a result of:

1. A conflict, immediately after choosing a new literal (line 9 of Algorithm 1); or

2. Conflict analysis (line 7 of Algorithm 3).

Definition 43 (Conflict-driven flipped variable)**.** *A flipped variable is* conflict-driven *iff the function Flip was invoked as a result of a flip following conflict analysis (line 7 of Algorithm 3).*

In implication graph-based terminology a flipped variable is conflict-driven iff it was assigned as a result of a failure-driven assertion, rather than as a result of BCP. Proposition 9 shows that it also holds in our framework that a flipped variable is conflict-driven iff it was not assigned as a result of BCP.

Suppose that the algorithm is analyzing a certain conflict in function AnalyzeConfBtAndFlip. Suppose that the last decision level contains at least one conflict-driven flipped variable. Assume that r is the assignment level of a conflict-driven flipped variable with the highest assignment level. The idea of local conflict clause recording is to simulate a situation, when the last assigned conflict-driven flipped variable v was first assigned the opposite polarity. When the current backtracking clause contains only one variable, assigned at r or after r, the local conflict clause recording algorithm records this clause as a conflict clause. This simulates a situation in which the last assigned conflict-driven flipped variable becomes a non-flipped decision variable by recording a 1UIP conflict clause with respect to the fake decision level.

Local conflict clause recording (invoked just after line 4 of Algorithm 3):

Require: parent-based conflict clause recording algorithm, provided in Section 2.2.4, is used. (Recall that conflict clauses are recorded in L.)

if a local conflict clause has not yet been recorded at this invocation of AnalyzeConfBtAndFlip **then**

$d :=$ the current decision level

if there exists a conflict-driven flipped variable at decision level d **then**

$r :=$ the assignment level of a conflict-driven flipped variable with the highest assignment level

if ρ^T contains only one variable assigned at assignment level $\geq r$ **then**

$L := L \cup \{\rho^T\}$

Memorize that local conflict clause has been used at this invocation of AnalyzeConfBtAndFlip.

Note that in our example in Figure 3.5, parent-based conflict clause recoding in conjunction with local conflict clause recording would record both clauses $\neg f \vee \neg a \vee b$ and $\neg e \vee \neg a \vee \neg b$, independently of the polarity of b.

Applying local conflict clause recording results in a more balanced conflict clause recording scheme, in the sense that it depends less on polarity selection. In addition, this scheme records only selected clauses, so BCP is

Table 3.3: Effect of local conflict clause recording (time is in sec.; the "cut" column indicates the number of instances that timed out)

Family	Thr.	1UIP		1UIP + LCC	
		Time	Cut	Time	Cut
sat04_ind_maris03_gripper_sat [5]	3 hrs	2238	0	986	0
sat04_ind_goldberg03_hard_eq_check [5]	3 hrs	30336	2	15353	0
sat04_ind_maris03_gripper_unsat [5]	4 hrs	30135	4	17842	2
velev_fvp-unsat.3.0 [66]	3 hrs	18199	2	10928	2
velev_fvp-sat.3.0 [66]	3 hrs	9041	0	7155	0
velev_vliw_sat_2.0 [67]	3 hrs	5970	0	4715	0
barrel [7]	3 hrs	260	0	226	0
velev_pipe_unsat_1.0 [67]	3 hrs	15880	0	13094	0
velev_vliw_unsat_4.0 [67]	3 hrs	17260	0	14810	0
longmult [7]	3 hrs	5413	0	5076	0
velev_vliw_sat_4.0 [67]	3 hrs	5116	0	6882	0

not overwhelmed by the number of conflict clauses. Now we demonstrate that local conflict clause recording contributes to faster SAT solving on real-life industrial benchmarks. Table 3.3 shows the effect of local conflict clause recording on 11 industrial families. The technique is helpful overall in ten out of the eleven families. Table 3.4 shows that local conflict clause recording is particularly useful on real-life hard formal verification instances of the family *goldberg03-hard_eq_check* [5]. Accordingly, local conflict clause recording can be recommended as a default strategy for a modern SAT solver, especially in the formal verification domain.

Now we show that a flipped variable is conflict-driven iff it was not assigned as a result of BCP.

Proposition 9. *Consider Algorithm 1, enhanced by BCP, as implemented in Section 2.2.1. A flipped variable is conflict-driven iff it was not assigned by BCP before the flip.*

Proof. BCP assigns literals appearing in unit clauses to 0. Hence, there must be a conflict following each assignment by BCP; thus any variable assigned as a result of BCP is flipped immediately after choosing a new

Table 3.4: Local conflict clause recording on formal verification instances (time is provided in sec.; the time-out is 3 hours

Instance	Eureka without LCC	Eureka
rotmul	50	38
term1mul	74	42
desmul	90	89
frg2mul	139	103
c3540mul	213	104
dalumul	835	409
frg1mul	886	289
alu4mul	1077	1076
i10mul	1176	732
i8mul	1560	1089
x1mul	2636	906
c6288mul	Time-out	4911
k2mul	Time-out	5565

literal. Therefore, a conflict must follow and a flip must occur at line 9 of Algorithm 1. Consequently, a variable assigned by BCP must not be a conflict-driven variable.

Consider a variable v_s, assigned at assignment level s not as a result of BCP. Recall that BCP is applied whenever there are unit clauses. Hence, no unit clause exists at the time v_s is assigned, since otherwise BCP would have been invoked, preventing the assignment of v_s. Consequently, there cannot be a new falsified clause, immediately after assigning v_s, hence the function Flip at line 9 is not invoked. Suppose that the algorithm flips v_s before exiting. We show that the flip can only happen during conflict analysis (function AnalyzeConfBtAndFlip).

Suppose to the contrary that the flip occurs at line 9.

The assignment and the parent invariants must hold immediately before the assignment of v_s. Indeed, they hold before the first assignment. Thus it follows from repeatedly applying Lemma 3, that they must hold before the assignment of v_s.

Suppose that the algorithm is situated just before assigning v_s and the

termination function is $f = \langle t, s \rangle$. The pre-conditions of Lemma 3 hold at this point. It follows from repeated applications of Lemma 3 that the algorithm must reach line 4, at which it chooses new decision variables, again and again. The termination function is increased between each two visits. Suppose that the algorithm is situated just before assigning a new variable for the last time before flipping v_s at line 9. There must be a conflict, immediately after the assignment to ensure that v_s is flipped at line 9. Let the termination function at this point be $f' \langle t', s' \rangle$. Yet we know that $f' > f$. The termination function could have grown because either:

- $t' = t$; $s' > s$. In this case, the conflict occurs at assignment level $s' \neq s$, and the flipped variable is v'_s, rather than v_s. A contradiction; or
- $t' > t$. In this case, a flip at decision level $s'' < s$ must have occurred by definition of termination function. This means that at some stage the assignment level was decreased beyond s, hence v_s was unassigned, rather than flipped. A contradiction.

□

The correctness of local conflict clause recording follows from Theorem 5 on page 40.

3.6 Conflict Clause-Based Assignment Stack Shrinking

Conflict clause-based assignment stack shrinking, known also as assignment stack shrinking, assignment shrinking, learned clause shrinking or simply shrinking, is a technique that was proposed by the author of this work in [47] and implemented in the Jerusat SAT solver [46]. This technique tries to dynamically reduce the size of conflict clauses and to unassign irrelevant literals from the assignment stack to improve both backward and forward pruning.

If certain conditions hold for a newly learned conflict clause, shrinking unassigns the literals of the conflict clause and reassigns them to 0. BCP follows each assignment. It is guaranteed that after shrinking is applied, a conflict occurs. Following this operation, the newly generated conflict clause tends to contain fewer literals. In addition, the assignment stack tends to be more relevant for the conflict clause in the sense that more assigned variables participate in the clause itself or appear as pivot variables in the resolution derivation of the clause.

Shrinking was fine-tuned in [40] and implemented in the Zchaff2004 SAT solver. Zchaff2004 had two versions: zchaff.2004.5.13 and zchaff_rand. The latter version performed better in SAT competition 2004 [5], hence we describe its implementation of shrinking. The interested reader is referred to [40] for more details on the differences between the two Zchaff2004 versions.

When a newly learned clause exceeds a certain length x, Zchaff2004 sorts the clause according to decision levels. The algorithm finds the lowest decision level that is less than the next higher decision level by at least 2. (If no such decision level is found, then shrinking is not performed.) The algorithm backtracks to this decision level, and the decision strategy starts re-assigning to 0 the unassigned literals of the conflict clause untill a conflict is encountered again. It was found that when reassigning the variables in the reverse order, i.e. in descending order of decision levels, the algorithm performed slightly better than when reassigning the variables in the same order as they were assigned in previously. Since some assigned variables that did not belong to the conflict clause, but which were unassigned during the backtrack, may not get reassigned, the number of assigned variables is likely to drop after this operation. As the assigned variables are more relevant to derived conflict clauses, new conflict clauses are expected to be shorter and more likely to share common literals. In the experiments of [40], no fixed value for x performed well. Instead, x was set dynamically using some measured statistics. Zchaff2004 measures the mean and standard deviation of the lengths of the recently learned conflict clauses and tries to adjust x to keep it at a value greater than the mean. More specifically, Algorithm 5 was used for adjusting

Algorithm 5 AdjustThresholdForShrinking(Threshold for shrinking x, Threshold for learned clauses number y)

Require: x is initialized with the value 95 in the beginning of SAT solving.

```
mean := mean of recent y learned clause lengths
stdev := standard deviation of last y learned clause lengths
center := mean + 0.5 * stdev
ulimit := mean + stdev
if x ≥ center then
  x := x − 5
if x < center then
  x := x + 5
if x > ulimit then
  x := ulimit
if x < 5 then
  x := 5
return x
```

x after each y conflicts. The threshold on the conflict number y is 600 for Zchaff2004. Eureka [48] uses the same algorithm with $y = 2000$. Shrinking often reduced the average length of learned conflict clauses and led to faster solving times, especially for the microprocessor verification benchmarks [40].

Shrinking was also discussed in the paper on the PicoSAT SAT solver [6]. Among the improvements, introduced in the PicoSAT solver, was a rapid restart strategy, triggering restarts with high frequency. PicoSAT does not use shrinking for the following reasons, provided in the paper: (1) shrinking is expensive and is partially subsumed by conflict clause minimization; (2) rapid restarts simulate shrinking, since restarts help the solver recover from mistakes as shrinking does.

We carried out experiments with assignment stack shrinking, conflict clause minimization and restart strategies, in order to answer the following three questions:

1. Can shrinking be considered a useful technique in terms of performance?

2. Is shrinking subsumed by conflict clause minimization?

3. Do rapid restarts simulate shrinking?

The results of the experiments are shown in Tables 3.5 and 3.6. An explanation for the abbreviated benchmark family names and information about the benchmark families appears in Table 3.7. All experiments were carried out on a machine with 4Gb memory and two Intel Xeon CPU 3.60 processors. We used the Eureka-2009 SAT solver for the experiments. The basic version of the solver, denoted by "base" in Table 3.5, employs the following algorithms:

- Assignment stack shrinking. The threshold on the length of clauses x is updated as shown in Algorithm 5. The threshold on the number of conflicts y is 2000.
- Conflict clause minimization.
- An arithmetic restart strategy: restarts are carried out every 2000 conflicts.

Consider first Table 3.5. The version base_no_min does not employ conflict clause minimization, while the version base_no_shr does not employ shrinking. Both minimization and shrinking are disabled in the version base_no_min_no_shr.

The main observations are as follows:

- Overall, the base version solves 214/228 benchmarks within the given time limit. The version without minimization solves 203 benchmarks – 11 fewer than the base version. The version without shrinking solves 187 benchmarks – 16 fewer than the version without minimization. Finally, the version without both techniques solves only 168 benchmarks – 46 fewer than the base version.
- The version without minimization performs worse than the base version on 10/12 families (the exceptions are svp and uv2). The version without shrinking performs worse than the base version on all the families. There is one family containing satisfiable instances, where it is worthwhile to turn off both techniques – svp.

- There are four families, where shrinking is an enabler for solving the instances, whereas minimization either helps only slightly or even causes deterioration of the performance: uv2, uv3, uv4, ug. Overall, turned off shrinking deteriorates the performance more than turned off minimization on 10/12 families.

Consequently, we reach two main conclusions as follows:

1. It is advantageous to employ both conflict clause minimization and assignment stack shrinking.

2. Assignment stack shrinking contributes to the performance more than conflict clause minimization.

Now consider Table 3.6. It shows the performance of a version of Eureka, employing rapid restarts, as implemented in the PicoSAT solver [6], and a version with rapid restarts, without shrinking. It can be seen that assignment stack shrinking is more powerful than rapid restarts. Indeed, switching off shrinking in the base version causes it to solve 27 fewer instances. Switching off shrinking and adding rapid restarts causes the solver to solve 32 fewer instances. Surprisingly, in our experiments rapid restarts were not helpful overall compared to the default arithmetic restart strategy of Eureka. It would be interesting to compare the impact of rapid restarts and shrinking in other solvers, such as PicoSAT, where rapid restarts were found to be helpful.

In our view, the key algorithmic advantage of shrinking over rapid restarts is the fact that shrinking not only unassigns some of the literals – in which case it could have been considered a partial restart strategy – but that it also reassigns the literals in the conflict clause to 0. This causes both the assignment stack and the conflict clause to shrink in size. This effect is not achieved by restarts, which do not handle conflict clauses recorded just before restarting the search in any special way.

Assignment stack shrinking can be implemented as follows:

1. Maintain a threshold for shrinking x and a threshold for learned clauses number y. Algorithm 5 shows how zChaff2004 and Eureka manage both thresholds, where $y = 600$ for zChaff2004 and $y = 2000$ for Eureka.

2. Maintain the following data:

   (a) A Boolean variable *IfApplyShinkingNow*, initialized to false in the beginning of Algorithm 1.

   (b) An array *LitsForShrinking*, containing the literals that should be used for shrinking in correct order.

   (c) An index for the array *LitsForShrinking*, called *ShrinkingInd*.

3. Update the literal selection code with the following algorithm, carrying out the shrinking, if required. We suppose that both BCP and conflict clause recording are in use.

   Shrink (invoked instead of line 4 of Algorithm 1):
   if $\exists C \in F \cup L : C = \neg A \vee v^{\kappa}$ is a unit clause **then**
   $\langle v_s, \sigma_s \rangle := \langle v, \neg\kappa \rangle$
   else
   if IfApplyShrinkingNow **then**
   $\langle v_s, \sigma_s \rangle := \langle LitsForShrinking[ShrinkingInd], 0 \rangle$
   $ShrinkingInd := ShrinkingInd + 1$
   else
   $\langle v_s, \sigma_s \rangle :=$ *ChooseNewLiteral*()

4. Add the following code that decides whether to apply shrinking before each flip. The current implementation applies shrinking only to asserting clauses. It is also possible to apply shrinking to local conflict clauses (in fact, this step is carried out by Eureka), but we omit the implementation for the simplicity of presentation. Also, we do not specify exactly the condition for when to apply shrinking; instead we let the user implement the function FindAssLevelForShrinking. This function is provided with the candidate clause and the threshold x on the size of the clause. It returns the assignment level, where the

algorithm should backtrack to apply shrinking. If shrinking is not to be applied, FindAssLevelForShrinking returns -1. The shrinking scheme of zChaff2004 or Eureka, described in this section, can be used for implementing FindAssLevelForShrinking. We suppose in this implementation that the shrunk literals should be sorted in decreasing order, as in zChaff2004, though other sorting schemes are also possible.

DecideIfShrink (invoked in the beginning of the function Flip (Algorithm 2):
if IfApplyShinkingNow **then**
$IfApplyShinkingNow := false$
else
$L := L \cup \{\rho^T\}$
$b := FindAssLevelForShrinking(\rho^T, x)$
if $b \neq -1$ **then**
$ShrinkingInd := 0$
$LitsForShrinking :=$ Literals of ρ^T that are assigned after b, sorted by assignment level in decreasing order.
$s := b$
Go to line 3 of Algorithm 1

Now it is left to prove the correctness of shrinking. First, we prove that the algorithm must enter the conflict analysis loop following shrinking:

Lemma 4 (Conflict analysis loop entry follows shrinking)**.** *Let the number of literals, used for shrinking (the length of LitsForShrinking), be $k > 0$. The algorithm must enter the conflict analysis loop when shrinking is applied (IfApplyShinkingNow = true) after exactly k literals are assigned and BCP is applied.*

Proof. Denote the conflict clause, recorded before applying the shrinking algorithm by $C = l_1 \vee l_2 \vee \ldots \vee l_z$. Denote the length of C by z. Suppose that C is sorted according to the assignment level, where l_1 is the literal with the lowest assignment level. The algorithm assigns 0 to k unassigned literals of

C that appear in *LitsForShrinking*. The other $z - k$ literals of C are already assigned 0 at lower decision levels.

First we show that the algorithm cannot enter the conflict analysis loop before all k unassigned literals are assigned 0. Indeed, the algorithm assigns 0 to literals that were assigned 0 when the last conflict was encountered. The algorithm enters the conflict analysis loop, only when an assigned literal must receive both values 0 and 1 in two different unit clauses. However, such a situation cannot occur after shrinking is applied, since it did not occur before shrinking was applied; otherwise the algorithm would have backtracked.

Now consider a situation when the single literal l_z remains to be assigned to complete the shrinking. All the other literals $l_f, f \neq z$ are assigned 0. We claim that after l_z is assigned 0, and BCP is applied a number of times, there must be an unassigned literal that appears in different polarities in two unit clauses. This would ensure that the solver would enter the conflict analysis loop after BCP picks this literal as an assignment literal. The simplest argument for the validity of our claim is based on the implication graph structure. Every conflict clause corresponds to a cut in the implication graph. When all the literals of a conflict clause are assigned 0, a situation, when one literal must be assigned different values, must follow BCP.

Note that the correctness of our lemma guarantees that the index *ShrinkingInd* does not go out of bounds. □

Now we prove the correctness and termination of Algorithm 1 with assignment stack shrinking.

Theorem 10 (Correctness and termination of SSS with assignment stack shrinking.)**.** *Given a satisfiable formula F, SSS with assignment stack shrinking will return that the formula is satisfiable with the model $\sigma_{1\ldots s}$. In this case, $\sigma_{1\ldots s}$ indeed satisfies F. Given an unsatisfiable formula F, SSS with assignment stack shrinking will return that the formula is unsatisfiable with the resolution refutation ρ. In this case, ρ is indeed a resolution refutation of F.*

Proof. The only lemma that could be affected by shrinking is Lemma 3. Lemma 3 does not hold at the iteration when shrinking is applied, since the termination function does not grow. However, our algorithm guarantees that

shrinking is not applied two times in a row. Hence, Lemma 3 holds for the next iteration of the main loop, when a new conflict is discovered following shrinking, which is sufficient for our purposes. □

Table 3.5: Interplay between assignment stack shrinking and conflict clause minimization. The first two columns contain the family name abbreviation, explained in Table 3.7, and the number of instances in the family. Each subsequent two columns present the results of one particular strategy, including the number of solved instances and the overall time required for solving. The time threshold, provided in Table 3.7, was added to the latter number, when a strategy timed-out on an instance.

		base		**base_nomin**		**base_noshr**		**base_nomin_noshr**	
Family	**Inst.**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**
ufc	24	24	22776618	22	46660416	23	32874542	18	70726847
ufi	12	11	21533064	10	26596304	9	35223194	9	39426202
ufm	21	20	22917097	19	25739589	18	35088018	18	35227260
svp	10	10	1680863	10	994609	10	3787265	10	1032727
uv2	8	7	10339000	7	9818000	1	27069000	0	28800000
uv3	6	6	11276000	5	17647000	1	61729000	1	62857000
sv3	20	20	492889	20	723851	20	1024769	20	969874
uv4	4	4	10722000	3	19977000	0	43200000	0	43200000
ug	13	13	11500000	13	19426000	10	60848000	4	86400000
mm	10	6	16456000	5	21647000	4	26383000	1	35011000
ms1	50	49	25592157	47	38217198	48	42117556	47	63249099
ms2	50	44	89852376	42	104193222	43	98387213	40	139336578
Sum	228	214	245138064	203	331640189	187	467731557	168	606236587

Table 3.6: Comparing the impact of assignment stack shrinking and rapid restarts. The format is identical to that of Table 3.5

		base_raprest		**base_raprest_no_shr**	
Family	**Inst.**	**Solved**	**Time**	**Solved**	**Time**
ufc	24	24	22034857	22	33093100
ufi	12	11	20131630	9	34375024
ufm	21	20	22059570	19	33090098
svp	10	10	414455	10	1640104
uv2	8	7	10015000	1	28659000
uv3	6	6	10932000	0	64800000
sv3	20	20	896858	20	689353
uv4	4	4	10943000	0	43200000
ug	13	13	12114000	8	64522000
mm	10	6	16770000	3	27835000
ms1	50	48	28745686	47	47084350
ms2	50	44	90286871	43	95274906
Sum	228	213	245343927	182	474262935

Table 3.7: Family information for Tables 3.5 and 3.6. The first three families are internal families of Intel benchmarks, generated in the process of bounded model checking for formal property verification. Other benchmark families were generated by Miroslav Velev [67, 66], used for SAT competition 2004 [5], or used for SAT race 2006 [61]. All the benchmarks, except the mixed benchmark set, used for SAT race 2006, come from the formal verification domain. The last column specifies the time-out per benchmark.

Abbreviation	Family name	SAT/UNSAT/Mixed	Time-out
ufc	fpv_cingr	UNSAT	3 hours
ufi	fpv_iotrk1	UNSAT	3 hours
ufm	fpv_mpiotrk1	UNSAT	3 hours
svp	sat04-ind-velev-pipe-sat-1-1 [67]	SAT	1 hour
uv2	sat04-ind-velev-vliw_unsat_2.0 [67]	UNSAT	1 hour
uv3	velev_fvp-unsat.3.0 [66]	UNSAT	3 hours
sv3	velev_fvp-sat.3.0 [66]	SAT	3 hours
uv4	velev_vliw_unsat_4.0 [67]	UNSAT	3 hours
ug	sat04-ind-goldberg03-hard_eq_check [5]	UNSAT	3 hours
mm	sat04-ind-maris03-gripper [5]	MIXED	1 hour
ms1	SAT-Race_TS_1 [61]	MIXED	3 hours
ms2	SAT-Race_TS_2 [61]	MIXED	3 hours

Chapter 4

A Clause-Based Heuristic for SAT

In this chapter, we propose a new decision heuristic for modern SAT solvers. The heuristic's core innovation is that both the initial and conflict clauses are arranged in a list and the next decision variable is chosen from the topmost unsatisfied clause. Various methods of initially organizing the list and moving the clauses within it are studied. Our approach is an extension of one used in Berkmin [27], and adopted by other modern solvers, according to which only conflict clauses are organized in a list, and a literal-scoring-based secondary heuristic is used when there are no more unsatisfied conflict clauses. Our approach, implemented in Eureka [48], in the 2004 version of the Chaff solver Zchaff2004 [40] and in the Chaff-like SAT solver SE, results in a significant performance boost on hard industrial benchmarks.

4.1 Existing Decision Heuristics

We now describe the most widely used decision heuristics, known to be efficient on real-world industrial benchmarks. Early static heuristics (e.g., Jeroslaw-Wang [33], Literal Count [59]) picked the next variable based on the number of appearances (scores) of different variables in unsatisfied clauses. A major drawback of such an approach is that score calculation requires

visiting all the clauses at the decision point, which implies a very significant overhead. Another disadvantage of static heuristics is that they do not consider information that can be retrieved during conflict analysis. Heuristics based upon such analysis were found to be several orders of magnitude faster [27, 45].

The first dynamic heuristic was the Variable State Independent Decaying Sum (VSIDS) [45]. According to VSIDS, each literal is associated with a counter $cl(p)$, whose value is increased once a new clause containing p is added to the database. Counters are initialized to 0. Every once in a while, all counters are halved. The next literal to be picked is the one with the largest counter. Ties are broken randomly. Two major advantages of VSIDS over previous heuristics are that: (1) VSIDS is characterized by a negligible computational cost; and (2) VSIDS gives preference to literals that participate in recent conflict analysis, i.e., it is dynamic. The Minisat SAT solver [19] implements a variant of VSIDS. Instead of infrequent halving of the scores, Minisat multiplies the scores after each conflict by 0.95. This makes the heuristic more dynamic. The authors of the Berkmin SAT solver [27] proposed a successful decision heuristic that has been partially or fully adopted by SAT solvers such as the 2004 version of Chaff Zchaff2004 [40], Satzoo [19], and Oepir [1]. We show, in the experimental section, that the Berkmin heuristic is indeed faster than VSIDS on hard industrial benchmarks. The Berkmin heuristic's main difference, when compared to VSIDS, is as follows: Conflict clauses are organized in a list, and every new conflict clause is appended to the head of the list. The next decision variable is picked from the topmost unsatisfied clause. If no such clause exists, the next decision variable is chosen according to a VSIDS-like heuristic. We now describe the Berkmin heuristic in detail and analyze why it is preferable to VSIDS.

Berkmin maintains a counter $cl'(p)$ measuring the contribution of each literal to the search. Unlike VSIDS, Berkmin augments $cl'(p)$, not only for literals that belong to the conflict clause itself, but also for literals that belong to one of the clauses that were resolved upon to generate the 1UIP conflict clause on backtracking. At intervals, Berkmin divides all the counters by 4 (compared to 2 for VSIDS). Let $cv'(p)$ be a counter measuring the contri-

bution of each variable to the conflicts, defined as $cl'(p) + cl'(\neg p)$. Berkmin maintains all conflict clauses in a list. After each conflict, the new conflict clause is appended to the top of the list. The next decision variable is the one with the highest $cv'(p)$ out of all the variables of the topmost unsatisfied clauses. If no conflict clauses have yet been generated, or if all the conflict clauses are satisfied, then the variable with the highest $cv'(p)$ of all unassigned variables is chosen.

Next, we describe how Berkmin decides which literal, out of the two possible literals of the already chosen variable, to pick. Berkmin maintains a counter $gcl(p)$, which measures the global contribution of each literal to the conflicts. The counter $gcl(p)$ is initialized to 0 and is increased whenever $cl'(p)$ is increased, but is not divided by a constant. If a topmost unsatisfied clause exists, Berkmin picks a literal with the highest global score $gcl(p)$. Ties are broken randomly. If there is no unsatisfied topmost clause, then Berkmin picks the literal with the highest value of $two(p)$, where $two(p)$ approximates the number of binary clauses in the neighborhood of literal p. The function $two(p)$ is computed as follows: First, the number of binary clauses containing p is calculated. Then, for each binary clause B, containing p, the number of binary clauses containing q is computed, where q is the other literal of B. The sum of all computed numbers gives the value of $two(p)$. To reduce the amount of time spent computing $two(p)$, a threshold value of 100 is used. As soon as the value of $two(p)$ exceeds the threshold, its computation is stopped. Once again, ties are broken randomly.

The most important advantage of the Berkmin approach over VSIDS, as stated by the authors of Berkmin, is its additional dynamicity. It quickly adjusts itself to reflect changes in the set of variables relevant to the currently explored branch. Indeed, Berkmin picks variables from fresh conflict clauses and thus uses very recent data. Our understanding is that the Berkmin heuristic has another important advantage over VSIDS: newly assigned variables tend to embrace more interrelated variables. By interrelated, we mean variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch and satisfying problematic clauses in satisfiable branches. According to the Berkmin heuristic, a series of new de-

cision variables appears in the newest conflict clauses. This means that these variables were the ones recently traversed during conflict analysis and consequently contributed to conflict derivation. Moreover, even if the topmost conflict clauses were recorded a long time ago, the fact that their variables appeared closely together during conflict analysis, hints that they are interrelated. However, the impact of this advantage is diluted by the fact that Berkmin does not put the initial clauses in the list, but instead uses VSIDS as a secondary heuristic. The novel CBH heuristic, described in the next section, takes advantage of this observation.

4.2 The Clause-Based Heuristic

In our clause-based heuristic (CBH), all clauses (both the initial and the conflict clauses) are organized in a list. After each conflict, the conflict clause is prepended to the top of the list. *Conflict-responsible clauses* are clauses used in the resolution process during backtracking to generate the new 1UIP conflict clause. Conflict-responsible clauses are placed just after the new conflict clause. The next decision literal is picked from the topmost unsatisfied clause in the list. One can see that CBH is highly dynamic, since recently visited clauses are placed at the top of the list. Also, CBH organizes the list in such way that clauses that were responsible for a recent conflict are placed together. Hence, when one picks a series of decision variables after backtracking, it will tend to embrace interrelated variables. Indeed, when literals are picked from the same clause they must be related, even if the clause is an initial clause. When literals are picked from adjacent clauses, they also tend to be related, since by placing conflict clauses at the top and moving conflict-responsible clauses towards the top, the list is organized such that interrelated clauses are near each other.

As a variant, CBH can also move clauses found to have exactly two unassigned literals during BCP to the top of the list. We refer to this strategy as *2LitFirst*. The added value of this strategy is: (1) More implications are learned during BCP; and (2) Short and potentially contradictory clauses tend to be immediately satisfied. The first point guides the solver to find conflicts

in an unsatisfiable area, and the second one is useful in eliminating conflicts in a satisfiable area. The disadvantages of 2LitFirst are: (1) It tends to separate clauses that contain interrelated variables; and (2) it may promote clauses that have never been responsible for conflicts.

Experimentally, we found that while usually 2LitFirst hurts performance, it may be helpful for instances having high clause/variable ratios. This can be explained by the fact that in instances having a high clause/variable ratio, variables tend to appear in a greater number of clauses, since there are fewer variables per clause overall. Hence, two chains of decisions made using different decision strategies tend to contain more common variables. This gives more weight to the order between variables and the local context of the search. One should prefer variables whose assignment can have an immediate impact; this is exactly what 2LitFirst does. The default version of CBH invokes 2LitFirst on instances where the clause/variable ratio exceeds 10.

One can see that the major differences of CBH compared with the Berkmin heuristic are:

1. CBH organizes both the initial and conflict clauses, rather than only conflict clauses, in a list; therefore, a second choice heuristic is not required. Moreover, any set of decision variables picked by CBH tends to contain more variables from the same clause.

2. After a conflict, in addition to the conflict clause, CBH moves a number of clauses responsible for the conflict (including initial clauses) towards the head of the list. Thus, clauses that are adjacent are likely interrelated. (This idea was proposed independently in [25, 26] and implemented in the HaifaSat solver. However, in contrast to our approach, HaifaSat's CMTF heuristic maintains only the conflict clauses in the list.)

3. As a variant, CBH moves clauses that were discovered to have two unassigned literals towards the top of the list.

CBH can easily be implemented using a doubly-linked list. A pointer to the currently watched clause C, initialized to the topmost clause, is main-

tained. When a decision is required, we seek the topmost unsatisfied clause D, starting from C moving towards the bottom of the list, and picking a literal from D (as described in Section 4.2.1). Observe that if no topmost unsatisfied clause exists, then we have a satisfying assignment, since all the clauses, including the original ones are satisfied. After each conflict, the solver updates the clause list and sets the currently watched clause to point to the top of the list.

Section 4.2.1 explains how CBH chooses the decision literal from the topmost unsatisfiable clause. Section 4.2.2 explains the initial organization of the clause list.

4.2.1 Choosing the Decision Literal from the Top-Most Clause

CBH maintains two counters, $lcl(p)$ and $gcl(p)$, which measure the local and global contributions of each literal to the conflicts, respectively. The counter $lcl(p)$ is initialized to 0 for each p, while $gcl(p)$ is initialized with the number of p's appearances in initial clauses. Both counters are incremented whenever a literal belongs to one of the clauses traversed in the implication graph during 1UIP conflict clause identification. Occasionally, the value of $lcl(p)$ is divided by 2.

CBH also maintains two counters for variables $lcv(p)$ and $gcv(p)$, which measure the contribution of each variable to the conflicts. We define:

$$lcv(p) = (lcl(p) + lcl(\neg p)) + 3 * min(lcl(p), lcl(\neg p)).$$

The first term gives preference to variables for which both literals are important, and the second term eliminates variables where only one literal is important. In a similar manner, we have:

$$gcv(p) = (gcl(p) + gcl(\neg p)) + 3 * min(gcl(p), gcl(\neg p)).$$

CBH chooses the decision variable from the topmost unsatisfied clause using the following algorithm: A variable p with maximal $lcv(p)$ is chosen, so as to give preference to variables that participated in recent conflicts. Ties are broken by preferring variables with the maximal global score $gcv(p)$. According to the next criterion, variables that used to have the maximal decision level when assigned the last time are preferred. (If there is still a tie, it is broken by picking the lexicographically smallest variable.)

CBH chooses the decision literal out of the two possible, based on the global contribution value $gcl(p)$.

4.2.2 Initial Clause List Organization

In general, we aim to:

1. Place clauses containing frequently appearing literals near the top of the list; and

2. Place clauses containing common literals nearby.

Point 1 guides the solver to start the search using frequent literals, and point 2 increases the chances of picking interrelated literals.

First, the initial global score $igs(p)$ is calculated for each literal p. The function $igs(p)$ is initialized to 0 and is augmented for each clause that contains the literal p. The initial global score reflects the overall frequency of a literal. In the process of clause list construction, we also maintain the initial local score $ils(p)$ for each literal p. It is calculated similarly to $igs(p)$, except that only clauses already placed on the clause list are considered. The local score reflects the involvement of p in clauses already appended to the clause list. Initially, no clauses are included in the clause list, hence $ils(p)$=0 for each literal p. We also define the initial overall score $ios(p) = igs(p) + ils(p)$ for each literal p. The initial overall score takes into consideration both the local and global influences of each literal.

So far, we have defined three functions for each literal reflecting its global, local and overall influence. Now, we can define the initial overall score for each variable p:

$$iosv(p) = (ios(p) + ios(\neg p)) + 3 * min(ios(p), ios(\neg p)).$$

The clause list is constructed by repeating the following procedure until all the clauses are placed in the clause list: Let p be the variable having the maximal variable overall score amongst all variables that have not already been picked. (Ties are broken by preferring the smaller variable according to lexicographical order.) Clauses containing the variable p, which have not yet been appended, are appended to the end of the clause list. Local and overall scores are updated for each literal participating in clauses that have been appended to the list. Such dynamic updating of scores does not require any overhead, given that we use a priority queue, indexed by the scores. Literals can be moved within the queues in constant time.

4.3 Experimental Results

First, we tested the impact of CBH inside the SAT solver Eureka [48] on 57 instances from MicroCode verification [2]. The result are shown in Figure 4.1. As one can see, CBH leads to a substantial improvement in the run-time of Eureka. In particular, Eureka with CBH was able to solve every instance within the time limit of 65 minutes, whereas Eureka without CBH timed-out on 17 instances.

Second, we implemented CBH in two other SAT solvers. The first is Zchaff2004 or, more specifically, zChaff_2004.11.15 [40]. zChaff won first place in the Industrial-Overall category of the SAT04 competition [5]. This new version of zChaff2004 implements the Berkmin heuristic, in contrast to the 2001 version of Chaff [45], which used VSIDS. The performance of zChaff2004 was measured on a machine with 4Gb of memory and two Intel XeonTM CPU 3.06GHz processors with hyper-threading. The second solver we used in our experiments was SE – a Chaff-like SAT solver that was in use at Intel. The performance of SE was measured on a stronger machine with 4Gb of memory and two Intel XeonTM CPU 3.20GHz processors with

hyper-threading. In what follows, we first analyze the overall performance of CBH versus the Berkmin heuristic, VSIDS and zChaff2004's new Berkmin-like heuristic. We show that CBH outperforms both the Berkmin heuristic and VSIDS within the SE SAT solver, and also that CBH significantly outperforms zChaff2004's new Berkmin-like heuristic. Then, we analyze how various strategies used by CBH contribute to its performance. We tested CBH inside two solvers to ensure that its measured impact on performance is independent of the implementation details of a particular solver. The main measure for success in our experiments is the number of solved instances within an hour on hard industrial families used during the SAT'04 competition [5]. We find this measure, which was used during the SAT'04 competition, more convincing than a comparison of the number of decisions or conflicts, since reducing the running time is the final goal of any practical heuristic. Our experiments required approximately 35 days of computation.

Table 4.2 compares the performance of CBH, VSIDS, VSIDSM – a Minisat-like VSIDS with frequent score decay – and the Berkmin heuristic, implemented in SE, on eight hard industrial families used during the SAT'04 competition [5]. The description of these families is provided in Table 4.1. VSIDSM multiplies the score by 0.95 after every 10 conflicts, rather than after each conflict. The latter rate is used within Minisat, but the former is preferable within SE. Other heuristics decay the scores every 6000 conflicts. CBH solved at least as many instances within each family, when compared to either the Berkmin heuristic or either version of VSIDS. CBH solved more instances than both versions of VSIDS in 7 out of 8 cases, and solved more instances than the Berkmin heuristic in 5 out of 8 cases.

Table 4.3 shows the performance of CBH within the new version of zChaff2004. The performance of a version of CBH that does not use 2LitFirst is also provided. One can see that zChaff2004, CBH-enabled, outperformed zChaff2004 in a very convincing manner for 6 out of 8 families and was inferior in only one case. Moreover, when the 2LitFirst strategy was not used, CBH was never inferior.

One can conclude that CBH definitely improves the performance of a modern SAT solver, outperforming both VSIDS and the Berkmin heuristic.

Our experiments also confirm that the Berkmin heuristic is preferable to VSIDS, though the gap narrows if VSIDS decays scores frequently. This is to be expected, but – to the best of our knowledge – has never been reported, despite the fact that the Berkmin heuristic has been partially or fully adopted by most modern SAT solvers [1, 19, 27, 40].[1]

What remains is to analyze the performance of CBH when disabling some of its specific strategies. Accordingly, we consider CBH_NM, a version that does not move conflict-responsible clauses to the top of the list (but still appends the conflict clause itself to the top of the list), and CBH_NI, which does not use the initial strategy, described in Section 4.2.2, but rather appends all clauses to the list in their order of appearance in the input instance. We also experimented with CBH_2L_A, which always uses 2LitFirst, and with CBH_2L_N, which never does. Tables 4.4 and 4.5 compare the performance of CBH within SE and zChaff2004, respectively.

Switching off the initial strategy resulted in a performance degradation for three families within SE, and in a performance gain for one family. In zChaff2004, switching off the initial strategy resulted in a performance degradation for two families. In general, the initial strategy improved the performance within both SE and zChaff2004, although it was not the most crucial factor contributing to CBH performance. Even if the initial strategy was switched off, CBH performed better than other decision heuristics. This can be explained by the fact that during the search, CBH quickly reorganizes the clause list to contain groups of interrelated clauses.

Switching off the moving of conflict-responsible clauses to the top of the list resulted in a performance degradation for four families and a performance gain for three families in zChaff2004. The overall number of solved instances was higher when the strategy was switched on. Switching off the moving of conflict-responsible clauses to the top of the list led to mixed results in the case of SE. Performance seriously degraded for the SCH family, and also for the ST2 family; however, there was a performance-boost for the GR, ST2B

[1]Now, in the year 2008, one can see that some of the academic solvers, including the recent SAT competition winners RSAT and TiniSAT have gone back to VSIDSM. CBH remains the default heuristic for Intel's SAT solver Eureka [48], since it is faster than other heuristics on Intel's internal benchmarks.

and VUN families. The overall number of solved instances remained the same. One can conclude that moving the conflict-responsible clauses to the top of the list can be useful for some families, but detrimental to others. We recommend invoking it by default, since it resulted in an overall performance boost in the case of zChaff2004 and did not hurt the overall performance of SE.

Regarding the impact of the 2LitFirst strategy, first observe that according to Tables 4.4 and 4.5, invoking 2LitFirst at every instance is not justifiable. Note that the default strategy used by CBH invokes 2LitFirst if the clause/variable rate is greater than 10. The motivating experimental observation for designing CBH in this manner is that SE without 2LitFirst performed the same as the default version on all families, except VUN, where performance seriously degraded. VUN is the only family, other than PST, for which the clause/variable ratio was greater than 10 in all instances. To confirm that SE performs better when 2LitFirst is invoked in instances with a high clause/variable ratio, we launched SE on 26 handmade families that were submitted to SAT'04. SE was able to solve at least one instance from 13 families. The default CBH strategy performed better than a strategy with 2LitFirst disabled on two out of 13 families, and it performed the same on other families. In the case of zChaff2004, we found that the 2LitFirst invocation hurt performance in a dramatic manner on families with a low clause/variable ratio, and left the performance the same or slightly degraded on families having a high clause/variable ratio. The families HEQ, GR, SCH and ST2 are those with a ratio for all their instances lower than 10. When 2LitFirst was always used, zChaff2004 was able to solve only four instances of these four families, compared to 20 instances solved by the version that never used 2LitFirst. The other four families had either a mixed or high clause/variable ratio. For these families, when 2LitFirst was always used, zChaff2004 was able to solve 12 instances, compared to 16 for a version that never used 2LitFirst. One can conclude that within zChaff2004, 2LitFirst usage is not justified. Overall, 2LitFirst performed much better on instances having a high clause/variable ratio.

Abbr.	Family Name	Num.	C/V Av.	C/V Mx	C/V Mn
HEQ	*goldberg03-hard_eq_check*	13	6.4	6.7	6.1
GR	*maris03-gripper*	10	9.1	9.7	8.5
SCH	*schuppan03-l2s*	11	3.2	3.3	3.0
ST2	*simon03-sat02*	9	3.3	4.2	2.7
ST2B	*simon03-sat02bis*	10	23.9	71.9	2.9
CLR	*vangelder-cnf-color*	12	42.9	195.4	4.0
PST	*velev-pipe-sat-1-1*	10	33.7	33.8	33.7
VUN	*velev-vliw_unsat_2.0*	8	15.6	20.1	10.7

Table 4.1: Description of the hard industrial benchmark families used in our experiments. Family name, number of instances in each family as well as the average, and maximal and minimal clause/variable ratios are provided

Family	CBH	Berkmin	VSIDSM	VSIDS
HEQ	5	4	4	3
GR	1	1	0	1
SCH	5	2	2	0
ST2	5	4	4	2
ST2B	2	2	2	1
CLR	6	4	4	4
PST	10	10	4	5
VUN	4	2	1	0
ALL	38	29	21	16

Table 4.2: Performance of CBH vs. two versions of VSIDS and the Berkmin heuristic, implemented in the SE SAT solver, on eight hard industrial families. The first column contains an abbreviated family name. Each pair of subsequent columns is dedicated to a specific heuristic. The number of instances, solved within one hour, is provided.

Family	zChaff2004+CBH	zChaff2004+CBH_2L_N	zChaff2004 default
HEQ	8	8	4
GR	3	3	0
SCH	5	5	2
ST2	4	4	1
ST2B	2	2	1
CLR	3	4	1
PST	5	8	8
VUN	2	2	2
ALL	32	36	19

Table 4.3: CBH vs. the default heuristic within zChaff2004_2004.11.15. CBH_2L_N is a version of CBH that does not use the 2LitFirst strategy.

Family	CBH	CBH_NM	CBH_NI	CBH_2L_A	CBH_2L_N
HEQ	5	4	5	3	5
GR	1	2	2	0	1
SCH	5	2	4	5	5
ST2	5	4	5	2	5
ST2B	2	3	1	2	2
CLR	6	7	5	5	6
PST	10	10	10	10	10
VUN	4	6	4	4	1
ALL	38	38	36	31	35

Table 4.4: Performance of different configurations of CBH in terms of solved instances within one hour in the SE solver.

Family	CBH	CBH_NM	CBH_NI	CBH_2L_A	CBH_2L_N
HEQ	8	4	8	4	8
GR	3	1	3	0	3
SCH	5	2	4	0	5
ST2	4	2	4	0	4
ST2B	2	3	2	2	2
CLR	3	3	3	3	4
PST	5	10	3	5	8
VUN	2	3	2	2	2
ALL	32	28	29	16	36

Table 4.5: Performance of different configurations of CBH in terms of solved instances within one hour in the zChaff2004 solver.

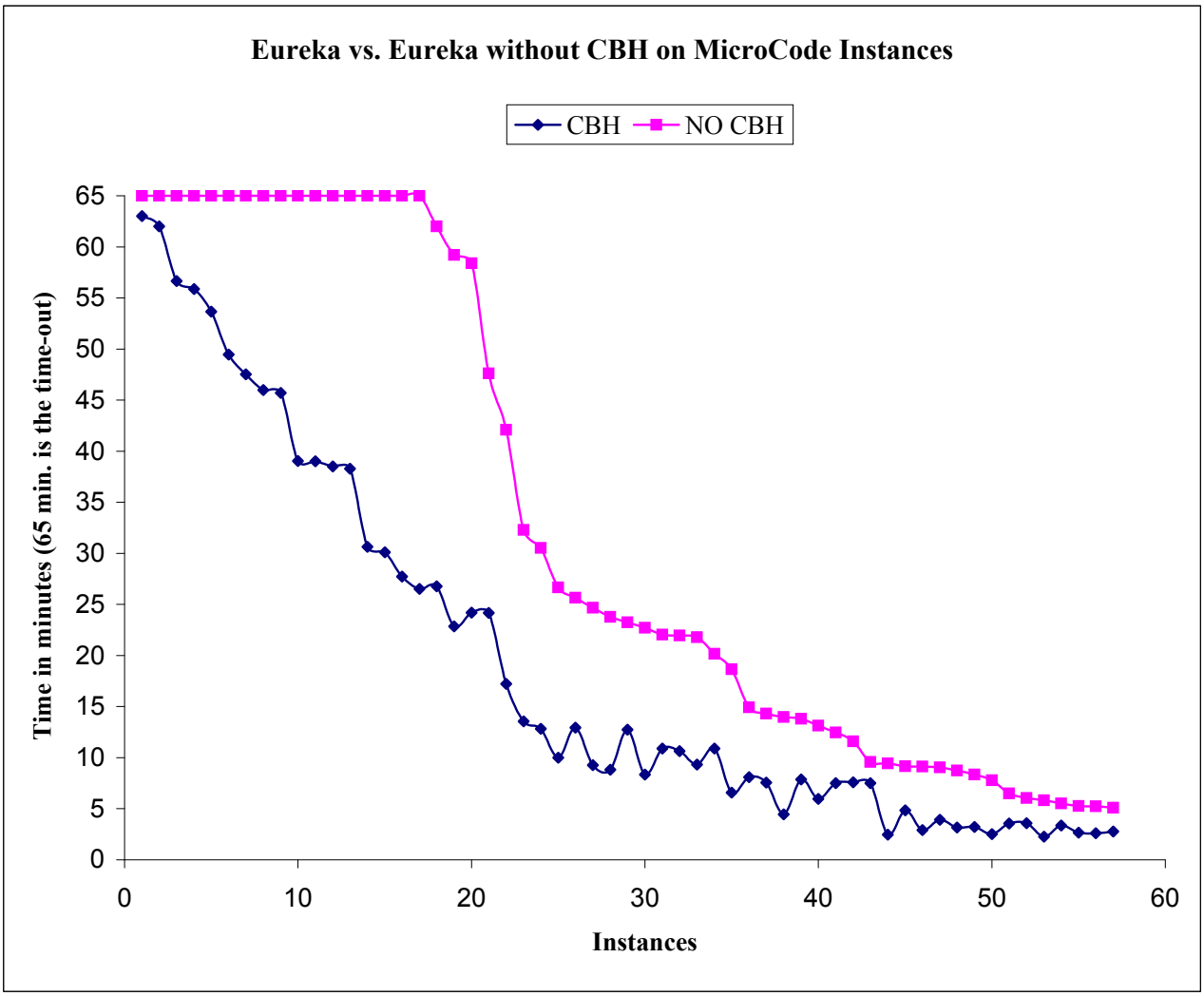


Figure 4.1: CBH effect on MicroCode instances.

Chapter 5

A Scalable Algorithm for Minimal Unsatisfiable Core Extraction

The only approach for unsatisfiable core extraction that scales well for formal verification benchmarks was independently proposed in [70] and in [28]. We refer to this method as the *EC (Empty-clause Cone)* algorithm. EC exploits the ability of modern SAT solvers to produce a resolution refutation, given an unsatisfiable formula. EC traverses a reversed refutation, starting with $\square$ and taking initial clauses, connected to $\square$, as the unsatisfiable core. Invoking EC until a fixed point is reached [70] allows one to reduce the unsatisfiable core even more. We refer to this algorithm as *EC-fp*. However, the resulting cores can be reduced further.

In this chapter we propose a new algorithm for minimal unsatisfiable core extraction, based on a deeper exploration of resolution refutation properties.

5.1 Related Work

Algorithms for unsatisfiable core extraction built on top of modern SAT solvers [70, 28] are the most relevant for our purposes for two reasons. First, this approach allows one to deal with real-world examples arising in formal

verification. Second, it serves as the basis of our algorithm. We have already described the EC and EC-fp algorithms above. Here we briefly consider other approaches.

Theoretical work (e.g., [64]) has concentrated on developing efficient algorithms for formulas with a small *deficiency* (the number of clauses minus the number of variables). However, real-world formulas have an arbitrary (and usually large) deficiency. A number of works considered the harder problem of finding the smallest minimal unsatisfiable core [39, 44], or even finding all minimally unsatisfiable formulas [38]. As one can imagine, these algorithms are not scalable for even moderately large real-world formulas.

In [8, 9], an "adaptive core search" was applied for finding a small unsatisfiable core. The algorithm starts with a very small satisfiable subformula, consisting of *hard* clauses. The unsatisfiable core is built by an iterative process that expands or contracts the current core by a fixed percentage of clauses. The procedure succeeded in finding small, though not necessarily minimal, unsatisfiable cores for the problem instances it was tested on, but these are very small and artificially generated.

Another approach that allows one to find small, but not necessarily minimal, unsatisfiable cores is called AMUSE [51]. In this approach, selector variables are added to each clause and the unsatisfiable core is found by a branch-and-bound algorithm on the updated formula. Selector variables allow the program to implicitly search for unsatisfiable cores using an enhanced version of DLL on the updated formula. The authors noted their method's ability to locate different unsatisfiable cores, as well as its inability to cope with large formulas.

The above described algorithms do not guarantee the minimality of the extracted cores. One folk algorithm for minimal unsatisfiable core extraction, which we dub *Naïve*, works as follows: For every clause C in an unsatisfiable formula F, Naïve checks if it belongs to the minimal unsatisfiable core, by invoking a SAT solver on $F \setminus C$. Clause C does not belong to MUC if and only if the solver finds that $F \setminus C$ is unsatisfiable, in which case C is removed from F. In the end, F contains a minimal unsatisfiable core.

The only non-trivial algorithm existing in the current literature that guar-

antees minimality is MUP [31]. MUP is mainly a prover of minimal unsatisfiability, as opposed to an unsatisfiable core extractor. It decides the minimal unsatisfiability of a CNF formula through BDD manipulation. When MUP is used as a core extractor, it removes one clause at a time until the remaining core is minimal. MUP is able to prove minimal unsatisfiability of some particularly hard classical problems quickly, whereas even just proving unsatisfiability is a challenge for modern SAT solvers. However, the formulas described in [31] are small and arise in areas other than formal verification. We will see in Section 5.5 that MUP is significantly outperformed by Naïve on formal verification benchmarks.

5.2 Multi-Resolution Refutation

A multi-resolution refutation is a resolution refutation, such that each resolvent clause C may have more than two sources, but it is guaranteed that a resolution derivation of C from the sources exists. In this chapter, it will be more convenient to view a SAT solver as on an engine, producing multi-resolution refutations.

A formal definition of a multi-resolution refutation appears below.

Definition 44 (Multi-resolution refutation)**.** *Let F be an unsatisfiable CNF formula (set of clauses) and let $\Pi(V, E)$ be a dag whose vertices are clauses.*[1] *Suppose $V = V^i \cup V^c$, where V^i are all the sources of Π, referred to as* initial clauses*, and $V^c = C^c_1, \ldots, C^c_m$ is an ordered set of non-source vertices, referred to as* conflict clauses*. Then, the dag $\Pi(V, E)$ is a* multi-resolution refutation of F *if:*

1. *$V^i = F$;*
2. *For every conflict clause C^c_i, there exists a resolution derivation $\{D_1, D_2, \ldots, D_k, C^c_i\}$, such that:*

[1] From this point on, we use the terms "vertex" and "clause" interchangeably in the context of multi-resolution refutation.

(a) for every $j = 1, \ldots, k$, D_j is either an initial clause or a prior conflict clause C^c_f, $f < i$, and

(b) there are edges $D_1 \to C^c_i, \ldots, D_k \to C^c_i \in E$ (these are the only edges in E);

3. The sink vertex C^c_m is the only empty clause in V.

A modern SAT solver employing conflict clause recording can generate a multi-resolution refutation as follows. Each conflict clause C, derived by resolution on a set of exiting clauses L, corresponds to a node in the multi-resolution derivation, whose sources are the clauses of L.

For the following discussion, it will be helpful to remember the notion of vertices that are "reachable", or "backward reachable", from a given clause in a given dag.

Definition 45 (Reachable vertices)**.** *Let Π be a dag. A vertex D is* reachable *from C if there is a path (of 0 or more edges) from C to D. The set of all vertices reachable from C in Π is denoted $Re(\Pi, C)$. The set of all vertices unreachable from C in Π is denoted by $\overline{Re}(\Pi, C)$*

Definition 46 (Backward reachable vertices)**.** *Let Π be a dag. A vertex D is* backward reachable *from C if there is a path (of 0 or more edges) from D to C. The set of all vertices backward reachable from C in Π is denoted by $BRe(\Pi, C)$. The set of all vertices not backward reachable from C in Π is denoted $\overline{BRe}(\Pi, C)$.*

For example, consider the multi-resolution refutation in Figure 5.1. We have $Re(\Pi, C^i_5) = \{C^i_5, C^c_2, C^c_3, C^c_4, C^c_5\}$ and $BRe(\Pi, C^c_4) = \{C^c_4, C^i_5, C^i_6\}$.

Multi-resolution refutations trace all resolution derivations of conflict clauses, including the empty clause. Generally, not all clauses of a multi-resolution refutation are required to derive $\square$, but only those that are backward reachable from $\square$. It is not hard to see that even if all other clauses and related edges are omitted, the remaining graph is still a multi-resolution refutation. We refer to such multi-resolution refutations as *non-redundant*

(see Definition 47). The multi-resolution refutation in Figure 5.1 is non-redundant.

To retrieve a non-redundant subgraph of a multi-resolution refutation, it is sufficient to take $BRe(\Pi, \square)$ as the vertex set and to restrict the edge set E to edges having both ends in $BRe(\Pi, \square)$. We denote a non-redundant subgraph of a multi-resolution refutation Π by $\Pi \restriction_{BRe(\Pi,\square)}$. Observe that $\Pi \restriction_{BRe(\Pi,\square)}$ is a valid non-redundant multi-resolution refutation.

Definition 47 (Non-redundant multi-resolution refutation)**.** *A multi-resolution refutation* Π *is* non-redundant *if there is a path in* Π *from every clause to* $\square$*.*

Lastly, we define the relative hardness of a multi-resolution refutation.

Definition 48 (Relative hardness)**.** *The* relative hardness *of a multi-resolution refutation is the ratio between the total number of clauses and the number of initial clauses.*

5.3 The Complete Resolution Refutation (CRR) Algorithm

Our goal is to find the minimal unsatisfiable core of a given unsatisfiable formula F. The proposed *CRR* method is displayed as Algorithm 6.

First, CRR builds a non-redundant multi-resolution refutation. Invoking a SAT solver for constructing a (possibly redundant) multi-resolution refutation $\Pi(V, E)$ and restricting it to $\Pi \restriction_{BRe(\Pi,\square)}$ is sufficient for this purpose.

Suppose $\Pi(V^i \cup V^c, E)$ is a non-redundant multi-resolution refutation. CRR checks, for every unmarked clause C left in V^i, whether C belongs to the minimal unsatisfiable core. Initially, all clauses are unmarked. At each stage of the algorithm, CRR maintains a valid multi-resolution refutation of F.

Recall from Definition 45 that $\overline{Re}(\Pi, C)$ is the set of all vertices in Π unreachable from C. By construction of Π, the $\overline{Re}(\Pi, C)$ clauses were derived independently of C. To check whether C belongs to the minimal unsatisfiable

Algorithm 6 (CRR). Returns a MUC, given an unsatisfiable formula F.

```
1: Build a non-redundant multi resolution-refutation Π(V^i ∪ V^c, E)
2: while unmarked clauses exist in V^i do
3:    C ← PickUnmarkedClause(V^i)
4:    Invoke a SAT solver on Re(Π, C)
5:    if Re(Π, C) is satisfiable then
6:       Mark C as a MUC member
7:    else
8:       Let G = Re(Π, C)
9:       Build multi-resolution refutation Π'(V^i_G ∪ V^c_G, E_G)
10:      V^i ← V^i \ {C}
11:      V^c ← (V^c \ Re(Π, C)) ∪ V^c_G
12:      E ← (E \ Re^E(Π, C)) ∪ E_G
13:      Π(V^i ∪ V^c, E) ← Π(V^i ∪ V^c, E) ↾_{BRe(Π,□)}
14: return V^i
```

core, we provide the SAT solver with $\overline{Re}(\Pi, C)$, including the conflict clauses. We are trying to *complete the multi-resolution refutation*, while not using C as one of the sources. Observe that $\square$ is always reachable from C, since Π is a non-redundant multi-resolution refutation; thus $\square$ is never input to the SAT solver. We let the SAT solver try to derive $\square$, using $\overline{Re}(\Pi, C)$ as the input formula, or else prove that $\overline{Re}(\Pi, C)$ is satisfiable.

In the latter case, we conclude that C must belong to the minimal unsatisfiable core, since we found a model for an unsatisfiable subset of initial clauses minus C. Hence, if the SAT solver returns *satisfiable*, the algorithm marks C (line 6) and moves to the next initial clause. However, if the SAT solver returns *unsatisfiable*, we cannot simply remove C from F and move to the next clause, since we need to keep a valid multi-resolution refutation for our algorithm to work properly. We describe the construction of a valid refutation (lines 8–13) next.

Let $G = \overline{Re}(\Pi, C)$. The SAT solver produces a new multi-resolution refutation $\Pi'(V^i_G \cup V^c_G, E_G)$ for G, whose sources are the clauses $\overline{Re}(\Pi, C)$. We cannot use Π' as the multi-resolution refutation for the subsequent iterations, since the sources of the refutation may only be initial clauses of F. However, the "superfluous" sources of Π' are conflict clauses of Π, unreachable from C,

and thus are derivable from $V^i \setminus C$ using resolution relations, corresponding to edges of Π. Hence, it is sufficient to augment Π' with such edges of Π that connect $V^i \setminus C$ and $\overline{Re}(\Pi, C)$ to obtain a valid multi-resolution refutation whose initial clauses belong to F. Algorithm CRR constructs a new multi-resolution refutation, whose sources are $V^i \setminus C$; the conflict clauses are $\overline{Re}(\Pi, C) \cup V_G^c$ and the edges are $(E \setminus (V_1, V_2) | (V_1 \in Re(\Pi, C)$ *or* $V_2 \in Re(\Pi, C))) \cup E_G$. This new refutation might be redundant, since $\Pi'(V_G^i \cup V_G^c, E_G)$ is not guaranteed to be non-redundant. Therefore, prior to checking the next clause, we reduce the new refutation to a non-redundant one. Observe that in the process of reduction to a non-redundant subgraph, some initial clauses of F may be omitted; hence, each time a clause C is found not to belong to the minimal unsatisfiable core, we potentially drop not only C, but also other clauses.

We demonstrate the process of completing a multi-resolution refutation on the example in Figure 5.1. Suppose we are checking whether C_1^i belongs to the minimal unsatisfiable core. In this case, $G = \overline{Re}(\Pi, C_1^i) = \{C_2^i, C_3^i, C_4^i, C_5^i, C_6^i, C_7^i, C_2^c, C_4^c\}$. The SAT solver receives G as the input formula. It is not hard to check that G is unsatisfiable. One multi-resolution refutation of G is $\Pi'(V_G^i \cup V_G^c, E_G)$, where $V_G^i = \{C_2^i, C_2^c, C_7^i, C_4^c\}$, $V_G^c = (D_1 = \Box, D_2 = a \vee b)$, and $E_G = \{(C_2^i, D_2), (C_2^c, D_2), (D_2, D_1), (C_7^i, D_1), (C_4^c, D_1)\}$. Therefore, C_1^i, C_1^c, C_3^c, C_5^c and related edges are excluded from the refutation of F, whereas D_2, D_1 and related edges are added to the refutation of F. In this case, the resulting multi-resolution refutation is non-redundant.

We did not define how the function *PickUnmarkedClause* should pick clauses (line 3). Our current implementation picks clauses in the order in which clauses appear in the given formula. Development of sophisticated heuristics is left for future research.

Another direction that may lead to a speed-up of CRR is adjusting the SAT solver for the purposes of the CRR algorithm, considering that the SAT solver is invoked thousands of times on rather easy instances. Integrating the data structures of CRR and the SAT solver, fine-tuning the SAT solver's heuristics for CRR, and holding the refutation in-memory rather than on disk (as suggested in [70] for EC), could be helpful.

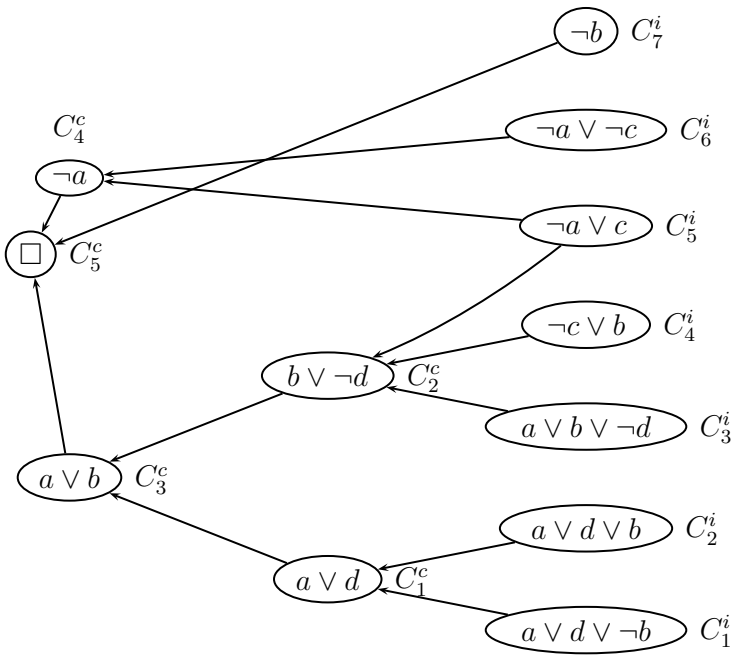


Figure 5.1: Multi-resolution refutation example.

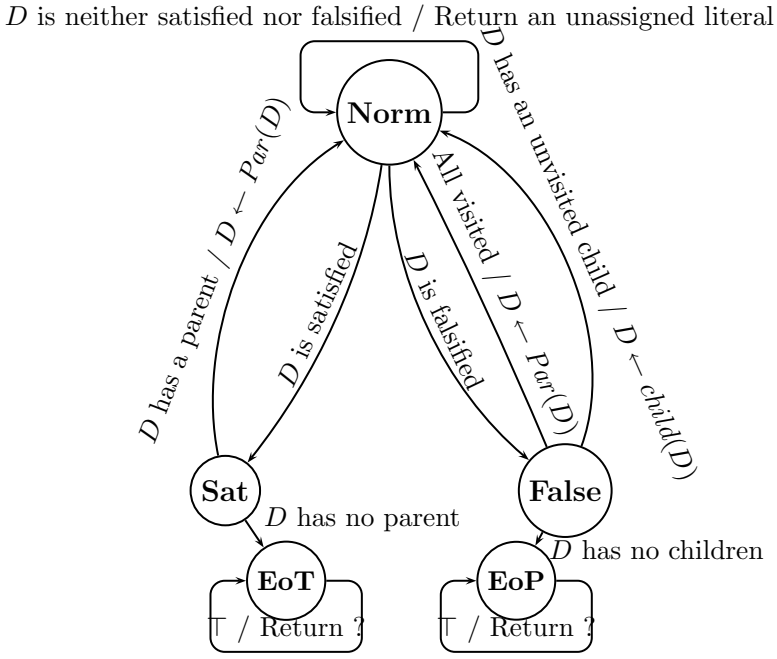


Figure 5.2: Function RRP_Decide represented as a transition relation. This function is invoked by the decision engine of a SAT solver, implementing the RRP pruning technique.

5.4 Resolution-Refutation-Based Pruning

In this section, we propose an enhancement of Algorithm CRR by developing multi-resolution refutation-based pruning techniques for when a SAT solver is invoked on $\overline{Re}(\Pi, C)$ to check whether it is possible to complete a refutation without C. We refer to the pruning technique, proposed in this section, as *Resolution Refutation-Based Pruning (RRP)*.

In this section, we suppose that the SAT solver uses BCP and non-chronological backtracking.

Recall from Definition 4 that an assignment σ *falsifies* a clause C, if every literal of C is *false* under σ. An assignment σ *falsifies* a set of clauses P if every clause $C \in P$ is falsified by σ. We claim that a model for $\overline{Re}(\Pi, C)$ can only be found under such a partial assignment, which falsifies every clause in some path from C to the empty clause in $Re(\Pi, C)$. The intuitive reason is that every other partial assignment satisfies C and must falsify $\overline{Re}(\Pi, C)$, since F is unsatisfiable. A formal statement and proof is provided in Proposition 10 below.

Consider the example in Figure 5.1. Suppose the currently visited clause is C_5^i. Two paths from C_5^i to the empty clause C_5^c exist – namely $\{C_5^i, C_4^c, C_5^c\}$ and $\{C_5^i, C_2^c, C_3^c, C_5^c\}$. A model for $\overline{Re}(\Pi, C_5^i)$ can only be found in a subspace under the partial assignment $\{a = 1, c = 0\}$, falsifying all the clauses of the first path. The clauses of the second path cannot be falsified, since a must be 1 to falsify clause C_5^i and 0 to falsify clause C_3^c.

Denote a subtree connecting C and $\Box$ by $\Pi \restriction_C$. The proposed pruning technique, RRP, is integrated into the decision engine of the SAT solver. The solver receives $\Pi \restriction_C$, together with the input formula $\overline{Re}(\Pi, C)$. The decision engine of the SAT solver explores $\Pi \restriction_C$ in a depth-first manner, picking unassigned variables in the currently explored path as decision variables and assigning them 0. As usual, BCP follows each assignment. Backtracking in $\Pi \restriction_C$ is tightly related to backtracking in the assignment space. Both events happen when a satisfied clause in $\Pi \restriction_C$ is found or when a conflict is encountered by the SAT solver. After a particular path in $\Pi \restriction_C$ has been falsified, a general purpose decision heuristic is used until the SAT solver

either finds a satisfying assignment or proves that no such assignment can be found under the currently explored path. This process continues until either a model is found or the decision engine has completed exploring $\Pi\restriction_C$. In the latter case, one can be sure that no model for $\overline{Re}(\Pi, C)$ exists. However, the SAT solver should continue its work to produce a multi-resolution refutation.

We need to describe in greater detail the changes in the decision and conflict analysis engines of the SAT solver required to implement RRP. The decision engine first invokes function *RRP_Decide*, depicted in Figure 5.2, as a state transition relation. Each transition edge has a label consisting of a condition under which the state transition occurs and an operation, executed upon transition. The state can be one of the following:

(**Norm**)	normal;
(**Sat**)	the currently explored clause is satisfied;
(**False**)	the currently explored clause is falsified;
(**EoT**)	subgraph $\Pi\restriction_C$ has been explored;
(**EoF**)	all clauses in the currently explored path are falsified.

The states are managed globally, that is, if *RRP_Decide* moves to state S, it will start in state S when next invoked. A pointer D to the currently visited clause of $\Pi\restriction_C$ is also managed globally. The state transition relation is initialized prior to the first invocation of the decision engine. Pointer D is initialized to C and the initial state is **Norm**.

State **Norm** corresponds to a situation when the algorithm does not know what the status of D is. If D is neither satisfied nor falsified, *RRP_Decide* returns a negation of some literal of D, which will serve as the next decision variable. If D is satisfied, the algorithm moves to **Sat**. Observe that a clause may become satisfied only as a result of BCP. Encountering a satisfied clause means that the currently explored path cannot be falsified, and we can backtrack. Suppose we are in **Sat**, meaning that D is satisfied. If D has a parent, the algorithm backtracks by moving D to point to its parent, and goes back to **Norm**; otherwise, the tree is explored and the algorithm moves to **EoT**. In this case, *RRP_Decide* returns an unknown value and a general purpose heuristic must be used. Consider now the case when the state is

Norm and D is falsified. The algorithm moves to **False**. Here, one of the three following conditions holds:

(a) D has an unvisited child S. In this case D is updated to point to S and *RRP_Decide* moves back to **Norm**.

(b) All children of D are visited. In this case, we backtrack by moving D back to its parent and go back to **Norm**.

(c) D has no children. In this case, all the clauses in the currently explored path are falsified. The algorithm moves to **EoF**; *RRP_Decide* returns an unknown value; and a general purpose heuristic must be used.

To complete the picture, we describe the changes to the conflict analysis engine required to implement RRP. One of the main tasks of conflict analysis in modern SAT solvers is to decide on the backtrack level (recall Definition 30 of a backtrack level on page 34). Let the backtrack level be bl. When invoked in RRP mode, the conflict analysis engine must also find whether it is required to backtrack in $\Pi\restriction_C$, and to which clause. The goal is to backtrack to the highest clause B in the currently explored path in $\Pi\restriction_C$, such that B has unassigned literals. Recall that D is a pointer to the currently visited clause of $\Pi\restriction_C$. Denote by $mdl(D)$ the maximal decision level of D's literals. If $bl \geq mdl(D)$, the algorithm does nothing; otherwise, it finds the first predecessor of D in $\Pi\restriction_C$, such that $bl < mdl(B)$ and sets $D \leftarrow B$.

We found experimentally that the optimal performance for RRP is achieved when it explores $\Pi\restriction_C$ starting from $\square$ and moving toward C (and not vice-versa). In other words, prior to the search, the SAT solver reverses all the edges of $\Pi\restriction_C$ and sets the pointer D to $\square$, rather than to C. (By default, the current version of RRP explores the graph only until a predefined depth of 50.) The next literal from the currently visited clause is chosen by preferring an unassigned literal with the maximal number of appearances in recent conflict clause derivations (similar to Berkmin's [27] heuristic for SAT). The next visited child is chosen arbitrarily. Further fine-tuning of the algorithm is left to future research.

Proposition 10. *Let $\Pi(V^i, V^c)$ be a non-redundant multi-resolution refutation. Let $C \in V^i$ be an initial clause and σ be an assignment. Then, if $\sigma \models \overline{Re}(\Pi, C)$, there is a path $P = \{C, \ldots, C_m^c\}$ in $Re(\Pi, C)$, connecting C to the empty clause[2], such that σ falsifies every clause in P.*

Proof. Suppose, on the contrary, that no such path exists. Then, there exists a satisfiable vertex cut U in Π. However, the empty clause is derivable from U, since it is a vertex cut; thus U is unsatisfiable, a contradiction. □

[2]The empty clause always belongs to $Re(\Pi, C)$, since $\Pi(V^i, V^c)$ is non-redundant.

Table 5.1: Comparing algorithms for unsatisfiable core extraction. Columns **Instance**, **Var** and **Cls** contain instance name (where, p/bl/lm stand for pipe/barrel/longmult), number of variables, and clauses, respectively. The next seven columns contain execution times (in seconds) and core sizes (in number of clauses) for each algorithm (AM is AMUSE). The cut-off time was 24 hours. Column **R. Hd.** contains the relative hardness of the final multi-resolution refutation, produced by CRR+RRP. Bold times are the best among algorithms guaranteeing minimality. Values "to" and "mo" stand for time-out and memory-out.

			Subopt.		**CRR**		**Naïve**		**MUP**	**R.**
Inst	**Var**	**Cls**	**EC**	**EC-fp**	**RRP**	**plain**	**EC-fp**	**AM**	**EC-fp**	**Hd.**
4p	4237		9	171	**3527**	4933	24111	to	to	1.4
		80213	23305	17724	17184	17180	17182			
4p_1_ooo	4647		10	332	**4414**	10944	25074	to	mo	1.7
		74554	24703	14932	12553	12515	12374			
4p_2_ooo	4941		13	347	**5190**	12284	49609	to	mo	1.7
		82207	25741	17976	14259	14192	14017			
4p_3_ooo	5233		14	336	**6159**	15867	41199	to	mo	1.6
		89473	30375	20034	16494	16432	16419			
4p_4_ooo	5525		16	341	**6369**	16317	47394	to	mo	1.6
		96480	31321	21263	17712	17468	17830			
3p_k	2391		2	20	**411**	493	2147	12544	mo	1.5
		27405	10037	6953	6788	6786	6784	6790		
4p_k	5095		8	121	**3112**	3651	15112	to	to	1.5
		79489	24501	17149	17052	17078	17077			
5p_k	9330		16	169	**13836**	17910	83402	to	mo	1.4
		189109	47066	36571	36270	36296	36370			
bl5	1407		2	19	93	**86**	406	326	mo	1.8
		5383	3389	3014	2653	2653	2653	2653		
bl6	2306		35	322	**351**	423	4099	4173	mo	1.8
		8931	6151	5033	4437	4437	4437	4437		
bl7	3523		124	1154	**970**	1155	6213	24875	mo	1.9
		13765	9252	7135	6879	6877	6877	6877		
bl8	5106		384	9660	**2509**	2859	to	to	mo	1.8
		20083	14416	11249	10076	10075				
lm4	1966		0	0	8	**7**	109	152	13	2.6
		6069	1247	1246	972	972	972	976	972	
lm5	2397		0	1	74	**31**	196	463	35	3.6
		7431	1847	1713	1518	1518	1518	1528	1518	
lm6	2848		2	13	**288**	311	749	2911	5084	5.6
		8853	2639	2579	2187	2187	2187	2191	2187	
lm7	3319		17	91	6217	**3076**	6154	32791	68016	14.2
		10335	3723	3429	2979	2979	2979	2993	2979	

5.5 Experimental Results

We implemented CRR and RRP in the Eureka SAT solver [48]. We used benchmarks from four well-known unsatisfiable families, taken from bounded model checking (*barrel*, *longmult*) [7] and microprocessor verification (*fvp-unsat.2.0, pipe_unsat_1.0*) [67]. All the instances we used appear in the first column of Table 5.1. The experiments on the *barrel* and *fvp-unsat.2.0* families were carried out on a machine with 4Gb of memory and two Intel Xeon CPU 3.06 processors. A machine with the same amount of memory and two Intel Xeon CPU 3.20 processors was used for experiments with the *longmult* and *pipe_unsat_1.0* families.

Table 5.1 summarizes the results of a comparison of the performance of two algorithms for suboptimal unsatisfiable core extraction and five algorithms for minimal unsatisfiable core extraction in terms of execution time and core sizes.

First, we compared algorithms for minimal unsatisfiable core extraction, namely, Naïve, MUP, plain CRR, and CRR enhanced by RRP. In preliminary experiments, we found that Naïve demonstrated its best performance on formulas that were first trimmed down by a suboptimal algorithm for unsatisfiable core extraction. We tried Naïve in combination with EC, EC-fp and AMUSE and found that EC-fp is the best front-end for Naïve. In our main experiments, we used Naïve, combined with EC-fp, and Naïve combined with AMUSE. We also found that MUP demonstrated its best performance when combined with EC-fp, while CRR performed the best when the first refutation is constructed by EC, rather than EC-fp. Consequently, we provide results for MUP combined with EC-fp and CRR combined with EC. MUP required a so-called "decomposition tree", in addition to the CNF formula. We used the c2d package [14] for decomposition tree construction.

The sizes of the cores did not vary greatly between MUC algorithms, so we concentrate on a performance comparison. One can see that the combination of EC-fp and Naïve outperformed the combination of AMUSE and Naïve, as well as MUP. Plain CRR outperformed Naïve on every benchmark, whereas CRR+RRP outperformed Naïve on 15 out of 16 benchmarks (the

exception being the hardest instance of *longmult*). This demonstrates that our algorithms are justified in practice. Usually, the speed-up of these algorithms over Naïve varied between 4 and 10x, but it was as large as 34x (for the hardest instance of the *barrel* family) and as small as 2x (for the hardest instance of *longmult*). RRP improved performance in most instances. The most significant speed-up of RRP was about 2.5x, achieved on hard instances of the *fvp-unsat.2.0* family. The only family for which RRP was usually unhelpful was *longmult*.

A natural question is why the complex instances of the *longmult* family are hard for CRR, and even harder for RRP. The key difference between *longmult* and other families was the hardness of the resolution proof. The relative hardness of a multi-resolution refutation produced by CRR+RRP varied between 1.4 to 2 for every instance of every family, except *longmult*, where it reached 14.2 for the *longmult7* instance. When the refutation was too complex, the exploration of $\overline{Re}(\Pi, C)$ executed by RRP was too complicated; thus, plain CRR is advantageous over CRR+RRP. Also, when the refutation is too complex, it is costly to perform traversal operations, as required by CRR. This explains why the advantage of CRR over Naïve was as small as 2x.

Comparing CRR+RRP on one side and EC and EC-fp on the other, we find that CRR+RRP always produced smaller cores than both EC and EC-fp. The average gain on all instances of cores produced by CRR+RRP over cores produced by EC and EC-fp was 53% and 11%, respectively. The biggest average gain of CRR+RRP over EC-fp was achieved on the *fvp-unsat.2.0* and *longmult* families (18% and 17%, respectively). Unsurprisingly, both EC and EC-fp were usually much faster than CRR+RRP. However, for the three hardest instances of the *barrel* family, CRR+RRP outperformed EC-fp in terms of execution time.

Chapter 6

Conclusion

Chapter 2 of this work proposed a new framework for presenting and understanding the functionality of modern SAT solvers. The framework exploits the inherent relationships between search and resolution. A formulation of the basic backtracking algorithm, including an exact description of the on-the-fly resolution refutation creation, has been provided. We called this formulation the SAT Solver Skeleton (SSS). We introduced a notion of a parent resolution derivation: a resolution proof for the validity of each flip operation. Chapter 3 demonstrated that the notion of parent resolution derivation is useful for analyzing and improving modern SAT solvers.

Another important feature of our framework in Chapter 2 is that it defined all the modern algorithms for conflict-driven learning, including 1UIP-based conflict-directed backjumping, non-chronological backtracking and conflict clause recording, without using the notion of an implication graph. Instead, our approach is based on resolution.

We also showed in Chapter 2 how to augment SSS with each one of the following six enhancements, used already in the Chaff-2001 SAT solver [45] and widely used in modern SAT solvers. These techniques include[1]:

- Boolean Constraint Propagation (BCP) [15]
- Non-Chronological Backtracking (NCB) [60, 3]

[1]The references are to first applications or important milestones of applying these technique in the context of SAT.

- 1UIP-based Conflict-Direct Backjumping (CDB) [60, 3, 45]
- Conflict Clause Recording (CCR) [60, 45]
- Restarts [29]
- Conflict Clause Deletion (CCD) [3]

Chapter 3 formalized the notion of search pruning, relating it to the size of the constructed resolution derivation. We introduced the concepts of backward pruning – the number of resolution derivation nodes skipped during backtracking; and forward pruning – the potential impact on reusing conflict clauses in the subsequent search. We showed that the 1UIP scheme [60, 3, 45] with conflict clause minimization [4, 62] for conflict-driven learning is better than other known schemes in terms of both backward and forward pruning, and explained its empirical advantage over other schemes.

We introduced an enhancement to the 1UIP scheme with minimization, called local conflict clause recording. This algorithm records additional conflict clauses to improve forward pruning by making it less dependent on the polarity selection heuristic. We demonstrated that local conflict clause contributes to the performance of a modern SAT solver.

In addition, we reaffirmed the empirical usefulness of assignment stack shrinking [47, 40], a technique for reducing the size of conflict clauses and removing irrelevant literals from the assignment stack. We showed that assignment stack shrinking contributes to the performance of modern SAT solvers. We also illustrated that the effect of assignment stack shrinking cannot be achieved by using conflict clause minimization and/or rapid restarts.

Chapter 4 presented a novel clause-based heuristic (CBH). This heuristic maintains a clause list organized in a manner that allows the algorithm to choose sequences of interrelated variables that were responsible for recent conflict derivation. CBH maintains both the initial and the conflict clauses in a single list. The next decision literal is picked from the topmost unsatisfied clause in the list. After each conflict, the conflict clause is prepended to the top of the list. Clauses visited during conflict-clause identification are placed just after the new conflict clause. As a variant, if the clause/variable ratio

of the input instance is greater than a predefined value (10 is a reasonable choice), newly identified binary clauses are moved to the top of the list. We demonstrated that using CBH results in a significant performance boost for hard industrial families, when compared with the Berkmin heuristic or VSIDS.

Chapter 5 proposed an algorithm for minimal unsatisfiable core extraction, called CRR. This algorithm builds a resolution refutation using a SAT solver and finds a first approximation of the minimal unsatisfiable core. Then it checks every remaining initial clause C to see whether it belongs to the minimal unsatisfiable core. The algorithm reuses conflict clauses and resolution relations throughout its execution. We demonstrated that our algorithm is faster than currently existing algorithms by a factor of six or more on large problems with non-overly hard resolution proofs, and that it can find minimal unsatisfiable cores for real-world formal verification benchmarks.

Index of Important Terms

Bibliography

[1] J. Alfredsson. The SAT solver Oepir. `http://www.lri.fr/~simon/contest/results/ONLINEBOOKLET/OepirA.ps`.

[2] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2005.

[3] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, 1997.

[4] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

[5] Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In Hoos and Mitchell [30], pages 321–344.

[6] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207, 1999.

[8] R. Bruni and A. Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *Proceedings of the Workshop on Theory and Application of Satisfiability Testing (SAT'01)*, 2001.

[9] Renato Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 547–560. Springer, 2007.

[11] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache conscious data structures for Boolean satisfiability solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:99–120, 2009.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[13] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[14] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence, (ECAI'04)*, pages 328–332, 2004.

[15] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[16] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A clause-based heuristic for SAT solvers. In Fahiem Bacchus and Toby Walsh,

editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2005.

[17] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, 2006.

[18] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In João Marques Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 287–293. Springer, 2007.

[19] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[20] Niklas Eén and Niklas Sörensson. MiniSat v1.13 a SAT solver with conflict-clause minimization, 2005.

[21] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[22] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *AMAI*, 2002.

[23] Allen Van Gelder. Pool resolution and its relation to regular resolution and dpll with clause learning. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 580–594. Springer, 2005.

[24] Allen Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In Kullmann [36], pages 141–146.

[25] Roman Gershman and Ofer Strichman. HaifaSat: A new robust SAT solver. In Ur et al. [65], pages 76–89.

[26] Roman Gershman and Ofer Strichman. HaifaSat: a SAT solver based on an abstraction/refinement model. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:31–51, 2008.

[27] Evgueni Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.

[28] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 886–891, 2003.

[29] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *AAAI '98/IAAI '98: Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[30] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2005.

[31] Jinbo Huang. MUP: A minimal unsatisfiability prover. In *Proceedings of the Tenth Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, pages 432–437, 2005.

[32] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2007.

[33] R. J. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–188, 1990.

[34] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[35] Zurab Khasidashvili, Alexander Nadel, Amit Palti, and Ziyad Hanna. Simultaneous SAT-based model checking of safety properties. In Ur et al. [65], pages 56–75.

[36] Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.

[37] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *J. Autom. Reasoning*, 9(1):25–42, 1992.

[38] Mark H. Liffton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 173–186, 2005.

[39] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[40] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In Hoos and Mitchell [30], pages 360–375.

[41] Vasco M. Manquinho and João P. Marques Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(5):505–516, 2002.

[42] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, pages 2–17, 2003.

[43] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[44] Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João P. Marques Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 467–674, 2005.

[45] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

[46] Alexander Nadel. Jerusat SAT solver. `http://www.cs.tau.ac.il/~ale1/Jerusat1.3.tgz`.

[47] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew Univeristy of Jerusalem, Jerusalem, Israel, November 2002.

[48] Alexander Nadel, Moran Gordon, Amit Palti, and Ziyad Hanna. Eureka-2006 SAT solver. `http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf`.

[49] Gi-Joon Nam, Fadi Aloul, Karem Sakallah, and Rob Rutenbar. A comparative study of two Boolean formulations of FPGA detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design (ISPD'01)*, pages 688–696, 2001.

[50] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–

Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[51] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41th Design Automation Conference (DAC'04)*, pages 518–523, 2004.

[52] C. H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). In *Proceedings of the Fourteenth Annual ACM Symposium on the Theory of Computing (STOC'82)*, pages 255–260, 1982.

[53] Knot Pipatsrisawat and Adnan Darwiche. RSat SAT solver. `http://reasoning.cs.ucla.edu/rsat/`.

[54] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

[55] Patrick Prosser. Hybrid algorithms for the constraint satisfaction. *Computational Intelligence*, 9(3):268–299, 1993.

[56] Lawrence O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004.

[57] Vadim Ryvchin and Ofer Strichman. Local restarts. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 271–276. Springer, 2008.

[58] Ofer Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70, London, UK, 2001. Springer-Verlag.

[59] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 62–74. Springer, September 1999.

[60] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[61] Carsten Sinz. SAT-Race 2006. `http://fmv.jku.at/sat-race-2006/`.

[62] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Kullmann [36], pages 237–243.

[63] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell.*, 9(2):135–196, 1977.

[64] Stefan Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, 2004.

[65] Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors. *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, volume 3875 of *Lecture Notes in Computer Science*. Springer, 2006.

[66] Miroslav N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, pages 28–35, 2002.

[67] M.N. Velev and R.E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 226–231, 2001.

[68] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275. Springer-Verlag, 1997.

[69] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. IEEE Press, 2001.

[70] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Preliminary Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.

ב. אם הפסוקיות אינן ספיקות, אזי C אינו שייך ללמב"ס. אפוא, החלף את Π ברב הפרכה ברזולוציה תקפה אשר אינה כוללת את C.

4. סיים כאשר כל פסוקיות הקלט, המקושרות לפסוקית הריקה מהוות למב"ס.

מחבר תזה זו הינו המחבר הראשי של המאמרים [16,17,18]. כמו כן, הוא השתתף בעבודות על ספיקות בו-זמנית בבדיקת מודלים [35] ועל ספיקות מערכי סיביות [10], אשר אינן מוצגות בתזה זו.

אכן יותר דינאמי מסיבתב"ה, אך לטענתנו, יתרון נוסף של מסייע ברקמין על סיבתב"ה הוא בכך שהוא נוטה לבחור משתנים קשורים – משתנים שבחירתם המשותפת מעלה את הסיכוי להגיע לסתירה בענפים אי-ספיקים וגם לספק פסוקיות במהרה בענפים ספיקים. ברם, יתרון זה איננו מנוצל עד תום בגלל העובדה שמסייע ברקמין איננו מוסיף לרשימה את פסוקיות הקלט ומשתמש במסייע משני דמוי סיבתב"ה. אנו מציעים מסייע ששמו מסייע מבוסס פסוקיות (ממ"פ), אשר מחזיק רשימה הכוללת הן פסוקיות קלט והן פסוקיות סתירה ומעלה בכך את הסבירות לבחירת משתנים קשורים. משתנה ההחלטה הבא נבחר מן הפסוקית הלא ספיקה העליונה. אין צורך במסייע משני. אנו מציעים שיטות שונות לארגון ראשוני של הרשימה וכן להזזת פסוקיות בתוך הרשימה. גישתנו גורמת להתייעלות ביצועים משמעותית בהשוואה לסיבתב"ה ולמסייע ברקמין.

בעיה שכיחה המתעוררת עבור נוסחאות בלתי ספיקות הינה מציאת הליבה הבלתי ספיקה – תת-קבוצה בלתי ספיקה של פסוקיות הקלט. דוגמאות ליישומיים לבעיה זו הן אימות פונקציונאלי של חומרה [43] ,ניתוב במערך שערים בר תכנון בשדה [49], עידון ההפשטה [42]. ליבה בלתי ספיקה הינה ליבה מזערית בלתי ספיקה (למב"ס), אם כאשר מוציאים פסוקית ממנה, היא הופכת לספיקה. תמיד רצוי למצוא ליבה בלתי ספיקה שהיא גם מזערית, אך בעיית מציאת למב"ס ידועה להיות מאוד קשה חישובית (הינה D^P-שלמה [52]). פרק 5 מציג אלגוריתם המסוגל למצוא למב"ס לבעיות מעשיות גדולות במיוחד מתחום אימות צורני של חומרה. העבודות [70] ו-[28] הציעו בצורה בלתי תלויה את הגישה היחידה היעילה מעשית למציאת ליבה בלתי ספיקה (אך לא בהכרח מזערית) עבור סימני מדידה גדולים מתחום האימות הצורני של חומרה. אנו מתייחסים לשיטה זו בשם חרוט פסוקית ריקה (חפ"ר). חפ"ר עושה שימוש ביכולת של פותרי ספיקות עכשוויים לחולל הפרכה ברזולוציה בהינתן נוסחא בלתי ספיקה. חפ"ר נעה לאחור על ההפרכה ההפוכה החל מן הפסוקית הריקה ומחזירה את פסוקיות הקלט המחוברות לפסוקית הריקה בתור הליבה הבלתי ספיקה. אלגוריתם, שאנו מתייחסים אליו בשם חפרנ"ק, מפעיל את החפ"ר עד לנקודה קבועה, אך ניתן לצמצם את הליבה המחוללת עוד יותר גם לאחר הפעלת חפרנ"ק. התזרים הבסיסי של אלגוריתמנו למציאת ליבה מזערית בלתי ספיקה הינו כדלקמן:

1. חולל רב הפרכה ברזולוציה Π של נוסחה בלתי ספיקה נתונה באמצעות הפעלת פותר ספיקות.
2. השמט את כל הפסוקיות הלא מקושרות לפסוקית הריקה מ-Π. כעת, כל פסוקיות הקלט שלא הושמטו מהוות ליבה בלתי ספיקה.
3. עבור כל פסוקית קלט C אשר נשארה ב-Π, בדוק האם היא שייכת ללמב"ס באופן הבא: הורד את C מ-Π יחד עם כל פסוקיות הסתירה שגזירתם כוללת את C. מסור את שארית הפסוקיות (כולל גם את פסוקיות הסתירה) לפותר ספיקות.

    א. אם הן ספיקות, אזי C שיים ללמב"ס וניתן לעבור לפסוקית הקלט הבאה

הפסוקית ההורית – לכל היפוך. מרבית התוצאות המוצגות בפרקים 2 ו-3, הושגו ע"י ניתוח ההשפעה של אלגוריתמי למ"ס על הבנייה של הרזולוציה ההורית.

פרק 3 עושה שימוש בגישה החדשה לבעיית הספיקות, אשר מתוארת בפרק 2, בכדי להבין ולשפר את אלגוריתם הלמידה מונחית סתירות של פותר ספיקות עכשווי. אנו מצרינים את המושג השימושי של גיזום החיפוש [41,58] ע"י הגדרתו כיכולת של שיטת למ"ס מסוימת לצמצם את מספר הצמתים בהפרכה ברזולוציה המחוללת ע"י האלגוריתם. גיזום החיפוש משמש כאמת מידה למדידת ההשפעה של שיטות למ"ס שונות. אנו מבדילים בין גיזום אחורי – מספר צמתי הרזולוציה אשר תהליך הנסיגה דילג מעליהם; וגיזום קדמי – ההשפעה האפשרית של שימוש חוזר בפסוקיות סתירה בהמשך החיפוש. כמו כן, אנו מבדילים בין שלושה סוגים שונים של גיזום אחורי – גיזום אחורי ברזולוציה, בנא"י ובנסיגה אי-כרונולוגית (נא"כ). אנו מפגינים כי שיטת הנא"י1 המשופרת ע"י צמצום גוברת על שיטות אחרות הן מבחינת הגיזום האחורי והן מבחינת הגיזום הקדמי. עובדה זו מהווה הסבר ליתרון הניסויי של שיטת הנא"י1 על שיטות אחרות.

אנו גם מציעים שיטה חדשה המשפרת את שיטת הנא"י1 המצומצמת, אשר נקראת הקלטת פסוקיות סתירה מקומיות ומפגינים את תועלתיותו על סימני מדידה תעשייתיים. הרעיון מאחורי הקלטת פסוקיות סתירה מקומיות היא לגרום לכך שהקלטת פסוקיות סתירה תהיה פחות תלויה במסייע עבור בחירת קיטוב משתנים.

כמו כן, אנו מאמתים את התוצאות של [40] ,לפיהם שיטת צימצום מחסנית ההשמות, אשר הוצגה ע"י מחבר עבודה זו ב-[47], משפרת את הביצועים של פותר ספיקות חדשני. אנו מראים כי השפעת צימצום מחסנית ההשמות גדולה יותר מהשפעת צמצום פסוקיות הסתירה והתחדשויות מקומיות.

מסייע החלטה יעיל ככל האפשר הינו רכיב קריטי לביצועיו של פותר ספיקות מודרני. מסייע ההחלטה קובע את משתנה הבחירה ואת ערכו הבוליאני בכל נקודת החלטה. פרק 4 מציג מסייע החלטה חדש, אשר נמצא להיות יעיל על סימני מדידה תעשייתיים. הוא תוכנן במטרה להעלות את הסבירות שמשתנים קשורים ייבחרו בסמיכות. לאחרונה, היינו עדים לפריצת דרך בתכנון מסייעי החלטה יעילים על דוגמאות תעשייתיות אמיתיות. תצפית המפתח היא שעל מסייעי ההחלטה להיות דינאמיים, כלומר עליהם לחזור ולמקד את החיפוש על פסוקיות סתירה אשר נגזרו זה לא מכבר. סכום יורד בלתי תלוי במצב המשתים (סיבתב"ה) [45] – המסייע הדינאמי הראשון – מחזיק ניקוד עבור כל ליטרל. ניקוד הליטרל עולה באחד, כאשר הוא מופיע בפסוקית סתירה. מעת לעת, מחלקים את ניקודי הליטרלים בשניים. אסטרטגיה זו מבטיחה שהפותר בוחר בליטרלים שהשתתפו בגזירת פסוקיות סתירה אחרונות. מסייע החלטות ידוע נוסף, אשר נמצא יעיל יותר מסיבתב"ה על סימני מדידה תעשייתיים, הינו זה של פותר הספיקות ברקמין [27]. יוצריו טענו, כי סיבתב"ה איננו דינאמי דיו, במובן שהוא עלול לבחור במשתנים הלא רלוונטיים לענף הנחקר באותו זמן. הם הציעו לארגן את פסוקיות הסתירה ברשימה ולבחור את משתנה ההחלטה הבא מן הפסוקית הלא מסופקת העליונה ברשימה. במקרה שפסוקית כזאת לא קיימת יש להשתמש במסייע משני דמוי סיבתב"ה. מסייע ברקמין הוא

תקציר

בעיית הספיקות עבור לוגיקה פסוקית הינה בעיית ההכרעה האם קיימת השמה מספקת למשתני נוסחא נתונה בלוגיקה פסוקית. בעיית הספיקות תופסת מקום מרכזי במשפחה גדולה של בעיות NP-שלמות, אשר ככל הנראה, אינן ניתנות לחישוב יעיל. לכן, נראה כי לא קיים אלגוריתם המסוגל לפתור את הבעיה בזמן סביר בכל המקרים. ובכל זאת, קיימים אלגוריתמים המסוגלים לפתור נוסחאות הנובעות מבעיות תעשייתיות באופן מהיר ויעיל. ישנם יישומים רבים לבעיית הספיקות באימות צורני, בינה מלאכותית ותחומים רבים אחרים במדעי המחשב והנדסת מחשבים.

פותרי ספיקות חדישים, כגון צ'ף [45], מיניסט [19] ואיריקה [48], מבוססים על גרסה משופרת של אלגוריתם החיפוש בנסיגה של דויס, לוגמן ולובלנד (דל"ל) [16]. דל"ל נחקר ושופר במשך השנים, אך פריצת דרך בביצועי פותרי ספיקות על סימני מדידה תעשייתיים מתחום האימות הצורני נעשתה ע"י יוצרי פותרי הספיקות גרספ [60] ורלסט [3]. גרספ ורלסט כללו מספר חידושים באלגוריתם הנסיגה, אשר הוצגו יחד תחת הכותרת "ניתוח סתירה". פותר הספיקות צ'ף [45] עידן חידושים אלה בצורה ניכרת. פותרי ספיקות עכשוויים מסוגלים להתמודד עם נוסחאות תעשייתיות הכוללות מיליוני פסוקיות ומשתנים.

האלגוריתם של הפותר צ'ף, אשר ממומש במרבית הפותרים החדישים, איננו מובן במלואו. ניתן לראות את צ'ף הן כאלגוריתם לחקירת מרחב ההשמות באמצעות בניית עץ חיפוש והן כאלגוריתם אשר בונה הפרכה ברזולוציה של נוסחא נתונה. לא היה ברור, כיצד ניתן לחבר בין שתי הגישות או, במילים אחרות, כיצד ניתן לנסח את האלגוריתם של צ'ף כך שניתן יהיה לעקוב אחר תהליכי חקירת מרחב ההשמות ובניית ההפרכה ברזולוציה גם יחד. מקור אחר להעדר הבהירות הייתה חוסר הבנה של האלגוריתם בעל העוצמה הרבה של למידה מונחית סתירות (למ"ס). ראה, למשל, את הטענה של יוצרי צ'ף, אשר נאמרה בהקשר של השוואה של שיטות למ"ס שונות, כדלקמן: "ניתן לקבוע את היעילות של שיטות חיפוש שונות אך ורק באמצעות מידע ניסיוני" [69].

פרק 2 של עבודה זו מציע מסגרת חדשה להצגה וניתוח של פותר ספיקות מודרני, המגשרת בין הגישה המבוססת חיפוש לבין הגישה המבוססת רזולוציה. מטרתנו היא לספק הצרנה נוחה עבור המחקר המעשי בנושא של בעיית הספיקות. אנו מראים כיצד ניתן להוסיף צעד-אחר-צעד את רכיבי אלגוריתם הלמ"ס של צ'ף לדל"ל הבסיסי בצורה מבודדת ובלתי-תלויה. התוצר הינו אלגוריתם הלמ"ס של צ'ף. צ'ף משייך פסוקית הורית לכל פעולת היפוך, אשר מהווה סיבה מספקת להיפוך המשתנה. למעשה, המטרה של אלגוריתם ניתוח הסתירה של צ'ף היא הסקת הפסוקית ההורית. אנו מציעים לשייך רזולוציה הורית – נגזרת רזולוציה של

ולבסוף, אנו מציגים שיטת חישוב חדשה עבור בעיית הוצאת ליבה מזערית בלתי ספיקה, אשר מסוגלת למצוא ליבה מזערית עבור נוסחאות תעשייתיות גדולות. משפחות הבעיות המופיעות באימות צורני של חומרה מעוררות עניין מיוחד עבורנו. פותר ספיקות חדיש מסוגל לייצר הפרכה ברזולוציה של נוסחה בלתי ספיקה נתונה, אשר מקוריה הן פסוקיות הקלט וכיורה הוא הפסוקית הריקה. שיטתנו בוחנת את פסוקיות הקלט המקושרות לפסוקית הריקה זו אחר זו ומסירה כל פסוקית הניתנת להסרה מן ההפרכה ברזולוציה. שיטתנו שומרת על נכונות ההפרכה ע"י השלמתה עם פסוקיות אחרות וקשרי רזולוציה אחרים. בסופו של התהליך, כל פסוקיות הקלט המקושרות לפסוקית הריקה מהוות ליבה מזערית בלתי ספיקה.

תמצית

בעיית הספיקות עבור לוגיקה פסוקית הינה בעיה NP-שלמה אשר תופסת מקום מרכזי במדעי המחשב (ראה, למשל, [21,12]). לבעיית הספיקות ישנן שימושים רבים באימות צורני [54], בינה מלאכותית [34] ותחומים נוספים. פותרי ספיקות עכשוויים, המשתמשים בגרסה משופרת של אלגוריתם החיפוש בנסיגה דויס-לוגמן-לובלנד (דל"ל) [15], מתמודדים בהצלחה עם נוסחאות בעלות מיליוני משתנים ופסוקיות. עבודה זו היא ניסיון לשפוך אור חדש על התפקודיות של פותר ספיקות חדשני. כמו כן, אנו מציעים מספר שיפורים מעשיים המתאימים במיוחד לתחום האימות הצורני.

ראשית, אנו מציעים מסגרת חדשה להצגה וניתוח של פותרי ספיקות חדישים מבוססי נסיגה. גישתנו עושה שימוש בקשר האינהרנטי בין הנסיגה לרזולוציה. אנו מראים צעד אחר צעד כיצד ניתן ליצור פותר ספיקות חדיש מאלגוריתם הדל"ל. גישתינו מבוססת על המושג של רזולוציה הורית – ההוכחה ברזולוציה לתקפות של כל היפוך.

שנית, אנו מגדירים אמת מידות חדשות להערכת ההשפעה המעשית של שיטות שונות עבור למידה מונחית סתירות: גיזום אחורי – מספר צמתי הרזולוציה אשר תהליך הנסיגה דילג מעליהם; וגיזום קדמי – ההשפעה האפשרית של שימוש חוזר בפסוקיות סתירה בהמשך החיפוש. אנו מראים כי שיטת נקודת אימפליקציה ייחודית 1 (נא"י1) [45] המוגברת ע"י צמצום פסוקיות הסתירה [4,62] עבור למידה מונחית סתירות טובה יותר הן מבחינת הגיזום האחורי והן מבחינת הגיזום הקדמי. עובדה זו מהווה הסבר ליתרון הניסיוני של שיטת ה-נא"י1 המצומצמת על שיטות אחרות.

שלישית, אנו מציעים שיפור יעיל מעשית לשיטת ה-נא"י1 הנקרא הקלטת פסוקיות סתירה מקומיות. שיטה זו משפרת את שיטת הנא"י1 המצומצמת ע"י הוספת פסוקיות חדשות. הקלטת פסוקיות סתירה מקומיות גורמת ללמידה להיות פחות תלויה במסייע עבור בחירת קיטוב משתנים.

רביעית, אנו מראים כי שיטת צימצום מחסנית ההשמות הינה שיטה המשפרת את הביצועים של פותר ספיקות חדשני. אנו משווים את ההשפעה של צימצום מחסנית ההשמות להשפעה של צמצום פסוקיות סתירה והתחדשויות מקומיות.

חמישית, אנו מציעים מסייע החלטות חדש לבעיית הספיקות, הנקרא מסייע מבוסס פסוקיות. מסייע זה תוכנן בכדי להעלות את הסבירות לבחירה מקורבת של משתנים קשורים. הוא מחזיק רשימת פסוקיות הכוללת פסוקיות התחלתיות (פסוקיות קלט) ופסוקיות סתירה כאחד. ליטרל ההחלטה הבא נבחר מן הפסוקית הבלתי ספיקה העליונה. אנו מציעים שיטות מגוונות לארגון ראשוני של רשימת הפסוקיות ולהזזת הפסוקיות בתוך הרשימה. גישתנו מובילה להתייעלות ביצועיים ניכרת על בעיות תעשייתיות אמיתיות בהשוואה למסייעים קיימים.

אוניברסיטת תל אביב TEL AVIV UNIVERSITY
הפקולטה למדעים מדויקים ע"ש ריימונד ובברלי סאקלר
בית הספר למדעי מחשב ע"ש בלבטניק

הבנה ושיפור של פותר ספיקות חדשני

חיבור לשם קבלת תואר "דוקטור לפילוסופיה"
מאת

נאדל אלכסנדר

עבודה זו נעשתה בהדרכתו של

פרופסור נחום דרשוביץ

הוגש לסנאט של אוניברסיטת תל אביב
אוגוסט 2009