TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

# Satisfiability with Hints

Thesis submitted as partial fulfilment of the requirements towards the M.Sc. degree

by

**Jonathan Kalechstain**

This work was carried out under the supervision of

**Professor Nachum Dershowitz**

August 2015

## Preface/Acknowledgement

I would first like to thank my supervisor prof. Nachum Dershowitz for many fruitful meeting and discussions we took together on accomplishing this paper and thesis. When things went hard in the first few months, Nachum insisted this was a good idea and gave me the motivation to continue this work. Special thanks to Vadim Ryvchin, who was the engine behind this thesis. Vadim brought up the original idea, and advised on how to accomplish it. Vadim and Nachum participated in writing the paper "Hints Revealed" which was accepted to SAT 2015. Many thanks to Alexander Nadel and prof. Ofer Strichman for reviewing the paper before it was submitted to SAT. And last thanks are for my dear parents Moshe and Ofra, whose support in my undergraduate and graduate studies was indispensable.

## Abstract

This thesis proposes a notion of *hints*, clauses that are not necessarily consistent with the input formula. The goal of adding hints is to speed up the SAT solving process. For this purpose, an efficient general mechanism for hint addition and removal is provided. When a hint is determined to be inconsistent, a hint-based partial resolution-graph of an unsatisfiable core is used to reduce the search space. The suggested mechanism is used to boost performance by adding generated hints to the input formula. described are two specific hint-suggestion methods, one of which increases performance by 30% on satisfiable SAT '13 competition instances and solves 9 instances not solved by the baseline solver.

# Contents

# List of Tables

# List of Figures

# List of Algorithms

# Chapter 1

# Introduction

Modern backtrack search-based SAT solvers are indispensable in a broad variety of applications [5]. In a classical SAT interface, the solver is given one formula in conjunctive normal form (CNF) and determines whether it is satisfiable or not. Performance of SAT solvers has improved dramatically over the past years [23]. The main advancements came as result of developing new heuristics for existing conflict-driven clause-learning (CDCL) solver techniques, like deletion strategies, decision heuristics, and restart strategies (plus preprocessing and in-processing).

In this work, we propose and investigate a novel method for cutting the search space explored by the SAT solver so as to help it reach a solution faster. The idea is to add *hints*, clauses that are not necessarily "correct", in the sense that they are not necessarily implied by the original input formula.

We call our hint-addition platform HSat (Hint Sat), and present two variants that have been implemented in HaifaMUC [21]. HaifaMUC is an adaptation of MiniSat 2.2 [8], which we will henceforth refer to it as Base.

The addition of hints $H$ to the original formula $F$ creates an extended formula $F'$. Hints can, of course, affect the satisfiability of the formula. As long as $H$ is implied by $F$, the extended formula $F'$ will be equi-satisfiable with the original $F$ (either both are satisfiable or neither is). This means that if $F$ is satisfiable but $F'$ is not, then there must be a contradiction between the added hints and the original formula.

In HSat, we try to solve only the extended formula $F'$. In case it is satisfiable, we are done, and the solver declares that the original formula was likewise satisfiable. Otherwise, the extended formula is unsatisfiable, in which case we need to understand whether the hints are the cause of unsatisfiability, that is, whether any hint is a necessary part of the proof of the empty clause. This is accomplished by an examination of the resolution graph that is built during the run of the solver on $F'$. In [22], the authors presented an efficient way (their "optimization **A**") of saving a partial resolution with respect to a given subset of input clauses. We use this ability to restrict tracking so that only the effects of hints are recorded in the partial graph. Marking clauses to track their origin is an old idea used in Chaff [15] and later reintroduced in [27], and is well adapted to cases when tracking of clauses is required. When the extended formula is unsatisfiable, we check the cone of the empty clause. If it includes a hint, then the status of the original formula remains unknown and additional operations are required (like deletion of the hints). Otherwise, the original formula is unsatisfiable, and we are done. Handling of inconsistent clauses was done in several other applications, like parallel solving [12, 14]; our solution differs, having the ability to track the full effect of the partial resolution tree.

In case the result is unknown and the UNSAT core contains only one hint, an additional optimization can be made by using the UNSAT core of the partial resolution graph. Suppose the UNSAT core contains only hint $h$, then $h$ must contradict $F$, and $\neg h$ is implied by $F$. As $\neg h$ is, in this circumstance, a set (conjunction) of unit clauses, each literal in $h$ can be negated and added as a fact to $F$, which will increase the number of facts and reduce the search space to be explored. This optimization can be generalized to include all graph dominators in the partial resolution graph. (See Theorem 9 below.)

We introduce two heuristics for hint generation. The first, "Avoiding Failing Branches" (AFB), is a purely deterministic hint-addition method. The main idea behind it is the same idea that drives restarts in modern SAT solvers, namely, the possibility that the solver is spending too much time on "bad branches", branches that do not contain the satisfying assignment to the problem. Our motivation is to prevent the solver from entering branches that have already been explored. In our algorithm, we describe an *explored branch* that is a subset of decision variables. We pick the most conflict-active decisions and add a hint that explicitly precludes choosing that set again. In this approach, we keep a score for each literal. The score is boosted every time a clause containing it participates in a conflict. The literals with the highest scores are added to a hint in their negated form. The hint is then added right after a restart, and the same set of active decision variables will never be chosen unless the hint is removed.

This approach leads to significantly improved solver times for satisfiable instances.

A second heuristic, "Randomize Hints" (RH), draws a given number of random assignments, and tries to create a set of hints that will contradict the instance. When the solver concludes unsatisfiability, all dominators of the partial resolution graph are extracted, and all literals in all dominators are added as facts in their negated form.

We continue in the next chapter with the formalization and various preliminaries. Chapter. 5 presents the HSAT algorithm, and, in Chapter. 7, we demonstrate its correctness. The two heuristic hint-generation methods of Chapter. 6 are empirically evaluated in Chapter. 8. We conclude and discuss future work in Chapter. 9.

This thesis contains several contributions. An efficient generic mechanism is introduced to add hints, the goal of which is to speed up the solver. It is based on the ability to remove clauses and all the facts derived from them. In HSAT, we use the partial resolution graph of BASE to remove the hints and their effect in case of an unsatisfiable conclusion. In [16–18] and later in [22], it was shown that the alternative, using selector variables for clause removal [9, 19], is inferior to the use of the resolution graph. We extend the path-strengthening technique published in [17]. Instead of using only immediate children of the removed clauses, we use all dominators in the partial resolution graph provided in BASE. We introduce two algorithms for hint generation, one of them (AFB) increasing performance for satisfiable instances by 19–30%. A paper on this work was accepted to the 18th International Conference on Theory and Applications of Satisfiability Testing [1] (SAT 2015).

# Chapter 2

# The Boolean Satisfiability Problem

The Boolean Satisfiability Problem, is the problem of determining whether a given Boolean formula is satisfiable. A Boolean formula is built of Boolean variables (over $\{0, 1\}$ or $\{true, false\}$), their conjunction (denoted by AND or $\wedge$), disjunction (denoted by OR or $\vee$) and negation (denoted by $\neg$). For example, the following formula is a Boolean formula: $(\neg x_1) \vee (x_2 \wedge x_3)$. An assignment is giving truth values to all Boolean variables in the formula. If there exists an assignment under which a formula evaluates to true, we say that the formula is *satisfiable*; otherwise, we say that the formula is *unsatisfiable*. Each clause $c = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_k$ is a disjunction of literals, and each literal $\ell_i$ is either a variable $v$ or its negation $\neg v$. A formula is in *conjunctive normal form* (CNF) if it is a conjunction (multiplication) of clauses. For example, the following formula is in CNF : $(x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_3 \vee x_4)$. A CNF formula is satisfied if and only if (iff) all of its clauses are satisfied. A clause is satisfied iff at least one of its literals are evaluated to true. We are only interested in CNF formulas. This "restriction" is in no way a limitation, since any Boolean formula can be transformed into an equivalently satisfiable CNF formula in polynomial time and size [20].

*SAT* is the language of all satisfiable CNF formulas. *UNSAT* is the language of all unsatisfiable CNF formulas. SAT was proven to be the first known NP-complete language [6]. Because SAT is NP-complete, every decision problem in NP is polynomially reducible to SAT, and can be solved by a *SAT solver*. A SAT solver $S$ is an algorithm that given a CNF formula $\varphi$, determines whether it is satisfiable or not, and if $\varphi$ is satisfiable, $S$ returns a satisfying assignment. Because SAT is an NP-complete problem, and the question of whether $P = NP$ is unknown, there is no known algorithm $S$ that solves all formulas in polynomial time. If the number of variables is n, the number of assignments is $2^n$. A straightforward implementation of a solver $S$ would be to iterate over all assignment, and return "SAT" if a satisfying assignment was found, or "UNSAT" otherwise. This algorithm is intractable for a large number of variables, so several optimizations and heuristics are used to trim the search space. These optimizations and heuristics are brought up in the following chapter.

# Chapter 3

# Modern SAT Solvers

Modern SAT solvers, like MiniSat and Glocuse [3, 8], are Conflict-Driven Clause-Learning (CDCL) solvers, based primarily on decisions, propagations, clause learning and non-chronological backtracking. CDCL solvers also use heuristics like restarts, deletion strategies, decision heuristics etc. To explain how a SAT solver like MiniSat works, we first explain DPLL solvers, the origin of modern SAT solvers.

## 3.1 DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is is a complete backtrack search SAT solver algorithm which uses several optimizations. It was introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland [7].

### 3.1.1 Backtrack Search

The backtrack search algorithm $S$ searches a tree called the *search tree*. Each level of the tree corresponds to decision of a variables value, when going left on a variable $x_i$ corresponds to assigning $x_i = false$ and going right corresponds to assigning $x_i = true$. The number of leaves in the tree is $2^n$ when $n$ is the number of variables, but DPLL will usually traverse a far smaller fraction of that tree. The algorithm $S$ chooses variables in a certain order and incrementally assigns a value to each variable. As long as the partial assignment does not contradict the formula, this process is continued. When the formula is contradicted, $S$ tries to assign the opposite value to the last chosen variable. If the resulting assignment contradicts the formula, there is no point in continuing in that search space and backtracking takes place. The algorithm backtracks to the previous decision, and either assigns the opposite value or backtrack again if both values were tested.

The algorithm continues the search in a similar fashion, assigning, reassigning, unassigning, and backtracking until either a satisfying assignment was found or no satisfying assignment was found and the results is unsatisfiable. The algorithm does not explicitly visits all assignments, since it backtracks once a partial assignment contradicts the formula.

Three trivial but crucial optimizations takes place in DPLL solvers at their early stages:

1. If a clause is of size one, its literal is taken as a fact.

2. Pure literal elimination - If a literal $\ell_i$ occurs with only one polarity in the formula, all clauses containing $\ell_i$ can be removed.

3. If a clause contains a literal and it's negation, then the clause will be satisfiable by all assignments and can be removed.

### 3.1.2 Boolean Constraint Propagation (BCP)

The original DPLL algorithm uses an iterative rule called *unit propagation.* A clause that has all but a single literal assigned to false is called a *unit clause.* In a unit clause, the remaining free literal must be assigned to true, otherwise the formula is falsified. When a unit clause is found, the algorithm immediately assign the free literal to true. Such assignments can result with more unit clauses, and the process continues until no more unit clauses exist or until a contradiction is found (a clause is evaluated to false). This iterative process is called *boolean constraint propagation* and the resulting assignments are called *implications.*

The variables assigned as a result of BCP are called *implication variables*, while the variables assigned by the decision process are called *decision variables.* When a clause is evaluated to false, a *conflict* takes place. When a conflict occurs, either the second possible value of the last decision variable is chosen, or if it has already been chosen then backtracking takes place. In both cases, the implications that followed the last decision are canceled. If no conflict occurs during BCP, a new decision variable is chosen, and BCP is performed again.

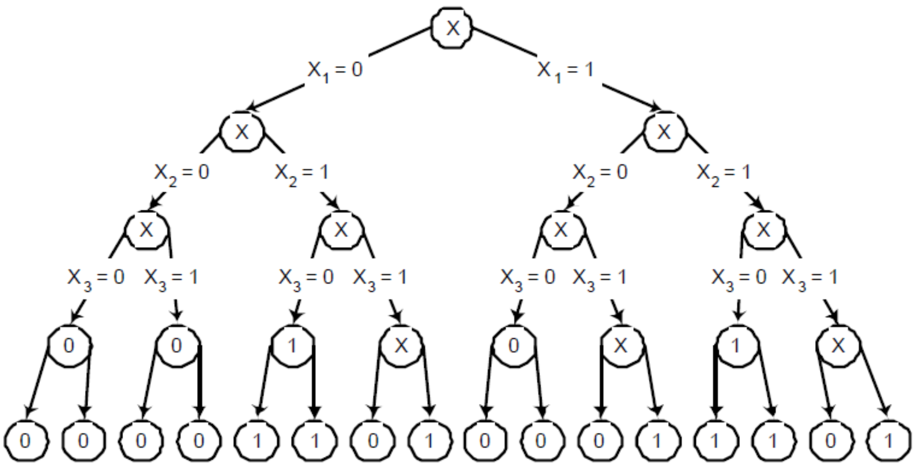

Figure 3.1: Search tree of $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4) \wedge (x_2 \vee x_3)$

The *decision level* is the number of decision variables assigned so far. The decision level is increased when a decision variables is assigned, and decreased when unassigned. An assignment to an implication variable does not change the decision level, the implication variable is associated with the current decision level. By construction, every decision level has a single decision variable and a list of implication variables (the list can be empty).

### 3.1.3 Search Example

Presented is an example of backtracking with BCP. In Figure 3.1 [1], the search tree of the formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4) \wedge (x_2 \vee x_3)$ is presented. The decision order in this example is $x_1, x_2, x_3, x_4$, but this is not compulsory and may change according to algorithm decision heuristic.

The following steps occur in a DPLL solver, when *decision_level* is the current decision level :

1. The decision $x_1 = 0$ is made. $decision\_level = 1$

2. No unit clauses are found and another decision is taken, $x_2 = 0$ is decided. $decision\_level = 2$

3. The clause $(x_1 \vee x_2 \vee \neg x_3)$ becomes unit, so $x_3$ is assigned to false. However, $(x_2 \vee x_3)$ is falsified. Backtracking is needed. $decision\_level = 1$

[1] Taken from Yulik Feldman's thesis, *Parallel Multithreaded Satisfiability Solver*, Tel Sviv University, 2005.

4. The decision $x_2 = 1$ is made. $decision\_level = 2$

5. No unit clauses are found and another decision is taken, $x_3 = 0$ is decided. $decision\_level = 3$

6. All clauses are satisfied and "SAT" is returned.

## 3.2 CDCL

Conflict-Driven Clause Learning (CDCL) is an advanced algorithm to solve the boolean satisfiability problem, primarily based on DPLL solvers. It was proposed in 1996 by Marques-Silva and Sakallah [25]. In CDCL SAT solvers, when a conflict takes place, *conflict analysis* is performed. In conflict analysis, the reason for the conflict is computed and added as a *conflict clause*. The conflict clause prevents the solver from returning to the same branch that caused the conflict.

As explained, conflict clauses are built during conflict analysis. The conflict clause is built by analyzing a data structure called the *implication graph*. After creating the conflict clause, backtracking takes place. However, instead of backtracking to the last decision level, the algorithm will usually backtrack to a lower decision level. This backtracking procedure is called *non-chronological backtracking*. Conflict analysis with non-chronological backtracking is one of the most important optimizations ever brought to modern SAT solvers, improving it by order of magnitude. We explain the process in more details in the coming sections.

### 3.2.1 Implication Graph

An implication graph is a directed acyclic graph (dag) that is implicitly created by recording the reason for every implication taken. Analysis of the implication graph takes place when a conflict occurs. In Figure 4.1 an implication graph is described. Each node $v$ represents an assignment or implication. An edge $u \to v$ exists if $u$ was one of the literals that became false in the unit clause that forced assigning $v$ to true. Note that more nodes from lower decision levels exist, however we are only concerned with the ones that are the cause of an implication at the current decision level. Therefore, decision variables or implication variables from lower decision levels have no incoming edges. In Figure 4.1, gray nodes represent decision variables or implications from lower decision levels, while black nodes represent implications from the current decision level.

Writing $v_i = a@b$ means that $v_i$ was assigned to $a$ at decision level $b$. The last node (on the right) is referred to as the *conflict node*, while the node whose decision led to a conflict is referred to as the *decision node*.

For example, the first decision was assigning $v_1 = 1$. Clause $c_1$ became unit, resulting with the implication $v_2 = 1$. Because $v_9$ was assigned to 0 at a lower decision level, another unit clause, $c_2$, leads to an implication $v_3 = 1$. Both implications turn $c_3$ into a unit clause, and lead to the implication $v_4 = 1$. This procedure is part of BCP, which continues until a clause becomes unsatisfied, in this case $c_6$.

In an implication graph, node $x$ is said to dominate node $y$ if all paths from the decision node to $y$ go through $x$. A *unique implication point* (UIP) is a node that dominates the conflict node. Intuitively, UIPs are points that can replace the decision node and still result with the same conflict. Note that the decision node is always a UIP. In Figure 4.1, $v_1$ and $v_4$ are UIPs. UIPs are ordered starting from the conflict, so $v_4$ is the first UIP and $v_1$ is the second UIP.

**Cut.** A *cut* in the implication graph is defined by a partition of the nodes into two sides. A cut is considered legal if one side contains the conflict node—the *conflict side*, while the other side contains all gray (decision) nodes—the *reason side*. For example, Figure 4.1 shows a cut where the conflict node, $v_5$, and $v_6$ are on the conflict side, while all other nodes are on the reason side.

### 3.2.2 Conflict Clauses

Conflict clauses are generated by finding a cut in the implication graph. Each cut corresponds to a disjunction of negation of the literals whose edges cross from the reason side to the conflict side. Since the conflict node can be deduced from all nodes in the reason side who have edges to the conflict side, the corresponding conflict clause prevents from entering the same problematic search space again. For example, the cut in Figure 4.1 represents the conflict clause $v_{10} \vee \neg v_4 \vee v_{11}$. In [15], it was shown that the best clause (best means minimize run-time) is the one created by finding the cut in the implication graph that has all nodes assigned after the first UIP (1UIP) in the conflict side, and all other nodes in the reason side. This cut is represented by a vertical line in Figure 4.1.

### 3.2.3 Non-Chronological Backtracking

In Section 3.2.2 we discussed conflict analysis and the construction of the first UIP conflict clause $U$.

$U$ will always have a single literal $\ell_U$ from the current decision level $d_U$, and other literals from lower decision levels. Denote $max_U$ to be the maximal decision level of literals of $U$ except $\ell_U$. If we were to backtrack to level $max_U$ (without canceling the decision of that level), then BCP will assign $\ell_U = 1$ and continue. Since all decisions between $max_U$ and $d_U$ had nothing to do with the conflict, the search continues from $max_U$. The decisions are then taken with respect to the used decision heuristic and implications resulted from the new conflict clause. In Figure 4.1, $max_U = 3$. The algorithm backtracks to level 3, assigns $v_4 = 0$ as an implication and continues with BCP.

### 3.2.4 Decision Heuristics

The strategy of deciding which variable to assign next is called a *decision strategy*. These strategies can be based on random selection, static heuristics or dynamic heuristics. A branching heuristic is considered good if it helps finding the satisfying assignment in branches that contain such an assignment, and finds good conflicts in branches that do not contain such an assignment. It is important that the overhead of the decision process be minimal to avoid slowing the entire algorithm.

Some examples for static heuristics are Jeroslaw-Wang [13], Literal Count [24], MOM (Maximal occurence on clauses of Minimal size) etc. A major drawback of these strategies is the big overhead they require. Another disadvantage is that they do not use the information gathered during conflict analysis. Example of heuristics that use conflict analysis information are VSIDS (Variable State Independent Decaying Sum) [15], VOX (Variable Ordering Extension) and the Berk-min heuristic [10]. In [10, 15], it was shown that decision heuristics based upon conflict analysis are faster by order of magnitude.

We describe VSIDS, the first dynamic decision strategy, introduced by Moskewicz in 2001 [15]. According to VSIDS, each literal $\ell$ is associated with a counter, whose value represents the number of clauses containing $\ell$. The initial counter for all literals is 0. When a conflict clause containing $\ell$ is added, the counter of $\ell$ is updated. The algorithm periodically decreases the value of all counters. The literal to be picked is the one with the highest score, when ties are broken randomly. Decreasing the value of counters every once in a while gives preference to literals participating in recent conflicts. The intuition behind this strategy is trying to satisfy the recent conflict clauses. This strategy is considered dynamic as it updates itself according to recent changes in the clause database. It has an extremely low overhead, because grades can be updated during conflict analysis.

### 3.2.5 Restarts

Most CDCL SAT solvers will use a strategy called *restart strategy*. In a global restart strategy, all decisions are canceled and the solver backtrack to its original base decision level. Information from the run prior to the restart is recorded in the form of conflict clauses. The logic behind this strategy is that it helps the solver avoid spending too much time in branches that have no easy-to-find satisfying assignment, or branches that do not lead to learning of strong conflict clauses. In [11] it was shown that restarts are very effective in real-world SAT instances.

Most restart strategies rely on a global criterion, such as the number of conflicts counted since the previous restart. The difference will usually be in the form of calculating the threshold after which another restart takes place. In this thesis, we use restarts to add additional clauses who are not necessarily implied by the formula.

# Chapter 4

# Hints Preliminaries

Let $\varphi$ be a CNF formula $c_1 \wedge c_2 \ldots \wedge c_m$. We write $c_i \in \varphi$ if $\varphi = c_1 \wedge \ldots \wedge c_i \wedge \ldots \wedge c_m$. Each clause $c = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_k$ is a disjunction of literals, and each literal $\ell_i$ is either a variable $v$ or its negation $\neg v$. We write $\ell_j \in c_i$ if $c_i = \ell_1 \vee \ldots \vee \ell_j \vee \ldots \vee \ell_k$. In what follows, $V$ denotes the set of variables occurring in $\varphi$, and $n = |V|$.

For two clauses $c_i = v \vee c$ and $c_j = \neg v \vee c'$, both involving the same variable $v \in V$, their binary *resolvent* is

$$\mathit{Resol}(c_i, c_j) \triangleq c \vee c' \,.$$

A conflict occurs when several solver decisions and subsequent implications result with a clause being unsatisfiable. In CDCL SAT solvers, a clause preventing the last conflicting set of decisions is created and added; as mentioned earlier, this clause is referred to as a conflict clause. In [15], it was shown that the best clause is the one created by finding the cut in the implication graph that includes the Unique-Implication-Point (UIP) closest to the conflict. That cut corresponds to a number of binary resolutions performed on clauses that are inside the cut or intersect it. For example, Figure 4.1 illustrates the cut and the clauses $c_4, c_5, c_6$ that participated in the resolutions that derived the conflict.

If $\varphi$ is a formula and $H$ is a set of hint clauses, then by $\varphi \wedge H$ we mean their conjunction: $\varphi \wedge \bigwedge_{h \in H} h$, which we will call a *hint-extended formula.*

In HSAT, we use a *resolution graph* to determine why $\varphi \wedge H$ is unsatisfiable, when it is, by extracting the *UNSAT core.*

**Definition 1** (Hyper-Resolution)**.** *Let $c_1, c_2, \ldots, c_i$ be all the clauses (from the implica-*

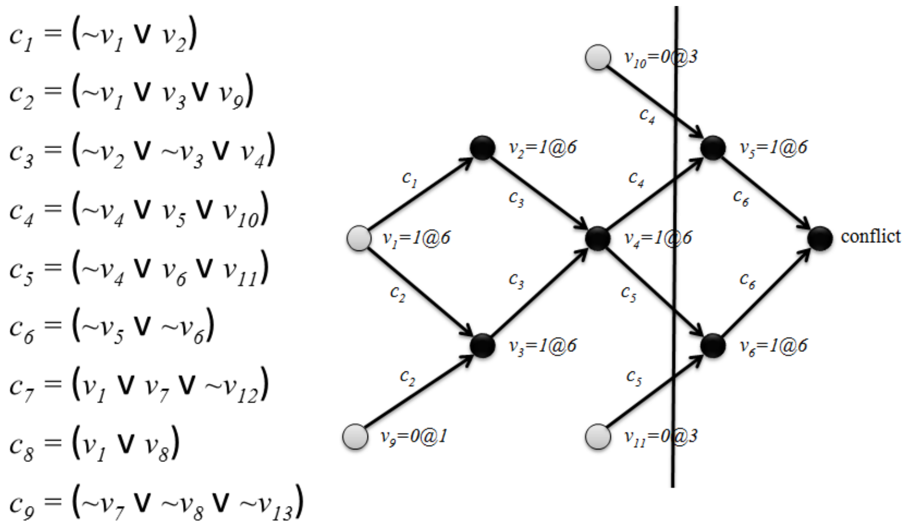
$$
\begin{aligned}
c_1 &= (\sim v_1 \vee v_2)\\
c_2 &= (\sim v_1 \vee v_3 \vee v_9)\\
c_3 &= (\sim v_2 \vee \sim v_3 \vee v_4)\\
c_4 &= (\sim v_4 \vee v_5 \vee v_{10})\\
c_5 &= (\sim v_4 \vee v_6 \vee v_{11})\\
c_6 &= (\sim v_5 \vee \sim v_6)\\
c_7 &= (v_1 \vee v_7 \vee \sim v_{12})\\
c_8 &= (v_1 \vee v_8)\\
c_9 &= (\sim v_7 \vee \sim v_8 \vee \sim v_{13})
\end{aligned}
$$

Figure 4.1: Conflict analysis graph. The gray nodes represent decision variables while the black nodes represent propagated values. The vertical line is the first UIP cut. Writing $v_i = a@b$ means that $v_i$ was assigned to $a$ at decision level $b$.

*tion graph) that participated in the binary resolutions that created the first UIP conflict clause $U$. The Hyper-Resolution function*

$$Hyper(c_1, c_2, \ldots, c_i) \triangleq U$$

*yields that resulting conflict clause $U$.*

As mentioned in Chapter. 1, we use a partial resolution graph to generate hints. This graph is used to determine whether there exists a directed path from $H$ to an empty clause.

**Definition 2** (Resolution Graph)**.** *The Resolution Graph $G = (V, E)$ is defined recursively as follows:*

$$
\begin{aligned}
V &:= \varphi \cup H \cup \{Hyper(c_1, \ldots, c_m) \mid c_1, \ldots, c_m \in V\}\\
E &:= \{(c_i, Hyper(c_1, \ldots, c_i, \ldots, c_m)) \mid c_1, \ldots c_m \in V\}\,.
\end{aligned}
$$

In words, the vertices are the initial clauses and hints closed under hyper-resolution and the edges point from participating clauses to their hyper-resolvent.

Determining whether a path exists from $H$ to the empty clause is possible by saving only the part relevant to hints. The partial resolution graph will consist only of hints or conflict clauses that were derived by some hint. To do so, we start just with the

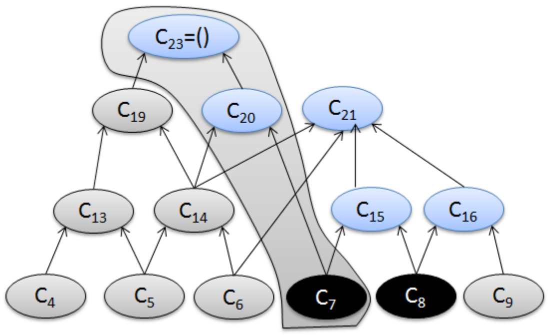

Figure 4.2: Resolution graph. Black nodes are the set $H$. Blue nodes are the hyper-resolvents of $V_P$. The gray nodes are the nodes in $V \setminus V_P$.

hints and define the relevant hint-based *Partial Resolution Graph* as follows:

**Definition 3** (Partial Resolution Graph)**.** *The Partial Resolution Graph* $G_P = (V_P, E_P)$ *is defined recursively as follows:*

$$\begin{aligned} V_P &:= H \cup \{Hyper(c_1, \ldots, c_i, \ldots, c_m) \mid c_i \in V_P,\ c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_m \in V\} \\ E_P &:= \{(c_i, Hyper(c_1, \ldots, c_i, \ldots, c_m)) \mid c_i \in V_P,\ c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_m \in V\}\,. \end{aligned}$$

In words, the vertices are the hints closed under hyper-resolution and the edges point from participating clauses to their hyper-resolvent. Figure 4.2 contains an example illustrating Definitions 2,3. The nodes $c_4, c_5, c_6, c_9$ form the set $\varphi$, while $c_7, c_8$ are the hints in $H$. The entire graph represents $G$ while the black and blue nodes and the edges between them are the restricted graph $G_P$.

Having defined the partial resolution graph, we are interested in isolating the proof of unsatisfiability. To do so, we define the UNSAT core (UC) of a resolution graph.

**Definition 4** (UNSAT Core)**.** *The UNSAT core is a subset* $UC$ *of* $\varphi \cup H$ *that is backward reachable from the empty clause in* $G$*.*

We are interested in finding that part of UC that is relevant to the hints:

**Definition 5** (Relevant UNSAT Core)**.** *The Relevant UnsatCore is the intersection* $RC = H \cap UC$*.*

We refer to the set of all dominator points (a vertex that lies along every path) between $RC$ and the empty clause in an UNSAT proof as $Dominator_{RC}$.

The negation $\neg h$ of a clause $h = \ell_1 \vee \ell_2 \ldots \vee \ell_k$ is the set (conjunction) of its negated literals $\neg\ell_1 \wedge \neg\ell_2 \ldots \wedge \neg\ell_k$, viewed as unit facts.

Chapter 5

# Hint Addition

We proffer a general platform for adding clauses without worrying that they might be inconsistent with the input formula. These "hint" clauses can be created using prior knowledge about the formula's origins or from information garnered during SAT solving, as explained in Chapter. 6. Our solution enjoys several benefits:

1. No additional literals are added.
2. We delay the effect of hints by using techniques from HaifaMUC as described in [22].
3. "Bad" hints, hints participating in the empty clause derivation, are used for search space reduction.

We use a resolution-graph-based solution to avoid the need for extra literals and to enable further optimizations in case the extended formula is unsatisfiable on account of the hints. In addition, we want to prevent any aggressive intervention of hints in the SAT solver's solution process, by using hints only when necessary, which is achieved by delaying their use. We discriminate in favor of the use of ordinary clauses because conflicts derived from hints are not necessarily consistent with the formula. The same motivation underlies modern Minimal Unsatisfiable Set (MUS) and Group Minimal Unsatisfiable Set (GMUS) solvers, which prefer to use clauses already known to be in the minimal core, to keep that as small as possible. Because of the similarity between hints and core clauses in MUS and GMUS solvers, we base our solver on HaifaMUC and use the optimization techniques described in [22]. These techniques allow us to prioritize ordinary clauses over hints and therefore reduce the run-time effect of hints.

The optimizations relevant to hints are the following:

1. Maintain only the partial resolution proof of clauses derived from the added hints. This prevents the keeping of the whole resolution proof in the memory and significantly reduces the memory footprint of the solver.

2. Selective clause minimization. Clause minimization [4, 26] is a technique for shrinking conflict clauses. If the learned clause is not derived from the hints, then during shrinking we prevent the use of hints in the minimization. The result is that no additional dependencies on hints are added even at the expense of longer learned clauses.

3. Postponed propagation over hints. This optimization is performed by changing the order of BCP (Binary Constraint Propagation). BCP first runs over ordinary (non-hint) clauses, and only if no conflict is found does it run over hints. The motivation is to prefer conflicts caused by ordinary clauses.

4. Selective learning of hints and selective backtracking. Both optimizations change the learning scheme by reducing the number of clauses effected by hints in case an ordinary clause can be learned.

We denote these optimization techniques collectively as HMucOpt.

One of the benefits of using a resolution-graph method is the availability of clause relation information, which can be used in case the extended formula is unsatisfiable on account of hints. In [17], a *path strengthening* technique was presented in relation to the MUS problem solution. It uses the partial resolution graph and is used to check whether a clause $c$ is part of the MUS. Checking whether $c \in$ MUS can be done by checking if the formula is unsatisfiable without using $c$. If it is, then $c$ cannot be part of the minimal core. To speed up the SAT solver run, the negation of the clause is added to the SAT Solver as assumptions. Path strengthening extends this set of assumptions by analyzing the resolution graph. If $c$ has only one derived clause in the cone of the empty clause, then the literals of this clause are added as assumptions as well. This operation is performed recursively until a clause with more than one child is reached. In HSat, we extend this by using all dominators between the hint clause and the empty clause in the partial resolution graph.

Algorithm 1 introduces the general workflow of HSat. Operation *Solve*() is a modification of a generic SAT solver with several additions. First, it allows the addition

**Algorithm 1** HSat– Solves an extended formula, negates dominators and cleans hints' effects.

```
Input:  instance – Boolean formula in CNF form
        H – Initial set of Hints (in our case ∅)
Output: SAT or UNSAT (ignore TIMEOUT)
1: while TRUE do
2:     model := Solve(instance ∧ H)        ▷ New h ∈ H can be added in Solve()
3:     if model ≠ NULL then
4:         return SAT                      ▷ We have the model
5:     else
6:         RC := GetRC()
7:         if RC.Size() = 0 then
8:             return UNSAT
9:         else
10:            Dominator_RC := GetDominators(RC)
11:            for each D ∈ Dominator_RC do
12:                instance := instance ∧ Negate(D)
13:            for each c_i ∈ RC do
14:                RemoveClauses(c_i)
```

of new hints and produces a partial resolution in case those hints are added. In addition, *Solve*() contains an implementation of HMucOpt. Operation *Solve*() can return satisfiable or unsatisfiable. In the satisfiable case, we are done, as the solver found a satisfying assignment to the formula. In case the result is unsatisfiable, we check the *RC* (UNSAT core of hints) created by the hints. The extraction of *RC* is performed using *GetRC*(). If *RC* is empty, then the solver found a proof of the empty clause without relying on hints, so the original formula is unsatisfiable. Otherwise, we find all dominators of the *RC* using *GetDominators*(). (See Algorithm 2 and the next paragraph.) For each dominator, we add its negation via *Negate*() to the input formula and create a new *instance*, which goes back to *Solve*(). Before the next call to *Solve*(), we clean the effect of hints in $G_P$ by means of *RemoveClauses*(). The correctness of Algorithm 1 is justified by the observations of Chapter. 7. As mentioned already, for *Solve*() we use a modification of Base, so all the optimizations HMucOpt are used, as was introduced in [22]. This way, we ensure an increased chance of finding the solution without hints if such a solution is easy to find.

The operation *GetDominators*() gets all nodes $v \in V_P$ such that all paths from $H$ to the empty clause go through $v$. At first, we save all nodes from *RC* in a list called

**Algorithm 2** *GetDominators*() – Gets all dominators in $G_P$. This set will be negated in Algorithm 1

```
Input:   $G_P$ – The Partial Resolution Graph
Input:   $workList$ – list of vertices. Initially set to $RC$
Output:  $Dominator_{RC}$ – The dominators with respect to
         $G_P$
1:  while $workList.Size() > 0$ do
2:      $v := GetAllParentsMarked(workList)$
3:      if $workList.Size() = 1$ then
4:          $Dominator_{RC}.Push(v)$                    ▷ A dominator
5:      $Mark(v)$                                       ▷ $v$ now marked
6:      for each $u \in Children(v)$ do
7:          if $\neg IsMarked(u)$ then
8:              $workList.Push(u)$
9:      $workList.Remove(v)$
10: return $Dominator_{RC}$
```

*workList*. The algorithm iterates until *workList* is empty. We get from the list some $v \in V_P$ that has all parent marked using *GetAllParentsMarked*(*workList*). Note that since *RC* has no parents, all members of *RC* have all parents marked. If the size of *workList* is 1, then $v$ is a dominator and we push it into $Dominator_{RC}$. We mark $v$ using *Mark*($v$) and push all its unmarked children into *workList*. Note that the empty clause is a child too.

# Chapter 6

# Hint Creation Algorithms

Heuristics for hint generation can vary from completely random selection to a purely deterministic selection algorithm.

## 6.1 Avoiding Failing Branches

In this section, we present a deterministic heuristic for hint creation based on the restart strategy and conflicts. We call this heuristic *Avoiding Failing Branches* (AFB). The idea is to track the most conflict-active decisions in the explored branch and add a hint that explicitly prevents choosing that set again. If a restart took place, it is reasonable to assume heuristically that the last explored branch is less likely to contain the satisfying assignment.

In AFB, we keep an array of variable activity to determine the most conflict-active decisions. When the solver encounters a conflict, we update the scores of all variables responsible for the conflict. We will explain what "responsible" means shortly.

The decision to add hints is taken upon backtracking. If the backtracking is actually a restart, then the most active literals are chosen to participate in a hint, which is added right after the restart. Because the literals are added in their negated form, all explored branches containing the set of literals in the hint will not be re-explored.

In Algorithm 3, which is implemented within the function *Analyze*() of MINISAT 2.2 [8], we update the score of the variables participating in a conflict. For this purpose, we keep an array of variables (*variableScores*), which is updated for all literals that are in the first UIP ($U$) computed in *ComputeFirstUip*(). We then iterate all variables $v \in U$. If $v$ is a decision variable (*DecisionVariable*($v$)), we increment its score by one.

**Algorithm 3** Update the score of a variable after a conflict. The score is updated for all decision variables in the first UIP, and for all variables in the reason clause for non-decision variables.

```
1:  $Analyze()$ {
2:  $\cdots$
3:  $U := ComputeFirstUip()$
4:  $\cdots$
5:  for each $\ell \in U$ do
6:      $v := Var(\ell)$                                  ▷ $v$ is the variable of $\ell$
7:      if $DecisionVariable(v)$ then
8:          $variableScores[v] := variableScores[v] + 1$
9:      else
10:         $c_v := Reason(v)$
11:         for each $\ell' \in c_v$ do
12:             $v' := Var(\ell')$
13:             $variableScores[v'] := variableScores[v'] + 1$
14: $\ldots$
15: }
```

Otherwise, we take the reason for $v$ being assigned ($c_v := Reason(v)$) and increment the score for all variables in $c_v$.

In Figure 4.1, the first UIP node is $U = v_{10} \vee \neg v_4 \vee v_{11}$. Algorithm 3 will first compute $U$ and iterate through all its literals. The scores of decision variables $v_{10}, v_{11}$ are increased in line 8; $v_4$ is not a decision variable, so its reason, $c_3$, is computed in line 10. The score of variables $v_2, v_3, v_4$ is increased in line 13.

The hints are added in the function *CancelUntil*() of MiniSat 2.2 [8]. If a restart is decided upon, we use the information acquired by Algorithm 3 to choose the most active literals to participate in the hint. A literal $\ell$ is chosen to participate if $variableScores[Var(\ell)]$ is greater than some threshold $\theta$. The integer *conflict* is the number of conflicts since *Solve*() was called. Three magic numbers, $\alpha \in [0..1]$, $x \in \mathbb{N}$, $y \in \mathbb{N}$, also appear in Algorithm 4. They are used in the following fashion:

1. A literal $\ell$ is added to the hint if $variableScores[Var(\ell)] > \alpha \times conflicts = \theta$.

2. We observed that, as time passes, it's advisable to increase $\theta$.

3. Parameter $x$ was added as a minimal threshold to prevent adding hints too "quickly". The idea is to prevent hints from being used when easy instances are solved.

4. Parameter $y$ is used to ensure that new hints are not too small. Small hints can be too influential in the search procedure.

**Algorithm 4** AFB hint addition – Adds a hint built of all negated literals with a score exceeding $\theta$.

```
 1: $CancelUntil(backtrackLevel)\{$
 2: if $backtrackLevel > 0$ then                                   ▷ This is not a restart
 3:     Performing backtracking until $backtrackLevel$...
 4:         Upon freeing variable $v$:
 5:             $variableScores[v] := 0$
 6:         ...
 7: else                                                           ▷ This is a restart
 8:     if $conflicts > x$ then
 9:         for each decision variable $v$ with decision $\ell$ do
10:             if $variableScores[v] > \alpha \times conflicts$ then
11:                 $hint.Push(\neg l)$
12:     for each variable $v$ with decision $\ell$ do
13:         $variableScores[v] := 0$
14:     Perform backtracking until $backtrackLevel = 0$...
15:     if $hint.Size() > y$ then
16:         $AddClause(hint)$
17:     $hint.Clear()$
18: ...
19: $\}$
```

We maintain a vector of literals, *hint*, to store the clause that might form the future hint. Function *AddClause*() adds the hint to the input instance.

## 6.2 Randomized Hints

We introduce next a completely random selection algorithm for hint creation, based on random assignments and satisfiability checking. We call this heuristic *Randomize Hints* (RH). In this algorithm, we use random assignments to see if we can learn literals that are likely untrue, that is, if chosen, a conflict is reached. We add these literals to form a new hint, that will hopefully lead the solver to an unsatisfiable conclusion. This hint is then negated, and the explored search space is reduced.

The randomized hint is created before HSAT is called. First, $k$ random assignments are drawn, each with uniform distribution over $\{0,1\}^n$. These assignments are then checked on every clause. If some clause is unsatisfied, we bump the grade of all literals in the clause. We keep a vector of grades, *literalsGrades*(), and track the maximal graded literals that will be chosen to participate in the hint. We encourage the solver to pick the literals of the hint as decisions by increasing the activity of the variables involved in MINISAT's *VarBumpActivity*($v$).

**Algorithm 5** Create randomized hints – draws random assignments and boosts score for all literals in a clause unsatisfied by an assignment. The literals with the highest scores are chosen to participate in hints.

```
Input:  sizeOfHint – Size of the hint
Input:  assignments – Number of assignments to draw
1:  $DrawRandomAssignments(assignments)$
2:  for each Assignment $\sigma$ do
3:      for each Clause $c$ do
4:          if $\neg ClauseSatisfied(c, \sigma)$ then
5:              for each literal $\ell \in c$ do
6:                  $literalsGrades[l] := literalsGrades[l] + 1$
7:                  $VarBumpActivity(Var(\ell))$
8:  for $i \in [0..sizeOfHint - 1]$ do
9:      $hint[i] := literalsGrades.PopMax()$
10: $AddClause(hint)$
11: $hint.Clear()$
```

The following functions and variables are used in Algorithm 5 for random hints:

1. *DrawRandomAssignments*(*num*) creates *num* random assignments.

2. *ClauseSatisfied*$(c, \sigma)$ returns true iff $\sigma(c) = \text{TRUE}$.

3. *PopMax*() returns and removes the literal with the highest score.

# Chapter 7

# Theoretical Basis

For completeness, a few observations are in place, which should serve to convince readers that correctness is being maintained.

**Proposition 6.** *For any formula $\varphi$, a set of hints $H$ and assignment $\sigma : V \to \{0,1\}$ of truth values to the variables of $\varphi \wedge H$,*

$$\sigma(\varphi \wedge H) \Rightarrow \sigma(\varphi)\,.$$

**Proposition 7.** *For any formula $\varphi$ and set of hints $H$,*

$$\varphi \wedge H \in \mathit{UNSAT} \Rightarrow \varphi \wedge \neg H \equiv \varphi\,.$$

By $\neg H$, we mean $\bigvee_{h \in H} \neg h$.

*Proof.* If $\varphi \wedge H \in \mathit{UNSAT}$, then $\neg(\varphi \wedge H)$, which is equivalent to $\varphi \Rightarrow \neg H$. □ □

From Proposition 7, we establish the following:

**Proposition 8.** *Given $\varphi \wedge H \in \mathit{UNSAT}$ and $|H| = 1$ where $h = \ell_1 \vee \ell_2 \cdots \vee \ell_k$*

$$\varphi \wedge \neg\ell_1 \wedge \neg\ell_2 \wedge \cdots \wedge \neg\ell_k \equiv \varphi\,.$$

This observation is critical for HSat. In this case, $k$ new facts are learned, which helps reduce the fraction of the search space that gets explored.

As mentioned earlier, this idea can be generalized to include all dominators.

**Theorem 9.** *If $\varphi \wedge H$ is unsatisfiable, then $\varphi \equiv \varphi \wedge \neg D$ for every $D \in \mathit{Dominator}_{RC}$.*

*Proof.* Since $D \in \mathit{Dominator}_{RC}$, it is sufficient to prove $\varphi \wedge D \in \mathit{UNSAT}$. By Proposition 7, $\varphi \equiv \varphi \wedge \neg D$. $\square$ $\square$

# Chapter 8

# Experimental Results

## 8.1 AFB Results: SAT 2013

We compare now the performance of HSat, with and without heuristic Afb. We find that hints have a positive effect for satisfiable instances but cause a moderate degradation for unsatisfiable ones. The positive results for satisfiable instances are in line with our presumption that, if a restart takes place, it is heuristically likelier that the satisfying assignment to the problem lies on another branch.

We ran over 150 *satisfiable* instances from SAT 2013, but the results reported below refer only to the 113 that were fully solved by at least one solver within half an hour. All of the instances are publicly available at [2]. We implemented all the algorithms in Base [22], which is built on top of MiniSat 2.2 [8]. The code is public and available at [21]. For the experiments, we used machines running Intel® Xeon® processors with 3Ghz CPU frequency and 32GB of memory.

Table 8.1 displays a 30% improvement in overall runtime for satisfiable instances. Furthermore, there are 9 instances solved by Afb that are not solved by the base solver, compared to 4 instances solved by Base but not by Afb.

In addition, 130 *unsatisfiable* instances from SAT 2013 were tested; the reported results refer only to the 60 that were fully solved by at least one solver within the 30-minute time limit. Table 8.1 shows a 7% degradation in overall runtime for unsatisfiable instances.

Figure 8.1 presents Base vs. Afb. The diagonal $y = x$ emphasizes the superiority of Afb. Figure 8.2 presents the time differential between Base and Afb. On average, Afb solves one of these problem instances $2\frac{1}{2}$ minutes faster than the baseline. The

Table 8.1: AFB performance results for SAT 2013: satisfiable (left) and unsatisfiable (right) instances. Run-time is in minutes.

| SAT | BASE | AFB |
|---|---|---|
| Run-time | 990 | **697** |
| Unsolved (by one) | 9 | **4** |

| UNSAT | BASE | AFB |
|---|---|---|
| Run-time | **727** | 779 |
| Unsolved (by one) | **2** | 3 |

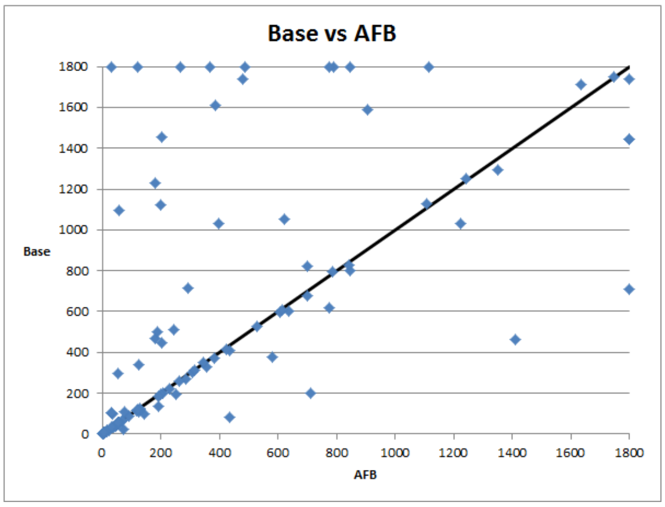

Figure 8.1: Comparing AFB to BASE.

graphs refer to satisfiable instances only.

Figure 8.3 shows three curves, plotted at one minute intervals. The lower curve ($A$) is the percentage of instances solved by BASE and AFB both; the middle ($B$) is the percentage of instances solved by BASE the upper ($C$) is the percentage solved by either one. The gap $B - A$ represents the percentage of instances solved by BASE but not by AFB; $C - B$ represents the percentage solved by AFB but not by BASE. Notably, $C - B$ is consistently larger than $B - A$.

We observe that the positive effect of AFB is due to successful branch cutting by hints and not because of HSAT's ability to negate dominators. Most of the hints added did not contradict the instance, so HSAT's UNSAT core abilities were not helpful in AFB.

For the SAT 2013 benchmark, we also measured the average size of hints (number of literals participating in a hint), the average number of hints per instance, and the number of dominators found in all instances:

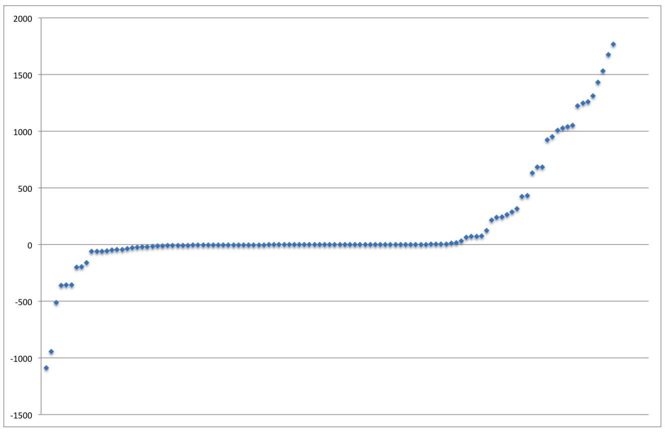

Figure 8.2: The time difference (in seconds) between Base and Afb

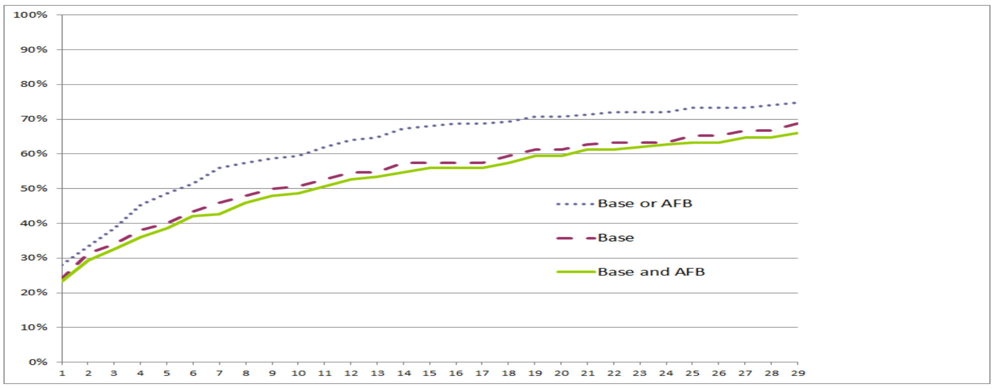

Figure 8.3: Comparing percentage of instances solved by Base and Afb.

| | SAT | UNSAT |
|---|---|---|
| Hint average size | 34 | 43 |
| Hints per instance | 0.84 | 1.16 |
| Dominators | 2 | 15 |

Hints were used in 34% of the satisfiable instances and 39% of the unsatisfiable cases.

## 8.2 AFB Results: SAT 2014

We used the same configuration when testing Afb on satisfiable instances from the SAT 2014 competition. Table 8.2 shows a 19% improvement in overall runtime for satisfiable instances. Furthermore, there were 9 instances solved by Afb that were not solved by the base solver, compared to 2 instances solved by Base but not by Afb. These results refer only to the 98 instances that were fully solved by at least one solver within 30 minutes.

Table 8.2: Afb performance results for SAT 2014: satisfiable instances. Run-time is in minutes.

| SAT | Base | Afb |
|---|---|---|
| Run-time | 833 | **681** |
| Unsolved (by one) | 9 | **2** |

Table 8.3: Rh performance results: satisfiable (left) and unsatisfiable (right) instances. Run-time is in minutes.

| SAT | Base | Rh |
|---|---|---|
| Run-time | 1080 | **988** |
| Unsolved (by one) | **12** | **12** |

| UNSAT | Base | Rh |
|---|---|---|
| Run-time | **757** | 888 |
| Unsolved (by one) | **3** | 9 |

## 8.3 RH Results: SAT 2013

The same configuration as in Chapter. 8.1 was used for the Rh heuristic on satisfiable instances from SAT 2013, and the same instances were tested. The results reported below refer only to the 116 instances that were fully solved by at least one solver within 30 minutes. Table 8.3 shows a 8% improvement in overall runtime for satisfiable instances.

We were admittedly surprised to see that satisfiable instances were solved faster because of "good" hints, hints that do not contradict the input. We were surprised because we tried to build hints that would contradict the input and have the negation of dominators drive the solution.

In addition, 130 *unsatisfiable* instances from SAT 2013 were tested; the results below refer only to the 60 that were fully solved by at least one solver within the 30-minute time limit. Table 8.3 shows a 15% degradation in overall runtime for unsatisfiable instances.

Combining the two heuristics, Afb and Rh, as though they would run in parallel for half an hour on the SAT 2013 benchmark, we obtain 16 SAT instances that are solved for which Base times out, versus 2 that only Base solves, and 5 UNSAT instances that Base fails on, versus 3 only by Base.

# Chapter 9

# Discussion

We have introduced a new paradigm and platform, called HSat, with which one can speed up SAT solving by means of added clauses. It enables the addition of "hint" clauses that are not necessarily derivable from the original formula but which can nevertheless help the solver reach a solution faster. HSat avoids the addition of new literals, using instead a partial resolution graph to keep track of the effect of hints. We have seen that the Afb hint heuristic, which causes the prover to avoid retaking the most conflict-active decisions, outperforms the (hintless) baseline system and introduces a significant improvement in the solver core. On a benchmark of 280 instances, 150 of which are satisfiable: Afb achieved a 30% runtime improvement over the baseline and solved 9 instances not solved by the baseline prover.

Though these results are very encouraging, we have reason to believe that future work can lead to further improvements. For example, we tried to increment conflict decision variable scores by an amount that is inversely proportional to its depth in the proof tree, so those closer to the root (which have greater impact) get greater weight. This approach did not work for the thresholds we looked at, but might work for others. Another example is that our hint heuristics do not work well for unsatisfiable instances, the main reason being that there are usually no dominator clauses, in which case unsatisfiability does not drive the subsequent search very well. In this case, the incremental running of Algorithm 1 just adds overhead. An interesting avenue for research would be to design hints that create multiple dominators or that lead the solver to a contradiction faster.

There are an endless number of ways to create hints, and many places in the process to add them; so far we have only explored a few options. It is likely that there remain even more interesting ways to create good hints for satisfiable instances and, hopefully, for unsatisfiable ones, too.

# Bibliography

[1] 18th international conference on theory and applications of satisfiability testing. http://www.cs.utexas.edu/~marijn/sat15/. 1

[2] Sat competition 2013. http://satcompetition.org/2013/downloads.shtml. 8.1

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. 3

[4] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *CoRR*, abs/1107.0044, 2011. 2.

[5] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. 1

[6] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971. 2

[7] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. 3.1

[8] Niklas Eén and Niklas Sörensson. An extensible SAT-solver [extended version 1.2]. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 512–518. Springer, 2003. 1, 3, 6.1, 6.1, 8.1

[9] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003. 1

[10] Evguenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. In *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pages 142–149. IEEE Computer Society, 2002. 3.2.4

[11] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. 3.2.5

[12] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 385–399. Springer, 2011. 1

[13] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990. 3.2.4

[14] Davide Lanti and Norbert Manthey. Sharing information in parallel search with search space partitioning. In Giuseppe Nicosia and Panos M. Pardalos, editors, *Learning and Intelligent Optimization - 7th International Conference, LION 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers*, volume 7997 of *Lecture Notes in Computer Science*, pages 52–58. Springer, 2013. 1

[15] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC), Las Vegas, NV*, pages 530–535. ACM, June 2001. 1, 3.2.2, 3.2.4, 4

[16] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Lugano, Switzerland*, pages 221–229. IEEE, October 2010. 1

[17] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *Formal Methods in Computer-Aided Design (FMCAD), Portland, OR*, pages 197–200. IEEE, October 2013. 1, 5

[18] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *JSAT*, 9:27–51, 2014. 1

[19] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41st Design Automation Conference (DAC), San Diego, CA*, pages 518–523. ACM, June 2004. 1

[20] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986. 2

[21] Vadim Ryvchin. HaifaMUC. `https://www.dropbox.com/s/uhxeps7atrac82d/Haifa-MUC.7z`. 1, 8.1

[22] Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing (SAT), Ann Arbor, MI*, number 6695 in Lecture Notes in Computer Science, pages 174–187, Berlin, 2011. Springer. 1, 2., 5, 5, 8.1

[23] Karem A. Sakallah and Laurent Simon, editors. *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing (SAT), Ann Arbor, MI*, Lecture Notes in Computer Science, Berlin, 2011. Springer. 1

[24] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Pedro Barahona and José Júlio Alferes, editors, *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999. 3.2.4

[25] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996. 3.2

[26] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), Swansea, UK*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, June 2009. 2.

[27] Ofer Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004. 1

**אוניברסיטת תל-אביב**

**הפקולטה למדעים מדויקים ע"ש ריימונד ובברלי סקלר**

**בית הספר למדעי המחשב ע"ש בלבטניק**

# ספיקות עם רמזים

חיבור זה הוגש כחלק מהדרישות לקבלת התואר "מוסמך אוניברסיטה" - .M.Sc באוניברסיטת תל-אביב
בית הספר למדעי המחשב

על ידי

**יונתן קלכשטיין**

העבודה נעשתה בהנחייתו של

**פרופסור נחום דרשוביץ**

אלול תשע"ה

## תקציר

התזה מציגה את המושג של רמזים, פסוקיות שאינן בהכרח קונסיסטנטיות עם נוסחת הקלט.

המטרה של הוספת רמזים אלו היא להאיץ את התהליך של הSAT solving.

למטרה זו, מנגנון יעיל וגנרי להוספה והסרה של רמזים אלו מוסף.

כאשר רמז נקבע כלא קונסיסטנטי עם נוסחת הקלט, ניתן להשתמש בגרף הרזולוציה המבוסס רמזים על מנת להקטין את מרחב החיפוש.

המנגנון הנ"ל משמש להאצת קצב הפתרון על ידי הוספת רמזים לנוסחת הקלט.

בתזה מתוארות שתי שיטות להוספת רמזים, כאשר אחת מהן קיצרה את זמן הריצה ב-30% על נוסחאות ספיקות מתחרות SAT '13 ופתרה תשע נוסחאות שלא נפתרו על ידי אותו solver ללא רמזים.