IMPROVEMENTS OF SAT SOLVING TECHNIQUES

Roman Gershman

Improvements of SAT solving techniques

Research Thesis

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

By
Roman Gershman

Submitted to the Senate of
The Technion - Israel Institute of Technology
Adar, 5767 Haifa March 2007

The Research Thesis Was Done Under The Supervision
of Dr. Ofer Strichman in the Faculty of Computer Science

The Generous Financial Help Of The Technion
Is Gratefully Acknowledged.

Contents

List of Tables

List of Figures

Abstract

We present two algorithms that improve current state-of-art SAT solving techniques. The first algorithm is an improvement of Bacchus and Winter's [2] Binary Hyper-Resolution preprocessor algorithm for simplifying industrial SAT formulas. Unlike the original algorithm, we restrict the application of Unit-Propagation to the roots of the Binary Implications Graph, and learn stronger implications by finding dominators that are responsible for the implications. Our algorithm HyperBinFast is typically faster and more cost-effective.

The second algorithm is a new decision heuristic in the DPLL framework. We present a theoretical model, based on abstraction-refinement, which is helpful in explaining clause-based decision heuristics such as Berkmin. Based on this model, we suggest a different heuristic, called Clause-Move-To-Front (CMTF), which attempts to keep the refinement focused on one path, in contrast to Berkmin. We also suggest a new algorithm for scoring variables, based on their activity in the internal resolution process that the SAT solver performs. Together these heuristics perform better on average than the well-known VSIDS and Berkmin heuristics, based on a large set of industrial problems.

The algorithms described in the thesis are implemented in our SAT solver HaifaSat. HaifaSat won the third place in the 2005 SAT competition in the industrial-benchmarks category.

Chapter 1

Introduction

1.1 The SAT problem

The satisfiability (SAT) problem is to decide whether there exists a truth assignment that satisfies a given propositional formula in Conjunctive Normal Form (CNF). Since every propositional formula can be translated to an equivalent CNF formula in polynomial time, and since experience of decades has shown that CNF formulas are easier to solve than their original representations, CNF has become the standard de-facto for competitive SAT solvers. The SAT problem is fundamental for solving many other problems in Computer-Aided Design, Verification, Automated Reasoning and so forth. The ever-growing need to solve larger and harder CNF formulas led through the years to a vast amount of research and consequently to exceptionally powerful SAT solvers, which can solve many real-life CNF formulas with hundreds of thousands of variables in a reasonable amount of time. Figure 1.1 shows the progress of these tools through the years. Of course, there are also instances with several hundreds of variables that they cannot solve. In general it is very hard to predict which instance is going to be hard to solve. It seems that SAT solvers are very good in identifying the important variables, those variables that once given the right value, simplify immensely the problem (Williams, Gomes and Selman coined the term *back-door variables* to refer to these variables[12]).

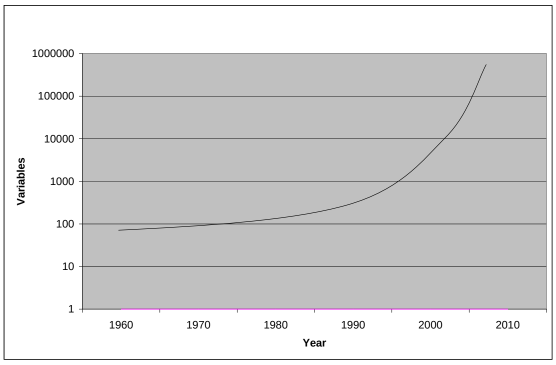

Figure 1.1: The size of industrial CNF formulas (instances generated for solving various realistic problems such as verification of circuits and Planning problems) that are regularly solved by SAT solvers in a few hours, according to year. Most of the progress in efficiency was made in the last decade.

Modern SAT solvers can be classified into two main categories. The first category is based on the *Davis-Putnam-Loveland-Logemann* (*DPLL*) framework: in this framework the tool can be thought of as traversing and backtracking on a binary tree, in which internal nodes represent partial assignments, and the leaves represent full assignments, i.e., an assignment to all the variables. The second category is based on *stochastic search*: the solver guesses a full assignment, and then, if the formula is unsatisfied, starts to flip values of variables according to some (greedy) heuristic. Typically it counts the number of unsatisfied clauses and chooses the flip that minimizes this number. There are various strategies that help such solvers avoid local minimums and repeating previous bad moves. DPLL, however, is considered better in most cases, at least at the time of writing this thesis (2006), according to annual competitions that measure their respective success in solving numerous CNF instances. DPLL solvers also have the advantage that, unlike stochastic search methods, they are complete. Stochastic methods seem to have an average advantage in solving randomly generated CNF instances, which is not surprising: in these instances there is no structure to exploit and learn from, and no obvious choices of variables and values, which makes the heuristics adopted by DPLL solvers ineffective. We focus on DPLL solvers only.

1.2 DPLL SAT solvers

In its simplest form, a DPLL solver progresses by making a decision on a variable and its value, propagates the implications of this decision using *Boolean Constraint Propagation* (BCP), and backtracks in case of a conflict. Viewing the process as a search on a binary tree, each decision is associated with a *decision level*, which is the depth in the binary decision tree in which it is made, starting from 1. The assignments implied by the decision are associated with its decision level. The assignments implied regardless of the current decisions (due to *unary clauses*, which are clauses with a single literal) are associated with decision level 0, also known as the *ground level*.

During the solving process, a clause in the CNF formula under partial assignment is exactly in one of the following four states:

1. Satisfied - if at least one of its literals is satisfied.

2. Conflicting - if all its literals are set to FALSE.

3. Unit - if one of the literals is undefined and all other are set to FALSE.

4. Unresolved - otherwise.

Example 1. *Given the Partial Assignment*

$$(x_1 = 1, x_2 = 0, x_4 = 1)$$

$(x_1 \vee x_3 \vee \neg x_4)$	*is satisfied*
$(\neg x_1 \vee x_2)$	*is conflicting*
$(\neg x_1 \vee \neg x_4 \vee x_3)$	*is unit*
$(\neg x_1 \vee x_3 \vee x_5)$	*is unresolved*

□

Given a partial assignment under which a clause becomes unit, it may be extended so it satisfies the last literal of this clause: this observation is known as the *unit clause rule*. Following this requirement is necessary but obviously not sufficient for satisfying the formula. BCP is the algorithm that is responsible for identifying all unit clauses and by setting their undefined literals to TRUE. During the run of this algorithm the unit clause rule is iteratively applied until there are no unit clauses under the current assignment or a conflicting clause is found. Each time a unit clause becomes a reason for assigning some variable x, we say that this clause is an *antecedent* clause of x and it stays so until x is unassigned during the backtracking.

Example 2. *Given the clause* $C : (\neg x_1 \vee \neg x_4 \vee x_3)$ *and the partial assignment* $(x_1 = 1, x_4 = 1)$, x_3 *is implied and Antecedent*$(x_3) = C$. □

Algorithm 1 The DPLL SAT framework

```
1: procedure DPLL
2:     while (TRUE) do
3:         if (BCP() = 'CONFLICT') then
4:             if decision-level = 0 then return 'Unsatisfiable';
5:             end if
6:             backtrack-level := ANALYZE-CONFLICT();
7:             BackTrack(backtrack-level);
8:         else
9:             if ¬DECIDE() then return 'Satisfiable';
10:            end if
11:        end if
12:    end while
13: end procedure
```

A framework followed by most DPLL modern solvers is presented, for example, by Zhang and Malik in [8], and reappears in Algorithm 1. Function DECIDE() at line 9 is responsible for choosing a next unassigned variable and assigning a truth value for it. In case all variables are assigned, it returns FALSE and DPLL() returns SATISFIABLE. Each such decision is associated with a *decision level*, which can be thought of as the depth in the search tree. There are numerous heuristics for making these decisions. Some of these are described later in Section 1.3. Function BCP() at line 3 is responsible for applying the Unit Clause Rule. If during the propagation of the current assignment a conflicting clause is found, it returns the answer 'conflict' and saves the conflicting clause for further processing. In case the conflict was found at the ground level, DPLL() returns UNSATISFIABLE. Otherwise, ANALYZE-CONFLICT() at line 6 creates a *conflict clause* (see below), adds it to the clause database, and computes the decision level to which DPLL() should backtrack.

We now demonstrate BCP, reaching a conflict and backtracking, following ANALYZE-CONFLICT as presented in Algorithm 2. Each assignment is associated with the decision level in which it occurred. If a variable x_i is assigned TRUE (either due to a decision or an implication) in decision level dl, we write $x_i = 1@dl$. Similarly, $x_i = 0@dl$ reflects a FALSE assignment to this variable in decision level dl. When appropriate, we refer only to the part of the label that refers to the assignment, without the decision level, in order to make the notation simpler.

The process of BCP can be illustrated with an *implication graph*. An implication graph represents the current partial assignment, and the reason for each of the implications.

Definition 1.2.1. An *implication graph* is a labelled directed acyclic graph $G(V, E)$ where

- V represents the literals of the current partial assignment (we refer to a node and the literal it represents interchangeably). Each node is labelled with the literal it represents and the decision level in which it entered the partial assignment.

- $E = \{(v_i, v_j) | v_i, v_j \in V \wedge \neg v_i \in antecedent(v_j)\}$. Each edge (v_i, v_j) is labelled with $antecedent(v_j)$.

- G can also contain a single conflict node labelled with κ and incoming edges $\{(v, \kappa) | \neg v \in c\}$ labelled with c for some conflicting clause c.

In an implication graph the root nodes correspond to decisions, and the internal nodes to implications through BCP. A conflict node with incoming edges labelled with

Algorithm 2 Analyze-Conflict

```
Require: $current\text{-}decision\text{-}level > 0$
1: $cl$ := $current\text{-}conflicting\text{-}clause$;
2: while (¬STOP-CRITERION-MET ($cl$)) do
3:     $lit$ := LAST-ASSIGNED-LITERAL ($cl$);
4:     $var$ := VARIABLE-OF-LITERAL ($lit$);
5:     $ante$ := ANTECEDENT ($var$);
6:     $cl$ := RESOLVE ($cl$, $ante$, $var$);
7: end while
8: add-clause-to-database($cl$);
9: return clause-asserting-level($cl$);          ▷ 2nd highest decision level in $cl$
```

c represents the fact that the BCP process has reached a conflict by assigning FALSE to all the literals in the clause c (i.e. c is conflicting). In such a case we say that the graph is a *conflict graph*. The implication graph corresponds to all the decision levels lower or equal to the current one, and is dynamic: backtracking removes nodes and their incoming edges, while new decisions, implications, and conflict clauses continue the construction of the graph. Note that the implication graph is sensitive to the order in which the implications are propagated in BCP, which means that there is more than one possible such graph given a specific state and a decision. In most SAT solvers this order is arbitrary and determined by the order of clauses in propagation queue.

A partial implication graph, which illustrates the BCP in a specific decision level, is sufficient for describing a particular popular method for implementing ANALYZE-CONFLICT. The roots in such a partial graph represent assignments in decision levels lower than dl, in addition to the decision at level dl, and internal nodes correspond to implications at level dl. The description that follows mainly uses this restricted version of the graph.

Consider, for example, the following subset of clauses:

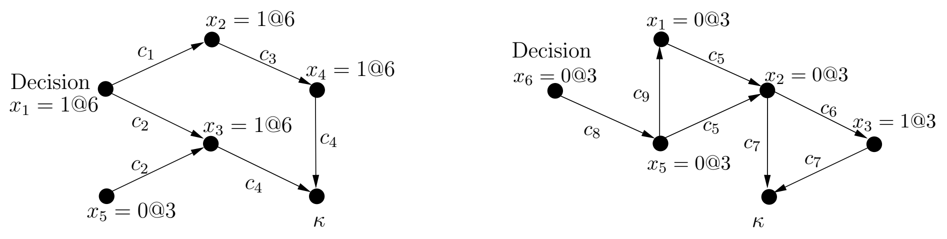

Figure 1.2: A partial Implications Graph for decision level 6, corresponding to the clauses in Formula 1.1, after a decision $x_1 = 1$ (left) and a similar graph after learning the conflict clause $c_9 = (x_5 \vee \neg x_1)$ and backtracking to decision level 3 (right).

$$\begin{aligned}
c_1 &= (\neg x_1 \vee x_2) \\
c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\
c_3 &= (\neg x_2 \vee x_4) \\
c_4 &= (\neg x_3 \vee \neg x_4) \\
c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\
c_6 &= (x_2 \vee x_3) \\
c_7 &= (x_2 \vee \neg x_3) \\
c_8 &= (x_6 \vee \neg x_5) \\
&\vdots
\end{aligned} \tag{1.1}$$

Assume that at decision level 3 the decision was $x_6 = 0@3$, which implied $x_5 = 0@3$ through clause c_8 (hence $\mathit{Antecedent}(\neg x_5) = c_8$). Further assume that the solver is now at decision level 6 and assigns $x_1 = 1$. In decision levels 4 and 5, variables other than $x_1, \ldots, x_6$ were assigned, and are not listed here as they are not relevant to these clauses. The implication graph on the left of Figure 1.2 demonstrates the BCP process at the current decision level 6 until, in this case, a conflict is detected. The roots of this graph, namely $x_5 = 0@3$ and $x_1 = 1@6$, constitute a sufficient condition for creating this conflict. Therefore, we can safely add the *conflict clause* $c_9 = (x_5 \vee \neg x_1)$ to our formula: while it is logically implied by the original formula and therefore does not change the result, it prunes the search space because it forbids partial assignments that contradict it. The process of adding conflict clauses is generally referred to as *learning*, reflecting the fact that this is the solver's way to learn from its past mistakes. As we progress in this chapter, it will become clear that conflict clauses not only prune the search space, but also impact the decision heuristic, the backtracking level and the set of variables implied by each decision.

Algorithm Analyze-Conflict (see algorithm 2) is the function responsible for deriving new conflict clauses and computing the backtracking level. It traverses the implication graph backwards, starting from the conflict node κ, and generates a conflict clause through a series of steps that we describe later in §3.2.1. For now assume that c_9 is indeed the generated clause.

After detecting the conflict and adding c_9, the solver determines to which decision level to backtrack according to the *conflict-driven backtracking* strategy. According to this strategy the backtracking level is set to the *second most recent decision level in the conflict clause* [1], while erasing all decisions and implications made *after* that level. In the case of c_9, the solver backtracks to decision level 3 (the decision level of x_5), and erases all assignments from decision level 4 onwards, including the assignments to x_1, x_2 and x_3.

The newly added conflict clause c_9 becomes a unit clause since $x_5 = 0$, and therefore the assignment $x_1 = 0@3$ is implied. This new implication re-starts the BCP process at level 3. c_9 is a special kind of a conflict clause, called an *asserting clause*: it forces an immediate implication after backtracking. With the right stopping condition in line 2, Analyze-Conflict can be designed to generate only asserting clauses. Indeed, most competitive solvers nowadays only add asserting clauses. After asserting $x_1 = 0$ the solver again reaches a conflict as can be seen in the right drawing of Figure 1.2. This time the conflict clause (x_6) is added, the solver backtracks to decision level 0, and continues from there.

Conflict-driven backtracking raises several issues:

- *It seems to waste work*, because the Partial Assignments up to decision level

[1] In case of learning a unary clause, the solver backtracks to the ground level.

5 can still be part of a satisfying assignment. However, empirical evidence shows that conflict-driven backtracking, coupled with modern decision heuristics performs very well. A possible explanation of the success of this heuristic is that the encountered conflict can influence the decision heuristic to decide different values or different variables than those in deeper decision levels (levels 4 and 5 in this case). Thus, keeping the decisions and implications made before the new information (i.e., the new conflict clause) arrived may have skewed the search to areas not considered best anymore by the heuristic.

- *Is this process guaranteed to terminate?* In other words, how do we know that a partial assignment cannot be repeated forever? The learned conflict clauses cannot be the reason, because in fact most SAT solvers erase many of them after a while to prevent the formula from growing too much. The reason is the following: *it is never the case that the solver enters decision level dl with the same Partial Assignment*. Consider a Partial Assignment up to decision level $dl - 1$ that does not end with a conflict, and falsely assume that this state is later repeated, after the solver backtracks to some lower decision level dl^- $(0 \leq dl^- < dl)$. Any backtracking from a decision level dl^+ $(dl^+ \geq dl)$ to Decision level dl^- adds an implication at level dl^- of a variable that was assigned at decision level dl^+. Since this variable was not so far part of the Partial Assignment up to decision level dl, once the solver reaches dl again, it is with a different Partial Assignment, which contradicts our assumption.

1.3 Decision heuristics

As we noted before DECIDE() is responsible for choosing variables and their truth values during the search. In fact, this function plays a crucial role in the performance of DPLL solvers. An improvements to decision heuristics in the past have led improvements of orders of magnitude on average and to solving problems that were considered unsolvable before. We mention some of the modern heuristics that are implemented in the best SAT solvers known to the research community to date.

Variable State Independent Decaying Sum (VSIDS) [8]: Each literal has its own score. At each decision, a literal with the biggest score is chosen to be satisfied. Each time a conflict clause is added to the database, the score of its literals is increased by 1. In addition, every constant number of conflicts (say 100), all scores are divided by 2 - thus gradually giving priority to those literals whose score was increased lately. Initially a score for each literal is set to the number of occurrences of that literal in the formula. This heuristic was implemented in Zchaff in 2001 and led to a very significant improvement. The authors of [8] explain VSIDS by saying that literals which appear most in the latest conflict clauses are chosen to be satisfied, thus satisfying "problematic" clauses. This was the first *conflict-driven* decision heuristic, and virtually all modern decision heuristic follow this principle.

Berkmin [6]: A score list is maintained as in VSIDS. Conflict clauses are pushed into a stack. When a decision has to be made, the first unsatisfied clause from the top is identified; From this clause the unassigned literal with the highest score is chosen. If the stack is empty, the literal with the highest score is chosen, as in VSIDS. This heuristic appears to be very robust for a wide range of industrial problems.

Variable Move To Front VMTF [7]: All variables are maintained in a list. Initially, variables are ordered in the list according to their frequency in the formula, where the most frequent variable appears at the top of the list. Each time a decision is made, variables are scanned from the top of the list and a first unsatisfied variable is chosen. Its value is chosen according to the relative frequency of its positive and negative occurrences (similarly to VSIDS). At each conflict a small constant number (say 8) of variables from the conflict clause is moved to the front of the variable list. These variables are chosen randomly from the conflict clause. This simple and fast heuristic in practice is very efficient as well.

1.4 The current work

This thesis improves modern DPLL-based algorithms in two ways: by applying a preprocessing algorithm on the initial CNF formula before DPLL is applied, and by using a new decision heuristic in the function Decide().

1. The first algorithm is our improvement to the preprocessing algorithm by Bacchus and Winter Hypre [2]. The authors used the fact that most industrial formulas have a lot of binary clauses with which many syntactical and semantical simplifications can be done. Often these simplifications help the SAT solver afterwards and can be seen as an orthogonal heuristic that helps DPLL. However, even for polynomial preprocessing algorithms, sometimes it is very hard to compete with SAT solvers on large formulas. Consider, for example, instances with $100 - 200K$ variables and $500 - 800K$ clauses. Even with complexity of $O(n^3)$ the preprocessing algorithm will run too long compared to aggressive heuristics of SAT solvers. Therefore, there is a need of more robust preprocessor, which can run faster in most cases. We improve Hypre by constraining its derivation rule to a subset of the variables, and learn from it more than the original Hypre would learn.

2. The second algorithm is a new decision heuristic, based on our hypothesis that explains why SAT solvers are able to be so efficient in solving industrial problems. We suggest a model based on abstraction-refinement that helps explain the progress of modern SAT solvers. Based on this model, we analyze the Berkmin decision procedure and suggest an improved procedure, using the insights given by the suggested model. In addition, we describe a new scoring scheme for the decision heuristic, which is based on the activity of a variable in the resolution process conducted by the SAT solver.

These improvements and some others are all implemented in HaifaSat, a SAT solver that was developed for this thesis. HaifaSat competed in the 2005 SAT competition and won the 3rd place in the industrial benchmarks category [2].

The rest of the thesis is organized as follows. Chapter 1 introduces HyperBinFast as an improvement to the algorithm Hypre. Chapter 2 describes our new decision procedure for DPLL solver. Chapter 3 describes a new variable scoring procedure and Chapter 4 provides the experimental results of our SAT solver HaifaSAT.

[2]the first two were variations on the same code base

Chapter 2

The HyperBinFast algorithm

2.1 Introduction

Given the power of modern SAT solvers, most CNF preprocessing algorithms [5, 11] are mostly not cost-effective time-wise. Since these solvers are so effective in focusing on the important information in a given CNF, it is particularly challenging to find the right balance between the amount of effort invested in preprocessing and the quality of information gained, in order to positively impact the overall solving time.

One of the only preprocessors that succeeds to do so, at least when combined with *some* of the modern SAT solvers, is Hypre [2]. An early version of Hypre was implemented in 2cls+eq as an inference rule with impressive success (it solved instances that could not be solved by any other solver in the SAT'02 competition). There it was invoked in each node of the decision tree (before the call to Decide()), hence making it part of the solver rather than a preprocessor. But given the very large SAT instances that now solvers need to cope with, this approach was too costly in practice. In [2] Bacchus and Winter improved the implementation of this algorithm and tried it very successfully as a preprocessor. Their experiments show that in most cases there is a benefit in using this preprocessor prior to invoking a state-of-the-art solvers like Berkmin and zChaff. It seems, however, that on larger CNF formulas this is no longer true: running Hypre on the benchmarks given in the

SAT'04 competition, which were larger on average than the benchmark set attempted in [2], we noticed that it is not cost-effective in most cases.

Hypre processes a CNF instance in three interconnected ways:

1. It adds binary clauses using Hyper Resolution [2].
2. It finds *failed literals* (variables with forced value) and propagates them, and
3. It identifies equivalent variables by a traversal of the binary implications graph (a graph in which the edges correspond to binary clauses) and performs substitutions accordingly.

Overall we refer to these actions as deriving *auxiliary information* from the CNF instance that simplifies its solution later by the SAT solver. One of the major reasons for the success of Hypre comparing to previous preprocessors like 2-simplify [5] is that it is more selective in the information that it adds, and take less time to generate.

In this chapter we present HyperBinFast, which is similar to Hypre, but improves it in two dimensions. First, to avoid cases in which preprocessing takes disproportional time, HyperBinFast is implemented as *anytime* algorithm, which means that it produces meaningful auxiliary information even when interrupted before termination. This enables us to control the amount of preprocessing, and in particular, to allocate a certain percentage of the overall solving time to preprocessing. Second, HyperBinFast sacrifices some of the preprocessing power in order to terminate faster. We consider this as an adaptation to the new reality of SAT solvers being so efficient by themselves.

2.2 Definitions

We begin with several definitions.

Definition 2.2.1 (Binary Implications graph)**.** Given a CNF formula φ with a set of binary clauses B, a *Binary Implications Graph* is a directed graph $G(V, E)$ such that $v \in V$ if and only if v is a literal in φ, and $e = (u, v)$ is an edge if and only if B contains a clause $(\overline{u}, v)$.

A Binary Implications Graph allows us to follow implications through binary clauses. Note that for each binary clause (u, v), both $(\overline{u}, v)$ and $(\overline{v}, u)$ are edges in this graph (thus, the total number of edges in the initial graph, before further processing, is twice the size of B). For this reason we say that Binary Implications Graphs are *symmetric*.

Definition 2.2.2 (Binary Transitive Closure of a literal)**.** Given a literal v, a set of literals denoted by $BTC(v)$ is the *Binary Transitive Closure of* v if it contains exactly those literals that are implied by v through the Binary Implications Graph.

Definition 2.2.3 (Failed literal)**.** A literal v is called a *Failed Literal* if setting its value to $TRUE$ and applying BCP causes a conflict.

Definition 2.2.4 (Propagation closure of a literal)**.** Given a non-failed literal u, a set of literals denoted by $PC(u)$ is the *Propagation closure of* u if it contains exactly those literals that are implied through BCP by u in the given CNF (not only the binary clauses).

It is easy to see that $BTC(v) \subseteq PC(v)$ for every literal v, because $PC(v)$ is not restricted to what can be inferred from binary clauses. Note that $v \in PC(u)$ implies that $u \rightarrow v$ and hence $\overline{v} \rightarrow \overline{u}$, but it is not necessarily the case that $\overline{u} \in PC(\overline{v})$, due to the limitations of BCP. For example, in the set of clauses $(\overline{x} \vee y), (\overline{x} \vee z), (\overline{y} \vee \overline{z} \vee w)$, it holds that $x \rightarrow w$ and hence $\overline{w} \rightarrow \overline{x}$, but BCP detects only the first direction. It disregards $\overline{w} \rightarrow \overline{x}$ because $\overline{w}$ does not invoke any unit clause. Hence, BCP lacks the symmetry of Binary Implications Graphs.

Definition 2.2.5 (The HyperBinRes Hyper resolution rule)**.** The HyperBinRes inference rule:

$$\frac{(l_1, \ldots, l_n) \quad (\bar{l}_1, l), \ldots, (\bar{l}_{n-1}, l)}{(l, l_n)} \qquad \text{for } n \geq 2 \tag{2.1}$$

HyperBinRes is a hyper resolution rule (resolution from more than two clauses). It is possible to compute the HyperBinRes *closure* (add all possible clauses according to this rule) in polynomial time, by analyzing the binary sub-theory of the formula, and invoking BCP.

2.3 Hyper Resolution

Hyper resolution can be thought of as a shortcut to a long sequence of standard resolution steps, that results only in the last clause in this sequence rather than in all intermediate results. It allows to retrieve much more information than otherwise possible by pure binary reasoning, while staying polynomial.

In general, any sequence of implications $l_1 \rightarrow \ldots \rightarrow l_n$ on the Binary Implications Graph that does not end with a failed literal, can potentially lead to hyper resolution through a standard application of BCP on the given formula (including all clauses). The following lemma proves this:

Lemma 2.3.1. *Given a CNF formula F and a set $PC(v)$ as defined before, there is a sequence of HyperBinRes derivations for every literal $u \in PC(v)$ that proves $(\overline{v}, u)$.*

Proof. By induction on the literals that are propagated with BCP. First, BCP is seeded by v itself and, of course, $v \in PC(v)$. Clause $(\overline{v}, v)$ is universally true in the trivial way.

Suppose now that the clause $(l_1, \ldots, l_n, u)$ propagates u and the lemma holds for all literals propagated before u. It must be that l_i is FALSE for $1 \leq i \leq n$. Since all propagated literals are a result of setting v to TRUE, then it must be that $\bar{l}_i \in PC(v)$. We have by induction that there exists a sequence S_i of HyperBinRes derivations that proves $(\overline{v}, \bar{l}_i)$. Now, we use HyperBinRes again to derive

$$\frac{(\overline{v}, \bar{l}_1), \ldots, (\overline{v}, \bar{l}_n), (l_1, \ldots, l_n, u)}{(\overline{v}, u)},$$

and add it to the sequence $S_1 \ldots S_n$, which results in $(\overline{v}, u)$. □

□

Hyper Resolution can solve the problem of lack of symmetry in BCP. In the example above, in which BCP cannot detect that $\overline{w} \to \overline{x}$, applying Hyper Resolution on the given clauses produces the new binary clause $(\overline{x} \vee w)$. Clearly, now both $w \in PC(x)$ and $\overline{x} \in PC(\overline{w})$. In addition, realizing this new clause as a syntactic primitive allows algorithms that work on Binary Implications Graphs to use the fact that $x \to w$ or $\overline{w} \to \overline{x}$. For example, an algorithm that looks for equivalences between literals searches for cycles in this graph, and this new clause can lead to additional such cycles.

Each added binary clause (recall that this corresponds to two edges in the Binary Implications Graph) adds implicants not only to $BTC(v)$ for some v, but also to $PC(v)$ because of the symmetry discussed above of binary clauses. The goal of Hypre is to add such clauses until both of these sets are equal, or, in other words:

> Goal: *Build a Binary Implications Graph such that $BTC(v) = PC(v)$ for every literal v.*

The Hypre algorithm

Hypre is a recursive algorithm, which processes a node v in the Binary Implications Graph only after returning from processing its children, i.e. in a *post-order*. It then performs BCP in order to compute the propagation closure of v unless v is a Failed Literal. If v is a Failed Literal then a new unit clause $(\overline{v})$ is added to the formula, and unit propagation is applied. Otherwise, the set $NewDescendants := PC(v) \backslash BTC(v)$ is computed. For each literal $u \in NewDescendants$, a new binary clause $(\overline{v}, u)$ is added to the formula and $NewDescendants$ is updated to be $NewDescendats \setminus BTC(u)$ (this is an optimization, aimed at reducing *forward edges*, i.e., edges to nodes to which we already have a path from v). When Hypre leaves v, it is guaranteed that either v is a Failed Literal or $PC(v) = BTC(v)$.

We call a node v *strong* if Hypre already concluded that $BTC(v) = PC(v)$ with respect to the current formula. There is nothing to be done for strong nodes before the formula changes again. A *weak* node is a node that is not strong. Note that it is invariantly true that if a node v is weak, then all of its ancestors are also weak. HYPRE begins by marking all nodes as weak, and then gradually processes them and changes their marking to strong. When a binary clause $(\overline{v}, u)$ is added as described above, $\overline{u}$ and its ancestors have to be marked as weak, which means that HYPRE has to process them again. Note that v should not be marked as weak, since HYPRE currently processes it.

To detect equivalent literals, occasionally HYPRE searches for Strongly Connected Components (SCC) in the graph. It is easy to see that literals in a cycle are equivalent. Each SCC is replaced with a single node (a 'representative literal'), not only in the binary clauses, rather in the entire formula. This operation, as well as unit propagation (when a new unit clause is added), can simplify clauses (shorten them) which means that HYPRE needs to reconsider nodes that were already visited before. When an n-ary clause $(l_1, \ldots, l_n)$ is shortened to, e.g., $(l_1, \ldots, l_k), 2 \leq k < n$, then the literals $\bar{l}_1 \ldots \bar{l}_k$ and all their ancestors are marked as weak, since further processing of these nodes may lead to more hyper resolutions given the new shortened clause. A more elaborated justification of this step can be found in [2].

The main computational cost of HYPRE is due to the need to perform BCP on each node at least once, but on average many more times, due to the iterative nature of the algorithm. It is still far more efficient than previous approaches like 2-SIMPLIFY and 2CL_SIMP [5, 11] that stored the full transitive closure of the binary sub-theory and are therefore incapable of handling large instances.

HYPRE also preserves an interesting optimality property: If $y \in PC(x) \cap PC(z)$ and $x \xrightarrow{*} z$ (read: there is a path from x to z in the binary Implications Graph), then HYPRE guarantees, by the post-order in which it progresses, that it adds the edge $z \rightarrow y$ and not $x \rightarrow y$. The former is a stronger implication, because from this implication it is possible to infer the latter implication through the Binary Implications Graph.

2.4 The HyperBinFast algorithm

As stated in the introduction, our algorithm HyperBinFast builds upon and improves Hypre in two dimensions: it is capable of giving useful auxiliary information even when stopped before termination, and it is more efficient for the price of generating less information. Our experiments, as shown in Section 2.6, prove that this shifting of emphasis is worth while: although the SAT solving time after Hypre can be smaller, the total time is typically larger.

Algorithm 3

HyperBinFast

```
1: Mark all root vertices as weak;
2: while there are weak roots, unit clauses, or binary cycles do
3:     while there are unit clauses or binary cycles do
4:         Detect all SCCs and collapse each one of them to a single node;
5:         Propagate all unit clauses and simplify all clauses accordingly;
6:         For each new binary clause $(u, v)$ mark as weak $roots(\overline{u})$ and $roots(\overline{v})$;
7:     end while
8:     Choose a weak root node $v$;
9:     $FailedLiteral$ =FastVisit ($v$);
10:    Undo assignments caused by BinaryWalk and clear bQueue;
11:    if $FailedLiteral \neq$ undefined then        ▷ $FailedLiteral$ holds a failed literal
12:        Add unit clause ($\overline{FailedLiteral}$);
13:    end if
14:    Mark $v$ as strong;
15: end while
```

Algorithm 3 gives a bird-eye view of HyperBinFast. HyperBinFast iterates over all root nodes in the Binary Implications Graph ($roots(v)$ denotes the set of all ancestor roots of v in such a graph). It has two main stages. In the first stage (lines 4 - 5) it iteratively finds equal literals (by detecting SCCs and unifying their vertices to a single 'representing literal'), propagates unit clauses, and simplifies the clauses in the formula. Simplification in this context corresponds to substituting literals by their representative literal in all clauses (not only binary), removing literals that are evaluated to FALSE and removing satisfied clauses. The simplification may result in shortening of some n-ary clauses to binary clauses, which change the Binary Implications Graph. In line 6 we perform a restricted version of what HYPRE does in such cases: while HYPRE marks as weak all ancestor nodes, HYPERBINFAST only marks root ancestor nodes. Further, while HYPRE invokes this process every time an n-ary clause is being shortened, HYPERBINFAST only does so for clauses that become binary. The reduced overhead due to these changes is clear. In the second stage (line 9), we invoke FASTVISIT for some weak root node, a procedure that we will describe next. FASTVISIT can change the graph as well, so HYPERBINFAST iterates until convergence.

Algorithm 4

```
1: procedure BINARYWALK (Literal $t$, Antecedent clause C)
2:     if value($t$)=True then
3:         return $undefined$;
4:     end if
5:     if value($t$)=False then
6:         return $\bar{t}$;                ▷ a failed literal which was marked before
7:     end if
8:     $value(t) \leftarrow TRUE$;
9:     $antecedent(var(t)) \leftarrow C$;
10:    Put $t$ on assignment stack;
11:    Put $t$ into bQueue;
12:    for each binary clause $(\bar{t}, u)$ in 2-CNF sub-theory do
13:        $res \leftarrow$ BINARYWALK ($u$, $(\bar{t}, u)$);
14:        if $res \neq undefined$ then
15:            return $res$;
16:        end if
17:    end for
18:    return $undefined$;
19: end procedure
```

Computing the Binary Transitive Closure

Before describing FASTVISIT, we concentrate on the auxiliary function BINARYWALK 4, which FASTVISIT calls several times. The goal of BINARYWALK is to mark all literals that are in TBC(v) or return a failed literal, which can be either v itself or some descendant of v. It also updates a queue, called *bQueue* with those literals in TBC(v) for future processing by FastVisit. BinaryWalk performs DFS from a given literal on the Binary Implications Graph. In each recursive-call, if t is already set to false (i.e. $\bar{t}$ is already set to true in the current call to FastVisit), it means that there is a path in the binary implication graph from $\bar{t}$ to t, and hence $\bar{t}$ is a failed literal. This is a direct consequence of the following lemma:

Lemma 2.4.1. *In a DFS-traversal on a Binary Implications Graph from a literal u that marks all nodes it visits, if when visiting a node t another node $\bar{t}$ is already marked, and this is the* first time *such a 'collision' is detected, then $\bar{t} \xrightarrow{*} t$.*

Proof. Falsely assume that there is no path $\bar{t} \xrightarrow{*} t$. The fact that traversal from u leads to marking of both $\bar{t}$ and t, implies that $u \xrightarrow{*} \bar{t}$ and $u \xrightarrow{*} t$. By the symmetry of Binary Implications Graphs, there is also a path $\bar{t} \xrightarrow{*} \bar{u}$. Since the DFS traversal visits $\bar{t}$ before t then it must visit $\bar{u}$ before leaving $\bar{t}$. Moreover, it cannot visit t before leaving $\bar{t}$ because otherwise this would contradict our assumption that no such path exists. This leads to a collision between u and $\bar{u}$, before a collision is detected between t and $\bar{t}$, which contradicts our assumption that the latter was detected first. Hence, $\bar{t} \xrightarrow{*} t$. □

□

When BinaryWalk detects such a failed literal it returns $\bar{t}$ all the way out (due to lines 14-15) and back to FastVisit and then to HyperBinFast.

The other case is when t does not have a value yet. In this case BinaryWalk sets it to true and places it in *bQueue*, which is a queue of literals to be propagated later on by FastVisit. It also places t in the (global) assignment stack, and stores for $var(t)$ its antecedent clause (the clause that led to this assignment), both for later use in FindUIP.

From Binary Transitive Closure to Propagation Closure

We now describe FastVisit. Recall that FastVisit is invoked for each root node in the Binary Implications Graph. FastVisit combines unit propagation with binary learning based on single assignments, i.e. learning of new clauses by propagating a single decision at a time. It relies on the simple observation that if $u \in PC(v)$ then $v \rightarrow u$. It is too costly to add an edge for every such pair v, u, because this corresponds to at least computing the transitive closure [2]. Since our stated goal is to form a binary graph in which $PC(v) = BTC(v)$ for each root node, it is enough to focus on a vertex u only if $u \in PC(v)$ but $u \notin BTC(v)$. Further, given such a vertex u, although adding the edge $v \rightarrow u$ achieves this goal, we rather find a vertex w, a descendant of v that also implies u, in the spirit of the First Unique-Implication-Point (UIP) scheme that is used by most modern SAT solvers. The FindUIP function (see Algorithm 6) called by FastVisit can in fact be seen as a variation of the standard algorithm for finding first UIPs [9]: unlike the standard usage of such a function in analyzing conflicts, here there are no decision levels and the clauses are binary. On the other hand it can receive as input an arbitrary set of assigned literals, and not just a conflict clause. In contrast to Hypre, which performs unit propagation from each node in a post order, HyperBinFast only propagates from the roots, and the edges that it adds depend on the specific DFS run it performs. It therefore cannot guarantee the optimality property discussed in the end of the previous section. Invoking FindUIP attempts to compensate on this fact, but it cannot guarantee it.

In line 6 FastVisit starts to process the literals in *bQueue*. For each literal p in this queue, it checks all the n-ary clauses ($n > 2$) watched by p. As usual, each such clause can be of interest if it is either conflicting or unit. If it is conflicting, then FastVisit calls FindUIP, which returns the first UIP causing this conflict. This UIP is a failed literal and is returned to HyperBinFast, which adds its negation as a unit clause in line 12. If the processed clause is a unit clause, the unassigned literal, denoted by *toLit*, is a literal implied by v that is not in $BTC(v)$ (otherwise it would be marked as TRUE in BinaryWalk). In other words, $toLit \in UP(v)$ and

Algorithm 5

```
procedure FastVisit (Literal $v$)
    $res \leftarrow$ BinaryWalk ($v$, NULL);
    if $res \neq undefined$ then
        return $res$
    end if
    while !bQueue.empty() do
        Literal $p \leftarrow$ bQueue.pop_front();
        for each n-ary clause $\in$ watched(p) do                    $\triangleright\ n > 2$
            if $clause$ is conflict then
                Literal $fUIP \leftarrow$ FindUIP ($clause$);
                return $fUIP$;
            else if $clause$ is unit then
                Literal $toLit \leftarrow$ undefined literal from $clause$.
                Literal $fromLit \leftarrow$ FindUIP ($clause \setminus \{toLit\}$);
                Add clause ($\overline{fromLit}, toLit$)
                Mark $roots(\overline{toLit}) \cup roots(\overline{fromLit})$ as weak
                $res \leftarrow BinaryWalk(toLit, (\overline{fromLit}, toLit))$;
                if $res \neq undefined$ then
                    return $res$
                end if
            end if
        end for
    end while
    return $undefined$;
end procedure
```

Algorithm 6

```
1: procedure FINDUIP (Literal set $S$)
Require: implication graph is binary only.
Ensure: $res$ is first UIP of $S$
2:     mark all variables in $S$;
3:     $count \leftarrow |S|$;
4:     while $count > 1$ do
5:         $v \leftarrow$ latest marked variable in the assignment stack
6:         unmark $v$ and decrease $count$ by one.
7:         Let $(u, L)$ be antecedent clause of $v$, s.t. $var(L) = v$.
8:         if $var(u)$ not marked then
9:             mark $var(u)$ and increase $count$ by one.
10:        end if
11:    end while
12: end procedure
13: $res \leftarrow$ last marked literal in assignment stack.
14: unmark $var(res)$;
15: return $res$;
```

$toLit \notin BTC(v)$, which is exactly what we are looking for. At this point we can add a clause $(\overline{v}, toLit)$ but rather we call FINDUIP, which returns a first UIP denoted by $fromLit$. The clause $(\overline{fromLit}, toLit)$ is stronger than $(\overline{v}, toLit)$ because the former also adds the information that $\overline{t}olit \rightarrow fromlit$. Note that this is an unusual use of this function, because *clause* is not conflicting. Because the addition of this clause changes the Binary Implications Graph, we need to mark as weak all the ancestor nodes of $fromLit$ and of $\overline{toLit}$, and to continue with BINARYWALK from $toLit$. This in effect continues to compute $BTC(v)$ with the added clause.

2.5 discussion

2.5.1 Differences from HYPRE

One of the differences between fast and HYPRE can be demonstrated with the following example. Suppose we have a direct sub-graph with n vertices, rooted at a node

r. Hypre goes recursively over all descendants of r and apply UP at each one of them and finds *all edges* that are needed to achieve the stated goal of making the binary graph represent the propagation closure of r and its descendants. There can be n propagations, which can many times find only a very small number of edges, sometimes even zero such edges. Moreover, it can also generate edges which connect descendants of r between them. HyperBinFast, on the other hand, applies only one UP at the root r, and finds *all literals* that should have new edges leading to them. It will not connect existing descendants of r but concentrate on those which do not have any path from r to them.

Hypre generally adds more information than HyperBinFast. For example, if two descendants of v, p and s prove a new literal t, then HyperBinFast finds only one of them, e.g. it could add $p \rightarrow t$. In other words, it finds only one edge for each UP-implied literal. Also, while FindUIP tries to find the best binary clauses to add, it does not guarantee optimality. It could add $p' \rightarrow t$, where p' is some literal between v and p.

HyperBinFast can be seen as a restricted version of DPLL with non-conflict learning. Indeed, it "decides" on the root variables and never goes below decision level 1. Also, it learns a new clause every time an implication by an n-ary clause occurs. It is possible to change HyperBinFast so it uses decision heuristics other than just choosing the roots. We leave this option for future research.

2.5.2 Bounding the runtime of HyperBinFast

Given that HyperBinFast is an anytime algorithm, there are various strategies to decide when to stop it. The most naive method, of course, is to use a time-limit. While being extremely simple, it still enables us to balance between the preprocessing and the solving stage, according to the efficiency of the SAT solver. One somewhat unexpected disadvantage of this method is that it can lead to a nondeterministic solving process and run time[1]. The reason for this is the imperfection of measuring

[1]It is the policy of the SAT competition's organizers to not accept non-deterministic solvers.

time: differences in mili-seconds as measured by the computer clock can stop the preprocessor in different stages and hence lead to a different instance for the SAT solver. We therefore developed a heuristic function that decides when to stop the preprocessor according to its progress. In particular, it measures the number of deduced unit clauses and equivalent literals, versus decaying rate of the weak nodes. This way we do not allow HyperBinFast to work for a long time without producing evidence of its efficiency. If the weak nodes are not decreased fast enough comparing to what is expected from the number of reduced variables, HyperBinFast is stopped and its output is forwarded to the SAT solver. HyperBinFast performs this check every constant number of invocations of FastVisit. There are numerous possible heuristics to perform this check. Currently our heuristic, which is still under development, is better or equally good as a fixed time-out of 300 seconds, although in a few instances it is worse (it stops the preprocessing too early). Altogether for the set of benchmarks reported in the next section the total solving time of the automated and fixed timeouts are comparable. We believe that with further experiments and tuning we will be able to make this technique dominant over a fixed timeout strategy.

2.6 Experiments, conclusions and directions for future research

We ran HyperBinFast and the original preprocessor Hypre [1] with several SAT solvers: siege_v1 [7], zChaff 2004 and our experimental SAT solver HaifaSat. We do not present results for the latest version of Siege, siege_v4, because we do not know if this undocumented solver already uses similar learning rules internally. Nevertheless, we ran siege_v4 with both Hypre and HyperBinFast and saw that it is not cost-effective to run either one of them. The overhead of HyperBinFast, however, was much smaller than that of Hypre.

Table 2.1 shows experiments on an Intel 2.5Ghz computer with 1GB memory running Linux. The benchmark set is comprised of 165 industrial instances used in various SAT competitions. In particular, *fifo8, bmc2, CheckerInterchange, comb, f2clk, ip, fvp2, IBM02 and w08* are hard industrial benchmarks from SAT02; *hanoi* and *hanoi03* participated in SAT02 and SAT03; *pipe03* is from SAT03 and *01_rule, 11_rule_2, 22_rule, pipe-sat-1-1, sat02, vis-bmc, vliw_unsat_2.0* are from SAT04 [10, 3, 4]. The number in brackets for each benchmark set denotes the number of instances. The timeout for each instance was set to 3000 seconds. When relevant, the timeout for the preprocessor itself was set to 300 seconds and the timeout for the SAT solver was dynamically reduced to 3000 minus the time spent during preprocessing. All times in the table include preprocessing time when relevant. We count each failure as 3000 seconds as well.

The table shows that:

1. HyperBinFast helps each of the tested solvers to solve more instances in the given time bound.
2. When the instance is solvable without HyperBinFast, still HyperBinFast typically reduces the overall run time.
3. Whenever HyperBinFast does not help, its overhead in time is relatively small.
4. It is very rare that an instance can be solved without HyperBinFast but cannot be solved with HyperBinFast.
5. On average, the total gain in time is about 20-25%.

Table 2.2 compares the performance of Hypre and HyperBinFast. Since Hypre is not implemented as an anytime algorithm, and as a preprocessor it always takes more time than HyperBinFast (empirically), its only advantage when compared to HyperBinFast can be that it produces better information that compensates on the extra preprocessing time. In order to test this possibility we ran Hypre with a timeout of 3000 seconds and the SAT solver with a timeout of 3000 seconds minus the time spent by HYPRE. In other words we compared three configurations for each SAT solver: plain SAT, SAT solver+HYPRE, and SAT solver+HYPERBINFAST, all with a global timeout of 3000 seconds per instance. The time policy of HYPERBINFAST was left as in the previous experiment. Table 2.2 compares these configurations with both HaifaSat and Siege_v1. The times include both the preprocessing and the SAT solving run times. The table shows that sometimes HYPERBINFAST is not 'strong' enough (it does not simplify the formula enough), so the SAT solver fails on the corresponding instance but succeeds after applying HYPRE. Nevertheless, the total time is always smaller with HYPERBINFAST. Moreover, it can be seen that with HaifaSat, HYPRE is not cost-effective, neither in the total number of failures or the total run time, while HYPERBINFAST reduces HaifaSat's failures by 35% and reduces its total solving time by 25%.

More data: For the above benchmark, it took HaifaSat 97,909 seconds after HYPERBINFAST and only 54,567 seconds after HYPRE, which indicates that indeed the quality of the CNF generated by HYPRE is better, as expected. But these numbers may mislead because, recall, the timeouts for the two preprocessors are different, which, in turn, is because HYPERBINFAST is an anytime algorithm. In Table 2.3 we list several benchmarks for which both preprocessors terminated before their respective timeouts, together with the time it took the preprocessor and then HaifaSat to solve them. To the extent that these instances are representative, it can be seen that typically the solving time is longer after HYPERBINFAST, but together with the SAT solver time it is more cost effective than HYPRE.

Conclusions and directions for future research.

Preprocessing can be cost-effective when combined with modern SAT solvers, as is evident from our experiments with 165 industrial CNF instances from previous SAT competitions. We pointed to two directions for future research: develop more efficient dynamic strategies for determining the amount of time spent for preprocessing, and make preprocessing *decide* on the set of variables from which it begins its traversal of the Binary Implications Graph (and not just choose all the root nodes as we do now). This concept can be generalized to preprocessing in general: while SAT solvers focus on the *semantics* of the formula, that is, they attempt to find the ‘important’ variables, preprocessors focus on the *syntactical* characteristics of the formula, and are therefore much more sensitive to its size. Hence, attempting to build a *semantic preprocessor* seems like a worth while direction to pursue next: the CNF instances that are hard to solve with modern solvers become larger every year, so becoming less affected by their sheer size seems like the only way for preprocessors to stay in the game.

SAT solver →	HaifaSat				Siege_v1				zChaff 2004			
Preprocessor →	—		H-B-Fast		—		H-B-Fast		—		H-B-Fast	
Benchmark:	Time	F	Time	F	Time	F	Time	F	Time	F	Time	F
01_rule(20)	19,172	2	**7,379**	0	20,730	4	**11,408**	1	20,779	4	19,196	5
11_rule_2(20)	22,975	6	**7,491**	0	29,303	8	**17,733**	2	36,042	10	**27,500**	8
22_rule(20)	27,597	8	**22,226**	5	31,839	10	29,044	9	31,279	9	**25,377**	6
bmc2(6)	1,262	0	**81**	0	3,335	1	**85**	0	3,174	1	**83**	0
CheckerI-C(4)	**682**	0	902	0	4,114	0	**3,541**	0	816	0	**703**	0
comb(3)	4,131	1	4,171	1	5,679	1	6,027	1	6,363	2	6,237	2
f2clk(3)	4,059	1	4,060	1	6,105	2	6,063	2	6,090	2	6,047	2
fifo8(4)	1,833	0	**554**	0	5,555	1	**2,420**	0	5,206	1	**3,390**	1
fvp2(22)	1,995	0	2,117	0	1,860	0	2,009	0	7,078	0	**3,830**	0
hanoi(5)	**131**	0	285	0	**357**	0	1,231	0	**2,151**	0	2,435	0
hanoi03(4)	**427**	0	533	0	6,026	2	6,028	2	6,022	2	6,016	2
IBM02(9)	**3,876**	0	5,070	0	10,442	4*	**7,881**	0	9,596	3	**8,132**	1
ip(4)	203	0	**172**	0	630	0	**548**	0	**1,065**	0	3,538	1
pipe03(3)	1,339	0	1,266	0	2,006	0	**1,275**	0	4,106	1	**1,254**	0
pipe-sat-1-1(10)	**3,310**	0	5,147	0	**2,445**	0	5,249	0	**4,568**	0	8,950	0
sat02(9)	17,330	4	**14,797**	4	24,182	7	**18,843**	5	23,632	7	**20,333**	6
vis-bmc(8)	13,768	3	**10,717**	2	10,449	2	**6,989**	1	18,358	6	**13,389**	4
vliw_unsat_2(8)	19,425	5	19,862	6	16,983	6*	17,891	6*	21,867	7	21,364	6
w08(3)	2,681	0	**1,421**	0	4,387	1	**1,711**	0	4,316	1	**2,223**	0
Total(165)	146,194	30	**108,251**	19	186,426	49	**145,978**	29	212,508	56	**179,997**	44

Table 2.1: Run-times (in seconds) and failures (denoted by 'F') for various SAT solvers with and without HyperBinFast. Times which are smaller by 10% than in competing configurations with the same SAT solver are bolded. Failures denoted by * are partially caused by bugs in the SAT solver.

SAT solver →	HaifaSat						Siege_v1					
Preprocessor →	—		HYPRE		H-B-FAST		—		HYPRE		H-B-FAST	
Benchmark:	Time	F	Time	F	Time	F	Time	F	Time	F	Time	F
01_rule(20)	19,172	2	10,758	1	**7,379**	0	20,730	4	**5,318**	0	11,408	1
11_rule_2(20)	22,975	6	21,247	0	**7,491**	0	29,303	8	20,178	1	**17,733**	2
22_rule(20)	27,597	8	**16,825**	2	22,226	5	31,839	10	**17,510**	3	29,044	9
bmc2(6)	1,262	0	163	0	**81**	0	3,335	1	156	0	**85**	0
CheckerI-C(4)	**682**	0	989	0	902	0	4,114	0	**2,492**	0	3,541	0
comb(3)	4,131	1	4,056	1	4,171	1	5,679	1	**4,439**	1	6,027	1
f2clk(3)	4,059	1	**3,448**	1	4,060	1	6,105	2	**5,078**	1	6,063	2
fifo8(4)	1,833	0	1,756	0	**554**	0	5,555	1	**1,159**	0	2,420	0
fvp2(22)	1,995	0	3,288	0	2,117	0	1,860	0	2,431	0	2,009	0
hanoi(5)	131	0	119	0	285	0	357	0	802	0	1,231	0
hanoi03(4)	**427**	0	979	0	533	0	6,026	2	6,014	2	6,028	2
IBM02(9)	**3,876**	0	11,072	3	5,070	0	10,442	4	12,653	3	**7,881**	0
ip(4)	203	0	365	0	**172**	0	630	0	**349**	0	548	0
pipe03(3)	1,339	0	1,809	0	1,266	0	2,006	0	1,822	0	**1,275**	0
pipe-sat-1-1(10)	**3,310**	0	27,130	10	5,147	0	2,445	0	30,029	10	5,249	0
sat02(9)	17,330	4	16,669	4	**14,797**	4	24,182	7	17,662	4	18,843	5
vis-bmc(8)	13,768	3	10,139	2	10,717	2	10,449	2	**5,715**	0	6,989	1
vliw_unsat_2(8)	19,425	5	20,421	6	19,862	6	16,983	6	20,375	6	17,891	6
w08(3)	2,681	0	2,899	0	**1,421**	0	4,387	1	2,726	0	**1,711**	0
Total(165)	146,194	30	154,132	30	**108,251**	19	186,426	49	156,910	31	145,978	29

Table 2.2: Run-times (in seconds) and failures (denoted by 'F') for HaifaSat and Siege_v1, without preprocessing and when combined with HYPRE and HYPERBIN-FAST. All run-times include both the preprocessing and the SAT solving times. Times which are smaller by 10% than in competing configurations with the same SAT solver are bolded.

	Hypre		HyperBinFast	
Benchmark:	Hypre	SAT	HyperBinFast	SAT
01_rule.k95.cnf	377	1679	4	1504
11_rule2.k70.cnf	1387	47	71	285
22_rule.k70.cnf	671	251	51	1410
fifo8_400.cnf	164	1226	12	309
7pipe.cnf	651	258	147	416
ip50.cnf	109	82	6	79
w08_14.cnf	1231	5	267	298
Total:	4590	3548	558	4301

Table 2.3: Few representative instances for which both Hypre and HyperBinFast terminated before their respective (different) timeouts. The SAT times refer to HaifaSat's solving time. It can be seen that typically the solving time is longer after HyperBinFast, but together with the SAT solver time it is more cost effective than Hypre.

Chapter 3

The CMTF decision heuristic

3.1 Introduction

A SAT solver can be thought of as a search engine based on *enumeration* of solutions, but also as a *proof engine* based on inference through the resolution rule. Traditionally the first view was dominant, hence the emphasis in designing SAT solvers and explaining their success was on pruning search spaces. Decision heuristics and learning schemes can all be interpreted as aiming at this goal. Yet the harder and larger the CNF instances are, pruning alone cannot account for the success of modern SAT solvers. It is their ability as proof engines that makes them succeed. This distinction has practical implications, too. For example, for many years decision heuristics gave higher priority to variables in shorter clauses, and to learning shorter conflict clauses. The reasoning was that such clauses can potentially prune larger search-spaces. Although this claim is true, all modern decision heuristics (VSIDS [8], VMTF [7], Berkmin [6]) ignore the length of the clauses, after reaching empirically the conclusion that there are more important considerations. Ryan experimented in his thesis [7] with first-UIP and all-UIP learning schemes, and although the latter generate on average shorter clauses, the former is empirically better. He hypothesized that the learning scheme should be geared towards resolution rather than for pruning. In this chapter we extend this approach by looking on clause-learning and the decision heuristic as one complete mechanism and refer to a SAT solver as a prover rather than as a search engine. It turns out, empirically, that when conflict clauses are effective, which is the case in all real-world instances, this is the right way to go.

Not only that a DPLL-based SAT-solver can be seen as a proof engine, various strategies, we argue, can be explained through the popular abstraction/refinement framework, which is very common in verification. The connection between these two worlds is due to the fact that conflict clauses are derived through a process of resolution. If a clause c is derived by resolution from a set of clauses $c_1 \ldots c_n$ then

$$c_1 \wedge \cdots \wedge c_n \rightarrow c$$

while the other direction does not hold. As we will show in Section 3.2, we can view c as an over-approximating abstraction of the resolving clauses $c_1 \ldots c_n$. Attempting to satisfy c first, therefore, can be seen as an attempt to satisfy the abstract model first. And like any abstraction/refinement technique, a successful assignment to c is one that satisfies the concrete model (the $c_1 \ldots c_n$ clauses) as well. Further, an unsuccessful assignment leads to a refinement step, or, in our case, to derivation of new conflict clauses which further constrain the abstract model. According to this model, Berkmin is only one of many possible strategies to refine the abstract model. In Section 3.3 we suggest one such alternative clause-based decision heuristic called Clause-Move-To-Front (CMTF), which attempts to follow the order of the clauses in the *resolve-graph* rather than their chronological order in which they were created. In Section 4 we also show a resolution-based score function for choosing the variable from the selected clause and a similar function for choosing the sign. In Section 4.3 we report experimental results on hundreds of industrial benchmarks that prove the advantage of our approach.

3.2 Background

The explanation of our methods and the analysis of various heuristics later on will require some basic definitions.

The Abstraction-refinement model: from structures to formulas

The classic use of the terms abstraction and refinement in the context of model-checking is the following. Let M be a Kripke structure, $L(M)$ the set of propositions labeling its states and $\mathcal{L}(M)$ the language defined by M. A model $\hat{M}$ is an over-approximating abstraction of M such that $L(\hat{M}) \subseteq L(M)$, if for every property φ

$$\hat{M} \models \varphi \rightarrow M \models \varphi. \tag{3.1}$$

Equivalently, for every string s,

$$s \in \mathcal{L}(M) \rightarrow s \in \mathcal{L}(\hat{M}). \tag{3.2}$$

The inclusion relation is defined with respect to the alphabet of the language, e.g., $s \in \mathcal{L}(M)$ is defined with respect to the projection of s to $L(M)$.

M_1 is a *refinement* of $\hat{M}$ with respect to M, if for every string s,

$$s \in \mathcal{L}(M) \rightarrow s \in \mathcal{L}(M_1), \tag{3.3}$$

and

$$s \in \mathcal{L}(M_1) \rightarrow s \in \mathcal{L}(\hat{M}). \tag{3.4}$$

Abstraction-Refinement is a process in which we find increasingly accurate models (closer to the concrete model M) until proving the property or, in the worst case, reaching the original model M.

In this chapter we wish to bridge between the terminology and notations of models and strings on one hand, and the terminology and notations of formulas and satisfying assignments on the other hand. Thus, consider now formulas rather than models.

For two formulas f and $\hat{f}$ such that $var(\hat{f}) \subseteq var(f)$, we can restate an implication of the form

$$f \rightarrow \hat{f}, \tag{3.5}$$

by saying that for every assignment α,

$$\alpha \models f \rightarrow \alpha \models \hat{f}. \tag{3.6}$$

As usual satisfaction is defined with respect to a projection of α to the variables of the formula.

Due to the resemblance to Formula 3.2, we now say that $\hat{f}$ is a conservative abstraction (over-approximation) of f.

Further, for a formula f_1 such that $var(\hat{f}) \subseteq var(f_1) \subseteq var(f)$, we can restate

$$f \rightarrow \hat{f_1} \tag{3.7}$$

and

$$\hat{f_1} \rightarrow \hat{f} \tag{3.8}$$

by saying that for every assignment α,

$$\alpha \models f \rightarrow \alpha \models f_1, \tag{3.9}$$

and

$$\alpha \models f_1 \rightarrow \alpha \models \hat{f}. \tag{3.10}$$

Once again, due to the resemblance of Formulas 3.3 and 3.4 to Formulas 3.9 and 3.10, we now say that f_1 refines $\hat{f}$ with respect to f.

Continuing with this terminology, abstraction-refinement for formulas is an iterative process, in which one begins with some abstract formula $\hat{f}$ of a concrete formula f and gradually refines it through a series of formulas $\hat{f_1}, \ldots, \hat{f_n}$ until proving or disproving the desired property of f. Here again, in the worst case $\hat{f_n} = f$. Thus, there is a parallelism between abstraction refinement of structures, and the process described here for formulas.

3.2.1 Conflict clauses and resolution

The well-known binary resolution rule is:

$$\frac{a_1 \vee \ldots \vee a_n \vee \beta \qquad b_1 \vee \ldots \vee b_m \vee (\neg\beta)}{a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m}$$

where $a_1, \ldots a_n, b_1, \ldots b_m, \beta$ are literals. β is known as the *resolution variable* of this derivation. Clauses $(a_1, \ldots, a_n, \beta)$ and $(b_1, \ldots, b_n, \overline{\beta})$ are called *resolving clauses* and clause $(a_1, \ldots, a_n, b_1, \ldots b_n)$ is a *resolvent*. It follows by the soundness of the rule, that the resolvent is always implied by its resolving clauses and, using the terminology of §3.2, can be thought of as an abstraction of the clauses that participated in the derivation.

Algorithm 7 The First-UIP resolution algorithm

```
procedure AnalyzeConflict(Clause: conflict)
    $currentClause \leftarrow conflict$;
    $ResolveNum \leftarrow 0$;
    $NewClause \leftarrow \emptyset$;
    repeat
        for each literal $lit \in currentClause$ do
            $v \leftarrow var(lit)$;
            if $v$ is not marked then
                Mark $v$;
                if $dlevel(v) = CurrentLevel$ then
                    $++\ ResolveNum$;
                else
                    $NewClause \leftarrow NewClause \cup \{lit\}$;
                end if
            end if
        end for
        $u \leftarrow$ last marked literal on the assignment stack;
        Unmark $var(u)$;
        $--\ ResolveNum$;
        $ResolveCl \leftarrow Antecedent(u)$;
        $currentClause \leftarrow ResolveCl \setminus \{u\}$;
    until $ResolveNum = 0$;
    Unmark literals in $NewClause$;
    $NewClause \leftarrow NewClause \cup \{\overline{u}\}$;
    Add $NewClause$ to the clause database;
end procedure
```

We now show why the process of generating conflict clauses indeed can be seen as a sequence of resolution steps. Algorithm 7 shows a simple and efficient implementation of the First-UIP resolution scheme, which is implemented in most competitive SAT solvers, including our solver HaifaSat. We will refer to this algorithm simply as the *resolution* algorithm. First, a conflicting clause is set to be the current resolved clause. The main loop processes literals in the current clause. All literals from the previous decision levels are gathered into $NewClause$ at line 13 and marked. Literals from the current level are marked in order to resolve on them further (i.e., make them the resolution variables). In every iteration a new marked (yet unprocessed) literal u is chosen in line 17. This literal must be from the current decision level. The algorithm resolves on u by setting *currentClause* to be the antecedent clause without u.

ResolveNum counts the number of the marked literals from the current decision level that still have to be processed. When $ResolveNum = 0$ at line 22, then u is the *FirstUIP* or the *asserted* literal. The negation of this literal is added to the $NewClause$ causing u's value to be flipped after backtracking. For more details on the resolution algorithm see [8, 7].

We will use the following definition in order to denote the initial state of $NewClause$:

Definition 3.2.1 (Asserting clause)**.** Suppose a new conflict clause C was created in Alg. 7 with asserted literal u. Suppose also that the solver backtracks after the conflict to level dl. Then C becomes an *asserting* clause when it implies $\overline{u}$ for the first time at level dl, and stops being asserting when the solver backtracks from dl.

It follows from definition that every conflict clause becomes asserting exactly once.

Example 3. *Consider the following partial implication graph and set of clauses. Denote by* $Resolve(s, t, x)$ *the binary resolution of clauses* s *and* t *with the resolution variable* x*. Then the conflict clause* $c_5 : (x_{10}, x_2, \neg x_4)$ *is computed through a series of binary resolutions, starting from the conflicting clause* c_4*, and going backwards on the implication graph until all literals in the conflict clause are either from previous decision levels or the* $firstUIP$*.*

$$Resolve(Resolve(Resolve(c_4, c_3, x_7)), c_2, x_6), c_1, x_5) = (x_{10}, x_2, \neg x_4)$$

Algorithm 7 implicitly performs these resolution steps while computing the conflict clause c_5*.* □

$$\begin{array}{lcl} c_1 & = & (\neg x_4 \vee x_2 \vee x_5) \\ c_2 & = & (\neg x_4 \vee x_{10} \vee x_6) \\ c_3 & = & (\neg x_5 \vee \neg x_6 \vee \neg x_7) \\ c_4 & = & (\neg x_6 \vee x_7) \\ \vdots & & \vdots \end{array}$$

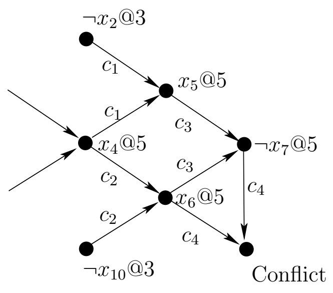

Figure 3.1: A partial implication graph and set of clauses demonstrate Analyze-Conflict. x_4 *is the FirstUIP, and* $complx_4$ *is the asserted literal.*

NewClause is derived through a series of binary resolutions that can be seen as a tree: every time the solver reaches line 21, an intermediate clause (consisting of all marked literals) is resolved with the antecedent clause of the chosen resolution variable. We can treat this process as one atomic action of *Hyper-resolution* (resolution between more than two clauses). Since each conflict clause is derived from a set of other clauses, we can keep track of this process with a *Resolve-Graph.* Here we define a variation of the well-known resolve-graph that distinguished between two types of resolutions:

Definition 3.2.2 (Colored Resolve Graph)**.** A *Resolve Graph* is a directed acyclic graph where each node corresponds to a clause, and there is an edge (u, v) if and only if v participated in the Hyper-resolution of u as a *CurrentClause* at line 6 of Alg 7.

The *color* of the edge (u, v) is defined to be blue if v was an asserting (conflict) clause during the resolution and red otherwise.

In this graph, edges come from the resolvent to its resolving clauses. The leafs of the graph correspond to the original clauses in the formula. Notice that since a conflict at level *dl* necessarily implies that the solver backtracks from *dl* and unassigns all the variables that were resolved on, any asserting clause which participated in the

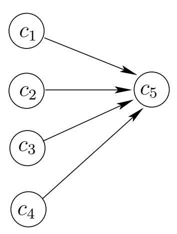

Figure 3.2: A resolve-graph corresponding to the implication graph in Figure 3.1

$$\begin{array}{lcl} c_1 & = & (\neg x_4 \vee x_2 \vee x_5) \\ c_3 & = & (\neg x_5 \vee \neg x_6 \vee \neg x_7) \\ c_4 & = & (\neg x_6 \vee x_7) \\ \vdots & & \vdots \end{array}$$

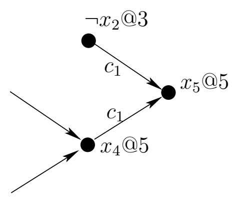

Figure 3.3: A partial implication graph corresponding to c_1, c_3, c_4 and the decision $x_4@5$.

resolution will stop being asserting. Therefore for any conflict clause there can be at most one incoming blue edge. The original clauses do not have outgoing edges, and only red incoming edges.

Example 4. *Consider once again the implication graph in Figure 3.1. Since $c_1 \dots c_4$ participate in the resolution of c_5, the corresponding resolve-graph is as appears in Figure 3.2. Assuming that $c_1 \dots c_4$ are original clauses, then all the edges in this graph are red, because original clauses cannot be asserting.*

Now consider a similar case in which c_2 is not an original clause, and at the time when $x_4@5$ is assigned it does not yet exist. The implication graph at this stage appears in Figure 3.3. Now assume that due to further decisions and implications in deeper decision levels a conflict is encountered, the solver creates the new conflict clause c_2, backtracks to decision level 5 and asserts $x_6@5$. This, in turn, completes the implication graph to the way it looks in Figure 3.1. But now, since c_2 asserts x_6,

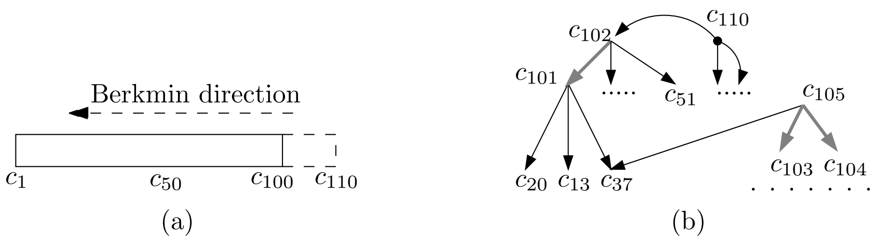

Figure 3.4: Berkmin's decision heuristic can be thought of as an abstraction-refinement process, where a range of the conflict clauses from the right end until c_i represents an abstract model of the clauses on the left of c_i. (a) Berkmin clauses stack: after encountering a conflict, the new resolved clauses are added on the right end. By the time the solver returns to c_{50}, it will have a partial assignment that satisfies a refined model, i.e. the clauses $c_{51} \ldots c_{110}$ (b) The resolve sub-graph of some newly created clauses. Grey thick edges denote the blue edges in the graph.

we consider its edge on the resolve-graph from c_5 *as blue.* □

The distinction between the two type of edges is important because a blue edge (u, v) indicates that the solver had to create u in order to later create v[1].

3.2.2 The Berkmin Decision heuristic

We have already described the Berkmin Decision heuristic in our introduction. Let see this heuristic in more detail.

Berkmin [6] pushes every new conflict clause to a stack, and makes a decision by choosing an unassigned variable from the last unsatisfied conflict clause in this stack (if there is more than one such variable, it uses the VSIDS score system). If all the conflict clauses are satisfied, it continues with a different heuristic.

In Fig. 3.4(a) we show a sketch of the progress of Berkmin, which is helpful in understanding why this process can be seen as abstraction-refinement. Clauses $c_1, \ldots, c_{100}$ are conflict clauses ordered by their creation time (c_1 is first). Berkmin tries to satisfy these clauses from last to first, i.e. from right to left. Suppose that all clauses $c_{51} \ldots c_{100}$ are already satisfied, and now Berkmin focuses on c_{50}. We refer to $S = \{c_{51}, \ldots, c_{100}\}$ as our current abstract formula of the original formula φ (it is

[1]By this we do not mean that this is the only way to create v.

abstract because each of the clauses in S is derived by a resolution chain from the clauses of φ). Clauses in S must be satisfied, since the decision heuristic reached c_{50}. Berkmin now makes a decision on a variable from c_{50} which leads to a conflict and learning of a new clause. The decision heuristic backtracks to the clauses on the end of the list, until finally, through possibly additional iterations of conflicts and added clauses, it reaches c_{50} again while all the clauses to its right are satisfied. Denote by S' the clauses to the right of c_{50} at this point, e.g. $S' = \{c_{51} \ldots, c_{110}\}$. Clearly $S \subseteq S'$ and S' is an abstraction of φ. We can therefore say that S' is a refinement of S with respect to φ.

This view of the process possibly explains why a strategy of giving absolute priority to variables in a specific clause is empirically better than previous approaches like VSIDS that used only a score function.

Fig. 3.4(a) shows a 'linear' view of the conflict clauses in the order that they are added, which is also the order in which they are considered by Berkmin. The Berkmin heuristic never tries to satisfy a clause before satisfying its resolvents and thus mimics a gradual process of refinement.

A different view of conflict clauses considers their partial order in the Resolve Graph. Fig.3.4(b) presents a possible Resolve sub-Graph corresponding to the same set of clauses. After the conflicts, Berkmin starts from satisfying c_{110}. c_{102} is a resolving clause that can potentially refine the initial model, however Berkmin first passes through $c_{105}, c_{104}, c_{103}$ to which c_{110} is not connected at all. Therefore Berkmin is dispersed trying to refine several abstractions. Such unfocused behavior can lead to longer proofs. This problem is exactly what our decision heuristic CMTF attempts to solve, as we soon show.

Our SAT solver HaifaSat makes a decision in three steps: it chooses an unsatisfied clause according to the CMTF heuristic, it then chooses an unassigned variables from this clause, and finally gives it a value. The next sections describe in detail these decision steps.

```
1: $S = roots(ResolveGraph)$;     ▷ The resolvent clauses that did not resolve other clauses.
2: Choose an unsatisfied clause (vertex) $v \in S$;
3: Process $v$;     ▷ Processing a clause, among other things, satisfies it.
4: $S = S \cup children(v)$;
5: Goto 2
```

Figure 3.5: A Resolve-Graph Based decision heuristic

3.3 The Clause-Move-To-Front (CMTF) decision heuristic

The description above of Berkmin's decision heuristic, and the alternative view of the conflict clauses as being part of a resolve-graph, hints towards the process which is described in Figure 3.5. In this general scheme a clause is processed only if at least one of its abstractions (its resolvent clauses) has already been processed. It is easy to see that Berkmin is an instantiation of the scheme. In fact, Berkmin is more strict and processes a clause only if *all* its abstractions are satisfied.

CMTF is a method that instantiates this scheme in a different way. It causes the decision heuristic to be more focused on the current refinement path, i.e. to satisfy children of the currently satisfied clause s. It works as follows:

- All the conflict clauses are stored in a list.
- During the resolution in Alg 7, a bounded number of resolving conflict clauses which are processed at line 6 are moved to the front (front corresponds to the right end of Fig 3.4(a)). The newly created clause $NewClause$ is also added to the list (can be done at line 25).
- Clauses are processed from right to left in the list, while ignoring satisfied clauses. If all the conflict clauses are satisfied then the original VMTF strategy (from Siege [7]) is applied.

The idea of this strategy is to keep clauses that participate in resolution adjacent to their resolvents (at least until the next time they participate in a resolution, a case in which they can be moved to a new location).

CMTF shows a big improvement on many industrial problems comparing to the Berkmin heuristic. Both are specific instantiation of the scheme showed above. The advantages of CMTF is its simplicity and the fact that the explicit storage of the resolve-graph is not required. However, it seems that there is still room for future research on how to use the general scheme. For example, classic AI search methods like best-first-search can be used to decide on the exploration order of nodes in S at line 2. It may happen that partial or full storage of the resolve-graph will improve the performance.

Chapter 4

Resolution-based-scoring

4.1 Introduction and some definitions

In the previous chapter we showed how HaifaSat decides which clause to satisfy first. Given a clause c there can still be several ways to satisfy it. HaifaSat computes dynamically an *activity score* for each variable and then chooses the variable with the maximal score. Then another *sign score* is used to determine its Boolean value.

We define a scoring heuristic based solely on the resolution algorithm (Algorithm 7). The idea, intuitively, is to give higher weights to variables that were frequently resolved on recently, while distinguishing between resolutions that were necessary for the progress of the solver, and those that were made due to the imperfection of the decision heuristic. We will need several definitions and lemmas to explain this heuristic more precisely.

Suppose that every time the solver makes a decision or processes a conflict it writes into a log the event $a_i = (dl, e)$ where dl is the decision level where the event occurred and $e \in Conflicts \cup Decisions$ is either a conflict event or a decision event. The global index i is incremented every time the event happens. We call the sequence $\{a_i\}_1^N$ the *flat log* of the solver's run. We will denote by $DL(a_i)$ the decision level of the event. We consider only the case in which $dl > 0$. All conflict events other, potentially, than the last one in an unsatisfiable instance are included by this definition[1]. It must hold that for any conflict c there exists a decision d at the same level as c. In such a case, we say that d is refuted by c. More formally:

Definition 4.1.1 (Refuted decision by a conflict)**.** Let $a_j = (dl, c)$ be a conflict event. Let $a_k = (dl, d)$, $k < j$, be the last decision event with decision level dl preceding a_j (note that for $i \in [k+1, j-1] : DL(a_i) > dl$). We say that d is the *refuted decision* of the conflict c, and write $\mathbf{D}(\mathbf{a_j}) = a_k$.

Note that because of non-chronological backtracking the opposite direction does not hold: there are decisions that do not have conflicts on their levels that refute them.

For any conflict event a_j, the range $(D(a_j), a_j)$ defines a set of events that happened after $D(a_j)$ and led to the conflicts that were resolved into the conflict a_j which, in turn, refuted $D(a_j)$. These events necessarily occurred on levels deeper than $DL(D(a_j))$.

Definition 4.1.2 (Refutation Sequence and sub-tree events)**.** Let a_j be a conflict event with $a_i = D(a_j)$. Then the (possibly empty) sequence of events $a_{i+1}, \ldots, a_{j-1}$ is called the *Refutation Sequence* of a_j and denoted by $\mathbf{RS}(a_j)$. Any event $a_k \in RS(a_j)$ is called a *sub-tree event* of both a_j and a_i.

Example 5. *Consider the conflict event $a_j := (27, c_{110})$ in Fig. 4.1. For every event a_i that follows decision $D(a_j) = (27, d_{202})$ until (but not including) the conflict c_{110} it holds that $a_i \in RS(a_j)$. Note that the solver can backtrack from deeper levels to level 27 as a result of conflict events. However no event between a_i and a_j occurred on levels smaller or equal to 27.* $\square$

The number of resolutions for each variable is bounded from above by the number of sub-tree conflicts that were resolved into the current conflict. However, not all sub-tree conflict clauses resolve into the current refuting conflict. Some of them could

[1]A conflict that occurs at level 0 proves that the instance is unsatisfiable.

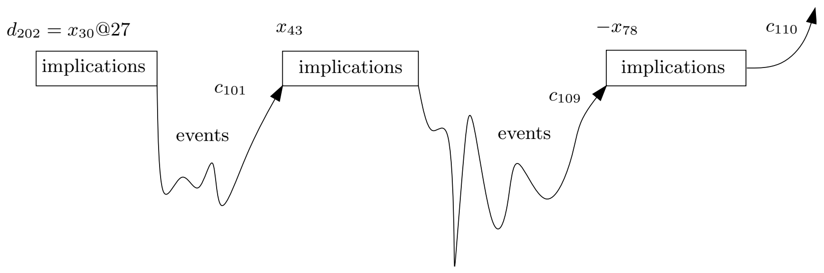

Figure 4.1: A possible scenario for the flow of the solver's run. After deciding x_{30} at decision level 27 the solver iteratively goes down to deeper decision levels and returns twice to level 27 with new asserted literals x_{43} and $\overline{x}_{78}$. The latter causes a conflict at level 27 and the solver backtracks to a higher decision level. Implications in the boxes denote assignments that are done during BCP after implying decision or asserting literal.

be caused by the imperfection of the decision heuristic and are therefore not used at this point of the search. Our goal is to build a scoring system that is based solely on those conflicts that contribute to the resolution of the current conflict clause. In other words, we compute for each variable an *activity score* which reflects the *number of times it was resolved-on in the process of generating the relevant portion of the refutation sequences of recent conflicts.* We *hypothesize that this criterion for activity leads to faster solution times.*

The information in the colored resolve-graph can enable us to compute such a score.

Definition 4.1.3 (Asserting set)**.** Let $G = (V, E)$ be a colored resolve-graph, and let $v \in V$ be a conflict clause. The *Asserting set* $B(v) \subset V$ of v is the subset of (conflict) clauses that v has a blue path to them in G.

The following theorem relates between a resolve-graph and sub-tree conflicts.

Theorem 4.1.1. *Let e_v be the conflict event that created the conflict clause v. Then the asserting set of v is contained in the refutation sequence of e_v, i.e. $B(v) \subseteq RS(e_v)$. In particular, since conflict events in $B(v)$ participate in the resolution of v by definition, they necessarily correspond to those sub-tree conflicts of e_v that participate in the resolution of v.*

Note that $B(v)$ does not necessarily include *all* the sub-tree conflicts that resolve into v, since the theorem guarantees containment in only one direction. Nevertheless, our heuristic is based on this theorem: it computes the size of the asserting set for each conflict.

In order to prove this theorem we will use the following lemmas.

Lemma 4.1.2. *Denote by stack*(a_j) *the stack of implied literals at the decision level* $DL(a_j)$*, where* a_j *is a decision event. Suppose that a literal* t *is asserted and entered into stack*(a_j)*, where* a_j *is a decision event. Further, suppose that* t *is asserted by the conflict clause* cl *(*cl *is thus asserting at this point) which was created at event* a_i*. Then it holds that* $j < i$*, i.e* cl *was created after the decision event* a_j *occurred.*

Proof. Right after the creation of cl, the DPLL algorithm backtracks to some level dl' with a decision event $a_k = (dl', d)$ and implies its asserted literal. It holds that $k < i$, because the solver backtracks to a decision level which already exists when cl is created. By the definition of an asserting clause, cl can be asserting exactly once, and since cl is asserting on level dl', it will never be asserting after the DPLL algorithm will backtrack from dl'. Therefore it must hold that $a_k = a_j$ $(k = j)$ and $dl' = dl$. □

□

Lemma 4.1.3 (Transitivity of RS)**.** *Suppose that* a_i, a_j *are conflict events s.t.* $a_i \in RS(a_j)$*. Then, for any event* $a_k \in RS(a_i)$ *it follows that* $a_k \in RS(a_j)$*.*

Proof. First, we will prove that $D(a_i) \in RS(a_j)$, or, in other words, that $D(a_i)$ occurred between $D(a_j)$ and a_j. Clearly, $D(a_i)$ occurred before a_i and, therefore, before a_j. Now, falsely assume that $D(a_i)$ occurred before $D(a_j)$. Then the order of events is $D(a_i), D(a_j), a_i$. However, this can not happen since $D(a_j)$ occurred on shallower (smaller) level than a_i and this contradicts the fact that all events between $D(a_i)$ and a_i occur on the deeper levels. Therefore, both $D(a_i)$ and a_i occurred between $D(a_j)$ and a_j. Now, since a_k happened between $D(a_i)$ and a_i it also happened between $D(a_j)$ and a_j and from this it holds that $a_k \in RS(a_j)$. □

□

Using this lemma we can now prove Theorem 4.1.1.

Proof. We need to show that any blue descendant of v is in $RS(e_v)$. By Lemma 4.1.3 it is enough to show it for the immediate blue descendants, since by transitivity of RS it then follows that for any blue descendant. Now, suppose that there exists a blue edge (v, u) in the resolve-graph. By the definition a blue edge, clause u was asserting during the resolution of v. On the one hand, u was resolved during the creation of v and, therefore, was created before v. On the other hand, by Lemma 4.1.2 it was created after $D(e_v)$. Therefore, $e_u \in RS(e_v)$. □

□

Definition 4.1.4 (Sub-tree weight of the conflict)**.** Given a resolve-graph $G(V, E)$ we define for each clause v a state variable $W(v)$:

$$W(v) = \begin{cases} \sum\limits_{(v,u)\in E} W(u) + 1 & v \text{ is asserting} \\ 0 & \text{otherwise} \end{cases}$$

The function $W(v)$ is well-defined, since the resolve-graph is acyclic. Moreover, since the blue sub-graph rooted at v forms a tree (remember that any node has at most one incoming blue edge), $W(v)$ equals to $|B(v)| + 1$. Our recursive definition of $W(v)$ gives us a simple and convenient way to compute it as part of the resolution algorithm. Algorithm 8 is the same as Algorithm 7, with the addition of several lines: in line 5 we add $W \leftarrow 1$, at line 24 we add $W{+}{=}W(\mathit{ResolveCl})$ and, finally, we set $W(\mathit{NewClause}) \leftarrow W$ at line 29. We need to guarantee that $W(C)$ is non-zero only when C is an asserting clause. Therefore, for any antecedent clause C, when its implied variable is unassigned we set $W(C) \leftarrow 0$.

4.2 Computing the scores of a variable

Given the earlier definitions, it is now left to show how activity score and sign score are actually computed, given that we do not have the resolve-graph in memory. For each variable v we keep two fields: *activity*(v) and *sign_score*(v). At the beginning of the run *activity* is initialized to $\max\{lit\_num(v), lit\_num(\overline{v})\}$ and *sign_score* to $lit\_num(v) - lit\_num(\overline{v})$. Alg. 8 shows the extended version of the resolution algorithm which computes the weights of the clauses and updates the scores. Recall that any clause weight is reset to zero when its implied variable is unassigned, so that any clause weight is contributed at most once. In order to give a priority to recent resolutions we occasionally divide both activities and sign scores by 2.

Our decision heuristic chooses a variable from the given clause with a biggest activity and then chooses its value according to the sign score: TRUE for the positive values and FALSE for the negative values of the sign score.

4.3 Experiments

Table 4.2 shows experiments on an Intel 2.5Ghz computer with 1GB memory running Linux, sorted according to the winning strategy, which is CMTF combined with the RBS scoring technique. The benchmark set is comprised of 165 industrial instances used in various SAT competitions. In particular, *fifo8, bmc2, Checker-Interchange, comb, f2clk, ip, fvp2, IBM02 and w08* are hard industrial benchmarks from SAT02; *hanoi* and *hanoi03* participated in SAT02 and SAT03; *pipe03* is from SAT03 and *01_rule, 11_rule_2, 22_rule, pipe-sat-1-1, sat02, vis-bmc, vliw_unsat_2.0* are from SAT04 [10, 3, 4]. The timeout for each instance was set to 3000 seconds. If an instance could not be solved in this time limit, 3000 sec. were added as its solving time. All configurations are implemented on top of HaifaSat, which guarantees that the figures faithfully represent the quality of the various heuristics, as far as these benchmarks are representative. The results show that using CMTF instead of Berkmin's heuristic for choosing a clause leads to an average reduction of 10% in run time and 12-25% in the number of fails (depending on the score heuristic). It also shows a 23% reduction in run time when using RBS rather than VSIDS as a score system, and a corresponding 20-30% reduction in the number of fails.

Algorithm 8 First-UIP learning scheme, including scoring

```
    procedure AnalyzeConflict(Clause: conflict)
2:      $currentClause \leftarrow conflict$;
        $ResolveNum \leftarrow 0$;
4:      $NewClause \leftarrow \emptyset$;
        $\mathbf{wght} \leftarrow \mathbf{1}$;
6:      repeat
            for each literal $lit \in currentClause$ do
8:              $v \leftarrow var(lit)$;
                if $v$ is not marked then
10:                 Mark $v$;
                    if $dlevel(v) = CurrentLevel$ then
12:                     $++\ ResolveNum$;
                    else
14:                     $NewClause \leftarrow NewClause \cup \{lit\}$;
                    end if
16:             end if
            end for
18:         $u \leftarrow$ last marked literal on the assignment stack;
            Unmark $var(u)$;
20:         $\mathbf{activity}(\mathbf{var}(\mathbf{u})) \mathrel{+}= \mathbf{wght}$;
            $\mathbf{sign\_score}(\mathbf{var}(\mathbf{u})) \mathrel{-}= \mathbf{wght} \cdot \mathbf{sign}(\mathbf{u})$;
22:         $--\ ResolveNum$;
            $ResolveCl \leftarrow Antecedent(u)$;
24:         $\mathbf{wght} \mathrel{+}= \mathbf{W}(\mathbf{ResolveCl})$;
            $currentClause \leftarrow ResolveCl \setminus \{u\}$;
26:     until $ResolveNum = 0$;
        Unmark literals in $NewClause$;
28:     $NewClause \leftarrow NewClause \cup \{\overline{u}\}$;
        $\mathbf{W}(\mathbf{NewClause}) \leftarrow \mathbf{wght}$ ;
30:     Add $NewClause$ to the clause database;
    end procedure
```

		BERKMIN+RBS		BERKMIN+VSIDS		CMTF+RBS		CMTF+VSIDS	
Benchmark	instances	time	fails	time	fails	time	fails	time	fails
hanoi	5	389.18	0	530.62	0	130.72	0	74.55	0
ip	4	191.02	0	395.52	0	203.24	0	324.27	0
hanoi03	4	1548.25	0	1342.1	0	426.87	0	386.28	0
CheckerI-C	4	1368.25	0	3323.16	0	681.56	0	3457.78	0
bmc2	6	1731.96	0	1030.9	0	1261.97	0	1006.94	0
pipe03	3	845.97	0	6459.62	2	1339.29	0	6160.12	1
fifo8	4	1877.57	0	3944.31	0	1832.65	0	3382.61	0
fvp2	22	1385.64	0	8638.63	1	1995.17	0	11233.7	3
w08	3	2548.62	0	5347.62	1	2680.96	0	4453.28	0
pipe-sat-1-1	10	1743.23	0	3881.49	0	3310.41	0	6053.84	0
IBM02	9	7083.55	1	9710.52	1	3875.64	0	7163.95	0
f2clk	3	4389.04	1	5135.25	1	4058.62	1	4538.15	1
comb	3	3915.15	1	3681.45	1	4131.05	1	4034.53	1
vis-bmc	8	15284.45	3	7905.9	1	13767.52	3	10119.34	2
sat02	9	17518.09	4	22785.77	5	17329.64	4	21262.25	4
01_rule	20	22742.11	4	33642.33	9	19171.5	2	23689.37	5
vliw_unsat_2.0	8	16600.67	4	24003.62	8	19425.41	5	22756.03	7
11_rule_2	20	31699.69	8	34006.97	10	22974.7	6	28358.05	6
22_rule	20	28844.07	8	33201.87	10	27596.78	8	30669.91	8
Total:	165	161706.5	34	208967.7	50	146193.7	30	189125	38

Figure 4.2: A comparison of various configurations, showing separately the advantage of CMTF, the heuristic for choosing the next clause from which the decided variables will be chosen, and RBS, the heuristic for choosing the variable from this clause and its sign.

4.4 Summary of chapters 3 and 4

We presented an abstraction/refinement model for analyzing and developing SAT decision heuristics. Satisfying a conflict clause before satisfying the clauses from which it was resolved, can be seen according to our model as satisfying an abstract model before satisfying a more concrete version of it. Our Clause-Move-To-Front decision heuristic, according to this model, attempts to satisfy clauses in an order associated with the resolve-graph. CMTF does not require to maintain the resolve-graph in memory, however: it only exploits the connection between each conflict clause and its immediate neighbors on this graph. Perhaps future heuristics based on this graph will find a way to improve the balance between the memory consumption imposed by saving this graph and the quality of the decision order. We also presented a heuristic for choosing the next variable and sign from the clause chosen by CMTF. Our Resolution-Based-Scoring heuristic scores variables according to their involvement ('activity') in refuting recent decisions. Our experiments show that CMTF and RBS either separately or combined are better than Berkmin and the VSIDS decision heuristics.

Bibliography

[1] http://www.cs.toronto.edu/∼fbacchus.

[2] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT 2003*, volume 2919 of *Lect. Notes in Comp. Sci.*, pages 341–355, 2003.

[3] D. Le Berre and L. Simon. The essentials of the sat'03 competition. . In editor A. Tacchella E. Giunchiglia, editor, *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, volume 2919 of *LNAI*, pages 452–467. Springer-Verlag, 2003.

[4] Daniel Le Berre and Laurent Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In *SAT (Selected Papers*, pages 321–344, 2004.

[5] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of the International Joint Conference on Artifical Intelligence*, 2001.

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

[7] L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.

[9] J.P.M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.

[10] L. Simon, D. Le Berre, and E. Hirsch. The sat2002 competition. *Accepted for publication in Annals of Mathematics and Artificial Intelligence (AMAI)*, 43:343–378, 2005.

[11] Allen Van Gelder and Yumi K. Tsuji. Satisfiability Testing with More Reasoning and Less Guessing.

[12] R. Williams, C. Gomes, and Selman. Backdoors to typical case complexity. *IJCAI03*, 2003.

קל לפתור או להפריך נוסחה אבסטרקטית, ולכן יש לכוון את הפותר לפתור פסוקית קונפליקט לפני שפותרים את הבנים שלה, בניגוד להיוריסטיקת Berkmin. בהתאם, אנו מגדירים היוריסטיקה חדשה בשם Clause Move To Front – CMTF ששייכת לאותה סכמה ופועלת טוב יותר, באופן אמפירי, מ Berkmin.

תוצאות המחקר מומשו בפותר SAT חדש בשם HaifaSat, אשר זכה במקום השלישי בתחרות הבין לאומית לפותרי SAT לשנת 2005.

פסוקיות אונריות אם ישנן על ידי פישוט של הנוסחה ומוצא ליטרלים שקולים בנוסחה על ידי מציאת מעגלים בגרף הבינארי.

השיפור שלנו התרכז בשני כיוונים:

א. ההפעלה של ה-BCP מוגבלת לשורשים של הגרף הבינארי בלבד. כך אנחנו מפעילים פעולה שיחסית פחות יקרה מבחינת זמן (כתלות של מספר שורשים בגרף), אבל בהינתן זה אנחנו יודעים ללמוד גרירות (ואחר כך קשתות) יותר חזקות מאשר באלגוריתם המקורי.

ב. האלגוריתם ממומש כ any-time, כלומר כאלגוריתם שיש לו תרומה גם אם הוא מופסק לפני סוף הריצה. האלגוריתם משולב עם היוריסטיקה אשר מנחשת מתי יעילותו יורדת ביחס לזמן שהושקע. נזכיר שמטרתנו העיקרית היא לפתור את הנוסחה ולא לפשט אותה, ולכן יש חשיבות לעצור את תהליך העיבוד המקדים כשדבר זה פוגע במהירות הכוללת של האלגוריתם.

2. <u>היוריסטיקת החלטה:</u> האלגוריתם השני הוא היוריסטיקה של שיטת החלטה בתוך האלגוריתם של פתירת DPLL – SAT. ההיוריסטיקה המוצעת מבוססת על מודל תיאורטי מסוים אשר אנו מציעים, אשר בעזרתו ניתן להבין טוב יותר את האופן בו DPLL מצליח לפתור בעיות SAT אמיתיות.

להלן מספר מושגים הדרושים על מנת להבין את האלגוריתם. לנוסחא בלתי ספיקה ניתן למצוא סדרת *רזולוציות* המובילות לפסוקית ריקה ("שקר" לוגי). רזולוציה היא מערכת הסקה עם כלל היסק יחיד, הנקרא כלל הרזולוציה:

$$\frac{(A \vee x) \wedge (B \vee \neg x)}{(A \vee B)}$$

בהינתן נוסחת CNF בלתי ספיקה לפותר SAT העובד בשיטת DPLL, ניתן לבנות *גרף רזולוציות* המתאים לנוסחא. בגרף זה צמתי המקור מייצגים פסוקיות מהנוסחא, וצמתים פנימיים מייצגים *פסוקיות קונפליקט* אשר נלמדו בזמן ריצת הפותר. פסוקיות אלה נובעות מפסוקיות מקוריות לפי כלל הרזולוציה. הקשתות היוצאות מן הצומת מייצגות את העובדה שהפסוקית של הצומת נובעת מהפסוקיות המתאימות לצמתים אליהם הקשתות נכנסות. במקרה של נוסחה לא ספיקה, לגרף יש שורש יחיד המייצג את הפסוקית הריקה. אחרת, יכולים להיות מספר שורשים המייצגים פסוקיות קונפליקט כלשהן.

בכל רגע נתון, ניתן לראות את גרף הרזולוציות כיחס סדר חלקי בין פסוקיות בנוסחה. כאשר פסוקיות אחרונות בסדר הן פסוקיות מקור שמגדירות את הנוסחה במדויק ופסוקיות קודמות בסדר הן החזית של הגרף ומייצגות אבסטרקציה של הנוסחה המקורית.

התיזה מראה כיצד ניתן לראות פסוקיות קונפליקט כאבסטרקציה של הפסוקיות מהן הן נגזרו (במלים אחרות, כל צומת פנימי בגרף הוא אבסטרקציה של הבנים שלו). תוך קישור למודל הידוע של הפשטה/עידון המשמש לתיאור אלגוריתמים רבים באימות פורמלי, אנו מתארים סכמה כללית להבנת תהליך התקדמות הפותר ויצירת פסוקיות הקונפליקטים. אנו מראים יישום של המודל על היוריסטיקת Berkmin, אחד היוריסיקות המובילות בעת כתיבת התיזה. התיזה מציגה השערה שלפותר SAT יותר

תקציר

בעיית SAT היא בעיית הכרעה של נוסחה בוליאנית: האם נוסחה בוליאנית נתונה ספיקה (כלומר, קיימת השמה מלאה למשתנים המספקת את הנוסחה). פותרי SAT על פי רב מקבלים כקלט נוסחאות בוליאניות ב Conjunctive Normal Form – CNF: חיתוך לוגי ("וגם") של פסוקיות, אשר כל אחת מהן היא איחוד לוגי ("או") של משתנים בוליאניים או שלילתם. לדוגמה:

$$(x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

כאשר x$x_1 \ldots x_4$ הם משתנים בוליאניים. השמה מספקת לנוסחה זאת, היא למשל

$$x_1 = 0, \quad x_2 = 1, \quad x_3 = 1, \quad x_4 = 0$$

.

לאור העובדה שמספר רב של בעיות תכנון, אימות, שיבוץ וכו' ניתנות למידול על ידי נוסחאות פסוקיות (למעשה, כל בעיה ב NP), ולאור העובדה שנוסחאות המגיעות מהתעשייה הן גדולות מאוד (מאות אלפי משתנים ויותר), קיים מחקר אינטנסיבי לשיפור יעילותם של פותרי SAT. בעשור האחרון חל שיפור של יותר מסדר גודל בגודל הנוסחאות האופייניות מהתעשייה הניתנות לפתרון בזמן סביר.

בתיזה זאת אנחנו מציגים שני אלגוריתמים לשיפור אלגוריתמים לפתירת בעיית SAT, אשר יסוכמו להלן בקצרה:

1. <u>עיבוד מוקדם (pre-processing)</u>: האלגוריתם המוצע, Hyperbinfast, הוא שיפור של האלגוריתם שהוצע על ידי Bachus & Winter, אשר מפשט את נוסחת ה - CNF לפני שפותרים אותה.
סקירת האלגוריתם המקורי דורשת את ההגדרה הבאה:
גרף בינארי הוא גרף מכוון המושרה על ידי הפסוקיות הבינריות בנוסחת ה – CNF באופן הבא: כל משתנה המופיע באחת מהפסוקיות הבינאריות, וכן שלילתו, הם הצמתים בגרף. הקשתות המכוונות מוגדרות כלהלן. לכל פסוקית מהצורה $(\alpha \vee \beta)$ נוסיף שתי קשתות:

$$\overline{\alpha} \rightarrow \beta \quad \overline{\beta} \rightarrow \alpha$$

(גרירות מסמנות קשתות מכוונות בגרף).

מעגל בגרף זה מראה שכל הליטרלים במעגל הם שקולים ולכן ניתן לפשט את הנוסחה בהתאם.
האלגוריתם המקורי משים ערך T למשתנה x, עבור כל משתנה x בנפרד, ומפעיל Boolean Constraint Propagation (BCP) כדי לבדוק אילו ערכים השמה זאת גוררת. לכל משתנה y שקיבל ערך T בתהליך זה, ניתן להוסיף לנוסחה המקורית פסוקית בינארית ולגרף קשת חדשה מ x ל y. (באופן דומה התהליך מבוצע עבור השמות של הערך F). אחרי הוספת הקשתות, האלגוריתם מבטל את ההשמה, מבטל

המחקר נעשה בהנחיית
דר' עופר שטריכמן
בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תודתי נתונה למשפחתי על אהבתם, סבלנותם ותמיכתם.

תודה לדר' עופר שטריכמן על ההדרכה והעזרה.

שיפורים לשיטות פתירה
של בעיות SAT

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

רומן גרשמן

הוגש לסנט הטכניון - מכון טכנולוגי לישראל
שבט תשס"ז חיפה פברואר 2007