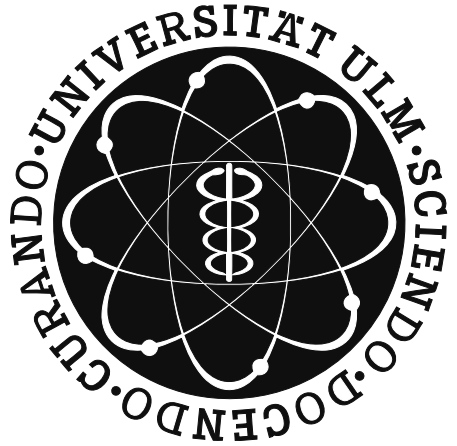
UNIVERSITY OF ULM
Faculty of Engineering and Computer Science
Institute of Theoretical Computer Science



DIPLOMA THESIS

TOWARDS THE DEVELOPMENT OF A HYBRID SAT SOLVER

Oliver Gableske

Towards the Development of a Hybrid SAT Solver

Oliver Gableske

1st Reviewer:
Prof. Dr. Uwe Schöning,
Institute of Theoretical Computer Science, University of Ulm, Germany

2nd Reviewer:
Prof. Dr. Jacobo Torán,
Institute of Theoretical Computer Science, University of Ulm, Germany

Consultant:
Dipl. Inf. Adrian Balint,
Institute of Theoretical Computer Science, University of Ulm, Germany

To my parents;

the brightest lights in the darkest nights.

Declaration of Authenticity

I hereby declare, that this diploma thesis is the result of my own work and that all sources have been duly acknowledged.

Ulm, February 13, 2009

Oliver Gableske

Man errs as long as he doth strive.
from "Faust: The Tragedy Part One"
– Johann Wolfgang von Goethe

Foreword

No scientist can begin a quest for truth alone and is expected to find it. The freedom to ask questions to others is one of the finest a scientist has these days, and he is blessed when others feel free to listen, to object, and to speak as they see fit. For keeping my "errs" as small as possible and patiently accepting the "strives" of my quest, I hereby want to thank:

- Dipl. Inf. Adrian Balint, who was my consultant during the time of this work and helped me with words and deeds,
- Prof. Dr. Uwe Schöning, who was the first reviewer of this work and gave me the opportunity to conduct research on this topic,
- Prof. Dr. Jacobo Torán, who was the second reviewer of this work and took some time to study a preliminary version of this thesis,
- Stefan Dietzel, who willingly spared various ideas with me without expecting anything in return,
- Moritz Gerlach, who listened well and asked the right questions,
- Markus Maucher, who helped me with some issues in probability theory,
- Simon Gog, who helped me with some programming issues,
- Marijn Heule, who helped me to understand March_ks,
- Knot Pipatsrisawat, who helped me to understand RSat.

Additionally, not only the content itself is important, but also the way in that it is presented. For proofreading and design recommendations, I want to thank:

- Michael Henn
- Stephanie and Gunther Gableske

I am deeply grateful for the support of my parents, who had to waive from my presence for a long time. I hope that the value of my scientific work makes up for their sacrifices.

Contents

List of Figures

Whatever you can do, or dream you can, begin it.
Boldness has genius, power, and magic in it.
– Johann Wolfgang von Goethe

PART I

Introduction and Preliminaries

The contribution of this part is twofold. In chapter 1 we will give an introduction to the work at hand. Therefore, we will outline its context, present its structure, and give a description of the main goals it pursues.

In chapter 2 we will give the necessary background knowledge for this work. We will introduce the basic terminology that is needed for an understanding of SAT.

In order to gain that understanding, we will explain propositional logic, the conjunctive normal form (CNF) as well as simplification and inference techniques for propositional formulas in CNF.

We then define SAT and k-SAT, followed by an explanation of the DIMACS format that is used to represent problem instances in CNF.

We will also put SAT solvers in the context of general problem solving to clarify their value for engineering and computer science.

Finally, we introduce the term of hardness for propositional formulas in CNF. At the end of this part, we will give a summary to recapitulate the most important details.

Chapter 1

Introduction

The main topic of this diploma thesis is the propositional satisfiability problem (SAT), which was shown to be $\mathcal{NP}$-complete in 1971 [Coo71]. While SAT is often used as a canonical example for $\mathcal{NP}$-complete problems, its practical relevance for computer science and engineering is fundamental. This is because any problem from these domains can be translated into SAT, for which a general purpose SAT solver algorithm can be used to find a solution for the corresponding SAT problem. The solution for the SAT problem can then be re-translated into the original problem domain in which it then acts as a solution for the initial problem. Examples for problems, that are already solvable using the means of SAT, are planning (Artificial Intelligence), VLSI verification (Engineering) and detection of graph isomorphisms (Math), just to name a few.

SAT has been researched for more than three decades now, yet no truly efficient SAT solver algorithm has been found. It is yet unknown, whether such an efficient algorithm can exist at all. It is believed, that the reason for the difficulties in finding such an algorithm lies within the $\mathcal{NP}$-completeness of SAT. Algorithms working on $\mathcal{NP}$-complete problems have an exponential runtime. Since little hope remains to find a truly efficient SAT algorithm without an exponential runtime, most of the research effort is put into the improvement of exponential runtime algorithms. The main goal of these improvements is to reduce the runtime in order to raise the limits for their practical application.

Two different types of SAT algorithms exist: those that can be theoretically analyzed and those, that elude themselves from such an analysis because they make use of heuristics. These heuristics can be seen as both a blessing and a curse. While they usually enable SAT solvers to solve problems in less time, they also make its behavior less predictable. Heuristics are, after all, not always true. Therefore, any decision based on a heuristic is questionable, what in turn makes the proper theoretical analyses of such an algorithm extremely difficult.

Since raising the limits of practically solvable formulas is considered one of the most important goals with respect to the feasibility of SAT solving, a considerable amount

of effort is put into the research on SAT algorithms using such heuristics. Two SAT solver paradigms exist within the domain of heuristic SAT algorithms: DPLL and SLS. Both algorithm families follow different approaches on how to solve a given SAT problem, and thus, they have different advantages and disadvantages.

DPLL algorithms perform a structured search by using binary search trees, which makes these algorithms complete (meaning that they will find a model for a given SAT problem if one exists or prove, that no such model exists). Furthermore, DPLL algorithms do not scale very well, since the creation and maintenance of a binary search tree uses up large amounts of memory and computation time.

SLS algorithms on the other hand follow the idea of local search, which makes these algorithms incomplete (meaning that they will not always find a model even though one might exist). Since SLS algorithms do not perform a structured search, they have no need for additional computation time and memory to maintain such a structure. Therefore, SLS algorithms scale much better than DPLL algorithms.

Today, the application of DPLL or SLS SAT solvers depends on the given task. If the unsatisfiability of a given problem is to be shown, a DPLL algorithm must be used. If a large scale problem is to be solved, that is way out of reach for DPLL algorithms, an SLS solver must be used. Yet no algorithm has emerged, that performs equally well in both tasks. In total, DPLL and SLS algorithms seem to complement each other quite well, which led to the idea of hybridizing both approaches in order to create a new algorithm that can fulfill both tasks.

Hybrid SAT solvers are supposed to inherit the advantages from both SAT Solver paradigms, while circumventing their disadvantages at the same time. Therefore, effort has been put into the development of such hybrid SAT solver algorithms. So far, the best known hybrid SAT algorithm yields only a speed-up of only a few percent (with respect to its DPLL component), which in turn means, that more effort is needed to further evolve the idea of hybrid SAT solvers. The work at hand is supposed to contribute to these efforts.

Before a new idea for hybridization can be identified, one is to understand the state-of-the-art in SAT solving these days, which means that one must understand the functioning of DPLL and SLS solvers. In order to gain such an understanding, one needs to understand SAT itself. Therefore, we will first provide the basic terminology and techniques that are common in conjunction with SAT these days in Part I of this work. After the technical basis to understand SAT have been provided, we will give a survey on the most successful SAT solvers these days in Part II. Therefore, we will present the functioning of two solvers for the DPLL and SLS paradigms respectively. Equipped with an understanding for modern SAT solvers, we will continue to develop a new idea on how these paradigms can be combined in Part III. This includes preliminary empirical experiments to provide a proof of concept for this new approach. This part also discusses the results provided in the empirical studies. Finally, the work is concluded in Part IV.

Chapter 2

Propositional Satisfiability (SAT)

In order to understand SAT, one needs to understand the basic terminology that is used for the definition of SAT. Today, the most common description for SAT is based on the propositional logic, and therefore, we will introduce it here.

After the propositional logic has been presented, we will describe the most common tools that are applied for SAT solving these days. These are mainly the conjunctive normal form, as well as simplification and inference techniques.

With these tools given, a definition for SAT is provided. Based on this definition, we give a more detailed description of problems and the term of problem hardness.

Finally, we will shortly outline the practical relevance of SAT in order to motivate the research on this topic.

2.1 Propositional Logic

We will now recapitulate propositional logic, which builds the basis for terms we need for the definition of SAT.

As any logic, the propositional logic consists of a formal syntax and a formal semantic.

The formal syntax of propositional logic consists of *variables* and *literals*, *truth values*, *connectors*, and *formulas*.

The set of propositional variables is denoted $\mathcal{V} = \{x_1, \ldots, x_n\}$. From variables, we construct literals. The two possible literals for x would be x itself and its negation $\neg x$. In other words, a literal is a signed variable.

The used truth values are TRUE (also 1) and FALSE (also 0).

Connectors are operators that work on variables and are used to build formulas. There are two forms of connectors: unary and binary connectors. An unary connector works on exactly one variable. In our case this would be the NEGATION

(symbolized by $\neg$). The binary connectors we are interested in the context of this work are the logical AND (symbolized by $\wedge$ or sometimes $\cdot$), the logical OR (symbolized by $\vee$ or sometimes $+$), the implication (symbolized by $\rightarrow$ or $\leftarrow$) and the equivalence (symbolized by $\leftrightarrow$). The connection of variables by AND is called a conjunction. The connection of variables by OR is called a disjunction.

The construction of a formula in propositional logic is conducted via rules [Hoo98]. These rules are:

- Any sole variable from $\mathcal{V}$, as well as TRUE and FALSE are formulas.
- If F_1 and F_2 are propositional formulas, then $\neg F_1$, $F_1 \vee F_2$, $F_1 \wedge F_2$, $F_1 \rightarrow F_2$ and $F_1 \leftrightarrow F_2$ are propositional formulas as well.

Any finite formula that is conform to the rules above is valid. We also allow the use of parenthesis to be able to bring structure to a formula and give priority to certain connectors.

The formal semantic of propositional logic is based on assignments.

Definition 1 *Given a propositional formula F that contains the variables in set $\mathcal{V} = \{x_1, \ldots, x_n\}$, an* `assignment` *$A$ is a mapping $A : \mathcal{V} \longrightarrow \{0, 1\}$ that assigns a truth value to each variable. We can denote this as a tuple with n components, whereas the i-th component of the tuple represents the value $v_i \in \{0, 1\}$ of the variable x_i, denoted $Val_A(x_i) = v_i$. If the assignment in question is clear from the context, we will omit the index naming this assignment and simply write $Val(x_i) = v_i$.*

An assignment B is called `partial`*, iff $B : \mathcal{V} \longrightarrow \{0, 1, ?\}$, whereas the '?' is a placeholder symbol indicating that the corresponding variable does not receive a value under B.*

Example 1 *Consider the formula $F = x_1 \wedge (x_2 \vee \neg x_3)$. Under the assignment $A = (0, 1, 1)$, we get $A(F) = 0$, which means that the formula evaluates to* FALSE*. Using $B = (1, 1, 1)$, we get $B(F) = 1$ and thus the formula evaluates to* TRUE*. Using $C = (?, 1, 1)$, we get $C(F) = x_1$, a reduced formula.*

Definition 2 *Given two formulas F and G. We call these formulas* `logically equivalent` *(denoted $F \equiv G$), iff F and G evaluate to the same truth value for under all assignments.*

There are two kinds of assignments: those that make F TRUE and those that make F FALSE.

The truth value of a formula is evaluated through inductive rules and the semantics of the used operators [Hoo98]. Let A be an assignment. Let F, G, H be propositional formulas.

- $F \equiv 0 \Rightarrow A(F) = 0$, $F \equiv 1 \Rightarrow A(F) = 1$.
- $F \equiv x_i \Rightarrow A(F) = Val(x_i) = v_i$.
- $F \equiv \neg G \Rightarrow A(F) = \neg A(G)$.
- $F \equiv G \wedge H \Rightarrow A(F) = A(G) \wedge A(H)$.
- $F \equiv G \vee H \Rightarrow A(F) = A(G) \vee A(H)$.
- $F \equiv G \rightarrow H \Rightarrow A(F) = \neg A(G) \vee A(H)$.
- $F \equiv G \leftrightarrow H \Rightarrow A(F) = (\neg A(G) \vee A(H)) \wedge (\neg A(H) \vee A(G))$.

Definition 3 *Given a propositional formula F and an assignment A, we say A is a* `model` *for F iff $A(F) = 1$. We then say, that A* `satisfies` *F, or A is a* `solution` *for F.*

Furthermore, there exist two kinds of formulas: those that have at least one model and those that do not have a model.

Definition 4 *We call a propositional formula F* `satisfiable` *iff $\exists A : A(F) = 1$. Otherwise we call it* `unsatisfiable`*.*

Sometimes we use abbreviations and call satisfiable formulas SAT and unsatisfiable formulas UNSAT.

Definition 5 *We call a propositional formula F a* `tautology`*, iff $A(F) = 1$ $\forall A$.*

A problem can usually be described in several ways, hence two different descriptions can actually describe the same problem. It is important to understand the difference between the terms *problem* and *formula*. A formula is a description of a problem.

Hence two different formulas can describe the same problem though they might have different structure. To further stress the difference between a problem and a formula, we sometimes call formulas *problem instances*.

Problem instances that are common these days are often aligned in a certain structure called CNF. This structure will be explained in the following section.

2.2 Conjunctive Normal Form (CNF)

As stated in section 2.1, any finite formula conforming to the formal syntax of propositional logic is valid. However, there are several ways in structuring a propositional formula into a certain pattern. One of these patterns is called the *conjunctive normal form* (or CNF for short).

Definition 6 *A propositional formula is in* `Conjunctive Normal Form (CNF)`*, iff it is a conjunction of disjunctions.*

The formula from Example 1 is in CNF. Another example would be:

Example 2 $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4 \vee x_3) \wedge (x_5 \vee x_1 \vee \neg x_2)$.

A disjunction in a formula in CNF is also called a *clause*. In Example 2, we therefore have three clauses: $c_1 = \{x_1, x_2, \neg x_3\}$, $c_2 = \{\neg x_2, x_4, x_3\}$ and $c_3 = \{x_5, x_1, \neg x_2\}$. Clauses are often written as sets, since the ordering of literals within a disjunction is of no interest (due to the commutativity of $\vee$).

Clauses containing exactly one literal are called *unit clauses*. Clauses that contain exactly two literals are called *binary clauses*. Clauses with exactly three literals are called *ternary clauses*. In general, we call a clause *m-ary*, when it contains $m > 3$ literals.

The set of clauses of a propositional formula F is denoted $\mathcal{C}$. That means that for Example 2 we have $\mathcal{C} = \{c_1, c_2, c_3\}$. A propositional formula in CNF is completely described by $\mathcal{C}$, since the ordering of the clauses is irrelevant (due to the commutativity of $\wedge$). According to the context, we will sometimes work with $\mathcal{C}$, and sometimes we will work with F.

It is interesting to note, that a propositional formula in CNF evaluates to TRUE under a given assignment A, iff all clauses evaluate to TRUE under this assignment.

There exist many more ways in structuring propositional formulas, like the DNF (Disjunctive Normal Form), which is a disjunction of conjunctions, or Horn Clauses, which structure the formula into a set of disjunctions with at most one positive literal. However, in the remainder of this work, we will only use formulas in CNF.

Definition 7 *Let $c_1, \ldots, c_k$ be a set of clauses. We call this set of clauses a* `contradiction`*, in case there is not a single assignment that satisfies all these clauses at once.*

An example for a contradiction could look like this:

Example 3 $c_1 = \{a, b\}, c_2 = \{\neg a, b\}, c_3 = \{\neg b\}$.

Propositional formulas often hold the opportunity for simplification. Some of the simplification techniques, along with the inference technique of Resolution, will be described next.

2.3 Simplification and Inference Techniques for Formulas in CNF

The mere description of a formula in CNF can be used for two applications: Simplification and Inference.

Simplification of propositional formulas in CNF is considered important [Pha06, SP05, EB05, Bra04], since it can reduce the size of such a formula. We therefore give a set of simplification techniques, that can be used to simplify formulas in CNF in section 2.3.1.

Inference is a tool to deduce new clauses from the clauses of a given formula. Inference can be seen as the complement to simplification, since it sometimes adds new clauses to the formula. However, inference can be used to further simplify formulas in conjunction with the simplification techniques. We therefore give an overview of Resolution, one of the most widely known inference rules, in section 2.3.2.

2.3.1 Simplification Techniques

Simplification techniques are techniques to reduce the size of a formula without altering the set of models it has. Given a propositional Formula F in CNF, the goal of simplification is to find a formula F‘ in CNF that is logically equivalent to F, but contains a reduced number of literals or clauses (or both). Some of these techniques are:

Purity Principle: A variable x_i is called *pure*, iff it occurs only as literal x_i or $\neg x_i$ within the elements of $\mathcal{C}$, but not in both polarities. We now set variable x_i to a truth value, such that all clauses in which x_i (respectively $\neg x_i$) occurs, become TRUE. More precisely, if x_i is a pure literal set $Val(x_i) = 1$, if $\neg x_i$ is a pure literal set $Val(x_i) = 0$. We can now save this in an assignment A. By applying this assignment on $\mathcal{C}$, we remove from $\mathcal{C}$ all clauses that contained x_i since all of these have now been satisfied. The reduced set of clauses is referred to as a simplified formula F‘. Further search is then conducted on the simplified formula. A correctness proof for this procedure is given in [Rob65b].

Example 4 *Let* $\mathcal{C} = \{\{x_1, x_2\}, \{\neg x_1, \neg x_3\}, \{x_3, x_2\}\}$. *Variable* x_2 *is pure. We set* $Val(x_2) = 1$, *since only literal* x_2 *occurs. The resulting set of clauses is then* $\mathcal{C}' = \{\{x_1, 1\}, \{\neg x_1, \neg x_3\}, \{x_3, 1\}\} = \{\{\neg x_1, \neg x_3\}\}$.

Unit Reduction Principle: A clause that contains only one literal is called a *unit clause*. Since this unit clause must be satisfied in order to satisfy the formula, we must set the corresponding variable to a value that satisfies this clause. We then can remove this clause from $\mathcal{C}$ and adapt the rest of the clauses that contain the just set literal accordingly.

Example 5 *Let* $\mathcal{C} = \{\{x_1, x_2\}, \{\neg x_1, \neg x_2\}, \{\neg x_2\}\}$. *The last clause is unit. We set* $Val(x_2) = 0$, *since this satisfies the last clause. The resulting set of clauses is then* $\mathcal{C}' = \{\{x_1, 0\}, \{\neg x_1, \neg 0\}, \{\neg 0\}\} = \{\{x_1\}\}$.

The process of iteratively applying the unit reduction principle until no further changes are possible is called *unit propagation*.

Unit Propagation on Literals: Consider a variable x_i that is neither pure nor part of a unit clause. Unit Propagation on literals (denoted UP) is the attempt of setting a variable to a value and examining the effect. Thereby $UP(x_i)$ means, we set $Val(x_i) = 1$, while $UP(\neg x_i)$ means we set $Val(x_i) = 0$. As a result, we can remove x_i (and eventually some now satisfied clauses from $\mathcal{C}$). The catch is, that if we decided wrong to what we set x_i, we can not satisfy the resulting set of clauses $\mathcal{C}'$ anymore.

Example 6 *Let* $\mathcal{C} = \{\{x_1, x_2\}, \{\neg x_1, \neg x_2\}, \{\neg x_1, x_2\}\}$. *Let us assume we do* $UP(x_1)$ *and thereby set* $Val(x_1) = 1$. *We then get*

$$\mathcal{C}' = \{\{1, x_2\}, \{\neg 1, \neg x_2\}, \{\neg 1, x_2\}\} = \{\{\neg x_2\}, \{x_2\}\}.$$

The result is, that we can not satisfy all clauses of $\mathcal{C}'$ *anymore. Whatever we assign to* x_2 *will leave one clause unsatisfied.*

In this case we say x_1 is a *failed literal*. However setting $Val(x_1) = 1$ was not useless. We now know that we must set $Val(x_1) = 0$, since $Val(x_1) = 1$ resulted in a contradiction. We can do another unit propagation: $UP(\neg x_1)$.

Example 7 *Let* $\mathcal{C} = \{\{x_1, x_2\}, \{\neg x_1, \neg x_2\}, \{\neg x_1, x_2\}\}$. *Let us assume we do* $UP(\neg x_1)$ *and thereby set* $Val(x_1) = 0$. *We then get*

$$\mathcal{C}' = \{\{0, x_2\}, \{\neg 0, \neg x_2\}, \{\neg 0, x_2\}\} = \{\{x_2\}\}.$$

The result is, that we now have a reduced set of clauses since only one (still satisfiable clause) is left.

In case $\neg x_1$ would become a failed literal as well, we would know that either assignments to variable x_1 would result in a contradiction. We call this a *conflict* on variable x_1. When a set of clauses contains a conflict, it is unsatisfiable. The propositional formula in CNF that is described by this set of clauses is then unsatisfiable as well.

Equality Reduction Equality reduction was proposed by [BW04] and is the process of identifying variables that are logically equal by the definition of the clauses in $\mathcal{C}$. Consider a set of clauses that contains $c_1 = \{\neg x_1, x_2\}$ and $c_2 = \{x_1, \neg x_2\}$. Both clauses are implications. c_1 is equivalent to $x_1 \rightarrow x_2$ and c_2 is equivalent to $x_2 \rightarrow x_1$. The combination results in $x_1 \leftrightarrow x_2$, and thus x_1 and x_2 will have the same value in a model (if one exists). The reduction is performed by replacing one variable by the other (say x_2 is replaced by x_1).

Then, all clauses containing both literals x_1 and $\neg x_1$ are removed, as well as clauses containing one literal multiple times are simplified so they contain their literals only once.

Example 8 *Let $\mathcal{C} = \{\{\neg x_1, x_2\}, \{x_1, \neg x_2\}, \{x_1, x_2, x_3\}\}$. We detect that the first clauses define an equivalence between x_1 and x_2, thus we replace x_2 with x_1 and get $\mathcal{C}' = \{\{\neg x_1, x_1\}, \{x_1, \neg x_1\}, \{x_1, x_1, x_3\}\} = \{\{x_1, x_3\}\}$.*

Subsumption Principle: We say a clause c_i is subsumed by a clause c_j, iff all literals from c_j also appear in c_i [Rob65b]. We can therefore remove c_i from $\mathcal{C}$. The result is a set $\mathcal{C}'$ that describes a formula logically equivalent to the formula described by $\mathcal{C}$. For an explanation, recall that we need to satisfy all clauses of a formula in order to find a model for it. Hence both clauses c_i and c_j must be satisfied. When c_i is subsumed by c_j, then c_j is included in c_i. If c_j evaluates to TRUE under a given assignment, than this holds for the part of c_i that is equal to c_j as well. Since clauses are sets of literals that are connected by OR, the rest of the literals in c_i can not change its truth value anymore, if c_i evaluates to TRUE. This clause is already satisfied by the subset of elements from c_j.

Example 9 *Clause $c_i = \{x_1, x_2, x_3\}$ is subsumed by $c_j = \{x_1, x_2\}$. If c_j is satisfied, so must be c_i. Since all clauses must be satisfied, we can remove c_i from $\mathcal{C}$.*

We have seen some of the simplification techniques and will now introduce an inference rule named Resolution.

2.3.2 Resolution

Resolution [Qui55, DP60, Rob65a, Rob65b] is a rule of inference. Resolution can infer a new clause from a given set of clauses. These new clauses represent relations of literals that are indirectly given with the original set of clauses. Thus, Resolution makes these implicit relations explicit.

The use of Resolution becomes clear in the context of simplification. The simplification techniques from section 2.3.1 do not apply on implicit literal relations. They only work with the given set of clauses. To enlarge this set, resolution is used, thereby aiding the simplification techniques to exploit more literal relations. The inference rule of Resolution is as follows [Pha06]:

Definition 8 *Given two clauses $c_1 \cup \{x\}$ and $c_2 \cup \{\neg x\}$ from $\mathcal{C}$, called the* `parent clauses`*. Each of these clauses can have arbitrary member literals and must have exactly one resolution literal (in this case x and $\neg x$ respectively). We call the underlying variable x the* `resolution variable`*. The Resolution inference rule is defined*

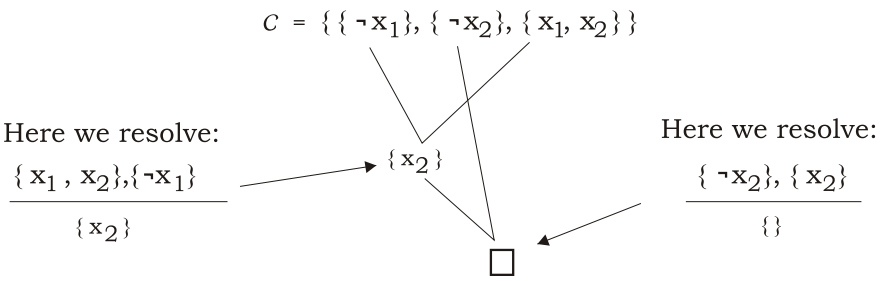


Figure 2.1: Inferring the empty clause with Resolution.

as follows.

$$\frac{c_1 \cup \{x\}, c_2 \cup \{\neg x\}}{c_1 \cup c_2}$$

The resulting clause $c_1 \cup c_2$ is called the `resolvent` *of the parent clauses.*

The resolvent clause can now be added to $\mathcal{C}$, thereby making the implicit literal relations, that where hidden in the parent clauses, explicit. This method is also referred to as *clause learning*.

However, the power of Resolution does not end here. Let us assume, that a set of clauses $\mathcal{C}$ is a contradiction (i. e. the formula described by $\mathcal{C}$ has no model). By using Resolution, we can infer the so called *empty clause* (denoted $\square$).

The empty clause is the elementary contradiction since it can not be satisfied. If we are able to infer the empty clause from $\mathcal{C}$, we know that $\mathcal{C}$ can not be satisfied [Pha06] as a whole.

Example 10 *Let $\mathcal{C} = \{\{\neg x_1\}, \{\neg x_2\}, \{x_1, x_2\}\}$. We can iteratively use Resolution and infer the empty clause. See figure 2.1.*

The attempt to infer all possible resolvents of a clause set $\mathcal{C}$ via Resolution is most likely to take exponential computation time (and space) [Pha06]. Therefore, a modified version of Resolution has been proposed: Hyper-Resolution.

Let $\mathcal{C} = \{c_0, \ldots, c_k\}$ be a set of clauses. Let x_i be a variable with its respective literals x_i and $\neg x_i$. Hyper-Resolution was proposed by [Rob65a] as well as [Bac02] and can be seen as a shortcut to normal Resolution adequate for some sets of clauses. Consider the clause $c_0 = \{x_1, x_2, \ldots, x_k\}$ as well as a number of binary clauses $c_i = \{\neg x_i, x\}, i \in [1, k-1]$. We can directly infer the hyper-resolvent clause $\{x, x_k\}$ without creating all the intermediate clauses that would be needed when using standard Resolution [Pha06].

We now have described a number of simplification techniques that can be used to simplify a propositional formula in CNF, as well as an inference rule that can be used to infer new clauses and detect contradictions. A wide variety of algorithms have been developed that apply the techniques described above. These algorithms are called preprocessors and will be topic of the next section.

2.3.3 Preprocessors

Simply applying simplification techniques and resolution from sections 2.3.1 and 2.3.2 is often not sufficient in order to maximize the simplification. For example, using unit propagation once may yield new unit clauses, so the unit propagation principle should be performed iteratively until no further changes can be made.

Furthermore, using multiple simplification techniques results in interdependencies. For example, using unit propagation might shrink a certain clause, that then suddenly subsumes another clause that it did not subsume before the unit propagation. Therefore, combined simplification techniques should be designed in a way that they are

- efficient (e.g. do not use rechecks with certain techniques where they are not necessary),
- useful (e.g. only combine techniques that do not foil each other),
- correct (i.e. do not alter the set of clauses in a way that the logical equivalency property is lost).

Algorithms that perform simplification according to these properties are called *preprocessors*. These preprocessors are algorithms, that receive a CNF formula (or a set of clauses) as input, and return a (maybe) reduced version that is logically equivalent to the input.

The reduced formula (or set of clauses) can then be used instead of the original one, since it still describes the same problem. Because the simplified formula is logically equivalent to the original formula, a model for the simplified version is always a model for the original version as well. Various preprocessors exist and will shortly be described next.

HypBinRes

In [Bac02], the concept of hyper Resolution (see section 2.3.2) is used in a preprocessor that simplifies a given formula in CNF: HypBinRes.

The first part of HypBinRes infers all resolvents that can be created with the set of binary clauses.

Then, it checks for unit clauses and performs a unit propagation on literals (see section 2.3.1) for each of the literals appearing in a unit clause.

This scheme is repeated until either no further changes happen or a contradiction is found. If the latter happens, the problem instance is unsatisfiable. The described procedure is sketched out in listing 2.1.

The procedure `hyperResolve(b, c)` either resolves `b` and `c` with normal Resolution

Listing 2.1: HypBinRes.

```
HypBinRes(F){
    changesHappened = true;
    while (changesHappened){
        changesHappened = false;
        for (binary clause b : clauses){
            for (clause c : clauses)
            {
                if (b and c can be resolved){
                    clause d = hyperResolve(b, c);
                    add d to the set of clauses from F;
                    changesHappened = true;
                }
            }
            for (unit clause u : clauses){
                unitPropagate(u);
                apply changes in F;
                changesHappened = true;
                if (the empty clause □ was
                    created during unit propagation){
                    return unsatisfiable;
                }
            }
        }
    }
    return simplified F;
}
```

in case these clauses are binary, or, if c is not a binary clause, performs hyper Resolution (with additional binary clauses) if possible.

As stated in [Bac02], using HypBinRes can result in an exponentially shorter search compared to the search performed on the original formula.

3-Resolution

In [LA97], the 3-Resolution algorithm is proposed, which consists of a restricted form of Resolution. This form only adds resolvents to the set of clauses, if this resolvent contains three or less literals.

Therefore, the preprocessor removes all tautologies first. Then, for all pairs of clauses, that have no more than three literals and can be resolved, a resolvent is computed. If the number of literals within that resolvent is no greater then three, it is added to the set of clauses.

Finally, a unit propagation is conducted for all literals appearing in a unit clause. The algorithm is sketched out in listing 2.2.

As stated in section 2.3.2, unrestricted resolution has exponential computation and space complexity. 3-Resolution counters this by only performing resolution on certain clauses and adding them, iff they do not increase the size of the formula.

Listing 2.2: 3-Resolution.

```
3-Resolution(F){
    remove tautologies;
    for (clause c, d : clauses){
        if (size of c ≤ 3 and size of d ≤ 3){
            if (c and d can be resolved){
                clause d = resolve(c, d);
                if (size of d ≤ 3){
                    add d to the set of clauses;
                }
            }
        }
    }
    perform unit propagation on all literals in unit clauses;
    return simplified F;
}
```

2-Simplify

2-Simplify was introduced in [Bra04]. In its core, the 2-Simplify preprocessor uses an implication graph.

This implication graph is constructed from all binary clauses in F and is a directed graph. The vertices of this graph represent literals and its edges represent a clause. For example the clause $(\neg x_1 \vee x_2)$ results in two vertices (one for x_1 and one for x_2) and a directed edge from the x_1 vertice towards the x_2 vertice. A binary clause $(x_1 \vee x_2)$ results in two implications, one for $(\neg x_1 \vee x_2)$ and one for $(\neg x_2 \vee x_1)$.

If there exists a transitive path from x_i to x_k, we can deduce $(\neg x_i \vee x_k)$. If an additional path exists between x_k and x_i, we say that x_i and x_k are strongly connected. That means, that x_i and x_k must be assigned the same value since they imply each other.

For a more detailed description of the usage of the implication graph see [Pha06, Bra04]. The algorithm is sketched out in listing 2.3.

Listing 2.3: 2-Simplify.

```
2-Simplify(F){
    construct the implication graph from all binary clauses;
    collapse all strongly connected components;
    generate transitive closure;
    derive shared implications;
    eliminate subsumed clauses;
    remove pure literals;
    compute transitive reduction;
    return simplified F;
}
```

HyPre

In [BW04], HyPre is introduced. HyPre, as well as 2-Simplify, uses the concept of an implication graph, but instead of using Resolution on binary clauses, it performs hyper Resolution to circumvent the space explosion, that normal Resolution can yield [Pha06].

Furthermore, HyPre uses unit reduction and equality reduction to infer more binary clauses. The algorithm is sketched out in listing 2.4.

Listing 2.4: HyPre.

```
HyPre(F){
    perform unit reduction;
    perform initial equality reduction;
    unmark all literals in the implication graph;
    while (there is an unmarked literal){
        for (unmarked root literal x){
            unit propagate on x;
            perform equality reduction;
            unmark potential literals;
        }
    }
    return simplified F;
}
```

A root literal is a literal in the implication graph that has no parents. A marked literal is a literal that will not yield any new deduction when unit propagation is performed on it. For more information on HyPre see [Pha06, BW04].

VER, NiVER and SATELITE

NiVER was introduced in [SP05], and is the successor of a method called VER. VER itself is a quite simple procedure, that eliminates variables from a formula using Resolution (see listing 2.5).

Listing 2.5: VER.

```
VER(F){
    for (variable x that occurs as x and ¬x in the formula){
        clause set C1 = all clauses that contain x;
        clause set C2 = all clauses that contain ¬x;
        clause set R = computeAllResolvents(C1, C2);
        remove from F all clauses C1 ∪ C2;
        add to F all clauses R.
    }
    return simplified F;
}
```

VER (Variable Elimination by Resolution) eliminates one variable after another until no further Resolutions can be conducted.

Clearly, this method has exponential space complexity [Pha06], and thus is impractical for large formulas. NiVER tries to combat this problem by only performing VER whenever the set of resolvents R is not larger than the set of $C_1 \cup C_2$ in terms of literal count. The name NiVER indicates this: Non-increasing Variable Elimination by Resolution.

This non-increasing property often occurs, since the clauses generated by VER are often tautologies[1] and must not be added to R. In detail, NiVER performs as outlined in listing 2.6.

Listing 2.6: NiVER.

```
NiVER(F){
    for (variable x in F){
        clause set C1 = all clauses that contain x;
        clause set C2 = all clauses that contain ¬x;
        cause set R = computeAllResolvents(C1, C2);
        if (number of literals in R < number of literals in C1 ∪ C2){
            remove from F all clauses in C1 ∪ C2;
            add to F all clauses in R;
        }
    }
    return simplified F
}
```

NiVER does not perform unit propagation and does not check for any subsumptions. SATELITE is an improvement of NiVER, introduced by [EB05], and adds these features to the basic NiVER algorithm. SATELITE is described in [Pha06, EB05] and the algorithm is sketched out in listing 2.7.

Listing 2.7: SATELITE.

```
SATELITE(F){
    unit propagate on all unit literals;
    for (newly added clause c){
        strengthen clauses using self-subsumption rule;
        unit propagate on new unit literals;
        remove subsumed clauses;
        eliminate variables by substitution;
    }
    return simplified F;
}
```

VER tries to reduce the size of F in terms of literal count and so does NiVER. SATELITE on the other hand, because of its subsumption and substitution checks, reduces F in terms of clause count [Pha06].

The question now arises, when it is useful to call for a preprocessor. This will be explained in the next section.

[1] At least in formulas with a reasonable amount of structure. We will come back to this in section 2.7.1

2.3.4 When to use Preprocessors

We have seen a variety of preprocessors in section 2.3.3 that try to simplify the input formula before the actual SAT solver searches for a solution. The question arises of when the application of a preprocessor is fruitful and what preprocessor should be used.

Clearly, the usage of a preprocessor makes no sense when it can not reduce the size of the formula anymore or when the computation time to reduce the formula excels the time to solve the original formula.

The usage of preprocessors is therefore only encouraged, when a problem instance is likely to contain certain structures a preprocessor can exploit (like pure literals or subsuming clauses). The catch hereby is, that this information is most likely not provided.

In [Pha06] a study is provided with numerous experiments on the effects a preprocessor can have on certain instances. These experiments show, that the usage of a preprocessor on randomly generated instances has little effect. Structured instances were often reduced quite effective (sometimes the preprocessor itself was able to solve the problem). We therefore conclude, that the usage of a preprocessor on randomly generated instances should be avoided. We will come back to the different types of problem instances in section 2.7.1.

The feasibility of using a preprocessor on structured instances depends on the structure, that is found in such an instance. A full overview on the effect of preprocessors on a variety of problems is given in [Pha06].

We now have recapitulated propositional logic, along with the conjunctive normal form. We have also introduced various simplification techniques and inference rules that work on CNF formulas along with a set of preprocessors. Additionally, we have defined the basic terms needed to understand a definition for SAT, which will be given in the next section.

2.4 Defining SAT and k-SAT

There are various ways to describe SAT [GPFW97]. Depending on its description, various ways of solving SAT can be presented. However, this work will concentrate on the following definition of the SAT problem:

Definition 9 *Given a propositional formula F. The* `propositional satisfiability problem (SAT)` *consists in finding a model for F, or providing a proof that no such model exists.*

k-SAT is a special case of SAT where F is in CNF and all clauses in F have at most k literals. The formula from Example 1 ($F = x_1 \wedge (x_2 \vee \neg x_3)$) would be part of the

2-SAT problem and the formula from Example 2 ($F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4 \vee x_3) \wedge (x_5 \vee x_1 \vee \neg x_2)$) would be part of the 3-SAT problem.

The reader should note that k-SAT is a problem in $\mathcal{P}$ for $k \leq 2$ [APT79] and $\mathcal{NP}$-complete for $k \geq 3$ [Coo71]. This means that finding a model for a 2-SAT formula can be done efficiently (i.e. polynomial computation time is needed regarding the problem size) but it can not be done efficiently (i.e. needs exponential computation time regarding the problem size) to find a model for a formula if $k \geq 3$.

An efficient algorithm to solve 2-SAT can be found in [APT79]. Unless $\mathcal{P} = \mathcal{NP}$, there is little hope for finding a truly efficient algorithm for k-SAT with $k \geq 3$. Because of this, the term k-SAT refers to the k-SAT problem with $k \geq 3$ throughout the rest of this work, since this is the part of the k-SAT problem domain for which no efficient solution exists.

For additional information on complexity theory in the context of SAT see [Hoo98, Coo00].

2.5 The DIMACS CNF Format

Todays widely used description scheme for problem instances is the *DIMACS CNF input format* [Sat02]. In its core, a problem instance described in this format is a propositional formula in CNF. This is not a limitation for the description of arbitrary problems, since all propositional formulas can be transformed into a logically equivalent formula in CNF [Poo84].

A problem instance in the DIMACS CNF input format has three types of lines: comments, problem descriptions and clause definitions.

A comment is any line starting with a "c" and will be ignored by further processing routines.

The problem description line must occur exactly once and starts with a "p". This line then gives three parameters: the type of input (usually "cnf"), the number of variables, and the number of clauses.

The definition of a clause contains of a set of numbers. Each number refers to a literal (depending on the sign of the number we have a positive or negative literal). A clause definition line ends with a zero.

Recall the formula from Example 2 ($F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4 \vee x_3) \wedge (x_5 \vee x_1 \vee \neg x_2)$). Listing 2.8 presents its description in the DIMACS CNF input format. A similar format for the representation of a solution to a problem instance exists as well and is called the *DIMACS CNF output format*.

In this format, we again have comment lines starting with a "c".

We have a line occurring exactly once, starting with a "s", followed by either "UN-

Listing 2.8: DIMACS CNF input example.

```
c Input file generated from Example 2.
p cnf 5 3
1 2 -3 0
-2 4 3 0
5 1 -2 0
```

KNOWN" (if the file makes no statement on the solvability of the formula), or "UNSATISFIABLE" (which means that the formula has no model), or "SATISFIABLE" (which means there is a model). In the second case, we do not have to present an unsatisfiability proof. In the last case, we must give a non-contradicting assignment for a set of variables that suffice in evaluating the formula to TRUE.

Such an assignment is given in lines that start with a "v". Each line then contains a set of numbers. A positive number means that the variable represented by this number is to be set to 1, a negative number means 0. The list of variable assignments must be terminated by a zero in the end. The ordering of variables in "v"-lines as well as the line ordering is irrelevant. A solution for Example 2 is presented in listing 2.9.

Listing 2.9: DIMACS CNF output example.

```
c Output file generated from the solution of Example 2.
s SATISFIABLE
v 1 -2 0
```

We have now finished our description of the DIMACS CNF input and output format and will continue by a description on how these formats are of use.

2.6 SAT Solvers in the Context of Problem Solving

As we have mentioned in section 2.4, finding an efficient algorithm to solve the SAT problem is a rather hopeless task. Nevertheless, scientists continue the search for algorithms that solve this problem more and more efficiently. These algorithms are called SAT solvers.

The usual application of modern SAT solvers is to receive a file containing the problem instance in the DIMACS CNF format (see section 2.5). They then start calculating until they either find a solution or abort due to certain time constraints. After performing their calculations they will output a file in the DIMACS CNF output format (see section 2.5). In case the solver found a solution, the file contains the variable assignment under which the problem instance evaluates to TRUE.

The general scheme for todays incorporation of SAT solvers and preprocessors into problem solving is outlined in figure 2.2.

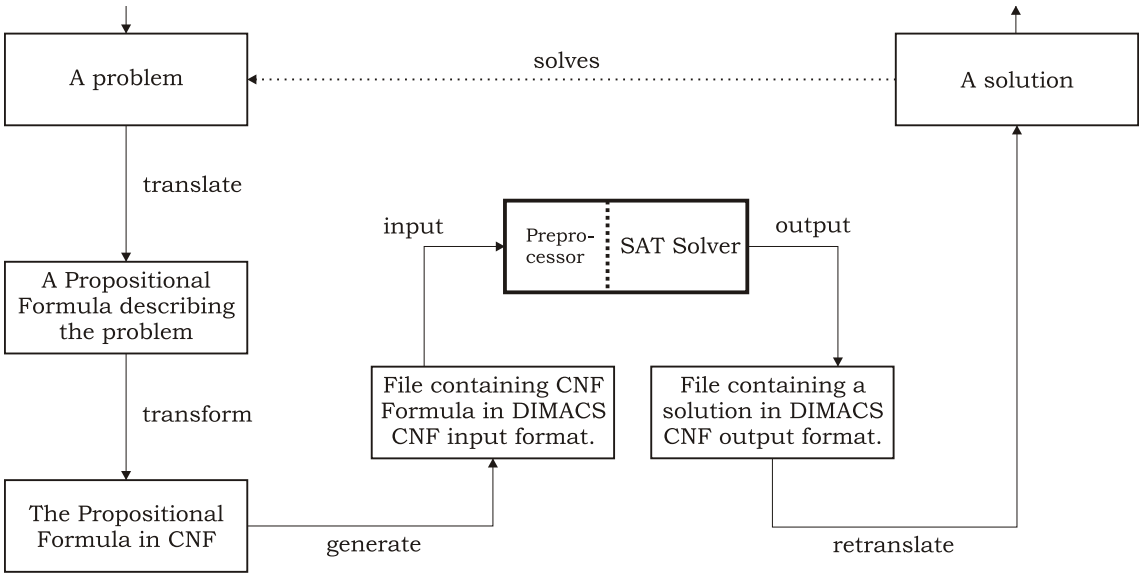


Figure 2.2: The general scheme for using SAT solvers.

We can see from figure 2.2, that the application of a SAT solver premises the translation of a problem into a propositional formula. However, the translation itself depends on the type of problem an no general translation scheme can be presented.

Nevertheless, a wide set of problems can already be translated into propositional formulas. A variety of such problems will be outlined in section 2.8.

We now have explained what SAT solvers are and have put them into the general context of problem solving. Clearly, not all problems are of the same nature or are equally hard to solve. To give some insight on the problem types and hardness of problem instances, we will explain some of their properties in the next section.

2.7 Problem Types and Problem Hardness

2.7.1 Problem Types

According to the SAT Competition[2], problem instances are sorted in three categories: random, industrial and handmade.

Random instances are formulas that are created randomly. Several schemes for such a creation exist [MSL92]. We will come back to random formulas in section 2.7.2.

Industrial instances are formulas, that are created by translating a real world problem into a CNF formula. They often contain exploitable structures and therefore offer the opportunity to use simplification techniques (see section 2.3).

Handmade instances are formulas that are hand-made and give the opportunity to

[2]http://www.satcompetition.org

Industrial	Handmade	Random
Industrial SAT+UNSAT	Handmade SAT+UNSAT	Random SAT+UNSAT
Industrial SAT	Handmade SAT	Random SAT
Industrial UNSAT	Handmade UNSAT	Random UNSAT

Table 2.1: Problem instance categories.

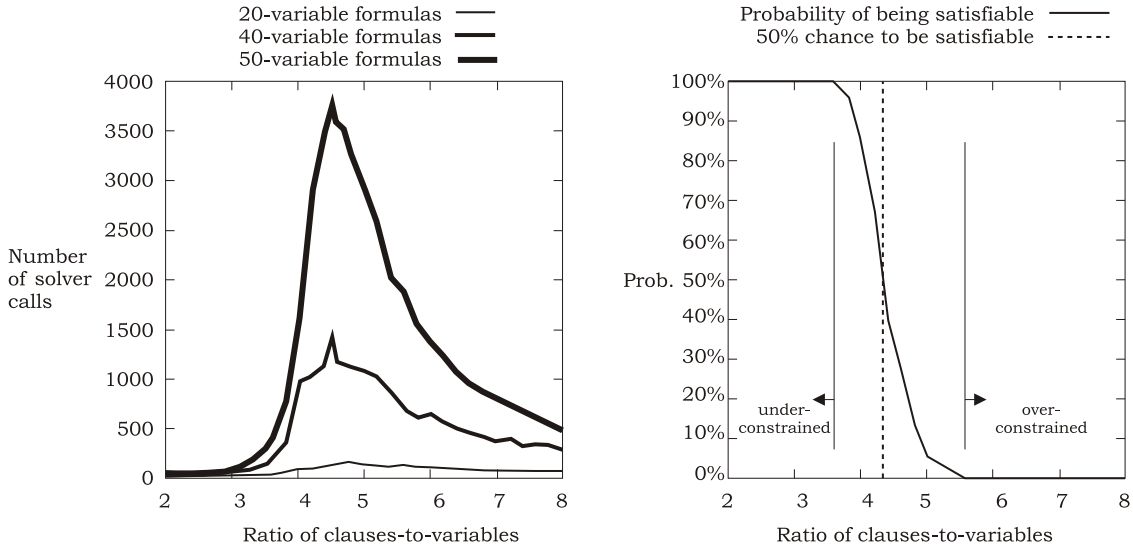


Figure 2.3: Distributions according to the clauses-to-variable ratio.

test the behavior of algorithms that work on these formulas according to certain formula properties.

Furthermore, each problem category is divided into satisfiable and unsatisfiable instances. Altogether, we have 9 sub-categories as presented in table 2.1.

2.7.2 Problem Hardness

The difficulty to find a solution for a problem may vary between different problems. The same is true for finding models for different formulas in CNF.

Various conditions influence the difficulty of a problem instance, for example its size (number of variables or clauses), as well as its structure (number of literals per clause). In section 2.4 we mentioned, that 2-SAT (at most two literals per clause) is a problem in $\mathcal{P}$, while 3-SAT (at most three literals per clause) is $\mathcal{NP}$-complete.

Another condition that influences the difficulty of a problem instance is its clauses-to-variables ratio (the number of clauses divided by the number of variables). In [MSL92], a report on experimental results is given concerning the probability of a randomly created formula to be satisfiable and how hard it is to solve this formula with a simple SAT solver.

A distribution of problem solving time for 3-CNF formulas is given according to the clauses-to-variables ratio (see left side of figure 2.3). As we can see, problems where the clauses-to-variables ratio is about 4.3 seem to be the hardest to solve.

In figure 2.3 (right), we can see that the probability for a randomly created formula with 50 variables is about 0.5 around the same ratio.

Formulas with a ratio much smaller than 4.3 are called *under-constrained*, since they have too few clauses to bring relations to literals, and thus, many solutions exist to solve these formulas [MSL92].

Formulas with a ratio much greater than 4.3 are called *over-constrained*. The set of clauses hereby introduces so many relations to the set of literals, that many contradictions exist, and thus, it is easy to show the formulas unsatisfiability [MSL92].

The region around the 4.3 clauses-to-variables ratio is considered the hardest. A possible explanation for this fact is, that when we start from a formula with a fixed number of variables and add randomly generated clauses, we reduce the number of solutions this formula has. On the other hand, the probability to introduce a contradiction to the formula gets greater. The region about the 4.3 ratio seems to be a place where we have not added enough clauses to create such a contradiction with a high enough probability, but have significantly reduced the number of solutions.

Throughout this work, we are only interested in formulas from the *hard region* where the variables-to-clauses ratio is about 4.26. More information on how to create random formulas, as well as a description of the above mentioned distributions, can be found in [MSL92].

2.8 The practical Relevance of SAT

As mentioned is section 2.6, scientists continue to search more efficient algorithms to solve the SAT problem. This research is mainly driven by the fact that many problems from engineering and computer science can be reduced to SAT [GPFW97] through the application of the general solving procedure (see figure 2.2).

Therefore, SAT can be seen as a fundamental problem whose solving would yield a speed-up in problem solving tasks for various domains. We will outline a few of these domains here and make an excerpt of the more detailed overview given in [GPFW97].

Mathematics finding n-ary relations such as transitive closure, detecting graph and subgraph isomorphisms

Computer science and artificial intelligence the constraint satisfaction problem, the n-queens problem, semantic information processing, theorem proving, neural network computing

Machine vision shape from shading problem, image matching problem

Robotics packing problem

Database systems database consistency maintenance, query-answering and redundancy checking

Integrated circuit design automation circuit modeling, testing and test generation, verification

This list is by far not complete but should give a reasonable motivation for conducting research on SAT solvers.

Summary of Part I

We have introduced a widely accepted definition of the SAT problem and explained the basic terms that are connected with it. We have put SAT solvers into the context of problem solving and explained how they can be of use. Therefore, we have described the commonly used DIMACS CNF format. Finally, we explained what types of problem instances exist, and which of them are of interest.

Two souls alas! are dwelling in my breast.
from "Faust: The Tragedy Part One"
– Johann Wolfgang von Goethe

PART II

The two SAT Solver Paradigms

We will now take an overview of the two SAT solver paradigms: DPLL and SLS. Since the construction of a hybrid solver will include concepts from both worlds, it is necessary to understand their respective benefits and drawbacks. Each paradigm contains two categories. A modern solver following one of the paradigms can be inserted into at least one of these categories (if not both). For DPLL Solvers, these two categories are *look-ahead* and *conflict-driven* [MvVW06]. For SLS we have the categories *random-walk* and *clause-weighting* [SSH01].

In chapter 3 we introduce DPLL solvers by presenting the basic idea behind Variable Elimination by Resolution and the first procedure that implemented it, the Davis-Putnam Procedure [DP60]. Then, two modern DPLL solvers are presented, both of them with unique features and a reasonable performance. We will explain March_ks in section 3.2, which is a representative for look-ahead solvers. This is followed by a description of RSat in section 3.3, which is a representative for conflict-driven solvers.

Chapter 4 is about SLS solvers. We will introduce the underlying idea of local search together with the first solver that implemented it, GSAT [SLM92]. This is followed by an explanation of two modern SLS solvers. In section 4.2, we describe adaptG^{2}WSAT$_0$, which is a random-walk solver. In section 4.3, we will explain gNovelty+, which is a representative for clause-weighting solvers.

The description of DPLL and SLS solvers is kept independently. The reader can continue with which is of more interest.

Chapter 3

DPLL Algorithms

The acronym DPLL stands for "Davis-Putnam Logemann and Loveland" and identifies the class of constructive backtracking algorithms that are used to solve the SAT problem.

The major advantage of DPLL algorithms as a whole is that they are complete. For a given formula, a DPLL algorithm will find a solution, iff one exists or can give a proof for unsatisfiability. The major disadvantages of DPLL algorithms are their relative complexity and the fact that they do not scale well. Therefore, the application of a DPLL solver must be conducted with caution. Some formulas from the under-constrained domain (see section 2.7.2 on page 22) can be solved far better by solvers following a different paradigm. But as soon as the unsatisfiability of a formula is to be shown or whenever a formula from the over-constrained domain is given (see section 2.7.2 on page 22), the use of a complete solver is essential.

Today, DPLL algorithms fall in one of two categories [MvVW06]: *look-ahead* and *conflict-driven*. After introducing the basic DP procedure (along with its successor DLL) on which all DPLL solvers are based, we give an example for a look-ahead solver (March_ks) in section 3.2 followed by an example of a conflict-driven solver (RSat) in section 3.3. Both categories have particular strengths and weaknesses on certain types of problems [MvVW06]. For example, look-ahead solvers can solve randomly created formulas more efficient than conflict-driven solvers. For other problem types, the opposite is true.

3.1 VER and the Davis-Putnam Procedure

The main objective for a DPLL solver is to find a model for a given formula. Recall, that a formula F evaluates to TRUE under an assignment A, iff all clauses in F evaluate to TRUE under A. Clearly, a formula that contains only pure literals would be the easiest case. This is true, because a pure literal can always be assigned in such a way that all clauses in which this literal occurs would evaluate to TRUE. The

problem is, that a general formula F will not consist of only pure literals. This means, that the appearance of a variable in both polarities (i.e. as a positive and a negative literal), prohibits the usage of only the purity principle to find a solution. Additional techniques are needed. The first algorithm that brought forward such additional techniques was the Davis-Putnam procedure (DP for short) [DP60]. The basic idea behind this procedure is to remove all variables from F that are not pure by using VER (see page 16). Thereby, DP gains a formula that consists of only pure literals on which the purity principle can be used to receive a model.

DP was originally designed to check the validity of a first-order logic formula, which is not necessarily a formula in CNF as we understand it here. However, adapted to our understanding of SAT to find a model for a propositional formula in CNF, a simplified version of the algorithm can be presented (see listing 3.1). The algorithm performs VER on the formula F. Two possible outcomes of VER exist:

VER creates the empty clause (by using resolution): If this case occurs, then the formula is unsatisfiable. As mentioned in section 2.3.2, deducing the empty clause from a set of clauses suffices in showing that there is no model for this set.

VER does not create the empty clause: If this case occurs, then all variables left within the simplified formula are pure. Hence we can apply the purity principle from section 2.3.1 to find a model for the formula as a whole.

Listing 3.1: The Davis-Putnam procedure for propositional logic.

```
DP(F){
    F‘ := VER(F, x);
    if (F‘ contains the empty clause){
        return unsatisfiable;
    } else {
        output a model using the purity principle;
    }
}
```

The use of VER has exponential space complexity (see section 2.3.3 on page 16). This is the basic reason why the DP procedure does not scale well. The successor of the DP procedure tries to circumvent this problem by following a slightly different approach. This approach is explained next.

3.1.1 DLL

DLL was introduced as the successor of DP in [DLL62]. The easiest way to circumvent the exponential space requirements of VER is simply not to use it. This leaves us with arbitrary formulas consisting of variables that are not necessarily pure. Another approach, different form VER/DP, is needed.

The approach of DLL to find a solution is to create a binary search tree from the formula. In such a tree, every node holds a set of information: the currently investigated formula F and an assignment A that has been created so far. Furthermore, each node corresponds to a (yet unassigned) variable x_i, that is to be assigned next.

When search begins with DLL, the first search node (also called the root node), contains the original formula F and an empty assignment A (with no variable assignments in it). Two edges now leave this search node: one representing the action of assigning $x_i = 1$ and one representing the action of assigning $x_i = 0$.

Whatever we assign to x_i, the assignment certainly affects the currently investigated formula. The computation of this effect is simply done by performing unit propagation on literals (see page 10). In case we decide on $x_i = 0$, we perform UP($\neg x_i$), while deciding on $x_i = 1$ results in UP(x_i). The decision $x_i = v$ ($v \in \{0, 1\}$) is saved in A, resulting in a new (extended) assignment A‘.

Applying A‘ on F results in a new (reduced) formula F‘. Applying A‘ on F is done in two steps. First, all clauses that get satisfied by $x_i = v$ in F are removed. Second, all literals in F that evaluate to FALSE under $x_i = v$ are removed from the clauses in F. Three possible results from such a propagation are possible:

- F‘ is a simplification of F that has no further clauses (i.e. F‘ is the empty formula).
- F‘ is a simplification of F that is not the empty formula and does not contain the empty clause.
- F‘ now contains the empty clause.

In case the resulting formula has no further clauses, we have found a model in A, because all clauses are removed. Removing a clause only happens, when an assignment in A makes this clause true. When all clauses got removed, all clauses got satisfied, and thus, the formula as a whole evaluates to TRUE. In this case, the search is finished and we can return A as a model for F.

In case the formula is not empty and the empty clause does not occur, we continue the search by repeating the scheme above (selecting a variable, decide on a value, propagate the decision).

In case the empty clause does occur, we have built an assignment A that does not suffice in satisfying F. This happens, when all literals in a clause evaluated to FALSE under the decisions made (i.e. under the assignment A created so far). As stated earlier, a literal evaluating to FALSE is simply removed from a clause it appears in. An empty clause evaluates to FALSE, since all literals in the conjunction this clause represents evaluate to FALSE. Since no further literals in the clause are left to change its truth value, the assignments in A will not suffice in making F TRUE, no matter what we assign to other variables. Therefore, under this assignment A, F as a whole can not evaluate to TRUE since not all of its clauses evaluate to TRUE.

In order to find a solution, we must revise an earlier decision that was made in order to create the assignment A. We therefore undo the last decision $x_i = v$, and instead propagate $x_i = \neg v$, thereby creating a different assignment B. The result is, that instead of F with assignment $x_i = v$ (yielding F‘ with an empty clause), we can check on F with the assignment $x_i = \neg v$ (yielding a different simplification F“). One could say that F“ is the result of taking upon a different opportunity to assign a value to x_i.

Now, F“ could as well contain the empty clause. This means, that both assignments to x_i (0 and 1) result in an assignment, that can not suffice in making F TRUE. If this is the case, we need to backtrack.

Backtracking identifies the process of moving back to older decisions made earlier in the search tree. Therefore, the current node is left (x_i gets unassigned again) upwards (i.e. to the parent node of the decision in x_i).

Let us assume this node is labeled x_k. We have just returned to x_k and thereby, have undone a decision made for x_k (lets say $x_k = v$).

Now two options arise. First, we have not tried $x_k = 1 - v$. If this is the case, we propagate this decision, again stepping further down in the search tree. Second, we have already tried $x_k = 0$ and $x_k = 1$. In this case, x_k does not yield any further opportunities to conduct search (i.e. all tried possibilities resulted in an empty clause). We then further backtrack to the predecessor of x_k, where we again check for possibilities and so on.

The root node is somewhat special. In case we backtrack to the root node and have no further opportunities left, the formula must be unsatisfiable. This is true, because all assignments made for the root node variable resulted in an empty clause via propagation on literals. Hence, at least one clause is not satisfied by the follow up decisions of assigning the root node variable.

To clarify the principle of the DLL search, we use the following example. In figure 3.1, we visualize the search for the following formula in a (binary) search tree:

$$F = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3).$$

To conduct search, we assume that the first variable we want to assign a value to is x_1, creating the first search node (i.e. the root node). We decide to assign the value 1 to it at first, so we leave the first search node via the UP(x_1) edge entering the second search node (see figure 3.1). This results in the formula

$$F‘ = (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3).$$

We gained this F‘ by performing UP(x_1) the following way. First, we remove $\neg x_1$ from the first and last clause in F, since literal $\neg x_1$ evaluates to FALSE under the assignment $x_1 = 1$. Furthermore, we can remove two clauses in which the literal x_1 occurs, since the corresponding clauses are now satisfied.

As we can see, F‘ is neither the empty formula nor does it contain the empty clause,

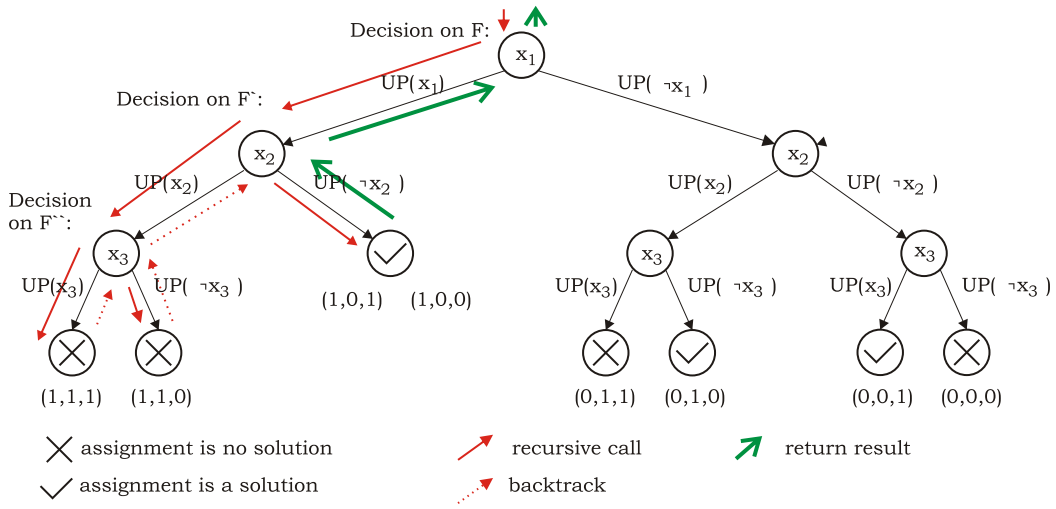


Figure 3.1: A complete search tree for a formula with three variables when searching for a solution using only unit propagation on literals.

so we continue the search by assigning a value to another variable, still unassigned in F‘. Let us pick x_2 now. We assign $x_2 = 1$ as well, which leads to the formula

$$F\text{“} = (\neg x_3) \wedge (x_3).$$

Again, no empty clause occurs. We continue the search by assigning 1 to the last variable that is left: x_3. We can now remove the second clause in F“, because it is satisfied. We remove $\neg x_3$ from the first clause, since this literal now evaluates to FALSE. The resulting formula

$$F\text{“‘} = ()$$

contains the empty clause.

Now at this point we realize that a somehow wrong decision on a variable assignment happened. We backtrack to the node, where x_3 receives an assignment (for formula F“), and continue with $x_3 = 0$ (we have not tried this yet). This again results in the empty clause. We again have to backtrack.

We have tried both assignments for x_3, and this means, we have to backtrack further, going back to x_2 and $F\text{‘} = (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$. At this point, we have not tried $x_2 = 0$, so we propagate this decision. The result is the empty formula, because we can remove all clauses with $x_1 = 1, x_2 = 0$. We have found a model for F. The assignment for x_3 is irrelevant under these assignments.

As we can see, it is quite cumbersome to navigate through a search tree using only unit propagation on literals. Especially the decision on $F\text{“} = (\neg x_3) \wedge (x_3)$ is dispensable. Adding the unit reduction principle would prevent a decision on x_3 here, since the unit reduction would suggest $x_3 = 0$ and $x_3 = 1$, already indicating

a conflict.

Furthermore, the decision for x_2 on $F' = (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$ could be avoided by incorporating the purity principle in our search. Here, $\neg x_2$ is pure, already indicating $x_2 = 0$. If we would have used the purity principle above, we would have avoided the search on x_3 completely, since $x_1 = 1$ (decided earlier) and $x_2 = 0$ (decided because of the purity principle) already suffices in solving F.

Because of this, DLL incorporates the use of the purity principle and unit reduction principle in each search node beforehand of a decision for a new decision variable. DLL works as outlined in listing 3.2.

Listing 3.2: The DLL procedure.

```
DLL(F, (partial) assignment A){
    apply assignment A on F;
    //thereby removing all satisfied clauses
    //and all literals from clauses that evaluate to false
    if (F contains the empty clause){
        //assignment A yields a contradiction
        return unsatisfiable;
    }
    for (every pure literal in F){
        apply the purity principle until no more changes appear;
        modify A accordingly;
    }
    for (every unit clause c in F){
        apply unit reduction until no more changes appear;
        modify A accordingly;
    }
    if (F == empty formula) {
        return A;
    }
    variable x_i = selectNewBranchVariable(F);
    F' = UP(x_i);
    A' = A ∪ {x_i = 1};
    if (F' == empty formula){
        return A'; \\model found
    }
    B = DLL(F', A');
    if (B == unsatisfiable){
        //UP(x_i) was a mistake
        //so we check with UP(¬x_i)
        F'' = UP(¬x_i);
        A'' = A ∪ {x_i = 0};
        if (F'' == empty formula){
            return A'' //model found
        }
        return DLL(F'', A'');
    } else {
        //UP(x_i) was not a mistake, a solution
        //was found using the partial assignment
        //with x_i = 1.
        return B;
    }
}
```

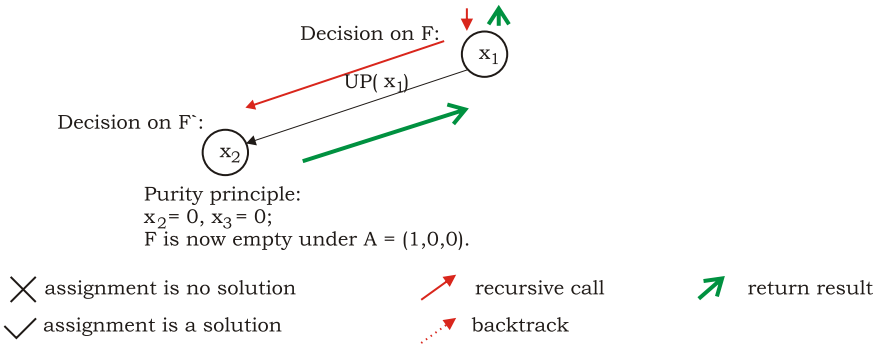


Figure 3.2: A DLL search tree for a formula with three variables when searching for a solution using all features.

In figure 3.2, we see the impact of the usage of the purity principle and the unit reduction principle. The search tree must be explored only in two nodes (instead of the six nodes that need to be checked when only using unit propagation on literals). This of course, saves memory, and is the big advantage over VER/DP, as already mentioned.

To summarize, the DLL procedure recursively constructs a binary search tree, in which every search-node (or simply node) holds a set of information: The currently investigated formula, the assignment A that has constructively been created so far and a variable x_i that is used for the next branch.

The root node therefore holds the original formula and an assignment A which is empty (currently holds no assignments to variables).

In each search node, the currently investigated formula is checked on pure literals and unit clauses, which might imply assignments for variables. After this, a variable is chosen for a new decision (via `selectNewBranchVariable(`F`)`). A branch can be seen as traveling down the search tree by assigning a value to the new decision variable and committing to this assignment via unit propagation on literals.

Backtracking can be seen as traveling back up to a previously created node, that still holds an untried variable assignment opportunity.

Creating a search tree and exploring it the way described above makes DLL a complete SAT algorithm. Hereby complete means, that it will explore the complete search tree, trying all assignments possible (if necessary) until a model is found. One can say that DLL performs a systematic search via a binary search tree.

Once all assignments have been tried, but none of them suffices in solving the formula, we can conclude the formula to be unsatisfiable.

Furthermore, if necessary, DLL can be used to find *all* models for F. This is achieved by continuing search even if a model has been found. DLL will then search through the complete search space, thereby finding all models one after another.

VER/DP and DLL together establish the basic procedure to solve SAT problems in a complete manner. Since DLL is the successor of VER/DP that concentrates on only the steps necessary for SAT but borrows heavily from the concepts of VER/DP, we call algorithms based on DLL the DPLL algorithms.

Many more complex algorithms [Cra], decision strategies and further simplification techniques have evolved. We now explain two solvers that are considered state-of-the-art in the DPLL domain. We thereby introduce various ideas that are considered valuable to solve SAT in a complete fashion.

3.2 March_ks

3.2.1 March_ks Background

March_ks was the winner of the SAT 2007 Competition[1] in the "Handmade SAT" and "Random UNSAT" category and became second in the "Random SAT+UNSAT" category. Therefore, March_ks is one of todays state-of-the-art SAT solvers.

Originating from a solver called "march", that was developed at the TU Delft, several versions have evolved over time. The first solver in this line with reasonable performance was called March_eq [HvZDvM04]. It was developed in 2004. Its successor became March_dl [HvM06] in 2006. The last in this line of developments was March_ks [HvM07b] in 2007.

Our description of March_ks will inherit the explanation of various features from the former march versions. We thereby aim for a complete description of the march solver family, using the functioning of March_ks as a leitmotif. March_ks is used as a representative for the look-ahead architecture of DPLL solvers.

The description of March_ks is structured as follows. The next section will give an overview of the functioning of March_ks. Since March_ks is a DPLL solver, we will use the basic DLL algorithm as explained in section 3.1.1 to point out the major improvements introduced in March_ks.

After the general ideas in March_ks have been given in this overview, we will continue by explaining the detailed functioning of March_ks and its various features.

After the detailed description of March_ks has been given, we take a more detailed look at the performance of March_ks at the the SAT 2007 Competition. The section is concluded by a summary of the March_ks review.

[1] http://www.satcompetition.org

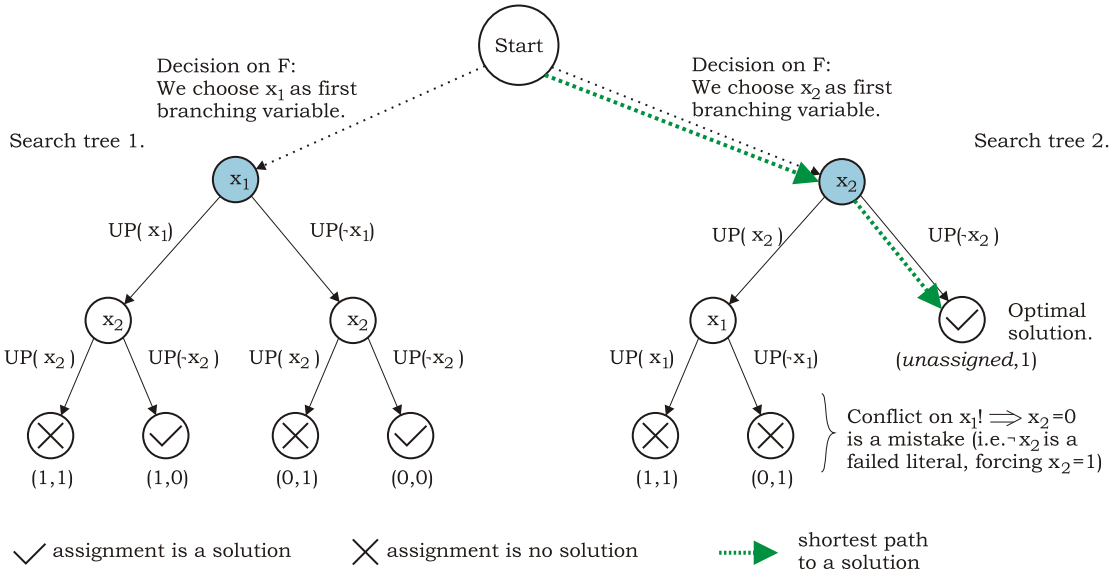


Figure 3.3: Two different search trees for F. The search tree structures depend on the first branching variable. The root nodes for their respective binary search trees are shaded. For search tree 2, we can see how look-ahead detects failed literals.

3.2.2 Ideas used in March_ks

Optimizing the Search

Like any DPLL solver, March_ks's main objective is to find a model for a given formula. In order to conduct search, it uses a binary search tree. As we have seen during the description of DLL in section 3.1.1, the construction of such a tree requires various actions, like

- the selection of a branching variable
- the assignment that is first explored for this variable
- backtracking to earlier explored nodes when a dead end is reached in the search tree (i.e. a contradiction is found).

Each of these actions forces us to make decisions, and, which is even more important, give us the opportunity to collect information in order to make the decision as optimal as possible. The question now is, what decisions are considered to be optimal when performing a certain action. Consider the formula

$$F = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2).$$

The complete search tree of this formula is presented in figure 3.3. Note that the root of this tree is a node not already holding a decision for a branching variable. In fact, we have two binary search trees here. Search tree 1 is used, when variable x_1 is chosen as first branching variable. Search tree 2 is used, when x_2 is used as

first branching variable. It is important to understand, that not the start-node is the root node for the search. We combine these two search trees in one figure here, to point out certain properties they have.

Since we have two binary search trees here, we need two root nodes (which are shaded in figure 3.3). Search is performed in exactly one search tree here, which is determined by the first branching variable we choose.

When starting the search in node start (before any decisions have been made), we would like to reach the solution closest to the start node (see figure 3.3). Such a solution would be reachable in a minimum number of decisions, and therefore, through a minimum amount of computation which in turn minimizes the search time the algorithm needs to find this solution. Hence, we need to branch on variables that are part of a path that is as short as possible. Furthermore, we need to assign the "right" value to the selected branching variable in order to stay on the path to such an optimal solution.

Look-ahead

When March_ks has to decide for a new branching variable, it collects information on that it bases its decision. This "collecting" of information is done via a *look-ahead*. A look-ahead is the process of performing temporary unit propagation on literals that correspond to free variables in the currently investigated formula. For our example formula $F = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$, we would perform look-ahead as follows:

- Look-ahead in F on variable x_1:
  - $\mathrm{UP}(x_1) \Rightarrow F\text{‘}_1 = (\neg x_2)$.
  - $\mathrm{UP}(\neg x_1) \Rightarrow F\text{“}_1 = (\neg x_2)$.
- Look-ahead in F on variable x_2:
  - $\mathrm{UP}(x_2) \Rightarrow F\text{‘}_2 = (x_1) \wedge (\neg x_1)$.
  - $\mathrm{UP}(\neg x_2) \Rightarrow F\text{“}_2$ is the empty formula.

As we can see, the look-ahead on x_2 via $\mathrm{UP}(\neg x_2)$ results in the empty formula (i.e. all clauses in this formula got removed since they all got satisfied). Therefore, March_ks would "know" that branching on x_2 would have the best chances in reaching a solution. Furthermore, the decision on what value must be assigned to x_2 first to reach this solution ($x_2 = 0$), becomes clear as well.

However, reaching a solution with using only a single look-ahead is not the general case. In most cases, the resulting formulas $F\text{‘}_i, F\text{“}_i$, from the unit propagation on the free variables x_i, will be simplified versions of F. The decision on what variable to branch on first is then not so clear. Nevertheless, a decision on what variable to use for branching is needed!

How to select a branching variable: March_ks uses the informations gathered by performing look-ahead to rank the variables via a so-called *look-ahead evaluation function*. The exact functioning is irrelevant at the moment. One must only understand that this evaluation function results in a variable ranking even if no solution is in sight.

The higher the rank of a variable, the smaller its resulting search tree is when branching upon it. The variable with the highest rank (smallest remaining search tree) is then used for branching.

Let us come back to the example again. After performing look-ahead, we decide that we branch on x_2. Therefore, we use search tree 2 for finding a model for F (we now stand at the root node of search tree 2). We now have to decide for what value we want to assign to x_2 at first. Recall the look-ahead result for x_2:

$$F\text{‘}_2 = (x_1) \wedge (\neg x_1) \text{ for } x_2 = 1$$
$$F\text{“}_2 \text{ is the empty formula for } x_2 = 0.$$

Now at this point it seems intuitive to assign $x_2 = 0$, since we thereby reach a solution. However, we also need to make a reasonable decision, even if no solution is in sight.

How to select the first assignment for a branching variable: The decision on what direction is followed first (UP(x_i) or UP($\neg x_i$)) is computed by a *direction heuristic*. This heuristic simply compares the number of reduced clauses in the resulting formulas F‘ and F“ after the respective unit propagation. We call a clause *reduced*, when it is still unsatisfied after the unit propagation, but got at least one of its literals removed.

The direction heuristic decides for following UP($\neg x_i$), when the number of reduced clauses in F“ is *less or equal* the number of reduced clauses in F‘. It follows UP(x_i), if the opposite is true.

To understand the usefulness of this decision, one must understand, that in general, the stronger the reduction, the higher the probability for the sub-formula to be unsatisfiable [HvM08]. Since we are interested in a solution, we should follow the path with the least reduction for a given variable.

In our example above, UP($\neg x_2$) resulted in the empty formula $F\text{“}_2$, and therefore, no remaining clause exists that can be counted as a reduced clause. The number of reduced clauses is therefore zero. UP(x_2) resulted in the formula $F\text{‘}_2$, by reducing two clauses in F. The reduced clause count for $F\text{‘}_2$ is therefore two. The direction heuristic will choose to follow UP($\neg x_2$), since the number of reduced clauses in $F\text{“}_2$ is zero, which is smaller than the two reduced clauses in $F\text{‘}_2$.

However, we do not have a guarantee for finding a solution inside the sub-tree we enter while performing search, despite of the calculations made for look-ahead along

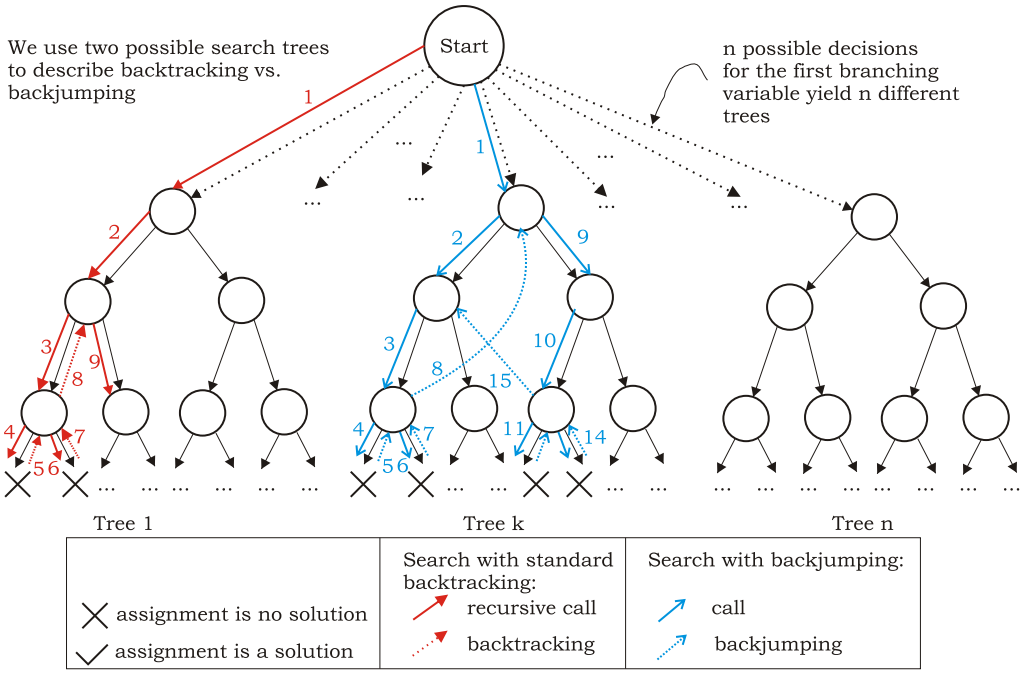


Figure 3.4: The proceeding of the search when using only standard backtracking (red), and backjumping (blue).

with the look-ahead evaluation function and the direction heuristic. Sometimes we need to undo a decision and perform backtracking.

Revising Decisions

In DLL, backtracking just moves back to the previously visited node, but March_ks tries to optimize this decision as well.

March_ks not only moves back one node, but will jump towards a different (yet partially unexplored) node of the search tree if this node seems to be more promising. This process is called *backjumping*. The idea behind backjumping is visualized in figure 3.4 for an arbitrary formula with n variables.

The standard backtracking, as performed by DLL, will always travel back to the next predecessing node that holds a still unexplored part of the search tree (see steps marked 5, 7, 8 in red). The standard backtracking is therefore called chronological backtracking (since it returns to the nodes that have been created most lately). Backjumping on the other hand can skip certain nodes when traveling back (see step marked 8 in blue), and return to nodes left behind later on (see step 15 marked in blue).

The decision on where to continue the search is controlled by the *jumping strategy*. The jumping strategy is used to rank search nodes according to their ability to hold a solution. The computations performed to obtain this ranking are derived from an empirical study [HvM08], that investigated the distributions of solutions in binary

search trees. In short, this study showed that there exists a better strategy in visiting nodes than the chronological one. This strategy is *called distribution* jumping, and allows March_ks to improve the way decisions are revised.

March_ks remembers partially unexplored nodes it left behind, so that it might return to them in the future when this becomes necessary. Therefore, March_ks is still a complete solver even though the structure of the binary search tree is not strictly followed.

We have now finished in roughly outlining the most important features of March_ks. The following section will give a more in-depth explanation for them, as well as for other features March_ks uses to improve its performance (for example a preprocessor).

3.2.3 The functioning of March_ks

Before March_ks actually starts its search for a model, it uses a preprocessor to try simplifying the formula at hand.

The Preprocessor of March_ks

The preprocessor of March_ks uses several tools for simplification and performs some preparations for the main solving procedure.

- Perform simplification operations (i.e. iteratively perform unit reduction)
- Construction of the CoE
- Root look-ahead and the detection of constraint resolvents
- 3-SAT translation
- Adding ternary resolvents
- Calculate the maximum depth for jumping d_{border}

These tasks will now be explained in more detail.

Primary simplifications: The primary simplifications done by the preprocessor are straight forward. It checks for unit clauses and performs unit reduction.

Furthermore, the preprocessor detects binary equivalences. A binary equivalence is the equivalence of exactly two variables. These equivalences are defined in a formula via so-called *equivalence clauses*. An equivalence clause is a clause that participates in defining an equivalence between variables. For example, the formula $F = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (x_3)$, has two implications: $x_1 \rightarrow x_2$ and $x_2 \rightarrow x_1$ (defined via the first two clauses) which together form the binary equivalence $x_1 \leftrightarrow x_2$.

When binary equivalences are detected, they get removed. Therefore, we replace one corresponding literal with the other and remove clauses containing both literals of a variable. For the example above, we could replace x_2 with x_1, resulting in the formula $F_{simplified} = (\neg x_1 \vee x_1) \wedge (\neg x_1 \vee x_1) \wedge (x_3) = (x_3)$.

The CoE: The so-called *CoE* (Conjunction of Equivalences) is constructed right after the primary simplifications have been performed. Therefore, the preprocessor detects equivalence clauses that define equivalences between more than two literals.

It removes these equivalence clauses from F and adds them to the CoE. This extracted set of equivalence clauses is a sub-formula of F and can be solved separately [HvM04]. This process of separately solving the CoE is described in [HvM04] and [WvM98] and due to its complexity, it will not be explained here. For the context of this review it is sufficient to know, that the CoE contains the detected equivalence clauses. The CoE is later used to collect information on equivalence clauses for various calculations [HvZDvM04].

Root look-ahead: The root look-ahead is a look-ahead on all variables in F. While the preprocessor performs its look-ahead operations, it detects *constraint resolvents*. Constraint resolvents are created when an iterative unit propagation on a variable yields a unit clause. This unit clause combined with the negation of the literal that was initially propagated then forms a constraint resolvent, iff the created unit clause was a ternary clause beforehand. The detection of a constraint resolvent is given in the following example.

Example 11 *Given the formula* $F = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4)$. *We iteratively perform UP*($\neg x_1$). *This yields the unit clause* (x_4). *Since* (x_4) *was in the ternary clause* $(x_2 \vee x_3 \vee x_4)$ *before the iterative unit propagation of* $\neg x_1$, *we can add the constraint resolvent* $(x_1 \vee x_4)$ *to* F.

Constraint resolvents are *learned* by March_ks (i.e. are added to the investigated formula). It is shown in [HvZDvM04], that adding such constraint resolvents outperforms search with no learning. The reason behind this is, that with constraint resolvents added, further look-aheads can detect failed literals sooner. Thereby, some search paths in a binary search tree are sooner identified as dead ends. Since these dead ends are then not further explored, search time is saved.

3-SAT translation: 3-SAT translation was introduced in March_eq [HvZDvM04], and became an optional feature in March_ks. It simply transforms a given k-CNF (with arbitrary k) into a 3-CNF formula ($k \leq 3$).

Therefore, each disjunction of literals, that occurs more than once, is substituted by a new *dummy variable* (starting with the most frequently occurring pair). After this substitution, three additional clauses are added for the dummy variable.

These clauses make the dummy variable logically equivalent to the disjunction it substitutes. To clarify this, consider the following example [HvZDvM04].

Example 12 *Lets assume we want to substitute the disjunction* $\neg x_2 \vee x_4$ *by the dummy variable* d_1 *in the formula* $F_a = (\neg x_2 \vee x_4 \vee x_1 \vee x_3) \wedge (\neg x_3 \vee \neg x_4)$*. We therefore replace all occurrences from* $\neg x_2 \vee x_4$ *by* d_1 *and add the three clauses* $(d_1 \vee x_2) \wedge (d_1 \vee \neg x_4) \wedge (\neg d_1 \vee \neg x_2 \vee x_4)$*. This results in the formula* $F_b = (d_1 \vee x_1 \vee x_3) \wedge (\neg x_3 \vee \neg x_4) \wedge (d_1 \vee x_2) \wedge (d_1 \vee \neg x_4) \wedge (\neg d_1 \vee \neg x_2 \vee x_4)$*. As we can see,* F_a *was originally a 4-SAT formula, but got translated in a logically equivalent formula* F_b*, which is a 3-SAT formula.*

Ternary Resolvents: The preprocessor of March_ks performs resolution for certain cases. The aim is to learn *ternary resolvents*. Ternary resolvents are resolvents that have exactly three literals and can be resolved from two parents that have exactly three literals as well.

Example 13 *Consider the two parent clauses* $(x_i \vee x_j \vee x_r), (x_i \vee \neg x_j \vee x_s)$*. Resolving on variable* x_j *yields the ternary resolvent* $(x_i \vee x_r \vee x_s)$*. [HvZDvM04]*

The motivation for adding ternary resolvents in the preprocessing phase is, that they reduce the overall computational cost in finding a solution by about 10% [HvZDvM04].

Maximum jumping depth d_{border} March_ks is not using chronological backtracking as explained in the previous section. Instead, it uses backjumping. The constant d_{border} is needed for backjumping and defines the maximum depth in the search tree for which backjumping is used. More information about the usage of d_{border} and backjumping will be given later. For now it is sufficient to know, that the preprocessor calculates this d_{border} for later usage.

The operations performed by the preprocessor are summarized in listing 3.3.

Listing 3.3: The March ks preprocessor.

```
Preprocess(F){
    propagate unit clauses;
    propagate binary equivalences;
    transfer equivalence clauses into the CoE;
    perform root look-ahead;
        //while performing root look-ahead:
        detect failed literals;
        add Constraint Resolvents;
    (optional) translate the formula into 3-SAT;
    add Ternary Resolvents;
    compute d_border;
    return simplified formula;
}
```

After preprocessing, the solver will call its main solving procedure to start its search on the simplified formula[HvM07b]. How this search is performed in detail will be explained next.

Performing search with March_ks

March_ks is a derivate of the DLL algorithm. Therefore, March_ks, as any DPLL solver, needs to choose a branching variable that is to be assigned for the next recursive step of the constructive search. In contrast to the DLL procedure from section 3.1.1, look-ahead solvers perform a variety of calculations to choose this variable "wisely".

The Application of Look-ahead

Look-ahead embodies the calculations that lead to the selection of a new branching variable. A look-ahead on variable x in formula F is done by performing the following steps:

- Perform UP(x) in F (i.e. set $x = 1$ in F). This includes the repeated application of the unit reduction principle until no further changes happen. The result is the formula F‘.
- Perform UP($\neg x$) in F (i.e. set $x = 0$ in F). This includes the repeated application of the unit reduction principle until no further changes happen. The result is the formula F“.

The aim of look-ahead: The aim of look-ahead is twofold. First, it tries to find failed literals in F as soon as possible to prune the search space for the constructive search. A failed literal is detected, if exactly one resulting formula of the look-ahead on a given variable contains the empty clause.

The pruning of the search space is achieved by replacing the original formula F by the formula that did not contain the empty clause. Thereby, an assignment for the variable from the look-ahead is forced.

If both formulas, that have been derived via the look-ahead, contain an empty clause, we have found a conflict. A conflict indicates, that the currently investigated formula F is unsatisfiable and search must be continued somewhere else in the search tree. This can be achieved by backtracking.

In case no failed literal or conflict is detected, search must be conducted for this variable.

The second aim of look-ahead is to rank the free variables in F (that have not yet received a forced assignment during look-ahead). This ranking represents the ability of a variable to yield a small remaining search tree when branched upon.

In section 3.2.2, we mentioned that it is intuitive to conduct further search with a variable that yields the smallest simplified formula when branched upon (i.e. the smallest remaining search tree). The reason is, that the smaller the remaining search tree is, the faster search will complete on it. To measure this ability for a variable, March_ks uses a look-ahead evaluation function [HvM07b].

The look-ahead evaluation function in March_ks is called `H` and relies on a heuristic called `Diff`. For an explanation of `Diff`, consider an arbitrary formula F and a variable x in F. We perform look-ahead on x (via iterative unit propagation on literals):

$$F\text{`} = \mathrm{UP}(x) \text{ and } F\text{``} = \mathrm{UP}(\neg x).$$

`Diff`(F,F`) now counts the number of clauses in $F$ that got reduced by the unit propagation UP($x$) to compute $F$`. This counting is done in a weighted fashion, such that a clause that became m-ary through the unit propagation will be weighted with $10.33 \cdot 0.44^m$. This value has been evaluated through an empirical study and resulted in the best variable selection behavior [HvM04].

The smaller the resulting clause, the more weight it will receive. We only count clauses that did not get satisfied by the unit propagation. `Diff`(F,F``) is similar. The heuristic `H` now computes

$$H(x) = 1024 \cdot \texttt{Diff}(F, F\text{`}) \cdot \texttt{Diff}(F, F\text{``}) + \texttt{Diff}(F, F\text{`}) + \texttt{Diff}(F, F\text{``}).$$

The higher the `H`-value for a variable, the smaller its remaining search tree is when performing a branch on this variable. The variable with highest `H`-value is therefore used for branching, because it creates the strongest reduction.

Two tasks have been achieved by look-ahead. First, a set of failed literals has been detected and the original formula F has been simplified accordingly. Second, the remaining variables in F have been ranked according to their ability to yield a small search tree. The variable that yields the smallest remaining search tree is used as the new branching variable.

Double Look-ahead

Double look-ahead is an improvement over the single look-ahead and checks whether a formula resulting from a look-ahead is unsatisfiable [HvM07a] (thereby it detects more failed literals). It does so by performing additional unit propagations on a previously reduced formula [HvM06]. To clarify this, consider the following formula:

$$F = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

The double look-ahead on F and x_1 then yields:

- Look-ahead: $\mathrm{UP}(x_1) \Rightarrow F\text{`}_1 = (x_2) \wedge (\neg x_2)$. Then, the double look-ahead on $F$`$_1$ and x_2 gives us:
  - $\mathrm{UP}(x_2) \Rightarrow F\text{`}_2 = ()$.
  - $\mathrm{UP}(\neg x_2) \Rightarrow F\text{``}_2 = ()$.

We have detected, that when $x_1 = 1$, any assignment to x_2 yields an empty clause (i.e. an unsatisfiable formula). Therefore, we know that we must not set $x_1 = 1$. x_1 becomes a failed literal.

The original formula would be immediately reduced by propagating $x_1 = 0$, resulting in a simplified formula $F := F_{\text{simplified}} = (x_2)$. Note, that we did *not* simplify the formula by advancing the search to a new search node (we did not yet decide on what variable to branch on). This simplification is the result of the double look-ahead. In other words, we have not left the current search node, but were able to prune the search path for UP(x_1), by forcing the assignment $x_1 = 0$.

When to use Look-ahead

Overall performance depends on the amount of (double) look-aheads that are performed during search, since these are costly operations [HvZDvM04]. Two opportunities to reduce the overhead on look-aheads exist. First, we can restrict the usage of look-aheads to a pre-selected set of variables. Second, we can restrict the application of double look-aheads.

Restricting look-ahead on certain variables: Let us first investigate the opportunities for a restriction of look-aheads to only a pre-selected set of variables. The optimal behavior of March_ks would be to use look-ahead, iff it can detect a conflict (i.e. a failed literal for the current search node).

Therefore, look-ahead is only performed on variables whenever there is a high probability for its literals to become such a failed literal.

When look-ahead is performed on only a subset $\mathcal{P}$ of the variables in $\mathcal{V}$, only a fraction of computational overhead is needed, but only a subset of failed literals can be detected either [HvZDvM04].

The question therefore is, what variable-subset should be considered for look-ahead and how large should this subset be in a certain node from the search tree.

The assembly of the set $\mathcal{P}$ is performed in a procedure called `LocalPreSelect(`F`)`. This method selects a number of variables from the currently investigated formula, that are to enter the look-ahead phase. Before `LocalPreSelect(`F`)` can be explained in detail, we must clarify the underlying pre-selection strategy.

In [HvM04], a pre-selection heuristic called "ACE" is introduced. ACE has the power to give a ranking of the free variables in a search node. This ranking orders the variables in a priority manner. The higher their priority, the more useful a look-ahead on this variable is.

To get the rank for a variable, one simply computes

$$Rank(x) = ACE(x) \cdot ACE(\neg x), \text{whereas}$$

$$ACE(x) = occ_3(\neg x) + \sum_{Q_i \in \varepsilon(x)} eq_{|Q_i|-1} + \sum_{\neg x \vee y \in F} \left(occ_3(\neg y) + \sum_{Q_i \in \varepsilon(y)} eq_{|Q_i|-1} \right)$$

Hereby, $occ_3(x)$ refers to the number of occurrences of x in ternary clauses. $\varepsilon(x)$ refers to the set of all equivalence clauses in which x occurs (can be requested by the CoE). eq_k defines the relative importance of an equivalence clause of length k. The exact value of eq_k is uninteresting for the context of this work, but is explained in [HvM04].

After the computation of the rank for every variable, we need to choose on how many variables we want to perform a look-ahead on. In other words, we need to decide how many variables should be part of $\mathcal{P}$.

Experiments in [HvM07a] show, that any fixed number for the size of $\mathcal{P}$ is suboptimal (even if it is a percentage of the number of variables in $\mathcal{V}$). Therefore, a new adaptive computation for the size of $\mathcal{P}$ has been proposed for March_dl [HvM06].

It has been observed, that the optimal number of variables in $\mathcal{P}$ is closely related to the frequency of detected failed literals in the preceded search. When more failed literals have been detected in the search nodes before the current search node, a higher number seemed optimal [HvZDvM04].

Let $\#\text{failed}_i$ be the number of detected failed literals in node i. We compute the maximum size of the pre-selected set in node w (denoted $\mathcal{P}^w_{max}$) as follows:

$$\mathcal{P}^w_{max} = \mu + \frac{\gamma}{w} \sum_{i=1}^{w} \#\text{failed}_i$$

Hereby, μ refers to the lower bound of $\mathcal{P}_{max}$ within each node. γ is a parameter modeling the importance of failed literals. Experiments in [HvM06] indicate, that $\mu = 5, \gamma = 7$ resulted in favorable performance on most instances. This adaptive pre-selection strategy was introduced in March_dl and is also used in March_ks. The procedure `LocalPreSelect`(F) embodies this pre-selection strategy and is described as follows.

Filtering the possible branching variables: The `LocalPreSelect(`F`)` method relies on the pre-selection strategy described above. However, before the pre-selection begins, the `LocalPreSelect(`F`)` procedure modifies the current formula. It performs pre-selection on $F_{reduced} = F \backslash F_{initial}$. Hereby, $F_{initial}$ is the original formula (the result of the preprocessor). By $F \backslash F_{initial}$ we denote the removal of clauses from F, that also appear in $F_{initial}$ (i.e. clauses that have not been altered by the search so far get removed from the currently investigated formula).

The reason why the formula is modified before the actual pre-selection is, that it is to support the branching strategy of March_dl, called *local branching*.

Recall that the branching strategy of March_ks (as mentioned in section 3.2.3 on page 43), is to select the variable with the highest `H`-value for the next branch. The variables used for branching only have a high `H`-value in common. In structured instances, this could result in branching variables that are scattered all over different

structures [HvM06]. This in turn increases the chance, that local conflicts must be resolved multiple times (since no conflict clauses are added in a look-ahead solver).

The `LocalPreSelect(`F`)` procedure now counters this, by only adding variables to $\mathcal{P}$ that appear within *reduced clauses* (since it applies `PreSelect` on $F_{reduced}$). Therefore, only variables within a structure that is already being investigated are candidates for $\mathcal{P}$, and thus, for branching.

The procedure for the local pre-selection is outlined in listing 3.4.

Listing 3.4: The LocalPreSelect procedure.

```
LocalPreSelect(F){
    F_reduced = F\F_initial;
    if (F_reduced is an empty formula){
        F_initial = F.
        restart;
    }
    return PreSelect(F_reduced); //returns P
}
```

Empirical studies have been conducted in [HvM06] to verify the usefulness of local branching. It is interesting to note, that on structured instances, this new branching strategy resulted in a dominating performance, while on instances that did not contain a reasonable amount of structure, no performance gains or losses were noticed.

Now after `LocalPreSelect(`F`)` has finished, we have a set $\mathcal{P}$ of pre-selected variables. This set of variables contains all variables that enter the look-ahead phase (and eventually, these variables are used for the double look-ahead as well). We have therefore reduced the amount of computation necessary for the look-ahaed phase by concentrating on the most promising variables.

However, we still have to decide whether a double look-ahead is to follow the first look-ahead on a variable. At this point, we have an additional opportunity to reduce the computational overhead.

When to use Double Look-ahead

Let us now investigate the opportunities to restrict the application of double look-aheads (which refers to looking even deeper ahead in the search tree).

Two strategies exist for deciding whether or not to perform a double look-ahead on the results F‘ and F“ from the first look-ahead on F. Both strategies rely on a parameter called $\triangle_{trigger}$, and both strategies count the number of *newly* created binary clauses, that have emerged during the first look-ahead (i.e. clauses that have been reduced to a size of two literals by the application of look-ahead). We denote this count by $|F\text{‘}_{\text{bin}} \backslash F_{\text{bin}}|$ (or $|F\text{“}_{\text{bin}} \backslash F_{\text{bin}}|$ respectively). The index “bin” refers to the usage of only the binary clauses from the formulas.

A static strategy: The first strategy is a *static check*. The static strategy was used in March_dl [HvM06]. This strategy is called static because it does not change the

$\triangle_{trigger}$ parameter.

To decide if we want to use double look-ahead, we simply check for $|F\text{`}_{\text{bin}} \backslash F_{\text{bin}}| > \triangle_{trigger}$. If and only if this is the case, a double look-ahead is performed on $F$`. The check for F`` is similar.

Whenever the double look-ahead yields the empty clause for F`, the original formula F (from before the first look-ahead) must be satisfiability equivalent to F``[HvM07b]. In other words, double look-ahead has the ability to detect failed literals sooner than single look-ahead.

The threshold $\triangle_{trigger}$ is a constant for this strategy. However, none of the settings for $\triangle_{trigger}$ is optimal for all kinds of instances [HvM07a], and therefore, the static strategy became obsolete in March_ks.

A dynamic strategy: The new strategy for March_ks is an *adaptive check*. The adaptive strategy was introduced in [HvM07a] and relies on three trigger update functions [HvM07a] that modify $\triangle_{trigger}$. $\triangle_{trigger}$ is updated according to the application of the double look-ahead and its result.

If no double look-ahead is performed (i.e. $|F\text{`}_{\text{bin}} \backslash F_{\text{bin}}| > \triangle_{trigger}$ does not hold), `TriggerDecrease()` is called. If a double look-ahead is performed (i.e. $|F\text{`}_{\text{bin}} \backslash F_{\text{bin}}| > \triangle_{trigger}$), `TriggerIncrease()` is called, independently of whether the double look-ahead detects any failed literals.

Lets assume we perform double look-ahead on F`. If the double look-ahead finds $F$` to be unsatisfiable (by performing further unit propagations), `TriggerSuccess()` is called.

The exact effects of the three trigger update functions on $\triangle_{trigger}$ are explained next.

`TriggerDecrease()` is called whenever the double look-ahead is *not* performed. This is because $|F\text{`}_{\text{bin}} \backslash F_{\text{bin}}| > \triangle_{trigger}$ does not hold. To ensure the application of double look-aheads in the future search, a degradation of $\triangle_{trigger}$ is needed. However, the study on how `TriggerDecrease()` should affect $\triangle_{trigger}$ is quite substantial [HvM07a], and therefore, can not be explained here in detail. The final result on how `TriggerDecrease()` acts is as follows:

$$\texttt{TriggerDecrease()} : \triangle_{trigger} = 0.9985 \cdot \triangle_{trigger}$$

`TriggerIncrease()` is always called whenever the double look-ahead is performed (i.e. $|F\text{`}_{\text{bin}} \backslash F_{\text{bin}}| > \triangle_{trigger}$ holds) and does not detect a conflict in $F$` by performing additional unit propagations. When a conflict is detected, the double look-ahead calls for the `TriggerSuccess()` update function and aborts. If no conflict is detected, we call for `TriggerIncrease()`. In [HvM07a], a rather drastical adaption for $\triangle_{trigger}$ is proposed:

$$\texttt{TriggerIncrease()} : \triangle_{trigger} = |F\text{`}_{\text{bin}} \backslash F_{\text{bin}}|$$

The reason to make $\triangle_{trigger}$ equal to the number of newly created binary clauses is as follows. When `TriggerIncrease()` is called, then no conflict was observed. This means that the trigger should have been at least the number of newly created binary clauses, because this setting would have prevented the additional computations for this (useless) double look-ahead. In other words, $|F`_{\text{bin}} \backslash F_{\text{bin}}| > \triangle_{trigger}$ should not have held for this double look-ahead since we were not able to prune the search space. Therefore, we correct the value of $\triangle_{trigger}$ as shown above.

`TriggerSuccess()` is called whenever a conflict is detected during the double look-ahead. `TriggerSuccess()` simply sets $\triangle_{trigger}$ to a value that depends on the number of variables in the original formula F. The effect of calling `TriggerSuccess()` (which is explained in detail in [HvM07a]) is as follows:

$$\texttt{TriggerSuccess()} : \triangle_{trigger} = 0.167 \cdot \#vars.$$

The three trigger update functions together make sure that double look-ahead is applied on pre-selected variables in $\mathcal{P}$ in a nearly optimal fashion. For more information on the functioning of the adaptive double look-ahead in March_ks see [HvM07a].

In summary, the adaptive double-look ahead in March_ks works as outlined in listing 3.5.

Listing 3.5: The AdaptiveDoubleLookahead procedure.

```
AdaptiveDoubleLookahead(F`, F){
    if (|F`_bin \ F_bin| > △_trigger){
        for (variable x_i in P){
            //perform the double look ahead
            F`` = IterativeUnitPropagation(x_i);
            F``` = IterativeUnitPropagation(¬x_i);
            if (F`` and F``` both contain the empty clause){
                //we have detected a failed literal:
                //the one that was used to create F`
                TriggerSuccess();
                return F``;
            }
            else if (F`` contains the empty clause){
                F` = F```;
            } else if (F``` contains the empty clause){
                F` = F``;
            }
        }
        TriggerIncrease();
    } else {
        TriggerDecrease();
    }
    return F`;
}
```

Performing a Branch on a Branching Variable

Whenever the look-ahead phase is finished for all variables in $\mathcal{P}$, we have performed all simplifications that occurred during the look-ahead phase for the current search

node. Thereby, F (the currently investigated formula) has been simplified. Furthermore, the variables from $\mathcal{P}$ are ordered according to the look-ahead evaluation function `H`. We can now use the variable with highest `H`-value for branching. Let us call this variable x.

March_ks now has to decide, what value is to be assigned to the branching variable x first (0 or 1). This decision is made according to a direction heuristic in a procedure called `GetDirection(`x`)`. A direction heuristic selects which boolean value is to be assigned first to a branching variable.

The direction heuristic used in March_ks bases its decision on the reduction caused by assigning a value to the branching variable x. The reduction from F to F‘ (by assigning $x = 0$) and from F to F“ (by assigning $x = 1$) is measured by the number of clauses that are reduced in size without being satisfied [HvM07a].

In general, the probability of a sub-formula to be unsatisfiable is higher, the more its clauses were reduced without being satisfied. Therefore, March_ks choses the direction (F‘ or F“), that yielded a lesser reduction compared to F, and therefore decides what literal (x or $\neg x$) is to be used [HvM07a].

Depending on the returned literal, the branching variable will be assigned different truth values first. In case $\neg x$ is returned, variable x will first be assigned to 0 (i.e. the UP($\neg x$) subtree is examined first). If x is returned, x will first be assigned to 1 (i.e. the UP(x) subtree is examined first).

Backjumping

However, taking into account all the ideas explained above is no guarantee, that the search-tree we investigate actually holds a solution. In case we have maneuvered our search into a dead end of the search tree, we need to perform backtracking. Such a dead end is ultimately detected, when the look-ahead on the currently investigated formula F results in formulas F‘, F“ that both contain the empty clause. In this case, search can not continue from the node we are currently at. We need to backtrack and continue search in a yet unexplored search node. As mentioned before, March_ks does not use chronological backtracking. It performs backjumping. This backjumping is explained next.

The direction heuristic gives valuable information on where to find solutions. Therefore, a study has been performed to evaluate the effectiveness of the direction heuristic [HvM08], and maybe, gain informations about redirecting the search when a conflict occurs. Before we can explain the results of this study, we need to introduce the terms of the *left* and *right branch*.

Definition 10 *Given the direction heuristic of a SAT solver, we call the branch on variable x that is performed at first (assigning to it either 1 or 0 depending on the direction heuristic), the* ***left branch****. The latter branch is called the* ***right branch****.*

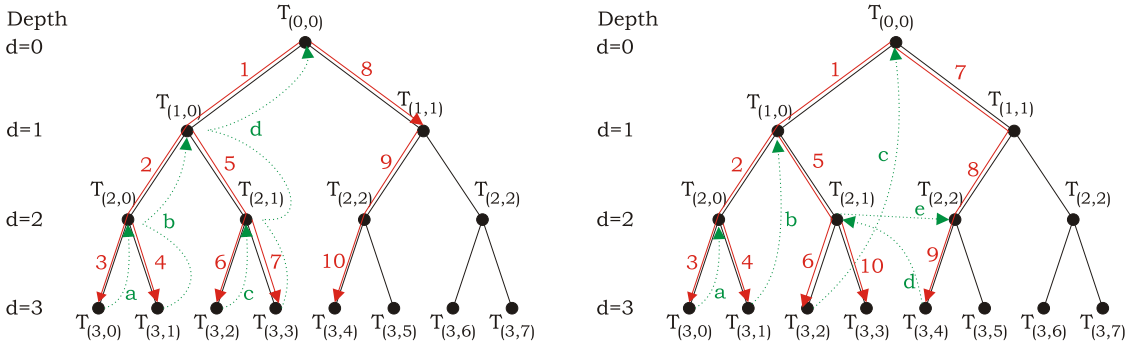


Figure 3.5: The branching trees for March_dl (left) and March_ks (right).

[HvM08] found out, that the solutions of random 3-SAT formulas are not equally distributed in search trees, that are ordered into left and right branches by the March_ks direction heuristic. We refer to these trees as branching trees.

In other words, the direction heuristic used by March_ks, does indeed guide the search into a direction, that is more likely to contain a solution.

The lower the number of right branches, that is needed to reach a node in a branching tree, the higher the probability, that its subtree contains a solution. The authors of [HvM08] concluded, that (in average), one should first check all subtrees that are reachable by the lowest (possible) number of right branches. Unfortunately, this can not be done by a simple backtracking algorithm (like the march versions before March_ks or DLL).

These algorithms perform a depth first search in the tree (see left side of figure 3.5). The depth first search follows the direction heuristic (steps 1,2,3) and as soon as a conflict is discovered, moves upwards the tree until an unexplored sub-tree is found. After step 3, backtracking 'a' is performed. After step 4, backtracking 'b' is performed. Then search continues with steps 5 and 6, resulting in backtrack 'c' and so on. As we can see, the subtrees visited are not ordered by the number of right branches, but by the chronology in which they have been visited.

Distribution Jumping: March_ks now uses a different way of exploring new portions of the search space [HvM08]. It jumps towards positions in the branching tree, that contain an unexplored sub-tree, which can be reached by a minimum number of right branches.

In figure 3.5 (right), steps 1, 2 and 3 follow the direction heuristic, and since all recursive steps can be performed without a right branch, the search continues. After step 3 no more unvisited nodes, that can be reached without a right branch, can be found. Therefore, jump 'a' is performed towards $T_{(2,0)}$. This node contains the nearest unvisited node that can be reached with a single right branch: $T_{(3,1)}$. After step 4 is performed, there are still unvisited nodes left that can be reached with

only a single right branch (for example $T_{(2,1)}$) and jump 'b' continues towards $T_{(1,0)}$. Then steps 5 and 6 are performed and so on. This ordering of nodes guarantees, that March_ks visits the unvisited nodes first, that can be reached by a minimum number of right branches. We refer to this jumping strategy as *distribution jumping*.

The jump calculations are performed by the procedure `Jump()`. In case that *all* nodes of the tree have been visited or pruned away, the `Jump()` procedure aborts the search and returns that the instance is unsatisfiable.

Distribution jumping (performed by `Jump()`) leaves partially unexplored nodes behind. For example, node $T_{(2,2)}$ is partially unexplored when jump 'd' is performed (see figure 3.5 (right)).

Since March_ks spends a lot of time, computing (double) look-aheads on this node, it would be disadvantageous to simply drop the node and forget everything that has been learned about this position in the search tree.

On the other hand, jumps drive the search to various positions within the search tree, and therefore, a lot of memory must be spent in case one wants to keep all information on nodes that are not completely explored yet.

To circumvent this, March_ks only jumps up to a certain depth d_{border} in the search tree. Below this depth, the chronological backtracking is performed [HvM08].

The setting of d_{border} is calculated by the preprocessor (see listing 3.3 on page 41) before the solver actually starts its search. The preprocessor estimates the average depth of the search-tree and then sets d_{border} to seven levels above this average [HvM08]. This results in storing only one out of hundred nodes when the search is performed [HvM08]. In case search is performed below d_{border}, `Jump()` simply initiates chronological backtracking to the previously visited search node.

Summarizing the functioning of March_ks

In summary, a look-ahead solver examines unexplored nodes in the search tree before committing to one of them. Double look-ahead investigates even deeper into the search tree before such a commitment is made.

The formula F from the current search node can then be simplified according to the knowledge received during look-ahead. This either results in a pruning of yet unexplored portions of the binary search tree, or reveals a conflict for the current search node (i.e. the currently investigated formula).

The information gathered during look-ahead is used to compute the rank of free variables. This rank is then used to choose a new branching variable.

Additionally, the direction (i.e. the assignment) we want to first investigate for this variable is calculated via a direction heuristic.

Depending on the literal that is returned by this heuristic, we either follow UP(x)

or UP($\neg x$) towards the next search node.

The overall advancing of the search, as just explained, can be summarized as outlined in listing 3.6.

Listing 3.6: The GetDecisionLiteral procedure.

```
GetDecisionLiteral(F){
    P = LocalPreSelect(F);
    for (variable x_i in P){
        //look-ahead:
        F‘ = IterativeUnitPropagation(x_i);
        F“ = IterativeUnitPropagation(¬x_i);
        //eventually double look ahead:
        if (F‘ does not contain the empty clause){
            F‘ = AdaptiveDoubleLookahead(F‘, F);
        }
        if (F“ does not contain the empty clause){
            F“ = AdaptiveDoubleLookahead(F“, F);
        }
        if (F‘ and F“ both contain the empty clause){
            Jump();
        } else if (F‘ contains the empty clause){
            F = F“;
        } else if (F“ contains the empty clause){
            F = F‘;
        } else {
            H(x) = 1024 · Diff(F,F‘) · Diff(F,F“)
                        + Diff(F,F‘) + Diff(F,F“);
        }
    }
    l_decision = GetDirection(x_i with highest H(x_i));
    return l_decision;
}
```

The `GetDecisionLiteral(`F`)` procedure now accumulates all the ideas explained so far:

- First, we pre-select variables to enter the look-ahead phase (done by procedure `LocalPreSelect(`F`)`).
- After that, we perform look-ahead (and double look-ahead done by procedure `AdaptiveDoubleLookahead(`F‘`, `F`)`) if adequate.
- In case a conflict is found, we continue search somewhere else in the search tree (this will be decided by procedure `Jump()`)
- If no conflict arises, we check whether the investigated variable yielded a failed literal (either F‘ or F“ contains the empty clause) and apply changes to the current formula F in this search node (i.e. simplify F or pruning certain paths in the search tree).
- In case the look-ahead did not yield a failed literal for the current variable, we compute the look-ahead evaluation function (`H-`)value for it.

- Then, we compute a branch direction for the variable with highest H-value (performed by `GetDirection(`x_i`)`).
- Finally, we return the calculated decision literal to the main solving procedure.

The main solving procedure of March_ks is outlined in listing 3.7.

Listing 3.7: The main solving procedure of March ks.

```
MARCHKS(F){
    if (F contains no more clauses){
        return satisfiable;
    } else if (F contains the empty clause){
        Jump();
    }
    l_decision = GetDecisionLiteral(F);
    //left branch
    if (MARCHKS(F with l_decision evaluating to TRUE) == satisfiable){
        return satisfiable;
    } else {
        //right branch
        return MARCHKS(F with l_decision evaluating to FALSE);
    }
}
```

This procedure is quite similar to DLL, since (beside `Jump()`) all the previously discussed features are implemented within the `GetDecisionLiteral(`F`)` procedure.

However, the performance of March_ks also relies on various implementation details like implication arrays, tree-based look-ahead and removal of inactive clauses. Since these features are mere implementation details, they are not explained here. For more information about these details see [HvZDvM04].

The next section will give some explanations for the favorable performance of March_ks on certain formulas.

3.2.4 Explaining the Performance of March_ks

As we have stated in the introduction to this section, March_ks performed quite well during the the SAT 2007 Competition[2].

March_ks was able to solve 257 of 511 random instances. It proved 111 of them to be unsatisfiable and provided correct solutions for 146 of the satisfiable ones. In total, it solved the second most instances from the combined sets of satisfiable and unsatisfiable formulas, and was therefore awarded the second place in the "Random SAT+UNSAT" category. Restricted to only the unsatisfiable instances, no other solver was able to outperform March_ks's 111 solutions. Therefore, it was awarded with the first place in the "Random UNSAT" category.

In the handmade category, March_ks was able to solve 47 of 201 instances. It proved 29 of them to be unsatisfiable and provided correct solutions for 18 of the

[2] http://www.satcompetition.org

satisfiable ones. Therefore, March_ks was awarded the 8th place on the "Handmade SAT+UNSAT" category. Restricted to only the satisfiable handmade instances, March_ks outperformed all other solvers in terms of solving time and was therefore awarded the first place for the "Handmade SAT" category.

March_ks did not participate in solving industrial instances during the SAT 2007 competition, and therefore, no results can be presented for this type of problems.

How Certain Features Affect the Search

We will now explain why March_ks was able to perform well on some of the instance types mentioned above.

The Benefits of Look-ahead

In [Li99], a study is presented that analyzed why a problem of a given size is harder than others, by empirically studying the shape of search trees. The knowledge, that is obtained in this study, suffices in explaining why March_ks performs well on random instances.

Before we can present the results of this study, we need to introduce two terms for binary search trees: tree *width* and tree *mean height*, as given in [Li99].

Definition 11 *The `width` of a search tree level is the number of nodes the search tree has at this level.*

Definition 12 *The `mean height` of a search tree is the sum of all path lengths from the root to a leaf, divided by the number of leaves of that search tree.*

Thereby, a leaf of a search tree is either a search node that contains a solution or a contradiction (in both cases, search can not continue at this search node). Obviously, the *size* of a search tree is determined by its width and mean height [Li99].

The mentioned study uses a set of randomly created 3-SAT problems. The problems are from the hard region (see section 2.7.2 on page 22). Several problem sizes have been used, ranging from 250 to 400 variables in increments of 50. The basic purpose of the study performed in [Li99] was to analyze the mean height and width of a search tree, when search was performed by a DPLL solver called Satz.

The runtime to solve these formulas were then used to split the instances of a certain variable size into two classes. The class of instances that took longer in solving was denoted $\mathcal{H}$ (for hard). The class of instances with shorter solving time was denoted $\mathcal{E}$ (for easy). The names hard and easy are somewhat misleading since all instances are from the hard region where the clauses-to-variables ratio is 4.3. We should consider the names with respect to the used solver, whereas hard means "hard to solve for Satz" and easy means "easy to solve for Satz". With respect to the classes $\mathcal{H}$ and $\mathcal{E}$, two results have been presented and are visualized in figure 3.6 [Li99].

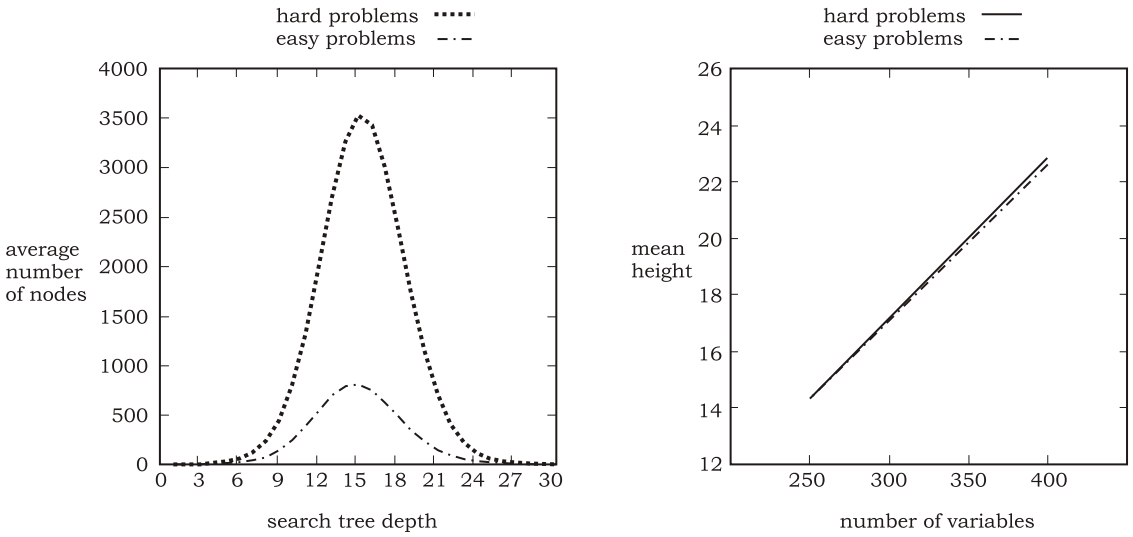


Figure 3.6: Average width of search trees at a certain tree depth (left) and mean hight of search trees (right) for problems with 300 variables.

First (right side of figure 3.6), the mean height of a search tree does not differ much between problems from $\mathcal{H}$ and $\mathcal{E}$. The much longer solving time on instances from $\mathcal{H}$ therefore must be connected to a different property of the search trees. In other words: the mean height of a search tree seems to be irrelevant for the estimation of a problems hardness for Satz.

Second (left side of figure 3.6), the average width of a search tree does differ strongly between problems from $\mathcal{H}$ and $\mathcal{E}$. The average tree width of problem from $\mathcal{H}$ is much larger than that of problems in $\mathcal{E}$.

There was no distinction between sat and unsat problems, since they yield the same shape of trees, and therefore, the same tree properties. [Li99] concluded, that a DPLL solvers priority should be the reduction of the search tree width, instead of the reduction of the search tree height.

Reducing the search tree width is done by detecting leaves of a search tree as soon as possible. As mentioned in the previous section, March_ks employs look-ahead and in certain situations, double look-ahead. The aim of double look-ahead is to detect conflicts in formulas that have been constructed by the single look-ahead (i.e. it tries to detect failed literals). Once it detects such conflicts and the resulting failed literals, it can effectively reduce the size of the search tree since it prunes away subtrees for the corresponding failed literals.

Therefore, March_ks employs a stronger reduction of the tree width than a solver that does not perform double look-ahead. The reduction of the tree width is essential for shorter problem solving times [Li99]. Since March_ks employs double look-ahead, it benefits from these shorter problem solving times, and therefore, has a reasonable performance on random formulas.

The Benefits of Distribution Jumping

The look-ahead feature of March_ks is not the only reason for its dominating performance on random instances. We mentioned a study that lead to the development of the distribution jumping strategy of March_ks [HvM08].

The results in this study clearly showed that the probability of a search node to contain a solution is higher, when it can be reached by a smaller number of right branches.

Distribution jumping, as it is employed by March_ks, guides the search towards unexplored nodes that can be reached by a minimum of right branches. March_ks therefore visits nodes first, that have a higher probability of containing a solution. This in turn reduces the computation time on satisfiable random formulas and adds to March_ks‘s performance on these instances as well.

The Benefits of Local Branching and other Features

However, March_ks did not only perform well on random formulas. It also proved to be competitive on some handmade formulas, namely the satisfiable handmade formulas. Handmade formulas often contain a number of structures.

The main reason why March_ks performs well on these instances is, that it exploits these structures by various techniques.

We explained, that equivalency reasoning is used to construct the CoE (in section 3.2.3 on page 40). This sub-formula is solved separately. Even though we did not explain the functioning of the CoE in detail, it greatly reduces the size of a formula with numerous equivalences, and therefore, reduces March_ks‘s solving time on these instances.

Additionally, we described the branching strategy of local branching that is employed by March_ks. We mentioned, that local branching focuses on selecting a branching variable from a structure, that is currently investigated. This prevents branches on variables that are scattered all over the formula, and thereby obviates the repeated detection of local conflicts. This in turn reduces computation time on instances that have multiple structures to be investigated.

March_ks did not participate in the industrial category during the SAT 2007 Competition. Therefore, an analysis of the behavior on these instances is skipped.

A summary of the most important features of March_ks will be given in the next section.

3.2.5 Summary of the Review of March_ks

March_ks is a look-ahead DPLL complete SAT solver, that proved to be very competitive during the SAT 2007 Competition.

It employs a preprocessor to simplify a given formula before it starts its search for a solution. The preprocessor applies unit propagation, a root look-ahead, and equivalency reasoning (manifested in the CoE). Additionally, the preprocessor adds constraint resolvents and ternary resolvents to guide the search.

March_ks utilizes the double look-ahead principle to effectively prune the search space while search is performed. To control the computational overhead, that is connected with look-aheads, it restricts the number of variables that enter the look-ahead phase by the use of a pre-selection heuristic.

Furthermore, it applies an adaptive strategy to decide for when to conduct a double look-ahead. To raise performance on satisfiable instances, it performs distribution jumping, local branching and employs a direction heuristic to guide the search.

We have now finished our review of the March_ks solver and will continue with a review of another DPLL solver called RSat.

3.3 RSat

3.3.1 RSat Background

RSat was developed at the University of California in Los Angeles. It was the winner of the SAT 2007 Competition[3] in the "Industrial SAT+UNSAT" and "Industrial UNSAT" category and became second in the "Industrial SAT" category. Therefore, RSat is one of todays state-of-the-art SAT solvers.

In this review, we will concentrate on RSat version 2.02, which is the version submitted to the SAT 2007 Competition with some minor bugfixes. However, we will use the functioning of RSat 2.02 as a leitmotif for the explanation of features not only part of this version, but also of features that have been presented for earlier versions of RSat. Thereby, we aim for a complete feature description of all versions up to 2.02.

The remainder of this section is structured as follows. The next section will give an overview of the functioning of RSat. Since RSat is a DPLL solver, we will use the basic DLL algorithm as explained in section 3.1.1 and point out the major improvements introduced in RSat. After the general ideas in RSat have been given, we will continue by explaining the detailed functioning of RSat and its various features. After the detailed description of RSat has been given, we take a more detailed look at the performance of RSat during the SAT 2007 Competition. The section is concluded by a summary of this review of RSat.

[3] http://www.satcompetition.org

3.3.2 Ideas used in RSat

Advancing the Search

Like any DPLL solver, RSat explores a binary search tree to find a model for a given formula F. In order to do so, RSat performs various actions like:

- decide, what variable is used for branching (via a variable selection heuristic),
- assign a boolean value to it (via a direction heuristic),
- backtrack, if our search leads to a dead end in the search tree (i.e. a conflict is detected for the decisions made so far).

When the basic DLL algorithm performs search on F, it will implicitly remember its decisions via the construction of the binary search tree through its recursive calls (see listing 3.2 on page 32).

RSat, however, does not perform recursive calls. Instead, RSat explicitly remembers the decisions it makes in a so called decision sequence, which is an ordered list.

This ordered list contains the assignments to branching variables in chronological order (i.e. in the order the decisions have been made), whereas the last element in that list is the newest one. Let us denote this decision sequence D.

The made decisions in D are remembered as literals, whereas the decision $x_i = 0$ is saved as $\neg x_i$ in D and $x_i = 1$ is saved as x_i in D. Each decision saved in D has a decision level, which identifies its position in D. Take a look at the following example.

Example 14 *Given a sufficiently large formula F. Let us assume, we made the following decisions: first $x_1 = 0$, second $x_2 = 0$, third $x_3 = 0$.*

The resulting decision sequence would be $D_3 = ((\neg x_1), (\neg x_2), (\neg x_3))$. The index 3 refers to the length of D, i.e. how many decisions are currently saved by it.

However, RSat performs unit propagation after each decision. In case a decision yields further implications via this unit propagation, we save them in the same decision level, right after the decision.

For example, let us assume the decision in decision level 3 ($x_3 = 0$), results in the implications $x_4 = 1, x_5 = 1$, then the decision sequence would be

$$D_3 = ((\neg x_1), (\neg x_2), (\neg x_3, \rightarrow x_4, \rightarrow x_5)),$$

whereas the "$\rightarrow x_i$" means, that the value for x_i is implied. Note how the index 3 did not change, since the number of made decisions is still 3. The other assignments saved in D_3 are mere consequences from these decisions.

Decision level 0 is somewhat special. When unit reduction can be applied to the original formula F, then these assignments are forced independently of any decisions.

Say we have a forced assignment $x_6 = 1$ in decision level 0. This would give us $D_3 = ((\rightarrow x_6), (\neg x_1), (\neg x_2), (\neg x_3, \rightarrow x_4, \rightarrow x_5))$.

In case there are implications in decision level 0, we can simplify the original formula F by forcing these assignments. Search must be conducted for only the resulting (simplified) formula.

D_i can be seen as a partial assignment, which RSat tries to expand to a model of the currently investigated formula.

Expanding the decision sequence is done via the selection of a branching variable and the assignment of a value to it, yielding a decision $x_i = v$ ($v \in \{0, 1\}$).

Once a decision has been made, it will be appended at the end of D_i. As a result, we get D_{i+1}, the modified version. To continue the search, RSat applies this decision sequence on the formula, yielding $F_{D_{i+1}} = D_{i+1}(F)$.

Applying the decision sequence to the currently investigated formula is the process of removing falsified literals and satisfied clauses from F under D_{i+1}. Since D_{i+1} holds more assignments than D_i, the formula $F_{D_{i+1}}$ will contain less literals and/or clauses than $F_{D_i} = D_i(F)$.

After a number of expansions to the decision sequence (i.e. after k decisions), it either suffices in solving F (in case $D_k(F)$ is the empty formula), or results in a conflict (in case $D_k(F)$ contains the empty clause). If D_k suffices in solving F, we have found a model and the search is finished.

Running into Conflicts

If $D_k(F)$ results in a conflict, at least one of our decisions in D_k was erroneous and must be undone. Undoing is performed by removing one (or more) decisions from the end of D_k (including their implications). In other words, if $D_k(F)$ resulted in a conflict, we move back to $D_{k-l}(l \in [1, k])$, since $D_{k-l}(F)$ did not yet result in a conflict. This undoing of lately made decisions is referred to as backtracking.

The question now is, what hinders us in making the same mistake again (i.e. again making a (set of) decision(s) leading to the previously explored D_k). Clearly, RSat has to learn something from the conflict in $D_k(F)$, in order to avoid it in the future.

For now, let us simply assume, that this knowledge can be saved in a set Γ, representing everything we have learned so far on how to avoid conflicts. After backtracking to D_{k-l}, a decision on $D_{k-l}(F \wedge \Gamma)$ would then result in a different decision sequence $D_k\text{`} \neq D_k$, since Γ has just been modified.

When a new (sequence of) decision(s) again leads to a conflict in $D_k\text{`}(F)$, Γ is extended further. Thereby, we gather more knowledge on how certain conflicts can be avoided. Because the search of RSat is strongly influenced by the conflicts it discovers and the learned knowledge on how to avoid these conflicts, RSat is called

a *conflict-driven* SAT solver.

Two things can happen when expanding our knowledge Γ. First, the knowledge will eventually enable us to avoid all conflicts when searching for a model, yielding a decision sequence that suffices in solving F. Second, the knowledge in Γ makes us backtrack to decision level 0, and without any decisions, $F \wedge \Gamma$ suffices in the identification of a conflict by only the application of iterative unit propagation. When this happens, the formula F must be unsatisfiable. In other words, no matter what decisions we make, we will always run into a conflict supported by Γ, before D suffices in solving F.

Let us use the following example to clarify the behavior of RSat mentioned above.

Example 15 *Given the formula*
$F = \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{1} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{2} \wedge \underbrace{(\neg x_1 \vee x_2 \vee \neg x_3)}_{3} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{4} \wedge$
$\underbrace{(x_1 \vee x_2 \vee x_3)}_{5} \wedge \underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{6}$
This formula has exactly two models: $A = (1, 1, 1)$ *and* $B = (0, 1, 1)$*. Additionally, the decision sequence* D *and the set of learned knowledge* Γ *are initialized to be empty at the beginning.*

A binary search tree for the chronological decisions on x_1, x_2 and then x_3 for F is given in figure 3.7. Note, that this tree is merely used for a visualization of the search-space. RSat itself does not construct this tree since it does not perform recursive calls like DLL. However, the structured character of the search is retained through the application of F, Γ and D, thereby making RSat a complete SAT solver. How this application looks like will be explained via an exemplary search on F as follows.

When RSat starts its search, it will first check for any unit clauses to perform unit propagation in decision level 0. Since no unit reduction is possible, it has to decide for a branching variable now.

RSat uses a simple literal count heuristic. It counts the occurrence of all literals, and selects the variable for branching, for which a corresponding literal has the highest count. In case of our example, all literals occur exactly three times. For such a constellation, RSat selects one of the variables with highest literal count at random.

The reason for RSat to select the variable with highest literal count is, that selecting such a variable and adding a value to it will result in a minimal remaining formula after this decision has been propagated. This in turn reduces the expected remaining work in order to find a model. Say we select x_1 here.

Now RSat needs to assign a value to the branching variable via a direction heuristic. Now it seems somewhat strange on the first glance, but RSat will always assign 0 to a branching variable. The assignment of value 1 can only happen during iterative

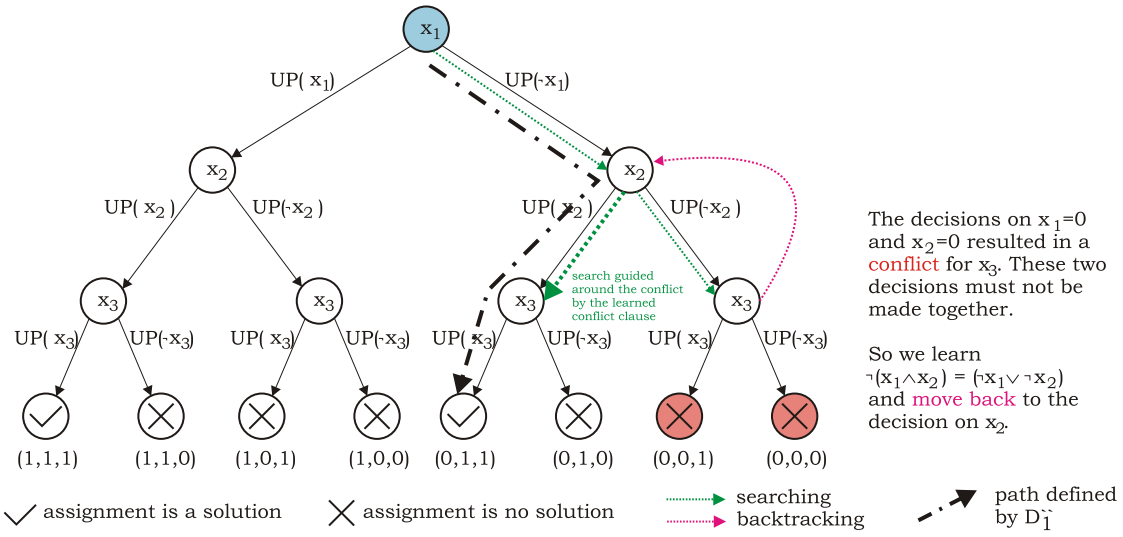


Figure 3.7: The search tree for F, when deciding first on x_1, then x_2 and then x_3.

unit propagation (i.e. as an implication).

However, after deciding on $x_1 = 0$, we will get $D_1 = ((\neg x_1))$. We apply D_1 on F (and F alone since Γ is empty) to see if this yields any further implications:

$$F_{D_1} = D_1(F) = (\neg x_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3).$$

We removed the clauses 1, 2 and 3 from F, since they got satisfied now. Furthermore we removed the literal x_1, since this literal now evaluates to FALSE.

Because no unit clauses appear in F_{D_1}, no further implications can be derived. It is time to choose a new branching variable.

Since the literal count for all remaining literals in F_{D_1} is two, we again select a variable at random. Say x_2 is chosen and since RSat always assigns 0 first, we add $x_2 = 0$ to our decision sequence. We get $D_2 = ((\neg x_1), (\neg x_2))$. We apply this decision sequence on F (and F alone since Γ is still empty) to see whether there are any further implications entailed.

$$F_{D_2} = D_2(F) = (x_3) \wedge (\neg x_3)$$

When now comparing F_{D_1} and F_{D_2}, we see that the application of D_2 on F acts like the removal of literal x_2 from F_{D_1}, since it evaluates to FALSE now. In other words, expanding D_1 is similar to further reduce the formula F_{D_1}, even though we applied D_2 on F.

Now two unit clauses remain from the original formula. Since unit clauses force implications, we try to propagate these, which results in a conflict on x_3.

Learning from Conflicts and Revising Decisions

At this point, we need to find out which of our decisions are the preconditions for this conflict to arise. Therefore, we analyze the *conflicting clauses*. Conflicting clauses are those, that participate in a conflict.

In our case, the clauses (x_3) and $(\neg x_3)$ where derived from the clauses 5 $(x_1 \vee x_2 \vee x_3)$, and 6 $(x_1 \vee x_2 \vee \neg x_3)$ in F.

In order to avoid the conflict on x_3 in the future search, we need to use these clauses to learn what decisions lead to the conflict. Taking a closer look at these clauses reveals, that both assignments $x_1 = 0$ and $x_2 = 0$ are neccesary preconditions for the occurrence of the conflict on x_3. We therefore learn, that we must not set $x_1 = 0$ and $x_2 = 0$ at the same time. More formal, we learn that

$$\neg(\neg x_1 \wedge \neg x_2) = (x_1 \vee x_2)$$

must be accomplished in order to avoid the conflict.

We therefore add this clause to Γ. After learning this so-called *conflict clause* (using the conflict**ing** clauses), we need to undo at least one decision in D_2 (i.e. perform backtracking), since we have reached a dead end in the search tree.

The question now is, what decision is to be undone. Clearly, undoing all decisions would suffice in resolving the conflict at hand, but this would put us back to the beginning of the search and might undo decisions that are no preconditions for this conflict. It would be better, if we could just undo a minimum number of decisions, since this would not put us back too far, holding us closer to a possible solution.

Recall that D is chronologically ordered, with the newer decisions being at the end of D. We now check the variables in the just learned conflict clause $(x_1 \vee x_2)$, and search for the variable with largest decision level (i.e. the variable that is next to the end of D). The decision level of x_1 is 1, the decision level for x_2 is 2. So x_2 is closer to the end of D_2.

We now want to backtrack in a way, that undoes this decision, but leaves all other decisions the way they are. In other words, we must backtrack to a predecessor of x_2 without changing this predecessor. The next predecessor of x_2, that also is a precondition for the conflict, is x_1. We therefore decide to backtrack to decision level 1 and undo $x_2 = 0$. The decision on x_1 is left unchanged.

Backtracking to this level retains the most decisions and suffices in undoing exactly one necessary precondition for the conflict.

After backtracking to level 1, we have $D_1 = ((\neg x_1))$. Recall, that Γ was changed, by adding the conflict clause $(x_1 \vee x_2)$, so Γ is not empty anymore. Therefore, a decision for a branching variable is now done on F and Γ, i.e. we decide on

$$F \wedge \Gamma = \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{1} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{2} \wedge \underbrace{(\neg x_1 \vee x_2 \vee \neg x_3)}_{3} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{4} \wedge$$

$$\underbrace{(x_1 \vee x_2 \vee x_3)}_{5} \wedge \underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{6} \wedge \underbrace{(x_1 \vee x_2)}_{7}$$

With clause number 7 being the learned conflict clause from Γ. At the current decision level we have $D_1 = ((\neg x_1))$, and therefore, use the formula

$$D_1(F \wedge \Gamma) = (\neg x_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_2).$$

for further search. Note how the conflict clause becomes a unit clause for the decision level we just backtracked to. This unit clause now implies an assignment for x_2 that helps us to avoid the conflict on x_3. Since we perform iterative unit propagation beforehand of a decision, we set $x_2 = 1$. This implication is a direct consequence of $x_1 = 0$ and the learned conflict clause, so $x_2 = 1$ is an implication of decision level 1 now. Our new decision sequence is $D\text{`}_1 = ((\neg x_1, \rightarrow x_2))$.

A clause that implied a variable assignment is called a *reason*. Here, the learned conflict clause becomes a reason for assigning $x_2 = 1$. As we can see, the learned conflict clause is used to prune away the previously explored part of the search tree (see figure 3.7).

This effect of conflict clauses is mainly the reason why RSat performs a structured and complete search, even though it does not explicitely create the structure of a binary search tree. Since unit propagation is performed in an iterative way (i.e. we perform unit propagation until no further changes happen), we again perform unit propagation on F under $D\text{`}_1$, which has the form

$$D\text{`}_1(F) = (x_3).$$

The only remainder left after the application of $D\text{`}_1$ is clause 4 from F. Since this is a unit clause as well, we force $x_3 = 1$, resulting in $D\text{“}_1 = ((\neg x_1, \rightarrow x_2, \rightarrow x_3))$. Checking the resulting formula $D\text{“}_1(F)$ reveals, that all clauses are now removed. This means, that $D\text{“}_1$ holds an assignment that suffices in solving F.

As already mentioned, RSat does not perform recursive calls when altering Γ and D. Therefore, RSat is not aware of the shape of the search-space or its position in it. It only uses its decision sequence D and the learned conflict clauses in Γ.

However, the decision sequence actually defines a path in such a binary search tree, even though the tree is never constructed. Refer to figure 3.7 for a visualization of our solution $D\text{“}_1 = ((\neg x_1, \rightarrow x_2, \rightarrow x_3))$.

RSat does not create the binary search tree, and can therefore be no backtracking algorithm like DLL. The term “backtracking” in RSat is used in the broader sense, meaning that we undo decisions, which is in total contrast to the way it is understood in DLL, where it means to literally go upwards in the search tree. Since we do not have a search tree here, we do not backtrack in the narrower sense.

The question now is, on how the basic framework for RSat looks like if it is not coherent with the DLL procedure. This brings us to the *iterative SAT procedure* from [PD08], which abstractly describes the functioning of RSat.

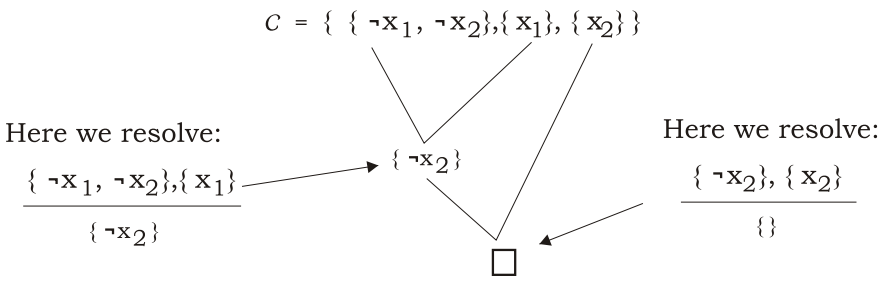


Figure 3.8: Using unit resolution to derive the empty clause.

The Iterative SAT Procedure

Before we can explain this procedure, we need to introduce some definitions and notations from [PD08]. These definitions will now formally introduce the terms we used above.

Let F be a propositional formula in CNF, let α be a clause, and let x_i be a literal. With $F \models \alpha$, we denote that α is entailed by F (meaning that α follows logically from F).

With $F \vdash x_i$, we denote that x_i can be derived from F using *unit resolution*. Unit resolution is resolution with at least one parent being a unit clause.

It is important to understand the similarities between unit resolution and unit propagation. The iterated application of unit resolution suffices in detecting a conflict, iff the same conflict can be detected by iterative unit propagation. To clarify this, consider the following example.

Example 16 *Given the formula $F = (\neg x_1 \vee \neg x_2) \wedge (x_1) \wedge (x_2)$. Using unit resolution will reveal this formula to be unsatisfiable (see figure 3.8), since we can deduce the empty clause. Likewise, using unit propagation on x_1 gives us first $F' = (\neg x_2) \wedge (x_2)$. Then, unit reduction on x_2 gives us $F'' = ()$, a formula containing the empty clause.*

We will continue to refer to Γ as the set of learned conflict clauses. But from now on, we will see Γ as a conjunction of these clauses. It is possible to see Γ as another CNF formula, that holds all the knowledge that has been learned from conflicts. Let us now give a more formal definition of the decision sequence.

Definition 13 *A `decision sequence` is an ordered set of literals $D = (x_1, \ldots, x_w)$, wherein a literal x_k is called the `decision` at level k. We write D_m (with $m < w$) to denote the `subsequence` $(x_1, \ldots, x_m)$. When appropriate, we will interpret D as a conjunction of literals (i.e. as a conjunction of unit clauses $x_1 \wedge \cdots \wedge x_w$) that are included in D.*

When we first introduced the term of a decision sequence, we enabled it to hold implications on variable assignments as well. However, this concession was merely

made in order to simplify the first example on how RSat performs search.

As we can see from the definition above, the decision sequence does not directly contain implications. It is simply an ordered list of literals representing the decisions we made so far. Nevertheless, detecting implications is an important part of RSat. In order to properly introduce how implications are saved, we need another term: the SAT state.

Definition 14 *A* `SAT state` *is a tuple* $S = (F, \Gamma, D)$*, with* $F \models \Gamma$*. To denote the SAT state at a certain decision level, we write* $S_k = (F, \Gamma, D_k)$*, with* D_k *being a subsequence of* D*.*

This notation simply models the progress of the search, whereas the formula F is being searched, Γ is the set of all conflict clauses that have been learned during that search, and D is the set of decisions that has been constructed so far. Now implications can be derived from a SAT state by simply using unit resolution (or iterative unit propagation likewise). Therefore, we write the components of the SAT state as a conjunction $F \wedge \Gamma \wedge D$. The term $F \wedge \Gamma \wedge D \vdash x_i$ then means, that we can derive the unit clause (x_i) using unit resolution (which can be seen as deriving $x_i = 1$ using unit propagation likewise). More formally:

Definition 15 *A literal* x_i *is* `implied` *by state* S_n *(written* $S \vdash_n x_i$*), iff* n *is the smallest natural number for which* $F \wedge \Gamma \wedge D_n \vdash x_i$*.* n *is called the* `implication level` *of* x_i*.*

When we write $S \vdash x_i$*, we mean* $S \vdash_k x_i$ *for some* k*. With* $S \nvdash x_i$*, we mean* $S \nvdash_k x_i$ *for all* k*.*

Furthermore, using unit resolution can uncover conflicts within the current SAT state.

Definition 16 *A state* $S = (F, \Gamma, D)$ *is called* `1-inconsistent`*, if and only if* $F \wedge \Gamma \wedge D \vdash$ FALSE*.*

In other words, the state is 1-inconsistent if the empty clause can be derived from the conjunction of its participants when using only unit resolution. In case a SAT state does not contain a conflict, we call it normal. More formally:

Definition 17 *A state* $S = (F, \Gamma, (x_1, \ldots, x_w))$ *is called* `normal`*, iff, for* $0 \leq k \leq w$*,* S_k *is not 1-inconsistent.*

In other words, a state S_n is normal when no sub-state S_k (for all $k \leq n$) of S_n is 1-inconsistent. Therefore, for all states S_k and any literal we know that $S_k \nvdash x_i$ AND $S_k \nvdash \neg x_i$ is impossible, meaning that no conflict can be derived using unit resolution in all previous states of S_n.

We have introduced the term of a conflict clause in an informal way already, stating that a conflict clause represents a condition that must be supported in order to avoid a certain conflict (i.e. the conflict clause must evaluate to TRUE). We will now give a more formal definition for conflict clauses.

Definition 18 *Let $S = (F, \Gamma, D)$ be a SAT state. A clause $\alpha = (x_i \vee \cdots \vee x_m)$ is a* `conflict clause` *for state S, iff*

- $F \wedge \Gamma \wedge \neg\alpha \vdash$ FALSE. *That means, that α is implied by $F \wedge \Gamma$ and this can be proven using unit resolution.*
- *For each x_i, $S \vdash \neg x_i$. That means, that the literals $\neg x_i$ in α are a subset of the implications derived by unit resolution in state S.*

Falsifying all literals in α will therefore result in a conflict. Let D be a decision sequence, that results in the discovering of a new conflict. Let $m = |D|$ be the last decision level (in which the conflict occurred).

Conflict analysis is then used to derive a conflict clause α in order to avoid this conflict in the future. The learned conflict clause α will contain at least one literal, that is falsified in decision level m, since this was the first decision level, that yielded this conflict under unit resolution.

Furthermore, the learned conflict clause will become empty under the decisions in D. A conflict clause in which exactly one literal is falsified in the last decision level of D is called an *asserting clause*. More formally:

Definition 19 *A conflict clause α of a state $S = (F, \Gamma, D)$ is an* `asserting clause`*, iff it has exactly one literal x_i with implication level $|D|$. The literal x_i is called the* `asserted literal` *of α. The* `assertion level` *of α is the highest implication level $k < |D|$ attained by some literal in α. If α consists of only one literal, than α is a unit clause. In case α is a unit clause, its assertion level is defined to be zero.*

To understand the difference between an arbitrary conflict clause and an asserting clause, take a look at the following example.

Example 17 *Given the formula*

$$F = \underbrace{(x_1 \vee x_2)}_{1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{2} \wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)}_{3} \wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4)}_{4}$$

and the decision $x_1 = 0$. With $D_1 = ((\neg x_1))$, we get

$$D_1(F) = \underbrace{(x_2)}_{1} \wedge \underbrace{(\neg x_2 \vee x_3)}_{2} \wedge \underbrace{(\neg x_2 \vee \neg x_3 \vee x_4)}_{3} \wedge \underbrace{(\neg x_2 \vee \neg x_3 \vee \neg x_4)}_{4}.$$

Clause 1 now implies $x_2 = 1$. Propagating this gives us clause 2 as another unit clause, forcing $x_3 = 1$. After propagating this, we have two unit clauses left: (x_4), derived from clause 3, and $(\neg x_4)$, derived from clause 4.

With this conflict given on x_4, we have to analyze the conflicting clauses 3: $(\neg x_2 \vee \neg x_3 \vee x_4)$ and 4: $(\neg x_2 \vee \neg x_3 \vee \neg x_4)$. We now have to decide on what we want to learn from this conflict.

Learning an arbitrary conflict clause: *Let us first learn an arbitrary conflict clause. The conflict on x_4 arose, because we set $x_1 = 0, x_2 = 1$ and $x_3 = 1$, so we learn*

$$\neg(\neg x_1 \wedge x_2 \wedge x_3) = (x_1 \vee \neg x_2 \vee \neg x_3) = \gamma$$

as the new conflict clause, and add it to Γ.

For backtracking, we analyze the conflict clause and check which of the included literals has the highest decision level. However, since x_2 and x_3 where implied by $x_1 = 0$, they all have the same decision level: 1.

When we backtrack to this level, we get the formula

$$F \wedge \Gamma = \underbrace{(x_1 \vee x_2)}_{1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{2} \wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)}_{3} \wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4)}_{4}$$
$$\wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}_{\gamma}$$

We can again set $x_1 = 0$, since no assignment to x_1 is forced yet. This will again imply $x_2 = 1$ (because of clause 1). Because of the newly learned conflict clause, the assignment $x_3 = 0$ is forced.

At this point we can see, that multiple literals in γ where falsified in this decision level (the level of the conflict). Therefore, γ is not an asserting clause.

However, under $x_1 = 0$ and $x_2 = 1$, clause 2 implies $x_3 = 1$ as well. So we now ran into another conflict. The learning of γ prevented us from running into the conflict on x_4 again, and thus saved computation time.

At this point we would have to start conflict analysis again. Anyway, let us see what would have happened, if we had learned an asserting clause γ_a instead.

Learning an asserting clause: *Again we use the original F and the decision $x_1 = 0$ as a start. This again implies $x_2 = 1$ and $x_3 = 1$, leaving us with the previously mentioned conflict on x_4. We again use the conflicting clauses 3 and 4 for the conflict analysis and again see, that we must not set $x_1 = 0, x_2 = 1$ and $x_3 = 1$.*

However, we can detect, that $x_2 = 1$ and $x_3 = 1$ are merely consequences from setting $x_1 = 0$, so we learn that

$$\neg(\neg x_1) = (x_1) = \gamma_a$$

We again backtrack to level 1, since this is the decision level of the only left literal in this conflict clause. After backtracking to this level, we have the formula

$$F \wedge \Gamma = \underbrace{(x_1 \vee x_2)}_{1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{2} \wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)}_{3} \wedge \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4)}_{4} \wedge \underbrace{(x_1)}_{\gamma_a}$$

As we can see, at the decision level of the conflict on x_4, only one literal is left within γ_a, and thus, only one literal could be falsified here. This in turn means, that γ_a is an asserting clause.

However, since we perform unit propagation beforehand of a decision, we immediately set $x_1 = 1$. This already removes all the clauses from the formula (i.e. the partial assignment $x_1 = 1$ is a model for F).

The benefit of learning the asserting clause $\gamma_a = (x_1)$ was its ability to empower unit resolution (i.e. the performed unit propagation) to evade the conflict on x_4 and, which is even more important, all other conflicts that had $x_1 = 0$ as a precondition.

The difference between asserting clauses and arbitrary conflict clauses is therefore the number of literals that become falsified in the decision level of the conflict. For arbitrary conflict clauses, this can be an arbitrary number of literals. For asserting clauses, it is exactly one literal.

Most modern SAT solvers following the iterative SAT procedure try to learn asserting clauses from a conflict. Now the question arises, what the benefit is when learning only asserting clauses.

We have introduced the term of 1-inconsistency. As we can see from the definition, a SAT state is called 1-inconsistent, when $F \wedge \Gamma \wedge D \vdash$ FALSE, meaning that under unit resolution, the empty clause can be derived.

As can be read in [PD08], unit resolution is incomplete, so it can not derive all possible conflicts for a given SAT state (in contrast to general resolution from section 2.3.2 on page 11). When adding conflict clauses to Γ, more candidate clauses for resolution are introduced, and in theory, more ways in finding conflicts become available.

The problem is, that unit resolution is restricted to the use of at least one clause that is unit. So for unit resolution, a learned conflict clause is only of direct use to derive more conflicts, if it is a unit clause. This is the case for an asserting clause at its assertion level. In contrast, this is not the case for arbitrary conflict clauses. One can think of asserting clauses as a tool to make unit resolution more complete, i.e. empower it to detect more conflicts [PD08]. Thereby, search is guided "around" already detected conflicts by the application of (asserting clause empowered) unit resolution.

Now all necessary terms have been introduced to present the functioning of the iterative SAT procedure in detail. This procedure initializes D and Γ to be empty sets. Then an iteration is started.

The start of an iteration is merely the check if the current SAT state, defined by the searched formula F, decision sequence D, and learned conflict clauses Γ, is *normal* or *1-inconsistent*.

If the state is 1-inconsistent, the iterative SAT procedure performs the following tasks:

- Check if the decision sequence is empty. If this is the case, than a conflict arose (was found using unit resolution) without any decisions. If so, the formula must be unsatisfiable.
- If the decision sequence contains some decisions, we need to analyze the conflict. The result of this analysis is an asserting clause α. This asserting clause then empowers the unit resolution in the future to detect the conflict it corresponds to. As a result, certain variable decisions will not be made in the future search again.
- With that α given, we calculate the assertion level of α by examining all its literals. The literal with the largest decision level defines the assertion level of this clause.
- With the calculated assertion level, we erase all decisions that have been made after this level in the decision sequence D. This decision sequence reduction is often referred to as backtracking even though the algorithm does not perform backtracks. This understanding of the reduction of D arises from the analogon of viewing the decision sequence as a path in the search tree. When erasing the last decisions, the path shrinks back towards a higher level in the search tree. This is usually what happens when a backtracking algorithm backtracks (like the basic DLL procedure from section 3.1.1 on page 28).
- When the decisions after the assertion level have been erased, we add the newly learned asserting clause to Γ. Now at this point, the newly inserted asserting clause becomes a unit clause, because we have revised all decisions but the last one that made the literals in this conflict clause false. This is true because we revised all decisions up to the assertion level of that clause. Therefore, the asserting clause implies a truth value for its last variable, guiding the search away from the conflict it corresponds to.
- After that, a new iterative step is started.

If the SAT state is found to be normal, the iterative SAT procedure performs the following tasks:

- We choose a new variable for a decision. By deciding on a literal corresponding to this variable, we decide, whether we set this variable to TRUE or FALSE. If no such variable exists, then no further decisions can be made (all variable values have either been set in D or are implied by these decisions). Since the SAT

state is normal, a (partial) assignment D has been found that, along with its implications on F and Γ, suffices in solving F. If this is the case, we can return the solution for F and exit.

- When a literal was chosen, we add this decision to D by appending it. Therefore, we enter a new decision level.
- After that, a new iterative step is started.

The procedure described above is sketched out in listing 3.8.

Listing 3.8: The iterative SAT procedure.

```
IterativeSAT(F){
    D = ();
    Γ = {};
    while(true){
        if (S = (F,Γ,D) is 1−inconsistent){
            //conflict because of the decisions in D and
            //their implications
            if (D is empty){
                return unsatisfiable;
            }
            α = an asserting clause for S;
            m the assertion level of α;
            //here we undo the decisions up to the assertion level
            //only the first m decisions are retained
            D = D_m;
            Γ = Γ ∪ α;
        } else {
            //no conflict, the state of S is normal
            //we need to make a new decision in order
            //to advance the search
            //find a new decision literal so that
            //S ⊭ l and S ⊭ ¬l.
            l = a new decision literal;
            if (no such l exists){
                //no more decisions are needed
                return satisfiable;
            }
            //we need to remember the decision
            D = D with l appended at the end;
        }
    }
}
```

Even though the algorithm iterates with an infinite loop, it must terminate. In order to understand that, one has to understand that asserting clauses learned by the algorithm never repeat. Learning an asserting clause twice would only be possible, if the same conflict could be reached twice by the same decision sequence. Since an asserting clause empowers unit resolution to find this specific conflict, the variable decisions (that are made while search advances) will not lead to the same conflict again. In other words, just before the conflict arises, an asserting clause implies an assignment for its asserting literal and thereby guides the search around this conflict.

Additionally, the following fact is given in [PD08]: A SAT state $S = (F, \Gamma, D)$, with $|D| > 0$ is 1-inconsistent, iff it has an asserting clause. With this fact given, it is clear that every conflict comes with an asserting clause. Since F only contains finitely many clauses, it can only have finitely many conflicts.

The iterative SAT procedure will eventually learn all asserting clauses connected with all conflicts, but since asserting clauses do not repeat and only a finite number of conflicts is available, the algorithm must terminate. Either it terminates, because it can not perform any more variable decisions due to unit resolution detecting conflicts for all assignments [PD08]. Or it terminates, because the expansion of D can be performed towards a solution without running into a conflict due to (asserting clause empowered) unit resolution implying all necessary assignments to evade these conflicts.

We have now explained the basic idea behind RSat and will continue to explain its functioning in detail.

3.3.3 The functioning of RSat

In order to explain the functioning of RSat in detail, we will follow the scheme of the iterative SAT procedure from the previous section. We will start by explaining the necessary operations for RSat in order to advance the search, by explaining:

- The preprocessor of RSat.
- The branching decision of RSat, that selects a new variable to continue search, and how iterative unit propagation is performed in order to check on the effects of such a branch.
- The conflict learning scheme of RSat. Therefore, we will explain the FirstUIP learning scheme used by RSat in more detail.
- The backtracking procedure of RSat, that is used to revise previous decisions. Therefore, we will explain the progress saving principle that RSat employs.

After the necessary operations have been explained, we take a more detailed look on some additional features of RSat, that add greatly to its performance. This includes:

- The maintenance of the learned clauses database Γ.
- The restarts that are performed by RSat.

All of our explanations will be given in connection with certain procedure names, that embody the mentioned tasks. These procedures are supposed to group the tasks in order to make it easier for the reader to draw connections between them.

This section is concluded with a summary of the functioning of RSat, and a presentation of its main solving procedure. This procedure will then combine the explained

procedures along with their names, to give a "complete picture" of the functioning of RSat.

The RSat Preprocessor

Before RSat begins its search, it will call for a preprocessor that will simplify the given formula. RSat makes use of the SATELITE preprocessor as explained in section section 2.3.3 (see page 17). Since the preprocessor and its use have already been introduced, we will not give any more details about it here.

Performing search: `SelectVariable()`, `MakeDecision()`, `SetDecision()`

The main goal of RSat, as for any SAT solver, is to find a model for a given formula. Let us call this formula F. In order to advance the search for such a model, RSat needs to choose a branching variable in each search iteration. Therefore, RSat uses a modification of the VSIDS (Variable Independent State Decaying Sum) strategy. The basic VSIDS strategy was first introduced in a solver called chaff [MMZ$^+$01]. It was revised in [LSB04]. The revised version of VSIDS works as follows.

Selecting a new branching variable with `SelectVariable()`: VSIDS uses a counter for each literal of a variable, which are initially set to the number of occurrences of the literals in formula F. These counters are updated in two ways.

First, whenever a conflict analysis is performed, a literal counter gets increased for every time the conflict analysis procedure looks at it, even if the literal does not appear in the finally learned asserting clause.

Second, the counters are frequently reduced (by dividing them by 2), after some threshold for variable scores is reached.

Whenever a new variable is needed for branching, `SelectVariable()` either returns the one with the highest counter value for one of its literals, or chooses one at random (though randomized choosing can be disabled).

In case no free variable can be found, the search is finished. To understand this, recall that the selection of a new branching variable is only done when the SAT state is normal (i.e. no conflicts are at hand). There are two possible reasons for why no more free variables exist in such a normal SAT state.

First, a value got assigned to each variable. Then, the normal SAT state implies that all clauses got satisfied (i.e. we have found a model).

Second, all free variables appear in clauses that are satisfied and are therefore ignored, because the assignment to those variables is irrelevant. The case, that an unassigned variable occurs in an unsatisfied clause does not occur here, otherwise

we could return this as a new branching variable. This in turn means, that no unsatisfied clause can hold an unassigned variable.

Since the SAT state is normal, no clause can consist of only falsified literals (i.e. we can not have the empty clause), because this would imply a conflict. Therefore, even though not all variables are assigned to a value, all clauses got satisfied already, indicating that we have found a model.

In summary, whenever a branching variable is to be chosen, but no free variable exists, that can act as such, we have found a model.

Selecting a direction with `MakeDecision()`: After a variable has been chosen, we need to decide what truth value will be assigned to it. Now two things can happen:

First, if the variable gets assigned for the first time, it will be assigned to 0.

Second, the variable is not assigned for the first time. If this is the case, we will assign the decision variable the value it had previously. The previous assignments are saved in an array called 'saved-literals'. The direction decision with respect to earlier assignments is called *progress saving*. The basic idea of progress saving is connected to the functioning of `Backtrack()`. We will describe progress saving when backtracking is described.

For now it is sufficient to know, that `MakeDecision(`x`)` selects either $\neg x$ or x as the new decision literal, depending on the assignment it chooses (if 0 is assigned, it returns $\neg x$, if 1 is assigned, it returns x).

After RSat decided on which assignment for the decision variable is used, it finalizes its decision by calling `SetDecision(`x`)`.

Propagating the decision with `SetDecision()`: Setting the decision now results in several actions.

The decision is added to the decision sequence D (by appending the decision literal) and setting its value in the assignment-array. The assignment-array holds all known assignments in a certain decision level (i.e. the decisions and all their implications).

Then, a *boolean constraint propagation* (BCP) is performed on this literal. Boolean constraint propagation is in its core an iterative unit propagation [MFM05].

BCP embodies the $\vdash$ operation from section 3.3.2 (see page 65). Hereby, BCP on a formula F and a literal x will yield a conflicts iff $F \wedge x \vdash \text{FALSE}$.

In order to perform BCP, the decision variable is assigned to its value, which might imply other assignments because some of the clauses from F or Γ could have become a unit clause now. Clauses implying assignments are called *reasons*. The value they imply is set in the assignment-array immediately (before their propagation is performed).

Formula F = $(\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_6 \vee x_7) \wedge (\neg x_1 \vee x_4) \wedge (x_5 \vee \neg x_2) \wedge (\neg x_7 \vee x_8) \wedge (\neg x_7 \vee \neg x_6) \wedge (\neg x_5 \vee \neg x_4 \vee x_1)$

Variable Count (positive, negative literal)	x_1 (1,3)	x_2 (0,1)	x_3 (1,0)	x_4 (1,1)	x_5 (1,1)	x_6 (1,1)	x_7 (1,2)	x_8 (1,0)

Decision level	1	2	3
Decision variable (& literal)	x_1 (x_1)	x_7 (x_7)	...
Implied assignments (queue)	x_4 x_3	$\neg x_6$ x_8	
Reasons	$(\neg x_1 \vee x_4)$ $(\neg x_1 \vee x_3)$	$(\neg x_7 \vee \neg x_6)$ $(\neg x_7 \vee x_8)$	
Assignment (x_1, x_2, ..., x_8)	(**1**,?,1,1,?,?,?,?)	(**1**,?,1,1,?,0,**1**,1)	

Figure 3.9: The connection between the terms *decision level*, *decision variable*, *implied assignments*, *reasons* and *assignment-array*.

All appearing implications are put on an implication stack for propagation. These implications must then (after the propagation of the decision literal is finished) be propagated as well, which might result in other implications etc.

The stack of found implications is priorized in such a way that BCP selects a literal to propagate, that has a high probability of creating a conflict. This priorized implication stack was first mentioned in [LSB04]. To distinct between priorized BCP and normal BCP, we will call the priorized version ECDB (Early Conflict Detection BCP).

ECDB uses the same literal counters as VSIDS for priorization. However, ECDB does not simply choose the literal for propagation next, that has the highest count. It rather chooses a literal whose *complement* has the highest count.

To clarify this, consider two queued implications on x and $\neg y$. Now ECDB will check for the counters of $\neg x$ (complement of x) and y (complement of $\neg y$). It then chooses x if $\neg x$ has a higher counter than y, or $\neg y$ if y has a higher counter than $\neg x$. The chosen implication is then propagated by ECDB (which might imply other assignments).

Two results from the ECDB are possible. First, the (iterated) propagation performed without any conflicts. In case this happens, ECDB returns without setting a conflicting clause. The iteration of the iterative SAT procedure is finished and a new one is started, based upon the updated assignment-array and decision sequence. For an image of a decision sequence in conjunction with some of the explained terms see figure 3.9.

Second, while performing additional propagations for implied variables, a conflict occurs. Hereby a conflict simply means, that an assignment is implied, that is contradictive to the one saved in the array of assignments.

If a conflict (an empty clause) is detected, ECDB dequeues all implied assignments and returns this clause as a new conflicting clause. Dequeueing all unprocessed

implied assignments is useful now, since we already know that there was something wrong with our decision. ECDB signals that a conflict has occurred by returning the conflicting clause (the clause that became empty during the propagation process), which in turn triggers the conflict handling.

For the detection of a conflict and implications, ECDB follows the *two watched literal scheme* (TWLS) from [MMZ+01]. The main task of the TWLS is to circumvent the necessity for visiting all clauses to see if one became unit or empty during propagation. How TWLS works will be explained next.

Let us assume we have a clause with N literals. A clause became unit, when $N-1$ of its literals evaluate to FALSE. The idea now is, to visit a clause only if its number of falsified literals goes from $N-2$ to $N-1$. This means, that it is not necessary to watch all of the literals in a clause. It is sufficient to watch exactly two literals that have not yet been set to a value, since, as long as we have two such literals, the count of falsified literals can not be greater than $N-3$.

If one of these "watched literals" is assigned so it evaluates to FALSE, we visit the clause. Now two things can happen [MMZ+01]:

First, the clause did not became unit. This can happen if there are other literals (not yet set), that can take the place of the just set watched literal. This unset literal will replace the other just set literal, thus we again have two unset (and watched) literals for this clause.

Second, the clause just became unit since no more unwatched and unset literals are available. In this case, a new implication is queued and an assignment for this implication is saved in the assignment-array. One should take note, that the implied literal is always the other watched literal, since it is the only one not yet assigned in this clause.

The following example is supposed to clarify the detection of implications as well as the detection of conflicts via the TWLS.

Example 18 *Let us assume we have the two clauses in an otherwise arbitrary formula:* $c_1 = (x_i \vee x_j \vee x_k), c_2 = (x_i \vee \neg x_j \vee x_k)$.

Let the two watched literals of the clauses be x_i, x_j *and* $x_i, \neg x_j$ *respectively. Both of them are unassigned at the moment.*

Let us further assume, that $x_k = 0$ *was already propagated. Its value is now saved in the assignment-array.*

At the next decision level, we assume that $x_i = 0$ *is assigned. Now this assignment touches one of the watched literals from* c_1 *and* c_2.

When c_1 *is investigated, RSat realizes it became unit and will immediately set* $x_j = 1$ *in the assignment-array. This clause is now satisfied and will be ignored. The implication* $x_j = 1$ *is still to be propagated and therefore saved in the priorized*

implication stack.

Now c_2 is still to be investigated, since we touched one of its watched literals. Here RSat detects, that there is no more free literal to be assigned. $x_j = 0$ would be the only way to satisfy this clause, but this literal just got assigned the opposite value when c_1 was analyzed (this assignment is already present in the assignment-array). This is why we have a contradicting assignment for x_j.

c_2 would become the new conflicting clause that is to be analyzed by the conflict handling procedures.

The conflict handling procedures will be explained next.

Conflict Handling: `AnalyzeConflict()`, `ComputeAssertionLevel()`, `AssertClause()`

We will now explain conflict handling in more detail. Conflict handling is performed in three steps.

First, from the given conflicting clause, decision sequence and clauses in F and Γ, a new conflict clause (which will be an assertion clause) must be learned. This is done in `AnalyzeConflict()`.

Then, the assertion level of that clause must be identified. This is done via the procedure `ComputeAssertionLevel()`. After the assertion level is computed, we backtrack to this decision level, thereby revising all decisions below that level.

Finally, the newly learned assertion clause must be asserted. This action is performed by `AssertClause()`. Asserting the clause basically inserts the clause to Γ and performs a unit propagation on this clause (since we backtracked to the assertion level of this clause, this clause must be unit here). This can result in further conflicts, where the conflict handling is repeated until no more conflicts arise or, a conflict arises within decision level zero.

A conflict within decision level zero means, that without any decisions, a conflicting assignment is forced (by two unit clauses in decision level zero), making the formula unsatisfiable.

We will now describe the function of the three conflict handling procedures in more detail.

Learning from conflicts: `AnalyzeConflict()`: The main goal of `AnalyzeConflict()` is to derive a new conflict clause. In our case, we are interested in only assertion clauses.

RSat uses the FirstUIP learning scheme which was introduced in [ZMMM01]. This scheme is explained next. To clarify the explanations given below, we will use an example from [CW03] (see figure 3.10).

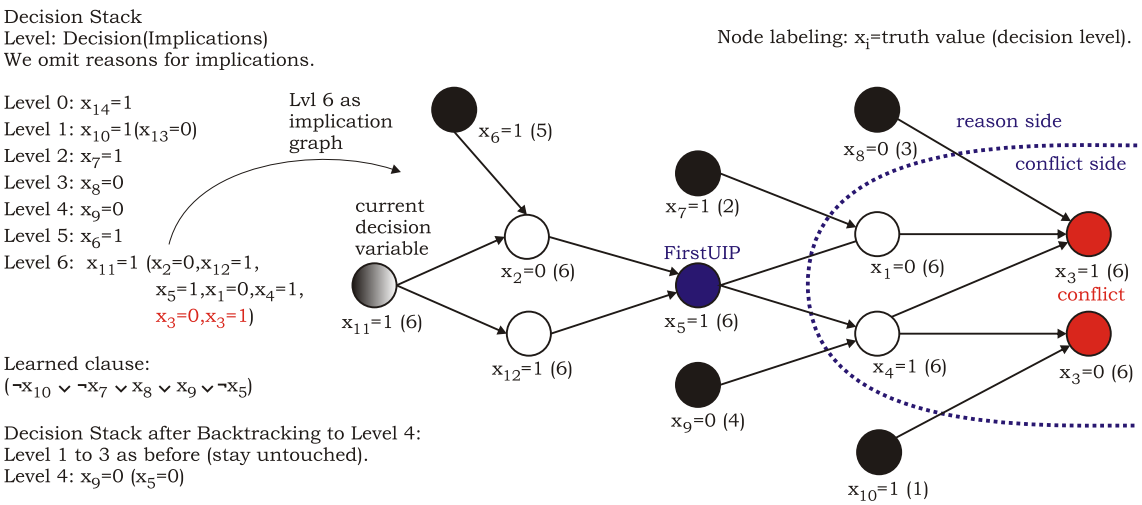


Figure 3.10: Example for the detection of a conflict in decision level 6 and an implication graph for this level.

Example 19 *Given the formula*

$$F = \underbrace{(\neg x_6 \vee \neg x_{11} \vee \neg x_2)}_{1} \wedge \underbrace{(\neg x_{11} \vee x_{12})}_{2} \wedge \underbrace{(x_2 \vee \neg x_{12} \vee x_5)}_{3} \wedge \underbrace{(\neg x_7 \vee \neg x_1 \vee \neg x_5)}_{4} \wedge$$
$$\underbrace{(\neg x_5 \vee x_9 \vee x_4)}_{5} \wedge \underbrace{(x_8 \vee x_1 \vee \neg x_4 \vee x_3)}_{6} \wedge \underbrace{(\neg x_{10} \vee \neg x_4 \vee \neg x_3)}_{7} \wedge \underbrace{(\neg x_{10} \vee \neg x_{13})}_{8} \wedge \underbrace{(x_{14})}_{9}$$

Let us assume we do the following decisions. The reader should note, that these decisions are made in an arbitrary manner, not taking into account the exact decision-finding procedure of RSat itself. This concession is made to simplify the example in order to be able to better concentrate on the learning scheme itself.

In level 0, we find unit clause 9 and propagate $x_{14} = 1$, this does not imply any other assignments.

In level 1, we assign $x_{10} = 1$ and this implies $x_{13} = 0$ because of clause 8.

Then, in level 2,3,4, and 5 we make the assignments $x_7 = 1$,$x_8 = 0$,$x_9 = 0$,$x_6 = 1$, which do not yield any further implications.

Arriving in level 6, we decide to assign $x_{11} = 1$. This triggers a set of implications: $x_2 = 0$, $x_{12} = 1$, $x_5 = 1$, $x_1 = 0$ and $x_4 = 1$ as well as $x_3 = 0$ and $x_3 = 1$. The last implications are of course contradictive, and we have found a conflict.

As presented in the previous example, making decisions, while search advances, results in implications and maybe a conflict. When a conflict arises, we first construct an implication graph for the decision level of the conflict (see figure 3.10).

An implication graph is a directed graph, that contains two types of vertices: those that do not have antecedents (nodes with no incoming edge), and those that have antecedents (at least one incoming edge).

Nodes without antecedents refer to decision variables and are labeled with the truth

value they received, and the decision level in that this decision was made (see figure 3.10).

Nodes with antecedents refer to variables, that are implied. They are labeled with the implied truth value and the decision level in which they got implied.

In figure 3.10, we color a node black to identify it as a decision variable. Nodes of implied variables are colored white. The node at the current decision level is shaded gray.

As we can see in figure 3.10, we have exactly one variable that is represented twice in the implication graph: x_3. It is represented twice, because it receives opposing truth values through the given implications.

The presented implication graph can be constructed using F, Γ and D, by revertedly following the reasons for implications starting with the conflict nodes. A reason then implies a variable for the currently investigated node. The reason itself might be the result of other implications, for which there must be reasons as well. The only assignments that come without a reason are those, that are decisions themselves. The construction of the graph is finished, when no more predecessors for nodes in the implication graph can be found (i.e. when all decisions with impact on the current conflict have been found).

As soon as the implication graph has been constructed, we will start to analyze this graph in order to find the preconditions for the conflict to arise. Such preconditions are a subset of all decisions represented in this implication graph (i.e. variable assignments).

As a result of the conflict analysis, we want to obtain a learned clause, that, when added to Γ, forbids this constellation of preconditions.

In order to find this set of preconditions, we apply the FirstUIP learning scheme as described below:
Perform a backward search through the implication graph, starting with the nodes representing the conflict ($x_3 = 0$,$x_3 = 1$). The backward search consists of the following steps:

1. Iteratively collect a set of nodes that is called the conflict side. Therefore, collect any antecedent nodes of the conflict that are not representing
   - decision variables,
   - unique implication points.

   Stop collecting iff:
2. No more direct antecedents can be found other than those of 1.

A *unique implication point* (UIP) is a node in the implication graph, that all implications towards the conflict must go through when starting at the current decision variable.

The UIP closest to the conflict is called the *first unique implication point* (FirstUIP) (see figure 3.10). Once the collection of the conflict side is finished, we have split the implication graph into two parts: conflict side and reason side, whereas the reason side are all nodes that are not in the conflict side.

The set of edges that cross the border between the reason and the conflict side are of special interest. They are linked to all the necessary preconditions for the conflict. So the setting of these nodes as given in the implication graph are the necessary preconditions for the current conflict.

In our example (see figure 3.10), these nodes are $x_8 = 0$,$x_7 = 1$,$x_5 = 1$,$x_9 = 0$,$x_{10} = 1$. We call these variables the conflict preconditions. To avoid the conflict, at least one of the conflict preconditions must receive a different assignment. In other words, we need to avoid $(\neg x_8 \wedge x_7 \wedge x_5 \wedge \neg x_9 \wedge x_{10})$ in order to avoid the conflict at hand. We therefore learn the clause

$$\neg(\neg x_8 \wedge x_7 \wedge x_5 \wedge \neg x_9 \wedge x_{10}) = (x_8 \vee \neg x_7 \vee \neg x_5 \vee x_9 \vee \neg x_{10}) = \alpha.$$

As soon as we have learned this clause, we are finished with the conflict analysis. The scheme described above is called the FirstUIP learning scheme (for more information see [ZMMM01, CW03]). It is interesting to note, that any clause learned with this scheme is an asserting clause.

Updating clause weights in `AnalyzeConflict()`: Conflict analysis obviously works with clauses currently acting as reasons for certain implications. RSat uses clause weights in order to measure the importance of clauses in Γ (i.e. how valuable they are in detecting conflicts).

Whenever a new clause is added to Γ, it will receive a basic clause score called `SCOREINC`. Adding of learned conflict clauses is not part of the `AnalyzeConflict()` procedure, but since it touches reasons, it can detect which of them are currently useful for conflict detection.

So whenever `AnalyzeConflict()` touches a clause (reason) that is part of Γ, it will raise the score of this clause. It does so by adding `SCOREINC` to this clause again. The more often a clause is touched (i.e. participating in detecting a conflict), the more weight it will have.

When the score of a clause becomes larger than a certain limit, *all* clause scores are updated. Updating is done by dividing all clause scores by a constant named `clauseScoreLimit`, whereas the `clauseScoreLimit` is the maximum allowed clause score.

The reason for measuring the importance of learned clauses is closely related to the way how the maintenance on Γ is performed. This will be explained later. For now it is sufficient to know, that each learned clause receives the basic score value `SCOREINC` when it is added to Γ. The score value is updated every time the clause

participates in detecting a conflict (by adding `SCOREINC`). Clause scores are reduced by division with `clauseScoreLimit`, whenever the weight for one clause becomes greater than the `clauseScoreLimit`.

However, after we have learned an assertion clause, we must revise some decisions in D in order to escape from the conflict at hand. Therefore, we need to know the assertion level of our just learned conflict clause.

Computing the assertion Level: `ComputeAssertionLevel()` After the clause is learned (which is always an asserting clause under the FirstUIP learning scheme), we need to compute its assertion level to backtrack to. One could argue, that backtracking to the topmost decision level (zero) is sufficient, and thus, the computation for the assertion level is useless. Of course, backtracking to the topmost decision level revises all decisions, and therefore, all decisions that resulted in the conflict at hand, but it will also revise decisions that did not participate in this conflict.

Even worse, backtracking to the topmost level might erase solutions for other parts of the formula that are independent from this conflict. Therefore, we want to revise as few decisions as possible (not even all decisions that are preconditions for the conflict).

It is sufficient to revise exactly one of the conflict preconditions (that are part of the just learned conflict clause). The goal now is to find the one conflict precondition, that has the largest decision level *and* is a decision variable itself.

Once we backtrack to this decision level (without altering it), the just learned conflict clause has but one unassigned variable left: the FirstUIP (which is the asserting literal for this asserting clause). How we compute this decision level is explained next.

`ComputeAssertionLevel`() now computes the assertion level of the learned assertion clause. We need to know this number since we want to backtrack to this level.

We simply check the literals within our just learned conflict clause

$$\alpha = (x_8 \vee \neg x_7 \vee \neg x_5 \vee x_9 \vee \neg x_{10}).$$

and compare their decision levels. In case of the example from above, this would be $x_8(3)$,$x_7(2)$,$x_9(4)$,$x_{10}(1)$. We detect, that x_9 has decision level 4 which is the greatest of all participating decision variables.

The reader should note, that the FirstUIP x_5 (which is also part of the learned conflict clause), has decision level 6, since it was implied in this level. But since it is no decision variable, we do not take its value into account when calculating the assertion level.

Now when we backtrack to level 4, the FirstUIP x_5 is the only variable, that did not yet receive a value and therefore, in level 4, the conflict clause is a unit clause.

The advantage of backtracking to level 4 (without altering level 4) now is, that this

unit clause implies a value for the FirstUIP. It forces to it the *opposite* value of the one it had when acting as a precondition for the conflict. Therefore, we just destroyed one of the necessary preconditions for the conflict and successfully guided our search "around" it.

Let us assume we have computed the assertion level of the conflict clause and backtracked to it. We now have to add this clause to Γ and need to detect any further implications that this action may cause (maybe another conflict as well). This is done by the `AssertClause()` procedure.

`AssertClause()`: Asserting a learned assertion clause α is based on two steps.

First, the clause is added to the set of all learned clauses Γ. When `AssertClause`() adds α to Γ, it will set its score to `SCOREINC` (i.e. gives it the basic clause score).

Second, we need to propagate the knowledge of our just added assertion clause. Since we backtracked to the assertion level of this clause, it will be unit here. Therefore, we will again call the ECDB procedure to perform a propagation on the assertion literal of our assertion clause.

This either results in a new conflict, which again triggers the conflict handling, or, if no conflict arises, finishes conflict handling for the conflict that was found. If no further conflicts are found, the iterative step of the iterative SAT procedure is finished and a new iterative step is started.

We have mentioned backtracking already. How backtracks are performed by RSat will be explained next.

Backtracking: `Backtrack()`

Backtracking denotes the revision of decisions made earlier (i.e. removing some decisions from the end of the decision sequence).

`Backtrack()` therefore unassigns the decision variable at the current decision level. Then, it unassigns all literals that were implied by this decision. When `Backtrack`() finishes with unassigning the decision variable and its implications of level k it goes back to level $k-1$ and repeats the unassignment scheme from above. Then, it goes back to $k-2$, then to $k-3$ and so on. Backtrack will continue to unassign the decisions and implications for the levels one by one, until it reaches the decision level it is supposed to backtrack to. The decision and its implications in the destination level are left unchanged.

We have mentioned earlier, that RSat remembers erased assignments via *progress saving* [PD07]. Therefore, `Backtrack()` just saves the values of the decision variable and its corresponding implications to an extra array, just before it unassigns them. This array is called the "saved-literal" array. One can understand this array as "earlier assignments made to the variables".

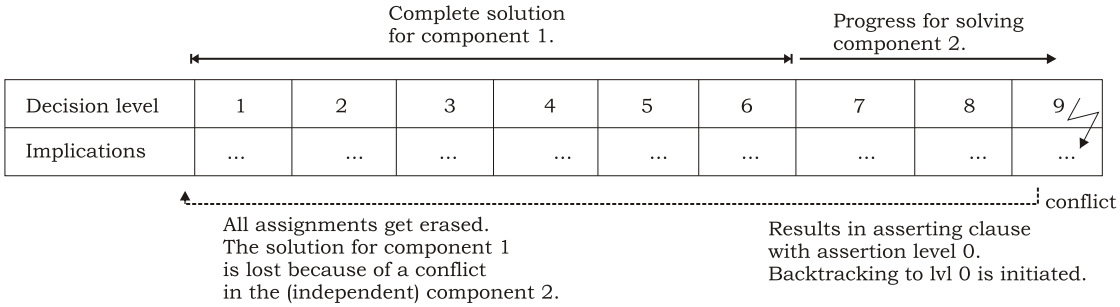


Figure 3.11: The effect of erasing decisions on earlier found solutions.

When search advances again, the `GetDecision()` procedure (that must return a literal for the chosen decision variable) will first check the saved-literal array. If this array contains a value for the variable that is about to be set, it uses this value. For example, if the saved-literal array contains a zero for the branching variable x_i, than `GetDecision()` will return $\neg x_i$ as the literal to propagate as new decision.

This seems to be useless on the first glance, because we erased these assignments to variables when escaping from a conflict. Therefore, one could ask what benefit it has when saving these old assignments. To explain why progress saving can be useful, we present an example from [PD07].

Example 20 *Let us assume we have a formula that consists of four identical copies of a satisfiable instance. We will call this combined formula a replicated instance, whereas the copies are its (independent) components.*

Let us furthermore assume, that a solution for component 1 is already saved in the decision sequence (see figure 3.11). Lets furthermore assume, that a solution for component 2 is searched. While this search is conducted, a conflict is detected and an assertion clause is learned. This assertion clause could have assertion level zero. Therefore backtracking is initiated all the way up to decision level zero (see figure 3.11).

The result is, that we erase our already computed solution for component 1 just because of a conflict in an independent other component.

Erasing all solutions for component 1 is completely useless, since all variables in component 1 do not occur in component 2 and vice versa. When using progress saving, we can not avoid the unassignment of variables for component 1, but we can save them in the saved-literals array.

When searching again for the solution of component 1, we use the values saved in the saved-literals array. Since this array now contains the solution, all decisions made by `GetDecision()` (to decide what truth value to assign to a decision variable), will probably yield the correct value. Therefore, the `MakeDecision()` procedure will most likely not run into a conflict again. The effect on solving a replicated instance is visualized in figure 3.12 [PD07].

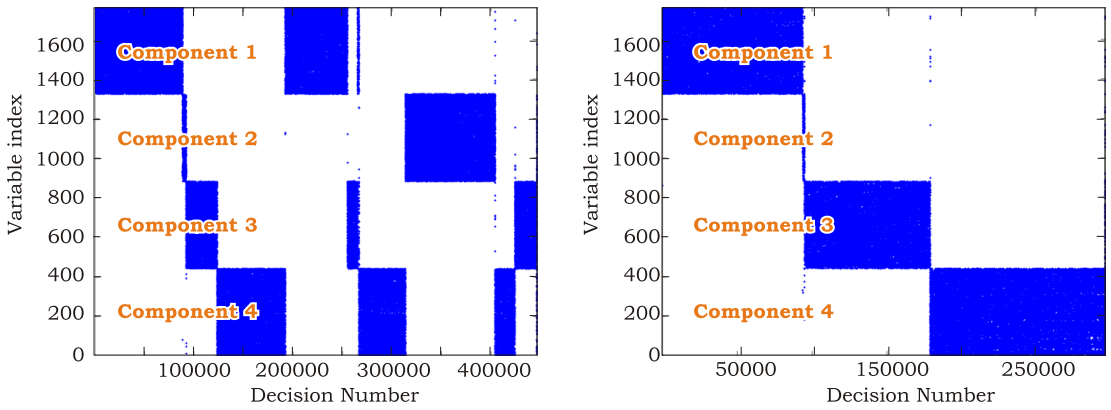


Figure 3.12: The effect of progress saving. Left, search without progress saving. Right, progress saving turned on.

As we can see from figure 3.12, when progress saving is turned off, the solver has to visit components more often to regain a solution for it. Additionally, the number of decisions is much higher when progress saving is turned off.

With progress saving turned on, components get visited in a much more constant fashion. Furthermore, the number of decisions is much lower with progress saving turned on. For more information on progress saving see [PD07].

We have now explained the primary operations that RSat performs while searching. We will now continue with the secondary features to maintain learned clauses and schedule restarts.

Removal of learned clauses: `MaximumNumberOfClauses()` and `DeleteClauses()`

While search advances, the number of learned clauses in Γ grows. The procedure `DeleteClauses()` removes clauses from Γ when a certain threshold of clauses is reached.

The procedure `MaximumNumberOfClauses()` checks, if a threshold for a maximum number of currently learned conflict clauses is reached. If this is the case, it returns true, and false otherwise (see listing 3.9).

Listing 3.9: The MaximumNumberOfClauses() procedure.

```
MaximumNumberOfClauses(Γ){
    if (size of Γ ≥ maximumNumberOfLearnedClauses
                        + numberOfAssignedVariables){
        return true;
    } else {
        return false;
    }
}
```

Hereby, `maximumNumberOfLearnedClauses` refers to a value that represents the de-

sired maximum size of Γ.

The number of assigned variables (parameter `numberOfAssignedVariables`) is simply the number of variables that are currently assigned (either by a decision or by an implication). The number of assigned variables is approximately the number of clauses which are currently acting as reasons for some implication (because the majority of assignments are implied).

The reason why we compare the size of Γ to `maximumNumberOfLearnedClauses` + `numberOfAssignedVariables` is to take into account, that some learned clauses are currently used for implying literals. These clauses from Γ (the reasons) must not be removed from Γ, since they are needed for conflict analysis, even if we have reached the maximum number of learnable clauses.

To circumvent the call for maintaining Γ with only such clauses, we raise the basic border of `maximumNumberOfLearnedClauses` with the number of assigned variables (which is roughly the number of reasons). We therefore assure, that Γ holds at least `maximumNumberOfLearnedClauses` dispensable clauses and the maintenance of Γ is not performed in vain.

The value for `maximumNumberOfLearnedClauses` is initialized to $|F|/3$ and updated during the maintenance of Γ in procedure `DeleteClauses()`. This procedure is explained next.

The main task of `DeleteClauses()` is simply to remove certain clauses form Γ, that seem to be of no use anymore. First of all, there are certain clauses that must not be removed. As already mentioned, clauses from Γ, that are currently acting as reasons, must not be removed. Furthermore, binary clauses must not be removed as well.

Removing reasons must be prohibited, because conflict analysis relies on them to construct the implication graph. Such clauses are called *locked clauses*.

Removing binary clauses must be avoided, because these clauses are the perfect candidates to create new implications, which in turn helps in rapidly advancing the search.

In case a clause is not binary or locked, it will be erased, iff its score is below a `limit`. This `limit` is a border that divides the number of clauses in Γ into two categories: active and inactive.

Recall, that the score of a clause gets larger, whenever it participates in resolving a conflict (when `AnalyzeConflict()` touches it). Clauses, that often participate in conflict analysis, will have a score high enough to be greater than `limit`.

Furthermore, recall that the score of *all* clauses is frequently reduced whenever procedure `AnalyzeConflict()` finds the clause score of *one* clause to be sufficiently large. When a clause fails to act in resolving conflicts for too long, its score will fall below `limit`. Clauses with a score below `limit` are considered inactive and get

removed.

In order to compute the `limit`, RSat calculates `limit = (SCOREINC)/`$|\Gamma|$. As mentioned before, `SCOREINC` is the basic clause score that every new conflict clause receives when being added to Γ.

After all clauses in Γ have been checked, whether they can be removed or not, RSat needs to update the `maximumNumberOfLearnedClauses` parameter (by multiplying it with `MAXLEARNEDCLAUSESMULT` $= 1.5$). To understand this, recall that RSat follows the iterative SAT procedure from section 3.3.2 (see page 70). This procedure can be proven to terminate only iff it is able to learn all assertion clauses for all conflicts. The increasing of the maximum number of learned clauses ensures, that this will be the case for some number of performed iterations.

The question then is, why clauses get deleted anyway. The answer to this question lies within the application of the ECDB procedure, that propagates implications. The more clauses in Γ exist, the more computation time is used up in ECDB. The aim of deleting clauses is to lower the computational overhead for ECDB. By removing only inactive clauses, we assure that no valuable knowledge on conflicts is lost.

The functioning of `DeleteClauses()` is summarized in listing 3.10.

Listing 3.10: The DeleteClauses() procedure.

```
DeleteClauses(Γ){
    if (size of Γ ≤ 0){
        return;
    }
    double limit = (SCOREINC)/|Γ|;
    for (clause c : Γ){
        if (c is a binary clause){
            continue;
        }
        if (c is a locked clause){
            continue;
        }
        if (score of c < limit){
            remove c from Γ;
        }
    }
    maximumNumberOfLearnedClauses *= MAXLEARNEDCLAUSESMULT;
}
```

We now have explained the maintenance procedures for updating the number of clauses in Γ. In the next section, we will take a look at the restarting strategy that RSat performs.

The restarting strategy: `ScheduleNextRestart()`

Before we can describe the restarting strategy of RSat, we need to clarify why restarts are useful. This is not clear at the first glance, since RSat is a complete

SAT solver and a restart is therefore not necessary by design. An explanation for this is given next.

Why restarts are useful: In [GSC97], structured combinatorial search problems are studied along with the behavior of search algorithms trying to solve them. SAT itself can be seen as such a combinatorial search for a model of a propositional formula in CNF.

In [GSC97] it is stated, that it is not uncommon to observe a search algorithm "hang" when trying to solve the problem, while a new run on the problem (using the same search algorithm) seems to solve the problem quickly.

In order to investigate this phenomenon, a variety of cost-distribution profiles of search methods on various problem instances are studied. One could expect, that the mean cost to solve an increasing number of problem instances of a certain type, is to stabilize. This is not the case.

The mean and the variance of the search cost often behave erratic, and do not stabilize when search is conducted for an increasing number of problem instances. In other words, it is hard to predict the behavior (in terms of runtime) of a search algorithm when searching specific combinatorial problems. These problems are called *heavy-tailed*.

In the context of our work, it is not relevant to understand the exact details of *heavy-tailed* search problems. It is sufficient to know, that such problems exist. The erratic behavior of search algorithms working on these problems is unwanted, since a large variance in solving time might yield a bad worst case runtime for some search runs at random.

A way of reducing this variance in the search behavior is the application of restarts [GSC97]. Therefore, RSat is performing them.

The question now is, *when* and *how* restarts are performed. We start by explaining when RSat performs restarts, and afterwards, we will explain how RSat performs them.

When to perform restarts: Before we can explain when restarts are to be performed, we need to introduce some terms.

Definition 20 *An Algorithm $\mathcal{A}$ is called a* `Las Vegas algorithm`*, if it is a randomized algorithm that always produces the correct answer when it terminates, but whose runtime is a random variable [LSZ93].*

In other words, such an algorithm will eventually stop with the correct result at hand, but one can not predict when this might be. RSat is a Las Vegas algorithm.

In [LSZ93], a restarting strategy is developed for Las Vegas algorithms. The idea is to create a sequence of runs for the algorithm, whereas in each run i the algorithm can use up a fixed amount of time t_i. When the algorithm (during one of these runs) outputs a result, the simulation stops successfully. More formally:

Definition 21 *Let t_i be a parameter indicating the amount of time a Las Vegas algorithm can use to compute a solution. A* `strategy` *$\mathcal{S} = (t_1, t_2, \dots)$ is a tuple of fixed numbers of runtimes the algorithm has in each run. When the computation time t_i is used up for run i, the algorithm is restarted in run $i + 1$ whereas it has the time t_{i+1} to conduct search.*

The task is to minimize the expected runtime of $\mathcal{A}$ to obtain an answer using a strategy $\mathcal{S}$, by identifying an optimal strategy [LSZ93].

As a result of the study presented in [LSZ93], the universal strategy (applicable in our case) is $\mathcal{S}_u = (t_1, t_2, \dots)$, with

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

Whereas k is depending on i. This would result in $\mathcal{S}_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \dots)$, meaning, that each time a pair of runs of a given length has been completed, a run of twice that length is immediately executed [LSZ93]. Now two things must be considered for an implementation in RSat.

First, the word "time" must be explained in more detail to explain when it is "time" to restart. Using real-time is obviously not a good idea, since this would impact the algorithms behavior on different computers. We therefore define a *pseudo-time* to model the search progress.

Second, depending on the understanding of pseudo-time, the rule to calculate t_i must be adapted to output values that correspond to this pseudo-time.

In order to define a useful time measurement, consider the way search advances in RSat. The algorithm performs unit resolution to find conflicts (with ECDB). Hence, the number of detected conflicts is a way to measure the progress of the search.

RSat therefore uses the number of conflicts it detected in a "run" to decide whether to restart or not. Once a certain number of conflicts is detected, the pseudo-time for this run is used up and a restart is performed.

Now the calculations for the pseudo-times of different runs must be aligned with S_u, since this is the optimal restarting strategy. The time-values t_i must now represent numbers of detected conflicts. We alter the calculation scheme from above the following way:

$$t_i = \begin{cases} 2^{k-1} \cdot c, & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

Whereas c is a constant representing a number of conflicts. RSat uses $c = 512$.

Using this constant, the following scheme for restarts is used (with respect to the number of detected conflicts): $\mathcal{S}_u = (512, 512, 1024, 512, 512, 1024, 2048, \dots)$. The check for whether a restart is to be performed is simply a check for whether the maximum number of conflicts for this run has been reached.

The procedure `ScheduleNextRestart()` performs the calculations to set the new maximum number of conflicts for the next step (i.e. calculates the runtime for the next run). `ScheduleNextRestart()` works as outlined in listing 3.11.

Listing 3.11: The ScheduleNextRestart() procedure.

```
ScheduleNextRestart(){
    numberOfRestarts++;
    restartConflictIncrease = GetLuby(numberOfRestarts);
    nextRestartConflictNumber =
        numberOfConflicts + restartConflictIncrease;
}
```

Whereas the `GetLuby()` procedure calculates the length of the next run, as presented in listing 3.12.

Listing 3.12: The GetLuby(numberOfRestarts) procedure.

```
GetLuby(numberOfRestarts){
    if (numberOfRestarts == 0){
        return 512;
    }
    double k = log(numberOfRestarts+1)/log(2);
    if(k==floor(k+0.5)){
        return (int)(pow(2,k-1))*512;
    }else{
        k = floor(k);
        return GetLuby(numberOfRestarts-(int)pow(2,k)+1);
  }
}
```

The calculations look complicated but are a straight forward implementation from the optimal restarting strategy $\mathcal{S}_u$.

How restarts are performed by RSat: The questions for why and when to restart are now answered, leaving only the how.

A restart is performed by revising all decisions in D through backtracking to the topmost level (zero). Learned assertion clauses will be remembered (Γ is not altered).

After backtracking, F will be simplified by removing satisfied clauses. This makes no sense on the first glance, since all decisions have just been revised. So when there are no decisions (i.e. values assigned to variables), it is questionable on how clauses can be removed from F. On the other hand, new assertion clauses might have been learned, which might imply the setting of variables in the topmost decision level.

To understand this, recall that any assertion clause has an assertion level at which this clause becomes a unit clause (see page 71). When an assertion clause was learned with assertion level zero, this clause is unit in level zero (the level we just backtracked to). This unit clause now implies the setting for the variable it contains via the unit reduction principle.

The procedure `Simplify(`F`)` just propagates this knowledge (by iterative unit propagation) and removes all clauses from F that become satisfied. Furthermore, all literals that got falsified by these implications are removed from clauses in F. The result is a simplified formula in decision level zero, on which search can be continued.

We have now explained all operations that RSat performs. The next section will summarize the explained details and give a description of the main solving procedure of RSat.

Summarizing the functioning of RSat

RSat follows the iterative SAT procedure from section 3.3.2 (see page 70). Before the iterative search starts, it will call for the preprocessor SATELITE in order to simplify the formula at hand. It might happen, that SATELITE finds the formula to be unsatisfiable or can provide a model for it, though it is highly unlikely that these cases occur.

Anyway, after SATELITE finishes, search will be conducted for a simplified formula. At first, RSat will initialize several data structures and parameters. After that, the iterative search is started, consisting of the following tasks:

- Check, whether it is time for a restart. If so, backtrack, schedule the next restart, and begin a new search iteration.
- Check, whether the maximum number of clauses has been reached. If so, perform clause database maintenance.
- Select a new branching variable. In case no such variable exists, we can output a model.
- Select a value for the branching variable. Therefore, check whether the saved-literals array already contains a value for the underlying variable. If not, assign 0 to it.
- Propagate the decision on the new branching variable.
- If no conflict was found during the propagation, continue with a new iteration. If there was a conflict in a decision level greater 0, initialize conflict handling, backtrack and continue with a new iteration. If there was a conflict in decision level 0, return “unsatisfiable”.

The functioning explained above can be summarized in the main solving procedure of RSat (see listing 3.13).

Listing 3.13: The main solving procedure of RSat

```
RSat(F){
    F = SATELITE(F);
    D = ();
    Γ = {};
    nextRestartConflictNumber = 512;
    numberOfConflicts = 0;
    numberOfRestarts = 0;
    savedLiteralArray = set all assignments to unknown;
    assignmentArray = set all assignments to unknown;
    while(true){
        if (numberOfConflicts ≥ nextRestartConflictNumber){
            Backtrack(0,D);
            ScheduleNextRestart();
            Simplify(F);
        }
        if(MaximumNumberOfClauses(Γ)){
            DeleteClauses(Γ);
        }
        v = SelectVariable();
        if (v == null){
            output(assignmentArray); //which now is a model
        }
        l = MakeDecision(v);
        conflictingClause = SetDecision(l, F, Γ, D);

        while(conflictingClause ≠ null){
            if (decisionLevel == 0){
                return unsatisfiable;
            }
            numberOfConflicts++;
            assertionClause c =
              AnalyzeConflict(F, Γ, D, conflictingClause);
            m = ComputeAssertionLevel(D, c);
            Backtrack(m, D);
            conflictingClause = AssertClause(c);
        }
    }
}
```

We have now finished our explanation on how RSat works. The following section will give some insight on how RSat performed during the SAT 2007 Competition.

3.3.4 Explaining the Performance of RSat

As we have stated in the introduction to this section, RSat performed quite well during the the SAT 2007 Competition[4].

RSat was able to solve 139 of the 234 industrial instances. It proved 76 of them to be unsatisfiable and provided correct solutions for 63 of the satisfiable ones. RSat

[4]http://www.satcompetition.org

and picosat solved the same amount of instances, but since RSat completed its task in much less time, it was awarded the first place in the "Industrial SAT+UNSAT" category.

Restricted to only the unsatisfiable instances, RSat proved the third most of them to be unsatisfiable, but it provided the results in the shortest time, and was therefore awarded the first place in the "Industrial UNSAT" category.

Restricted to only the satisfiable instances, RSat provided the second most models. RSat was therefore awarded the second place in the "Industrial SAT" category.

RSat also participated in solving the handmade instances and was awarded the 7th place in "Handmade SAT+UNSAT" for solving 55 out of 201 handmade instances.

RSat did not participate in the random category, therefore no results can be presented for this type of problem category.

We will now explain why RSat was able to perform well on the industrial instances.

The Benefits of using the Preprocessor

As we have already mentioned, RSat uses SATELITE as a preprocessor. In [Pha06] a study is presented on the impact of various preprocessors on industrial instances. SATELITE is stated to be the best preprocessor for industrial instances. Out of 64 test instances used in [Pha06], SATELITE produces the greatest variable elimination in 62.5% of the cases and the greatest clause elimination in 50% of the cases. RSat therefore benefits from solving smaller (more simplified) instances in the industrial category.

The Benefits of using Restarts

We have explained the universal restarting strategy $\mathcal{S}_u$ (see section 3.3.3 on page 87), that is preformed by RSat. In [Hua06], the impact of various restart strategies is studied, ranging from no restarts over fixed interval restarts and geometric restart policies to the described strategy of $\mathcal{S}_u$.

The study concludes, that $\mathcal{S}_u$ outperforms (in terms of solving time) all other tested restarting strategies when combating the problem of heavy-tailed combinatorial search problems. Therefore, RSat benefits from a dominating restarting policy as well.

The Benefits of Early Conflict Detection

We have already mentioned the importance of detecting conflicts fast when analyzing the performance of March_ks (see section 3.2.4 on pager 55). We concluded, that the reduction of the search tree width is of particular importance for short solving

times with look-ahead solvers.

RSat performs this "early search tree width reduction" by the application of ECDB. The ECDB procedure performs the boolean constraint propagation within RSat (see page 74).

When RSat commits to a decision literal, the ECDB procedure propagates this decision, which in turn might result in other implications that must be propagated. ECDB priorizes the detected implications with respect to their ability to yield a conflict. It prefers the implications that have a higher probability of creating such a conflict when being propagated.

In [LSB04], a study is performed to measure the impact of using ECDB instead of normal boolean constraint propagation (without a priorized implication stack). They concluded, that ECDB outperforms BCP on structured instances.

Using ECDB reduces the work performed in the BCP procedure to approximately 40%. Conflict-driven SAT solvers spend about 90% of their computation time in the BCP procedure [MMZ+01]. Hence, the total runtime of a conflict-driven SAT solver using ECDB is about 44% of the runtime of a SAT solver using normal BCP. RSat therefore benefits from early conflict detection as explained in section 3.2.4 (see page 55).

It is interesting to note, that look-ahead solvers as well as conflict-driven solvers, benefit from the "early search tree width reduction" by detecting conflict as soon as possible.

The Benefits of Deleting inactive Conflict Clauses

Learned clauses deletion adds to the reduction of the computation time of ECDB as well. ECDB (even though it uses the two watched literal scheme explained on page 75), has to perform more computations when more clauses are to be checked.

Not all of these clauses are equally participating in detecting or resolving conflicts. Therefore, RSat uses weights to measure their activity (see page 83). In order to keep the spend computation time as useful as possible, inactive clauses get deleted from time to time.

This effectively reduces the time spend in the ECDB procedure, and because most of the computation time is spend in this procedure, it effectively reduces the overall solving time of RSat.

The Benefits of using Progress Saving

Furthermore, we have explained how RSat performs progress saving in section 3.3.3 (see page 81). Progress saving allows for fast regaining an already computed solution after it got erased by backtracking. Therefore, RSat is able to cope with the

destruction of solutions through backtracking much better, than other solvers using the iterative SAT procedure.

The drawbacks of using a Priorized Implication Stack in ECDB

However, RSat also participated in solving handmade instances. Even though handmade instances often contain exploitable structure, RSat was not able to perform well on them compared to other solvers.

RSat ran out of time for 141 of the 201 handmade instances (the timeout was set to 5000 seconds). For the solved instances, RSat had an average CPU time of 1273 seconds per instance, which is by far the worst in the handmade category.

A reason for this might be the computational overhead of the priorized implication stack in ECDB. ECDB calculates priorities for all found implications in order to choose one with a high chance of creating a conflict. Whenever this is a large number, a reasonable amount of computation time is needed to calculate these priorities. However, when all implications end up in a conflict, the computation time for priorization was not spend very wisely.

For these instances, it would be better to simply pick an implication at random to propagate it. Here, chances are high that a picked implication yields a conflict anyway, which means that the computational overhead for stack-priorization can be saved.

Because we can not perform a detailed analysis here, we will skip any further explanation for RSat‘s behavior on handmade instances.

RSat did not participate in the random category during the SAT 2007 Competition. Therefore, an analysis of the behavior on these instances is skipped.

A summary of the most important features of RSat will be given in the next section.

3.3.5 Summary of the Review of RSat

RSat is an iterative conflict-driven DPLL complete SAT solver, that proved to be very competitive during the SAT 2007 Competition.

It employs the preprocessor SATELITE to reduce the size of a given formula. Then it applies the iterative SAT procedure to solve the simplified instance.

RSat utilizes ECDB, an improved version of the standard boolean constraint propagation, to propagate decisions and detect conflicts as soon as possible.

Discovered conflicts are used to learn assertion clauses via the FirstUIP learning scheme. These asserting clauses help the solver to avoid these conflicts in the future search. Once the asserting clause is learned, the solver performs backtracking along with progress saving.

To reduce the computation time of the ECDB method, learned clauses are deleted when they are found to be inactive for the current proceeding of the search.

In order to counter the effect of heavy tailed combinatorial search problems on the search behavior, RSat performs restarts according to the optimal restart strategy $\mathcal{S}_u$.

We have now completed the review of RSat, as well as our presentation of modern DPLL solvers. We explained the DPLL SAT solver paradigm, named its advantages and drawbacks and presented two modern DPLL solvers: a representative for look-ahead DPLL solvers (March_ks) and a representative for conflict-driven DPLL solvers (RSat). We will continue with the explanation of the SLS SAT solver paradigm.

Chapter 4

SLS Algorithms

The acronym SLS stands for "Stochastic Local Search" and identifies the class of SAT algorithms that utilize local search. The first overall successful SAT algorithm to use local search was invented by Bart Selman, Hector Levesque and David Mitchell [SLM92], and was called GSAT. GSAT, along with local search, will be explained in the next section.

One of the major advantages of SLS algorithms is that they scale very well. They can handle large size problem instances that are out of reach for DPLL algorithms [GPFW97]. Additionally, SLS algorithms are, compared to DPLL algorithms, much simpler in their design.

One major disadvantage of SLS algorithms is that they are incomplete. This means that a SLS solver might not find an assignment, that suffices in solving a formula even though such an assignment might exists [GPFW97]. The explanation for this incompleteness will be given in the next section.

Furthermore, an unsatisfiable instance will never be identified as such by the use of a SLS solver. Their search behavior then simply is an infinite runtime on these instances (or the termination of the algorithm without a result).

However, the usefulness of SLS solvers is undoubted for certain instances. Recall the under-constrained area from section 2.7.2 (see page 22). A SLS solver is most likely to outperform a DPLL solver in this region. This will be explained in the next section as well.

Since the introduction of GSAT [SLM92], a considerable amount of research has been conducted to enhance local search algorithms for SAT. Two major improvements have been proposed [SS01]: the incorporation of *random walks* and *clause weights*.

After the explanation of the basic ideas behind local search and GSAT, we will explain two modern SLS solvers: adaptG^{2}WSAT$_0$ and gNovelty+. The application of random walks will be explained in section 4.3, along with adaptG^{2}WSAT$_0$. The application of clause weights will be explained in section 4.3, along with gNovelty+.

4.1 Local Search and GSAT

4.1.1 Local Search for SAT

Local search itself is a technique to conduct search on a discrete search space [GPFW97]. This technique can be used to solve SAT by introducing an *objective function*. The objective function is a way to measure the quality of a given position in the search space.

Starting from a random position, the algorithm will change its position according to the objective function. It continues in the direction that yields the best improvement, i.e. improves the objective function the most.

This process of changing position and realigning the search direction is repeated, until a solution is found or the search can not improve its position any further. See figure 4.1 for a visualization of the search process.

The search space for SAT is the set of all assignments applicable for a given formula F. A typical objective function for SAT simply counts the number of clauses that are unsatisfied under a given assignment [HS00].

As mentioned in section 2.2, a formula in CNF only evaluates to TRUE under an assignment A, iff all clauses under this assignment evaluate to TRUE. Hence, a formula is solved as soon as an assignment is found under which the number of unsatisfied clauses is zero [HS00]. Local search therefore tries to find an assignment A that is a global minimum (for the objective function) with value zero.

The local search will start with a randomly created assignment A. It will then use the objective function to measure its quality. In most cases, this random starting assignment is not a solution, and thus, the search must continue.

Local search will then alter the starting assignment in a way, that a smaller value for the objective function is reached, thereby changing its position [GPFW97]. Altering the current assignment (changing the position) in the context of SAT is done via *flips*.

Flips are the basic action for local search to proceed. For the rest of this section we consider a problem instance F in CNF with variables $\mathcal{V} = \{x_1, \ldots, x_n\}$.

Definition 22 *Given an assignment $A = (v_1, \ldots, v_n)$. `Flipping` is the process of changing exactly one variable assignment to its opposite value. Let x_i be the variable to be flipped and let $Val_A(x_i) = v_i \in \{0, 1\}$ be its truth value under assignment A. Flipping x_i then results in a new assignment B such that $Val_B(x_i) = \neg Val_A(x_i)$ while all other variables retain their value from assignment A.*

Definition 23 *Given an assignment A and the set of all assignments $\mathcal{N} = \{B_1, \ldots, B_n\}$ that can be generated from A by flipping exactly one variable. We call $\mathcal{N}$ the `neighborhood` of A whereas $B_i \in \mathcal{N}$ is called a `neighbor` of A.*

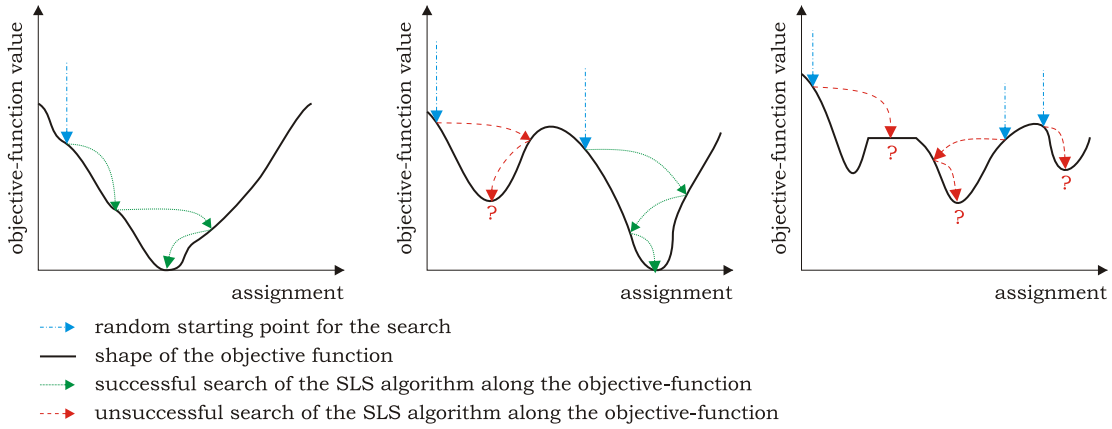


Figure 4.1: The proceeding of a search conducted by a SLS solver.

The choice of what variable is used for flipping is done by checking all possible flips [GPFW97]. Given an assignment A, we compute its neighborhood $\mathcal{N} = \{B_1, \ldots, B_n\}$.

Then, the quality of these new assignments (neighbors) is measured using the objective function. Let $o_i = \text{OBJECTIVEFUNCTION}(B_i)$ be the value, the objective function calculates for a given neighbor B_i. We commit to the assignment B_i for which $o_i < o_j, j \in \{1, \ldots, n\} \backslash \{i\}$ [GPFW97]. In other words, we flip the variable in A that minimizes the number of unsatisfied clauses.

Definition 24 *Given a formula F and an assignment A with its neighborhood $\mathcal{N} = \{B_1, \ldots, B_n\}$. Let $o = \text{OBJECTIVEFUNCTION}(A)$ be the objective function value for A. Let $o_i = \text{OBJECTIVEFUNCTION}(B_i)$ be the objective function values for all its neighbors $B_i \in \mathcal{N}$. A is called a local minimum, iff $o \leq o_i \; \forall i \in \{1, \ldots, n\}$.*

Local search repeats the procedure of improving the quality of its position until it either finds a solution, or gets stuck in a local minimum.

In figure 4.1, three different cases for the behavior of the local search are presented according to [GPFW97]. The left image shows the desired behavior, in which it starts at a random assignment. It then conducts flips until it reaches the global minimum of the objective function, which yields an assignment that solves the underlying formula F.

The center image shows the behavior that occurs whenever the local search gets stuck in a local minimum. Since it is committed to only flip towards a neighbor that improves the objective function value, it can not continue the search at this position. As we can see, the process of getting stuck in a local minimum depends on the random starting position of the search.

The right image visualizes the behavior when searching a solution for an unsatisfiable instance. We can see that this instance must be unsatisfiable, because the objective

function never reaches zero. In other words, whatever assignment we choose, the number of unsatisfied clauses is always greater zero. Therefore, the SLS solver will sooner or later get stuck in a minimum no matter what random starting assignment it chooses.

The right image also shows a special case of neighborhood formation: a *plateau*. Plateaus are sets of assignments that are neighbors and have the same objective function value. Since the SLS solver is committed to always improve the objective function value, it can not continue the search when arriving inside a plateau.

Whenever local search reaches a local minimum or a plateau, it gets stuck and the search is a failure. Therefore, local search is incomplete. This is the basic reason for incompleteness of SLS solvers, whose function rely heavily on local search.

Local search is conducted by improving an assignment. The generation of neighbors via flips, and the testing of the assignment quality via the objective function can be done efficiently [GPFW97]. Therefore, the local search in general can be computed efficiently. Since an assignment has exactly n different neighbors, the memory usage of local search is $O(n)$.

As can be seen from figure 4.1, the shape of the objective function (which depends on the formula) affects the behavior of the local search procedure. The more models a formula has, the higher the chances are that local search will create a random starting assignment in the vicinity of such a model. Since the process of searching is done efficiently, finding a model in the vicinity of such a starting assignment can be done in an efficient way. Therefore, formulas from the under-constrained region (see section 2.7.2 on page 22) are easy to solve for SLS solvers relying on local search.

For more information on local search in the context of SAT see [SLM92, GW93, GPFW97, Hoo99, SS01].

4.1.2 GSAT

GSAT (Greedy SAT) [SLM92] was the first SAT solver to utilize local search in order to find a model for a given formula. It is a straight forward implementation of the local search technique for SAT. However, since GSAT is an incomplete algorithm, we need to restrict its runtime, so it will not run forever. In order to perform this restriction, we need two parameters: `MAX-TRIES` and `MAX-FLIPS`.

`MAX-TRIES` is a number defining how many local searches are to be performed on the formula until the algorithm gives up searching for a model. With each try, a new random assignment is created as a starting point for the search. This is visualized by several "starting points" in figure 4.1. As we can see in the center image, performing more tries enables the search algorithm to investigate more areas of the search space, but must not necessarily yield a model as we see in the right image. The value of `MAX-TRIES` depends on the time one wants to spend for searching a model [SLM92].

`MAX-FLIPS` is a number defining how many flips are allowed before the search is given up in this specific area of the search space. When `MAX-FLIPS` is reached, a new try is started with a new randomly created assignment (if there is a try left). As a rough guideline, setting `MAX-FLIPS` equal to a few times of the number of variables is sufficient [SLM92].

Using these two parameters, one can fine-tune the behavior of GSAT and thereby adapting it to several types of instances [SLM92]. In case there are many local minima, one might want to spend more (but shorter) runs in order to escape from them soon. In case there are just a few but "far-reaching" minima, one might want to spend fewer (but longer) runs in order to be able to explore them completely. However, if the type of instance is unknown, one will most likely make a sub-optimal setting.

The necessary knowledge to understand GSAT has now been given. The algorithm is sketched out in listing 4.1.

Listing 4.1: GSAT.

```
GSAT(F, MAX_FLIPS, MAX_TRIES){
    for (k = 1 to MAX_TRIES){
        A = randomlyCreateAssignment();
        for (j = 1 to MAX_FLIPS){
            if (A is a model for F){
                return A;
            }
            x_i = getVariableToFlip(F, A);
            A = flip(A, x_i);
        }
    }
    return 'UNKNOWN';
}
```

In the previous section, we explained the method to choose a variable in a given assignment for flipping: We only flip a variables value if it yields an improvement w.r.t. the objective function. However, GSAT slightly differs from this approach.

The `getVariableToFlip`(F, A) method will return a variable that improves the objective function the most if existent. But if no such variable exists, a so-called "sideway-move" is allowed [SLM92].

A sideway-move is a variable flip that neither improves nor deteriorates the objective function value. In other words, when conducting a sideway-move, the objective function value stays the same.

The reason to allow these moves is, that the performance of GSAT with sideway-moves is by far better (solves more instances in less time) than that of GSAT without sideway-moves [SLM92].

Additionally, if several variables yield the same amount of improvement, the function `getVariableToFlip`(F, A) selects one of them at random, therefore it is highly unlikely that GSAT produces the same series of changes over and over again [SLM92].

Chances are, that (due to the random nature of restarts) the search will never enter an area where the objective function reaches zero. However, the more restarts are performed, the higher the probability is that GSAT will begin in such a preferable area.

To call GSAT "incomplete" is therefore not appropriate. In [Hoo99] a more formal introduction and distinction between the terms of "complete", "incomplete" and "probabilistic approximately complete" is presented. For the context of this work it is sufficient to know, that GSAT without restarts is incomplete.

However, allowing restarts, GSAT is probabilistic approximately complete. This means, that a model (if existent) will be found by GSAT with probability reaching 1, if `MAX-TRIES` reaches infinity. In practice, this is not applicable but underlines the importance of restarts.

We now have explained local search and presented a simple SLS solver, GSAT, along with its strengths and weaknesses.

Since GSAT was introduced, a number of improvements have been proposed. Many of these improvements deal with the escape from local minima and the selection of the next variable that is to be flipped.

Additionally, a variety of algorithms has been proposed to circumvent the need for parameter-tuning by dynamically adapting their values when a search is conducted. We will now review two modern SLS solvers and explain the features they use to improve the overall performance of local search for SAT.

4.2 adaptG^{2}WSAT$_0$

4.2.1 adaptG^{2}WSAT$_0$ Background

AdaptG^{2}WSAT$_0$ [LWZ07, LWZ06] was developed at the University of New Brunswick in Canada. It participated in solving random instances during the SAT 2007 Competition[1]. AdaptG^{2}WSAT$_0$ was awarded the second place in the "Random SAT" category and is therefore considered to be a state-of-the-art SAT solver.

As mentioned in the introduction to this chapter, SLS solvers can be categorized by the usage of random walks and clause weights.

The solver adaptG^{2}WSAT$_0$ is utilizing random walks (besides other features) to improve local search performance.

Throughout the next section, we will explain the developments that lead to the current version of adaptG^{2}WSAT$_0$. These developments include random walks, newly designed variable selection heuristics, the application of gradients, and an adaptive scheme to dynamically tune parameters of the search.

[1] http://www.satcompetition.org

The remainder of this section is structured as follows. In section 4.2.2, we will explain random walks and how they affect local search for SAT. Then, we explain gradients and how they can be of use when the objective function is computed. In section 4.2.3, we explain why and how search parameters can be adapted dynamically while search is performed. In section 4.2.4, we explain the functioning of adaptG^{2}WSAT$_0$ in detail. This is followed by an explanation of adaptG^{2}WSAT$_0$'s performance during the SAT 2007 Competition.

4.2.2 From GSAT over WalkSAT to G^{2}WSAT

As mentioned in section 4.1.2, the basic GSAT algorithm suffers from getting stuck in local minima. An obvious way to improve this algorithm would be, to change its functioning in a way that makes it less susceptible for them.

In [SKC94], WalkSAT is introduced which tries to escape from local minima by introducing randomness to the decision strategy in which the variable to be flipped is chosen. To explain this, we introduce some terms from [LH05] related to variables.

Definition 25 *Let x be a variable in the currently investigated formula F. Let* `break`*(x) be the number of clauses in F which are satisfied by the current assignment A but would be unsatisfied if x is flipped. Let* `make`*(x) be the number of clauses in F that are currently unsatisfied but would be satisfied if x is flipped. Let* `score`*(x) = make(x) - break(x).*

The randomness mentioned above is introduced in two ways:

1. The decision on what variable is chosen for a flip is no longer based on the complete neighborhood of the current assignment. Instead, we randomly select a clause from the formula that is *unsatisfied* under the current assignment. Let us call this clause c.

2. For c, we perform the `WalkSAT-Heuristic()` as follows. We check if a variable x exists within c, that has break(x) = 0. If such variables exists, we randomly pick one of them for flipping (greedy step). If such a variable does not exists within c, we perform the following action:

   - With probability p randomly pick a variable from c and flip it (random step).
   - With probability $1 - p$ pick a variable y from c such that break(y) is the smallest.

The resulting algorithm WalkSAT is quite similar to GSAT. The main difference between GSAT and WalkSAT is the way a variable is chosen for flipping. WalkSAT is presented in listing 4.2.

Listing 4.2: WalkSAT.

```
WalkSAT(F, MAX-FLIPS, MAX-TRIES,p){
    for (k = 1 to MAX-TRIES){
        A = randomlyCreateAssignment();
        for (j = 1 to MAX-FLIPS){
            if (A is a model for F){
                return A;
            }
            c = randomlyPickUnsatisfiedClause(F, A);
            x_i = WalkSAT-Heuristic(c, p);
            A = flip(A, x_i);
        }
    }
    return 'UNKNOWN';
}
```

The selection of c is done at random. This results in narrowing the decision for what variables are considered for flipping, since only variables that are part of this clause will be checked.

The check is done within the `WalkSAT-Heuristic()`. For this heuristic, the input parameter p is used. This parameter is often called "noise".

In case there is no variable within c, that has no destructive potential (break(x) = 0), the heuristic needs to decide what variable to select. Two options are thinkable: select the variable that has the least destructive potential (break(x) as small as possible) or simply select a variable at random in this clause.

Parameter p now represents the probability that a random move is done. The larger p is, the more random flips we get when performing search (i.e. the search performs less greedy).

On the other hand, the larger p is, the fewer damage limiting flips (choose the variable from c that has smallest break()) are made. If p is too large, the search behavior becomes erratic and the local search degenerates to random checking of assignments. If p is too small, the search still gets stuck in local minima.

The desired effect of randomly selected flips is visualized in figure 4.2

The `WalkSAT-Heuristic`(), as explained above, is not the only way to introduce randomness. In [MSK97], a study is conducted that analyzes different heuristics to choose a variable for flipping. Based on the experimental results, a new heuristic is proposed called Novelty.

Novelty works as follows. Sort the variables x in c by score(x), breaking ties in favor of the least recently flipped variable. Consider the best and second best variable under this sort. If the best variable is not the most recently flipped one in c, then pick it. Otherwise, with probability p, pick the second best, and with probability $1 - p$, pick the best variable.

The effectiveness of Novelty has been studied in [Hoo99]. Even though it performs

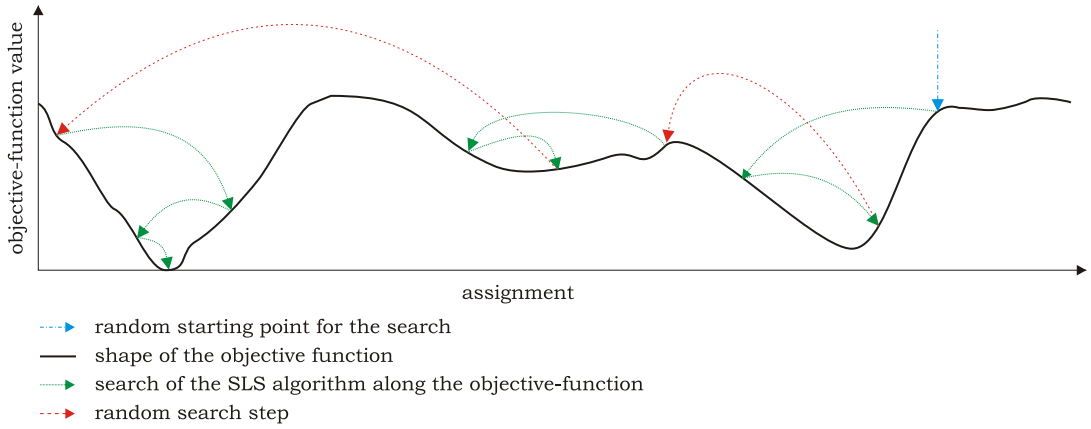


Figure 4.2: The proceeding of a randomness empowered search conducted by a SLS solver.

much better than the standard WalkSAT-Heuristic, Novelty sometimes gets stuck in a loop (fluctuating between the best and second best variable). For a clarification see the following example from [Hoo99].

Example 21 *Consider a formula with the following clauses:*
$c_1 = (\neg x_1 \vee x_2), c_2 = (\neg x_2 \vee x_1), c_3 = (\neg x_1 \vee \neg x_2 \vee \neg y), c_4 = (x_1 \vee x_2), c_5 = (\neg z_1 \vee y), c_6 = (\neg z_2 \vee y)$.

This formula has exactly one model: $x_1 = 1, x_2 = 1, y = 0, z_1 = 0, z_2 = 0$.

If Novelty starts with the random assignment $x_1 = 1, x_2 = 1, y = 1, z_1 = 1, z_2 = 0$, *it never reaches this solution because* y *is never flipped. It either flips* x_i *or* z_i, *depending on the clause* c_j *that is selected.*

A restart of the search could of course remedy this situation, but this feature relies heavily on the setting of `MAX-FLIPS`. The problem is, that an optimal setting for this constant cannot be found easily. Therefore, [Hoo99] proposes a modification of the Novelty heuristic called Novelty+.

In Novelty+, there is a new parameter called wp (for "walk probability") that is used to decide whether a completely random decision for a flip in clause c is to be done, without observing the variable scores.

In other words, with probability wp, randomly pick a variable from c to flip. With probability $1 - wp$, perform as Novelty. Novelty+ therefore has the chance to break free from setting either x_i or z_i repeatedly.

The problem now is, that this breaking free only occurs with a probability wp. But just entering the random selection does not mean that Novelty+ does indeed select the variable from the clause that is not part of the loop.

Take a look at clause c_3 from the previous example. In case this clause is selected and the random step is entered, the selection of y for a flip is only 1/3. It could still (randomly) pick one of the variables that is in the loop.

In [LH05], an attempt is made to circumvent this problem by altering Novelty in a different way. The resulting heuristic is called Novelty++ and performs as follows.

With probability dp (diversification probability), pick the least recently flipped variable in c. With probability $1 - dp$ do as Novelty.

Now breaking free of such a loop is done with probability dp, but since the least recently flipped variable is now used for flipping (which can not be one of the loop variables), it will surely exit the loop.

In the example above, y is flipped when the diversification decision is made in clause c_3.

In [LH05], an empirical study is performed and it shows, that Novelty++ is consistently better than Novelty+ in cases where stagnation behavior in local minima can occur.

However, the authors from [LH05] believe, that the main purpose of a local search procedure should always be a decrease of the number of unsatisfied clauses. Selecting an unsatisfied clause at random, only to see if there is a variable inside that clause that can improve the objective function value, is therefore not the optimal strategy.

It would be much better to select the variable to flip that improves the objective function value the most (as GSAT originally does), independently of the clause it is in. And as soon as there is no such opportunity of improvement, use a heuristic such as Novelty++.

The more variable flips we check, the higher the chance to find a flip that improves the objective function value the most for the current position in the search space. It would be best, if we could check all available flips for their consequences. Therefore, an efficient way to calculate and update many variable scores is needed. To explain this in more detail, we will introduce some terms from [LH05].

Definition 26 *A variable x in a formula F under assignment A is called* `decreasing`*, if flipping it would decrease the number of unsatisfied clauses in F. Or in other words, x is decreasing, when it has a positive score (score(x) = make(x) - break(x)).*

All decreasing variables are collected in a set `DecVar`.

Definition 27 *Let x, y with $x \neq y$ be two variables in F. Let y be a not decreasing variable. If y would become a decreasing variable after x is flipped, we call y a* `promising decreasing variable after` x `is flipped`*.*

Variable y will stay a promising decreasing variable (after x is flipped) as long as it is decreasing during other flips.

Definition 28 *A* `promising decreasing path` *is a sequence of moves in which every move flips a promising decreasing variable.*

Now we need an efficient procedure to compute the scores of all variables to get the decreasing variables for set `DecVar`. Additionally, we need an efficient way to maintain `DecVar`, i.e. a way to remove variables that are not decreasing anymore and adding all variables that become decreasing after a certain flip. This efficient procedure is well explained in [LH05] and works as follows.

Let F be a propositional formula in CNF with m clauses and n variables. Let A be an assignment. Let c_i be a clause in F with

$$c_i = (x_{i_1} \vee x_{i_2} \vee \ldots \vee x_{i_k} \vee \neg x_{i_{k+1}} \vee \ldots \vee \neg x_{i_{k+r}})$$

Note how the literals in c_i are ordered by their signs. The k positive literals are placed in the first k positions, followed by the r negative literals. Now consider all variables in c_i as integer variables taking values 0 or 1 depending on the assignment A. We define:

$$\mathcal{E}_i(x_{i_1}, \ldots, x_{i_k}, x_{i_{k+1}}, \ldots, x_{i_{k+r}}) = (1 - x_{i_1}) \cdot \ldots \cdot (1 - x_{i_k}) \cdot x_{i_{k+1}} \cdot x_{i_{k+r}}$$

Now $\mathcal{E}_i = 0$, iff c_i is satisfied under A. We now define:

$$\mathcal{E}(x_1, \ldots, x_n) = \sum_{i=1}^{m} \mathcal{E}_i \tag{1}$$

The value of $\mathcal{E}$ is the number of unsatisfied clauses in F under A. Let $v_f = Val_A(x_f)$ be the value of x_f under the assignment A. $\mathcal{E}(v_f)$ stands for $\mathcal{E}$ simplified after the substitution of x_f by v_f. $\mathcal{E}(x_f)$ is a linear function of the sole variable x_f (it is linear, because every variable occurs exactly once in a clause). Using Taylor's equation gives us

$$\mathcal{E}(x_f) = \mathcal{E}(v_f) + (x_f - v_f)\frac{\partial \mathcal{E}(x_f)}{\partial x_f} \tag{2}$$

Thereby $\frac{\partial \mathcal{E}(x_f)}{\partial x_f}$ indicates the variation of $\mathcal{E}$ when x_f changes. So we can compute all decreasing variables under A by computing $\frac{\partial \mathcal{E}(A)}{\partial x_f}$ for every variable x_f. Thereby we have

- If $\frac{\partial \mathcal{E}(A)}{\partial x_f} > 0$, then
  - x_f is a decreasing variable, iff v_f decreases.
  - x_f is not a decreasing variable, iff v_f increases.
- If $\frac{\partial \mathcal{E}(A)}{\partial x_f} < 0$, then
  - x_f is a decreasing variable, iff v_f increases.
  - x_f is not a decreasing variable, iff v_f decreases.

In other words: $|score(x_f)| = |\frac{\partial \mathcal{E}(A)}{\partial x_f}|$. The sign of the score depends on the change made to x_f.

Whenever x_f is a decreasing variable, its score is positive (see the scheme above), and negative otherwise. Using this, we can compute the score of every variable x_f, and collect them in `DecVar` iff their score is positive.

Once a variable is flipped, A changes. Lets assume we made t flips already and have A^t as well as all score values for all variables for this assignment.

We now flip variable x_f, which results in A^{t+1}. This affects the scores of the variables, but not necessarily all of them. We need a way to re-compute the scores of only those variables that are affected by flipping x_f. In order to do this, we rewrite equation (2) as:

$$\mathcal{E}(x_1, \ldots, x_f, \ldots, x_n) = \mathcal{E}(x_1, \ldots, v_f, \ldots, x_n) + (x_f - v_f)\frac{\partial \mathcal{E}(x_1, \ldots, x_f, \ldots, x_n)}{\partial x_f}$$

And for any variable $x_g \neq x_f$ we have:

$$\frac{\partial \mathcal{E}(x_1, \ldots, x_f, \ldots, x_n)}{\partial x_g} = \frac{\partial \mathcal{E}(x_1, \ldots, v_f, \ldots, x_n)}{\partial x_g} + (x_f - v_f)\sum_{i=1}^{m} \frac{\partial^2 \mathcal{E}_i}{\partial x_f \partial x_g} \quad (4)$$

The value of x_j for A^t is v_j^t. When moving from A^t to A^{t+1}, we have $v_j^t = v_j^{t+1}$ except for one variable $x_f = v_f^{t+1} = 1 - v_f^t$. Because we previously computed $\frac{\partial \mathcal{E}(x_j)}{\partial x_j}$ for every x_j at point A^t (in equation 2), we can use this value for an update of $\frac{\partial \mathcal{E}(x_j)}{\partial x_j}$ at point A^{t+1}. At this point

- $\frac{\partial \mathcal{E}(x_1,\ldots,x_f,\ldots,x_n)}{\partial x_g}$ is $\frac{\partial \mathcal{E}(A^{t+1})}{\partial x_g}$
- but $\frac{\partial \mathcal{E}(x_1,\ldots,v_f^t,\ldots,x_n)}{\partial x_g}$ is $\frac{\partial \mathcal{E}(A^t)}{\partial x_g}$, since A^{t+1} differs from A^t only in the value of x_f.

Since $x_f - v_f^t = v_f^{t+1} - v_f^t = 1 - v_f^t - v_f^t$ at point A^{t+1}, equation (4) becomes:

$$\frac{\partial \mathcal{E}(A^{t+1})}{\partial x_g} = \frac{\partial \mathcal{E}(A^t)}{\partial x_g} + (1 - 2v_f^t)\sum_{i=1}^{m} \frac{\partial^2 \mathcal{E}_i}{\partial x_f \partial x_g}$$

Now here we have to make an important note: $\sum_{i=1}^{m} \frac{\partial^2 \mathcal{E}_i}{\partial x_f \partial x_g} = 0$ for all x_g not occurring in any clause containing x_f. This means, that the score of a variable x_g, that has no relation with x_f (i.e. they are never together in one clause), will not change (i.e. is not influenced) when x_f is flipped. This greatly reduces the computational overhead to update the scores of a variable after a flip, since we only have to re-compute the scores of variables that occur in clauses that also contain x_f.

To summarize, assume we are at point A^t and go towards A^{t+1} by flipping x_f, we re-compute the variable scores:

$$\frac{\partial \mathcal{E}(A^{t+1})}{\partial x_g} = \begin{cases} \frac{\partial \mathcal{E}(A^t)}{\partial x_g}, \text{if } x_g \text{ not occurring in any clause with } x_f \\ \frac{\partial \mathcal{E}(A^t)}{\partial x_g} + (1 - 2v_f^t) \sum_{x_f, x_g \in c_i} \frac{\partial^2 \mathcal{E}_i}{\partial x_f \partial x_g}, \text{if } x_g, x_f \text{ have a relation} \\ \frac{\partial \mathcal{E}(A^t)}{\partial x_f}, \text{if } x_g = x_f \end{cases} \quad (6)$$

Equation (6) is used to maintain `DecVar`. To be precise, `DecVar` contains all variables, that have a positive score after equation (6) finished computation.

Let `MAINTAIN()` denote the procedure that uses equation (6) to efficiently maintain the variable scores of all variables.

`MAINTAIN()` represents a *gradient-based* variable score update scheme. The gradient $\nabla\mathcal{E}$ is a vector with n components (the number of variables). Component i of $\nabla\mathcal{E}$ is $\frac{\partial \mathcal{E}(A^{t+1})}{\partial x_i}$. One could say that $\nabla\mathcal{E}$ just holds all variable scores for the position A in the search space.

We have now introduced the WalkSAT architecture (see listing 4.2), and explained various heuristics that WalkSAT can use to select a variable to flip (like Novelty++). Furthermore, we explained the gradient-based variable score update scheme. In [LH05], these techniques are used to introduce a new SAT solver, called G^{2}WSAT (for Gradient-based Greedy WalkSAT). The functioning of that solver is outlined in listing 4.3

Listing 4.3: G2WSAT.

```
G2WSAT(F, MAX-FLIPS, MAX-TRIES, p, dp){
    for (k = 1 to MAX-TRIES){
        A = randomlyCreateAssignment();
        Compute ∂E(x_1,...,x_n)/∂x_j for all x_j at A;//init variable scores
        DecVar = all variables with positive score;
        for (j = 1 to MAX-FLIPS){
            if (A is a model for F){
                return A;
            }
            if (DecVar is not empty){
                //greedy step
                x_i = the variable in DecVar with maximum score
            } else {
                //random step
                c = randomlyPickUnsatisfiedClause(F, A);
                x_i = Novelty++(c, p, dp);
            }
            A = flip(A, x_i);
            MAINTAIN(); //update variable scores
            update DecVar;
            //By removing variables with non-positive scores.
            //Pushing new variables (≠ x_i) into DecVar,
            //that have a positive score now.
        }
    }
    return 'UNKNOWN';
}
```

One should note, that `DecVar` is actually a queue-like list, with the oldest decreasing variable at its beginning. For now, this has no consequence, since we always choose the variable with highest score in this list (greedy step).

If `DecVar` is empty (because the algorithm has arrived in a local minimum), the Novelty++ heuristic is used to escape from that minimum. G^{2}WSAT is therefore quite similar to GSAT, but it improves GSAT in two ways.

First, a strategy to escape local minima is integrated via Novelty++. Second, the way that the next flipping variable is chosen is optimized.

In section 4.1.2 we said, that a variable is to be flipped, if it improves the objective function value the most. The naïve way to implement this has been explained via the neighborhood $\mathcal{N}$ (see section 4.1.1), where one just has to compute all neighbors of the current assignment and chose the one that improves the objective function the most (thereby choosing which variable to flip).

The problem hereby is, that one has to create all these neighbors, and then, for all n neighbors, has to count the number of unsatisfied clauses (i.e. has to iterate over all m clauses to check if they are satisfied). This results in a time complexity of $O(nm) = O(n \cdot 4.26 \cdot n) = O(n^2)$ for each variable selection.

With the new scheme used in `MAINTAIN()`, the algorithm has to iterate over the r ($\leq n$) variables in `DecVar`, to select the one with the highest score. Furthermore, it has to update the scores of all s ($s \leq n$) variables, that occur in a clause that also holds the variable that has been chosen for a flip. In general, r and s are much smaller than n, and therefore, a speedup is gained. `MAINTAIN()` has a time complexity of $O(n + n) = O(n)$ for each variable selection.

As we can see from listing 4.3, several parameters are needed as input for this solver. Besides the formula F that is to be solved, we need `MAX-TRIES` and `MAX-FLIPS` in order to define how long we want to search and how many flips per search we want to allow.

The other parameters needed for G^{2}WSAT differ according to the used variable selection heuristic. When Novelty++ is used (as shown in listing 4.3) we need parameters that define the diversification probability dp and the noise p. In case no diversification step is performed in Novelty++, it behaves like Novelty. Novelty needs p to decide whether the best ore second best variable in a selected clause are chosen (with respect to variable scores).

We have already mentioned, that it is problematic to tune p (or dp likewise) towards an optimal value. For unknown instances, an optimal value can not be known a priori. Therefore, a dynamic parameter tuning scheme has been developed by [Hoo02], that adapts the parameters for the variable selection heuristic (like Novelty or Novelty++) while search is performed. This scheme will be explained next.

4.2.3 A Dynamic Parameter Tuning Scheme

The dynamic parameter tuning scheme presented in [Hoo02] is supposed to adapt the parameters involved in the variable selection heuristics dynamically while search is performed. To be precise, the noise value p is dynamically adapted.

The approach used is based on the following idea [Hoo02]. It appears that optimal noise settings are those that achieve a good balance between an algorithms ability to greedily find solutions by following the score gradient and the ability to escape from local minima. It appears reasonable to use the escape mechanism only when it is needed. Therefore it appears advantageous to use the following scheme.

At the beginning of the search, we use exhaustive greedy search only ($p = 0$). This will probably (unless a solution is found) result in a stagnation of the search in a local minimum. When this happens, the noise value must be increased in order to make the escape from a local minimum possible.

After increasing the noise value, we need to detect if this was sufficient for an escape. Therefore, we continue the search and monitor the further development of the objective function. If in a certain number of steps, the objective function value is not improved any more, we need to further increase the noise (since we did not yet escape from the minimum).

At some point, the noise value will be large enough to enable the escape from the current local minimum. After the escape, the noise can gradually be decreased, making the search more greedy again. The values, of how many steps are performed until we decide on the success of the search to escape from the minimum, as well as the values on how much the noise is increased or decreased, are computed as follows.

Given two parameters Θ and Φ. Let m be the number of clauses in the formula F that is currently searched.

- If the search did not improve the objective function value in the last Θm steps,
  - increase the noise according to $p := p + (1 - p) \cdot \Phi$.
- Whenever the objective function value is improved,
  - decrease the noise according to $p := p - p \cdot \Phi/2$.

While it appears that we just replaced the problem of tuning p with the tuning of Θ and Φ, the two latter parameter values where obtained in an empirical study [Hoo02] and can be held fixed. The advantage now is, that the values for $\Theta = 1/6$ and $\Phi = 0.2$ need not to be tuned for any single instance anymore, but the application of the scheme described above allows for a dynamic adaption of the noise while search is performed on a specific instance.

We already mentioned, that additional parameters (like dp) are sometimes needed, depending on the used variable selection heuristic. In [LWZ06], an automatic ad-

justment for an additional parameter is done via the (dynamic) noise value p. For example, using Novelty+ (requires the noise p and a walk probability wp), the wp parameter is calculated via $wp := p/10$. Similar calculations are thinkable for other parameters needed for other heuristics.

Furthermore, we need to decide on how to set the `MAX-FLIPS` and `MAX-TRIES` parameters. `MAX-TRIES` still depends on how much time the user wants to spend for a search, and therefore, depends on the application.

Furthermore, it has been shown [PW96, Hoo99], that `MAX-FLIPS` has little or no impact on the search performance when the noise setting p is set sufficiently high. One must only assure that `MAX-FLIPS` is not set to a too low value. As mentioned in section 4.1.2, setting `MAX-FLIPS` equal to a few times of the number of variables is sufficient.

Now that we have explained the functioning of the basic G^{2}WSAT algorithm and how dynamic parameter tuning can be done, we will continue with an explanation on how this adaptive parameter tuning can be included in G^{2}WSAT (yielding adaptG^{2}WSAT$_0$).

4.2.4 The functioning of adaptG^{2}WSAT$_0$

AdaptG^{2}WSAT$_0$ is explained in [LWZ06], as a variant of adaptG^{2}WSAT$_P$. Both algorithms differ only in the variable selection heuristic, that is used (adaptG^{2}WSAT$_0$ uses Novelty++', adaptG^{2}WSAT$_P$ uses Novelty+$_P$). adaptG^{2}WSAT$_0$ is very similar to the previously introduced G^{2}WSAT. They differ in the used variable selection heuristic and the incorporation of adaptive tuning for the noise parameter.

Furthermore, the decision on what variable is used for flipping is no longer determined only by the score of a variable. In G^{2}WSAT, the variable with highest score is used for flipping. In adaptG^{2}WSAT$_0$, the least recently flipped variable that is still a promising decreasing variable is used for flipping.

Since `DecVar` is a queue-like list, the least recently flipped promising decreasing variable is always found at its beginning. The reason for not searching for *the* best promising decreasing variable is, that all promising decreasing variables in `DecVar` usually have about the same score and thus yield about the same improvement of the objective function value [LWZ06].

Flipping the least recently flipped promising decreasing variable can improve the mobility and coverage [SS01] of a local search algorithm in the search space, which adds to the performance. This will later be explained in more detail.

Novelty++'(p, dp, c) works as follows [LWZ07]. With probability dp (diversification probability) exclude the best and second best variables in c (with respect to scores) and then randomly pick a variable of the remainders in c. With probability $1 - dp$, do as Novelty (i.e. with probability p use the best variable in c and with probability

$1 - p$ pick the second best variable in c with respect to scores).

Let `ADAPT(`p`)` be a procedure responsible for the parameter adaption as described in section 4.2.3. As mentioned before, dp is adapted using p ($dp = p/10$). The reason for this is, that when noise needs to be high, local search should also be well randomized, and when low noise is sufficient, random moves in the search space (i.e. random walks) are often not necessary.

The functioning of adaptG^{2}WSAT$_0$ is outlined in listing 4.4.

Listing 4.4: AdaptG2WSAT0.

```
ADAPTG2WSAT0(F, MAX_FLIPS, MAX_TRIES){
    for (k = 1 to MAX_TRIES){
        A = randomlyCreateAssignment();
        Compute ∂E(x_1,...,x_n)/∂x_j for all x_j at A;//init variable scores
        DecVar = all variables with positive score;
        p = 0;
        dp = 0;
        for (j = 1 to MAX_FLIPS){
            if (A is a model for F){
                return A;
            }
            if (DecVar is not empty){
                //greedy step
                x_i = the least recently flipped variable
                        in DecVar.
            } else {
                //random step
                c = randomlyPickUnsatisfiedClause(F, A);
                x_i = Novelty++'(p, dp, c);
            }
            A = flip(A, x_i);
            ADAPT(p); //and set dp = p/10;
            MAINTAIN(); //update variable scores
            update DecVar;
            //By removing variables with non-positive scores.
            //Pushing new variables (≠ x_i) into DecVar,
            //that have a positive score now.
        }
    }
    return 'UNKNOWN';
}
```

We have now explained the functioning of adaptG^{2}WSAT$_0$ and will continue with an explanation of its performance during the SAT 2007 Competition.

4.2.5 Explaining the Performance of adaptG^{2}WSAT$_0$

As mentioned in the introduction to this section, adaptG^{2}WSAT$_0$ participated in solving random instances during the SAT 2007 Competition[2]. Since adaptG^{2}WSAT$_0$ is an incomplete solver, it was only able to show the satisfiability of certain instances,

[2] http://www.satcompetition.org

and therefore, only participated in the "Random SAT" category.

In this category, it was able to solve 248 of the 511 instances, which was the second most amount in total. The needed solving times allowed it to award adaptG^{2}WSAT$_0$ with the second place in this category.

Since adaptG^{2}WSAT$_0$ did not participate in the handmade or industrial categories, no results can be reported here.

We will give some explanations for the performance of adaptG^{2}WSAT$_0$ on random instances in the remainder of this section.

The Benefits of the Gradient-based Score Update Scheme

As mentioned before, the gradient-based score update scheme is used to maintain the scores of all variables of a formula. We have seen, that this scheme only re-computes scores when it is necessary, and otherwise leaves the scores untouched.

This in turn saves computation time when the set of decreasing variables is to be maintained, while retaining the ability to have a complete overview of all the current variable scores.

The Benefits of using the Least Recently Flipped Variable for Flipping

The list of decreasing variables (i.e. those that can improve the objective function value) are saved in a queue-like list called `DecVar`. Whenever a variable is to be chosen for flipping, adaptG^{2}WSAT$_0$ simply uses the least recently flipped one in `DecVar`.

The reason for this is, that all variables saved in `DecVar` usually have about the same score and thus yield about the same improvement of the objective function value [LWZ06]. Since the least recently flipped variable is always found at the beginning of `DecVar`, no further computation is needed for choosing a variable after the variable scores have been computed.

Furthermore, choosing the least recently flipped variable has the advantage of improving the mobility and coverage of adaptG^{2}WSAT$_0$. To clarify this, consider the following terms from [SS01].

Definition 29 *The* `mobility` *measures, how rapidly a local search moves in the search space.*

Mobility is calculated via the Hamming distance between variable assignments that are k steps apart in a search sequence. Hereby, the Hamming distance is defined as follows.

Definition 30 *Given two assignments A and B. Recall the definition for assign-*

ments, where we understand an assignment as a tuple with n components. The `Hamming distance` *between the two tuples A and B is the number of components in which they differ.*

With such a sequence given, we average a quantity over the entire search sequence to obtain average distances at time lags $k = 1, 2, 3, \ldots$, etc. It is desirable to obtain a large value of mobility since this indicates that the search is moving rapidly through the search space.

Definition 31 *The* `coverage` *measures, how systematically a local search explores the entire search space.*

Coverage is calculated by first estimating the size of the largest unexplored "gap" in the search space (given by the maximum Hamming distance between any unexplored assignment and the nearest explored assignment) and the measuring how rapidly the largest gap size is being reduced. In particular, the coverage is defined to be $(n - \text{maxgap})(\text{searchsteps} - n)$. It is desirable to have a high coverage, since this indicates that the search is systematically exploring new regions of the search space.

Therefore, adaptG^{2}WSAT$_0$ benefits from using the least recently flipped variable in `DecVar`, since this makes its search more systematic and more rapid.

Selecting the least recently flipped variable means, that we do not take *the* best decreasing variable (that would improve the objective function value the most). However, since all variables in `DecVar` have about the same score, this is not a big disadvantage in terms of search greediness.

The Benefits of using an adaptive Parameter Tuning Scheme

We have mentioned the parameter tuning scheme to dynamically adapt p (and dp or wp). This scheme needs the parameters Θ and Φ, which are held fixed during the search. The settings $\Theta = 1/6$ and $\Phi = 0.2$ have been evaluated in [Hoo02].

The advantage now is, that the noise p and additional parameters like the walk probability wp or diversification probability dp (depending on the used variable selection heuristic) are dynamically adapted during the search. This makes parameter tuning for certain instances superfluous, while enabling the search to reach nearly optimal results.

Therefore, adaptG^{2}WSAT$_0$ benefits from an adaptive and thus nearly optimal parameter setting for each search, even if the parameters vary greatly between different instances.

The Benefits of using Randomness only when it is needed

As mentioned before, the behavior of local search should always aim for an improvement of the objective function value (i.e. should behave greedily) [LH05]. Randomness must therefore be used only when it is appropriate (i.e. when search must overcome a local minimum).

Since adaptG^{2}WSAT$_0$ only applies randomness when `DecVar` is empty (i.e. when no improvements are possible), its greedy search is only intermitted when it can not yield improvements anyway. One can say that adaptG^{2}WSAT$_0$ acts as greedy as possible, which in turn circumvents "useless" random steps. This in turn results in smaller solving times.

We have mentioned, that adaptG^{2}WSAT$_0$ uses Novelty++', whenever randomness is needed. This variable selection heuristic has the best trade-off between using randomness and the ability to break free from looping assignments that are repeated over and over again. In summary, the application of Novelty++' is most optimal for the application of randomness and therefore, the escape from local minima.

We have now finished our explanations for why adaptG^{2}WSAT$_0$ did perform well on random satisfiable instances. The next section will summarize our review on adaptG^{2}WSAT$_0$.

4.2.6 Summary of the Review of adaptG^{2}WSAT$_0$

AdaptG^{2}WSAT$_0$ is a randomized SLS incomplete SAT solver, that has proven to be very competitive during the SAT 2007 Competition.

AdaptG^{2}WSAT$_0$ employs a gradient based variable score update scheme, which allows it to efficiently maintain and use variables for greedy search (see section 4.2.2 on page 101). Additionally, this gradient based approach allows for efficiently detecting whether search got stuck in a local minimum.

AdaptG^{2}WSAT$_0$ uses randomness to escape from local minima, by using the Novelty++' variable selection heuristic. However, in order to keep randomness as minimal as possible, adaptG^{2}WSAT$_0$ calls for this heuristic only when no improvements can be made for a certain position in the search space (see section 4.2.4 on page 110).

In order to combat the necessity for parameter tuning, adaptG^{2}WSAT$_0$ employs a dynamic parameter tuning scheme. This scheme allows for nearly optimal search behavior, while working completely autonomous (see section 4.2.3 on page 109).

We have now finished our review of the adaptG^{2}WSAT$_0$ SLS solver and will continue with gNovelty+ in the next section.

4.3 gNovelty+

4.3.1 gNovelty+ Background

gNovelty+ [PG07] has been developed by the Queensland Research Lab and won the first place in the "Random SAT" category during the SAT 2007 Competition[3]. gNovelty+ is therefore considered one of todays state-of-the-art SAT solvers.

gNovelty+ is actually an improvement of G^{2}WSAT (as introduced in section 4.2.2), that has been derived by the incorporation of several improvements. In order to explain gNovelty+, we will first repeat G^{2}WSAT in short.

Then, we will explain the improvements for G^{2}WSAT, that have been proposed by [PG07].

After that, we show how these improvements have been incorporated in G^{2}WSAT in order to yield the new gNovelty+ solver.

After gNovelty+ and its functioning have been explained in detail, we will take a more detailed look at the performance of gNovelty+ during the SAT 2007 Competition.

The review of gNovelty+ is concluded by a summary of its most important features.

4.3.2 G^{2}WSAT as a Basis for gNovelty+

We have explained the functioning of G^{2}WSAT in section 4.2.2. We will shortly repeat it here, such that the reader can better recall its functioning. This is supposed to make it easier to understand the improvements that have been proposed in [PG07] in order to develop gNovelty+.

G^{2}WSAT is a WalkSAT variant, that incorporates a gradient based variable score update scheme to compute the objective function. The scores for all variables are then used to build a set called `DecVar`, which contains all variables that can be used for a greedy search step. G^{2}WSAT will then use the least recently flipped variable in `DecVar` to perform the next flip and continue search.

Whenever `DecVar` is empty, no improvements are possible from the current position in the search space. G^{2}WSAT will then call for the Novelty++ heuristic to escape from the encountered local minimum.

The functioning of G^{2}WSAT is summarized in listing 4.5.

[3] http://www.satcompetition.org

Listing 4.5: The G2WSAT procedure.

```
G2WSAT(F, MAX-FLIPS, MAX-TRIES, p, dp){
    for (k = 1 to MAX-TRIES){
        A = randomlyCreateAssignment();
        Compute ∂E(x_1,...,x_n)/∂x_j for all x_j at A;//init variable scores
        DecVar = all variables with positive score;
        for (j = 1 to MAX-FLIPS){
            if (A is a model for F){
                return A;
            }
            if (DecVar is not empty){
                //greedy step
                x_i = the variable in DecVar with maximum score
            } else {
                //random step
                c = randomlyPickUnsatisfiedClause(F, A);
                x_i = Novelty++(c, p, dp);
            }
            A = flip(A, x_i);
            MAINTAIN(); //update variable scores
            update DecVar;
            //By removing variables with non-positive scores.
            //Pushing new variables (≠ x_i) into DecVar,
            //that have a positive score now.
        }
    }
    return 'UNKNOWN';
}
```

4.3.3 Proposed Improvements for G^{2}WSAT

[PG07] examined the performance of G^{2}WSAT in the SAT 2005 Competition and concluded, that G^{2}WSAT has been the best solver for random 3-SAT instances. However, on random 5-SAT and 7-SAT instances, other solvers (like for example R+AdaptNovelty+) have been outperforming G^{2}WSAT. This leads to the question, on what techniques have been applied by such solvers to outperform G^{2}WSAT on these instances. One of these techniques is the application of a different variable selection heuristic. This new variable selection heuristic will be explained next.

A Better Variable Selection Heuristic

The performance of every WalkSAT variant critically depends on the setting of the noise parameter p [PG07]. This parameter controls the level of greediness for a WalkSAT solver. We have already introduced the dynamic parameter update scheme from [Hoo02] in order to adapt the noise value p dynamically while searching (see section 4.2.3).

This scheme uses two fixed parameters ($\Phi = 1/5$ and $\Theta = 1/6$) to update p. Whenever the search did not improve the objective function value in the last Θm steps,

it will increase the noise according to $p := p + (1 - p) \cdot \Phi$. Whenever the objective function value is improved, p is decreased according to $p := p - p \cdot \Phi/2$.

Which variable is selected for flipping is still depending on the used variable selection heuristic using p. Even though this heuristic is only called whenever search gets stuck in a local minimum, it can have great impact on the search performance.

Combining the parameter update scheme from above with the Novelty+ heuristic is called AdaptNovelty+. It works as follows.

Given a walk probability wp, a noise parameter p, and Φ, Θ for noise updates.

- With probability wp:
  - select a variable x, that appears in an unsatisfied clause (under the current assignment) at random and flip it.
- With probability $1 - wp$:
  - pick an unsatisfied clause c at random. Sort the variables x_i in c by score(x_i). Consider the best and second best variable under this sort.
  - If the best variable is not the most recently flipped one in c
    - then pick it and flip it.
    - else, with probability p, pick the second best, and with probability $1-p$, pick the best variable.
  - Adapt the noise p according to the parameter update scheme.

When explaining the parameter update scheme in section 4.2.3, we noted that it is possible to update additional heuristic parameters like wp in combination with p (for example setting $wp = p/10$). However, this is not done by the AdaptNovelty+ heuristic. In this heuristic, wp is a fixed constant that is never changed.

[PG07] concluded, that the R+AdaptNovelty+ solver [APSS05], which applies AdaptNovelty+ is outperforming G^{2}WSAT on random 5-SAT and 7-SAT instances. They therefore proposed, that G^{2}WSAT should also use AdaptNovelty+. So for their new gNovelty+ solver, they replaced the original Novelty++ heuristic in G^{2}WSAT with AdaptNovelty+.

An additional technique to improve the performance of G^{2}WSAT is the application of clause weights [PG07]. Clause weights will be explained next.

Incorporating Clause Weights

The original G^{2}WSAT uses variable scores in order to decide which of these variables improve the objective function. The score of a variable x is defined to be the difference between the number of clauses that get satisfied when x is flipped and the

number of clauses that get unsatisfied when x is flipped. In short:

$$\text{score}(x) = \text{make}(x) - \text{break}(x) \Leftrightarrow \text{score}(x) = \sum_{x \in c_i (\text{gets satisfied})} 1 - \sum_{x \in c_i (\text{gets unsatisfied})} 1.$$

A clause is therefore counted with a basic value of 1. Adding clause weights can be used to get a more sophisticated variable score, depending on the clause weights of the clauses the variable appears in. Therefore, each clause c_i gets a weight w_i. We then calculate

$$\text{score}(x) = \text{make}(x) - \text{break}(x) \Leftrightarrow \text{score}(x) = \sum_{x \in c_i (\text{gets satisfied})} w_i - \sum_{x \in c_i (\text{gets unsatisfied})} w_i.$$

This will result in a weighted count of clauses, which in turn results in a *weighted objective function*. This weighted objective function will calculate the variable score in dependency of certain clause weights.

One can think of the clause weights as priorities of clauses. When an unsatisfied clause has a high weight, any variable within this clause will receive a high *make*-score. Therefore, the chances of flipping such a variable will be higher, which in turn raises the chance that this clause gets satisfied.

On the other hand, if a variable is within a satisfied clause with high weight, it will receive a high *break*-score. Therefore, the chances of flipping such a variable will be lower, which in turn makes it more unlikely that the corresponding clause gets falsified.

In summary, the higher the weight of an unsatisfied clause is, the higher the chances are, that this clause get satisfied by flipping a variable it contains.

The question now is, on how to calculate the clause weights w_i. Obviously, clauses that are often unsatisfied should receive a higher weight, since satisfying them becomes more important. However, clause weights must be aligned from time to time in order to avoid an escalation of clause weights and an oscillating behavior of the search.

To clarify this, consider an ongoing search where certain clauses have already received a high weight. Let us assume we collect these clauses in a set C_a. The clauses in C_a will be the subject for the next flip in order to get satisfied.

As soon as these clauses have been satisfied, other clauses will come to the fore. Let us call this new set of clauses C_b. Their weights will increase over time. As a result, the clauses from C_b now receive a higher weight than those of C_a. The main priority for the solver then is to satisfy the clauses from C_b, ignoring those of C_a.

Probably, the solver will reassign variables to satisfy all clauses in C_b, which might result in a failure to satisfy the clauses from C_a. Then, the weights for the clauses from C_a will raise again, which gives them the new focus.

Altogether, the algorithm will oscillate between the solving of clauses from C_a and

C_b. During this back and forth, the clause weights of all the clauses from C_a and C_b will continuously raise to giant values, without being of any help.

In order to circumvent this oscillating behavior during the search, all clause weights are re-aligned from time to time. This re-aligning is called *smoothing* of clause weights.

The clause weighting scheme we are interested in was introduced in a solver called SAPS (for Scaling and Probabilistic Smoothing) [HTH02]. In this solver, all clause weights of clauses in the investigated formula F are initialized to 1. Then, weight updates are performed after each flip. This update was based on a multiplicative scheme as follows.

After each flip, the weight for every unsatisfied clause is multiplied by a constant α. This constant is provided by the user and gives the opportunity to fine-tune SAPS for different instances. The weight for satisfied clauses are left unchanged.

Reducing clause weights is done via a probabilistic strategy using a constant called *sp* (for "smoothing probability"). After a search step and after updating the weights of unsatisfied clauses, the algorithm performs smoothing with probability *sp* of *all* clauses. Therefore, it calculates

$$w_i := w_i \cdot \rho + (1 - \rho) \cdot \overline{w}.$$

Whereas $\overline{w}$ is the average over all clause weights and ρ is the smoothing factor (default value is $\rho = 0.99$).

However, in [TPBF04], a study was conducted in order to find out whether there is an alternative way to update clause weights. They concluded, that an additive weight update scheme tends to perform better than a multiplicative scheme on larger and more difficult problem instances. This additive weight update scheme was introduced with a solver called PAWS (for Pure Additive Weighting Scheme) [TPBF04].

The additive scheme performs like the multiplicative one explained above, but instead of multiplying weights with certain constants, it only adds or subtracts 1 (when increasing or smoothing clause weights, respectively). In the additive scheme, we therefore calculate

$w_i := w_i + 1$ for the weight increase of unsatisfied clauses c_i, and
$w_i := w_i - 1$ for the smoothing of clause weights for all clauses.

It is worth nothing, that the minimum for a clause weight is smaller 1, meaning that a clause weight will only be smoothed (reduced by 1) if it is greater 1.

Besides better performance on larger and more difficult problems, the additive scheme also has the advantage, that neither the parameter α nor the parameter ρ must be provided by the user. Only *sp* is needed for this scheme.

We have now explained how clause weights are applied in a local search solver. The question now is why this is a useful technique. In order to understand this, we

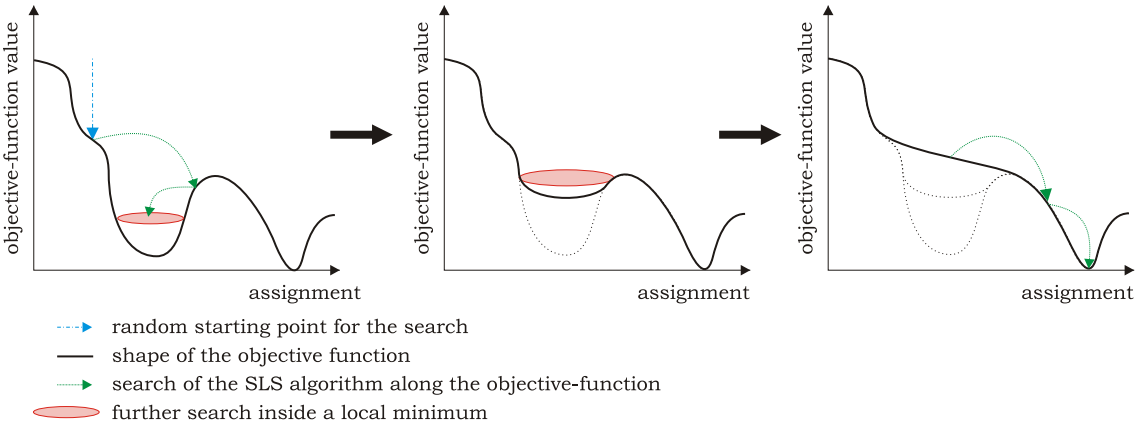


Figure 4.3: How clause weights affect the shape of the weighted objective function (and thereby the search behaviour).

provide the following example.

Example 22 *Consider an ongoing search on an arbitrary formula F. Furthermore consider, that this search has moved into a local minimum. Given the possibility, that no random walk is used (which appears only with probability wp), search will be trapped in this local minimum. See figure 4.3 (left).*

Since search will continue within this local minimum, clause weights of clauses, that are often unsatisfied (i.e. that define this local minimum), will become greater over time. To be exact, the weight of a clause is raised by 1 every time it is unsatisfied. This affects the shape of the weighted objective function (see figure 4.3 (center)).

When a sufficient amount of time has passed, the clause weights will become large enough to make the solver flip a variable within such an unsatisfied clause (with high weight). This is probably a variable that has (up to this point) not been very attractive (i.e. had a small score). This will then guide the search into a different region of the search space (see figure 4.3 (right)).

Even though the formula F is never changed during search, the shape of the objective function changes (due to change of clause weights). This change is visualized in the images of figure 4.3. As we can see, search is gently guided out of the local minimum when certain clause weight have become large enough.

In contrast to random walks, the mechanism of clause weights will generate knowledge (in form of clause weights), that indicate what area of the search space should be considered more important. This in turn helps to guide the greedy search without drastically interrupting its exploration of a certain search space portion (as random walks would do).

We have now introduced the major changes that gNovelty+ introduces to G^{2}WSAT. We will continue by giving a more detailed explanation on how gNovelty+ conducts search.

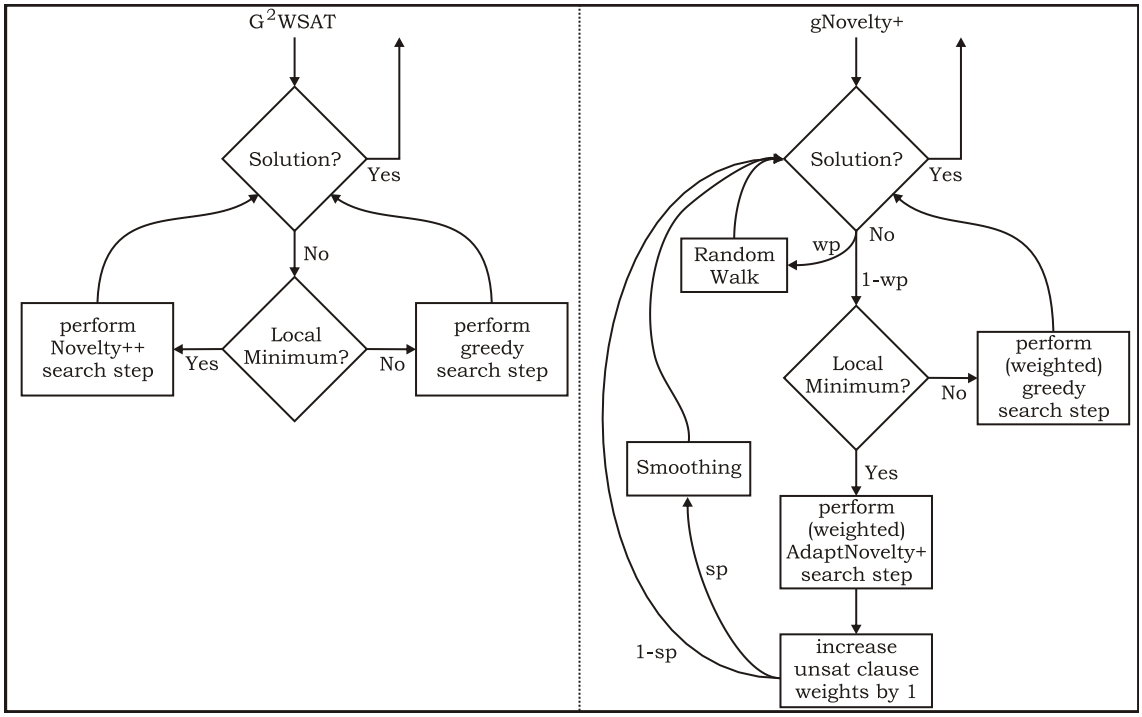


Figure 4.4: Comparing the functioning of G^{2}WSAT and gNovelty+.

4.3.4 The functioning of gNovelty+

In order to understand the functioning of gNovelty+ in detail, it is helpful to figuratively clarify its activity. In [PG07], a figure comparing G^{2}WSAT and gNovelty+ is given (see figure 4.4).

When gNovelty+ starts its search, it will first of all initialize all clause weights to 1. Then, it will generate a random starting assignment and calculate all variable scores. After this, the search procedure (as expressed in figure 4.4) is started.

At first, gNovelty+ checks whether the given assignment is a solution. If so, search is finished. If not, a new search step is performed.

In each search step, gNovelty+ will first perform a random walk with walk probability wp (default value for wp is 0.01).

If no random walk is to be performed, gNovelty+ will check whether it has arrived in a local minimum. This is done by checking whether the set of decreasing variables (`DecVar`) is empty. If this is not the case, the least recently flipped decreasing variable is chosen for flipping.

If no such variable exists, gNovelty+ calls for the AdaptNovelty+ heuristic. This heuristic then picks an unsatisfied clause c at random. For this clause it sorts the contained variables x_i by their scores. If the best variable in this clause is not the most recently flipped one, AdaptNovelty+ will use it for flipping.

If the best variable is the least recently flipped one, AdaptNovelty+ uses the noise p to select a variable. With probability p, it picks the second best variables in c. With probability $1 - p$, it flips the best variable.

After the AdaptNovelty+ heuristic finishes, the weights of clauses will be updated. Therefore, the weight of all unsatisfied clauses are raised by 1.

When this update finishes, a smoothing is performed with probability sp (default value for sp is 0.40). For this smoothing operation, the weight of all clauses (with weight greater 1) are reduced by 1.

Then, gNovelty+ will flip the selected variable, update p according to the parameter update scheme (see page 116), and maintain variable scores and `DecVar`.

After this, a new search step is started. gNovelty+ is summarized in listing 4.6.

Listing 4.6: The gNovelty+ procedure.

```
GNOVELTY+(F, MAX_FLIPS, MAX_TRIES, wp, sp){
    for (k = 1 to MAX_TRIES){
        A = randomlyCreateAssignment();
        Compute ∂ε(x_1,...,x_n)/∂x_j for all x_j at A;//init variable scores
        DecVar = all variables with positive score;
        for (j = 1 to MAX_FLIPS){
            if (A is a model for F){
                return A;
            }
            if (within a walk probability wp){
                x = randomly select a variable from an unsat clause c;
            } else if (DecVar is not empty){
                //greedy step
                x_i = the least recently flipped variable
                        in DecVar;
            } else {
                //AdaptNovelty+ step
                x = AdaptNovelty+(p);
                //Update clause weights
                for (the weight w_i of unsat clauses){
                    w_i = w_i + 1;
                }
                if (within a smoothing probability sp){
                    for (the weight w_i of all clauses){
                        w_i = w_i - 1;
                    }
                }
            }
            A = flip(A, x_i);
            ADAPT(p);
            MAINTAIN(); //update variable scores
            update DecVar;
            //By removing variables with non-positive scores.
            //Pushing new variables (≠ x_i) into DecVar,
            //that have a positive score now.
        }
    }
    return 'UNKNOWN';
}
```

We have now finished our explanation of the functioning of gNovelty+. We will continue by taking a more detailed look at its performance during the SAT 2007 Competition.

4.3.5 Explaining the Performance of gNovelty+

As mentioned in the introduction to this section, gNovelty+ participated in solving random instances during the SAT 2007 Competition[4]. Since gNovelty+ is an incomplete solver, it was only able to show the satisfiability of certain instances, and therefore only participated in the "Random SAT" category.

In this category, it was able to solve 242 of the 511 instances, which was the third most amount in total. However, the needed solving times allowed it to award gNovelty+ with the first place in this category.

Since gNovelty+ did not participate in the handmade or industrial categories, no results for these categories can be reported here.

We will give some explanations for the performance of gNovelty+ on random instances in the remainder of this section. As mentioned before, gNovelty+ is an improved version of G^{2}WSAT. Since the performance impacts of certain G^{2}WSAT features were already explained (see section 4.2.5), we will not discuss them again. Instead, we will now concentrate on the new features of gNovelty+.

The Benefits of using the AdaptNovelty+ Heuristic

G^{2}WSAT originally uses Novelty++ to escape from local minima. While this heuristic performed best during the SAT 2005 Competition for random 3-SAT instances, it was not competitive on random 5-SAT and random 7-SAT instances [PG07]. Especially R+AdaptNovelty+ outperformed G^{2}WSAT on these instances.

In [PG07], the conclusion is drawn, that the Novelty++ heuristic should be replaced by AdaptNovelty+ for gNovelty+ in order to raise the cumulative performance on all instances.

Furthermore, the usage of the dynamic parameter update scheme to adapt p makes gNovelty+ rather independent of parameter tuning when it comes to noise.

However, the AdaptNovelty+ heuristic needs the walk probability wp. The default value of wp is 0.01, meaning that approximately one out of hundred search steps is done completely at random among variables in unsatisfied clauses. This setting for wp has been determined experimentally in [HTH02]. The application of random walks adds to the *mobility* of a solver, which in turn results in smaller solving times [SS01].

[4] http://www.satcompetition.org

Altogether, gNovelty+ benefits from smaller run-times on larger instances, a dynamic noise adaption, and an improved mobility by the application of AdaptNovelty+.

The Benefits of using Clause Weights

As mentioned in section 4.3.3, gNovelty+ uses clause weights in order to guide the search out of local minima. As explained before, clauses that are unsatisfied more often receive a larger weight. Since the variable scores are computed via the clause weights of clauses in which they appear, variables in often unsatisfied clauses will receive a larger score.

Therefore, the chance, that such a variable is flipped and thus these unsatisfied clauses get satisfied, will raise over time. Because a local minimum is characterized by a number of unsatisfied clauses, the search will be guided away from this minimum.

We have already mentioned, that gNovelty+ uses an additive clause weight update scheme (PAWS) as presented in [TPBF04]. For this scheme, clause weights of unsatisfied clauses are increased by 1 in each search step. With a smoothing probability sp, clause weights get smoothed (decreased by 1).

Since addition and subtraction are cheap operations, the clause weight maintenance can be done in a very efficient way. [TPBF04] performed a study to see whether this additive scheme has any advantages, and they revealed, that it is a superior update scheme for large and more difficult instances. At least it performs better than the original multiplicative update scheme from SAPS on these instances.

The main benefit of clause weights is therefore an improved search guidance delivered by the weighted objective function (as used by the AdaptNovelty+ heuristic). The cost for weight updates are rather small, since a pure additive weight update scheme is used.

We are now done with explaining the impact of certain new features in gNovelty+, as well as our review on the functioning of it. We will finish this section by a summary on what we have presented for this review.

4.3.6 Summary of the Review of gNovelty+

gNovelty+ is an incomplete clause weighting SLS solver, that has proven to be very competitive during the SAT 2007 Competition.

gNovelty+ applies a gradient based variable score update scheme, that is further improved by the application of clause weights (see section 4.3.3 on page 117). Clause weights enable gNovelty+ to escape from local minima. Clause weights are maintained by an efficient pure additive weighting scheme (PAWS) as introduced in

[TPBF04].

Furthermore, gNovelty+ uses the AdaptNovelty+ heuristic (see section 4.3.3 on page 116). As has been revealed by [PG07], this heuristic delivers superior performance on random 5-SAT and 7-SAT instances, and gives gNovelty+ a better cumulative performance on all random SAT instances.

As part of this heuristic, gNovelty+ performs random walks. These random walks raise the mobility of gNovelty+ and make it probabilistic approximately complete.

We have now finished our review on gNovelty+ as well as our explanations on SLS solvers. We will now give a comprehensive summary of Part II, recapitulating the most important facts on the presented SAT solver paradigms.

Summary of Part II

The main topic of this part was the explanation of the two SAT solver paradigms DPLL and SLS. We have presented the functioning of various SAT solvers, in order to present the main benefits and drawbacks of the respective paradigm they belong to.

Summary of DPLL Algorithms

We started by explaining DPLL in chapter 3. We introduced the basic ideas behind the DLL algorithm (see section 3.1.1), that is the basis for all modern DPLL SAT solvers. We have explained two representatives for the main categories of DPLL: look-ahead and conflict driven.

The representative for look-ahead solvers was March_ks, as presented in section 3.2. March_ks has proven to be very competitive on random SAT and UNSAT instances during the SAT 2007 Competition.

The representative for conflict-driven solvers was RSat, as presented in section 3.3. RSat has proven to be very competitive on industrial SAT and UNSAT instances during the SAT 2007 Competition.

We used these two solvers to explain the major advantages of DPLL algorithms: their completeness and ability to provide unsatisfiability proofs.

As we have seen, completeness is achieved by structuring the search. For March_ks, this structure is achieved by organizing the search within a binary search tree. For RSat, structuring the search is done by learning assertion clauses.

The completeness then enables a DPLL solver to give a distinct answer on the satisfiability of an instance. Furthermore, it can be used to find not only one, but all models that a certain instance has.

The ability to provide a proof for the unsatisfiability of an instance might be useful for certain applications. For example, when using a DPLL solver for verification purposes, an unsatisfiability proof can be used to identify mistakes in order to correct them.

Furthermore, we have mentioned the major drawbacks of DPLL algorithms: complexity and scalability.

As we have seen, DPLL algorithms are often much more complex than SLS algorithms when it comes to an implementation. This is mainly because of the effort that is put into optimizing the structured search.

For March_ks, various features like a preprocessor, double look-ahead, distribution jumping, and local branching add to the performance, but make the solver more difficult to understand and to implement.

For RSat, a preprocessor, early conflict detection boolean constraint propagation (ECDB), the FirstUIP learning scheme, restarts, and clause database maintenance are essential for its performance, but make the solver more complex and difficult to implement as well.

When it comes to scalability, DPLL solvers behave much worse than SLS solvers. The main reason for this lies within the structuring of the search.

For March_ks, this structuring of the search is done via a binary search tree. This means, that the tree will become inevitably large for instances with more variables, even though early tree width reduction is performed via double look-ahead.

For RSat, the necessity for search structure causes the number of learned clauses to become inevitably large for instances with more variables, even though clause database maintenance is performed.

In summary, DPLL algorithms have the advantage of being complete, but this advantage is traded for high complexity and poor scalability. However, given the results of the SAT 2007 Competition, it is obvious, that DPLL solvers can be used for solving a variety of problems. It is interesting to note, that, except for the random SAT category, all categories have been won by a DPLL SAT solver.

DPLL solvers should therefore be used whenever an instance is supposed to be unsatisfiable, whenever an unsatisfiability proof is needed, or whenever all models of an instance are to be found.

The property of a random instance to be unsatisfiable is often given, when it is from the *over-constrained* region, i.e. when the clauses-to-variables ratio is greater than 4.26. Because SLS solvers can not detect the unsatisfiability of a random instance, they are inferior to DPLL solvers in the over-constrained region.

Summary of SLS Algorithms

After explaining DPLL, we continued with the SAT solver paradigm SLS in chapter 4. We introduced the basic idea behind stochastic local search, which is the basis for all modern SLS SAT solvers. We have explained two representatives for the main categories of SLS: randomized and clause weighted.

The representative for randomized solvers was adaptG^{2}WSAT$_0$, as explained in section 4.2. adaptG^{2}WSAT$_0$ has proven to be very competitive on random SAT instances during the SAT 2007 Competition.

The representative for clause weighting solvers was gNovelty+, as explained in section 4.3. gNovelty+ has also proven to be very competitive on random SAT instances during the SAT 2007 competition.

We used these two solvers to explain the major advantages of SLS algorithms: their scalability and performance on satisfiable random instances.

As we have seen, scalability is achieved by performing stochastic local search on complete assignments. For both solvers, this stochastic local search is realized via the application of a gradient based greedy search supported by an objective function.

Since SLS solvers do not come with a feature set as large as those of DPLL solvers, they are ofter much simpler in their design. Because of this, SLS solvers perform less computations in order to advance the search. This in turn makes it possible for them, to explore their search space much faster than DPLL algorithms. This then results in better performance on satisfiable instances, since a solution can be found with way less overhead.

Furthermore, we have mentioned the major drawback of SLS algorithms: their incompleteness. This incompleteness mainly comes from the application of stochastic local search. This in turn makes their application on unsatisfiable instances useless.

The incompleteness of SLS algorithms is manifested by the behavior of search getting stuck in local minima. Much effort has been spend in order to make SLS algorithms less susceptible for these local minima. For adaptG^{2}WSAT$_0$, this effort led to the development of the randomized Novelty++ variable selection heuristic with dynamic parameter tuning for noise. For gNovelty+, this effort led to the incorporation of a clause weighting scheme.

In summary, SLS algorithms have the advantages of being simple in their design and perform very well on random satisfiable instances, but this is traded for incompleteness.

SLS solvers should therefore be used whenever an instance is supposed to be satisfiable and when the application of a DPLL solver is prohibited by instance size.

SLS solvers seem to be effective for instances with a clauses-to-variables ratio much smaller than 4.26, i.e. instances from the *under-constrained* region (see section 2.7.2

on page 22). This is mainly because such instances have numerous models and since SLS solvers advance the search very fast without much overhead, it is likely that they find such a model without much effort.

DPLL versus SLS

Comprehensively, DPLL and SLS solvers seem to complement each other very well. DPLL algorithms perform better on unsatisfiable and over-constrained instances. SLS algorithms perform better on satisfiable and under-constrained instances.

This is the main reason why one might want to combine both approaches. The main goal would then be to retain the advantages of both paradigms, while overcoming their disadvantages. This could then result in a more robust, yet complete and fast hybrid SAT solver.

How this combination (in the form of a hybrid SAT solver) could be performed will be discussed in the next part of this work.

Enough words have been exchanged;
Now at last let me see some deeds!
from "Faust: The Tragedy Part One"
– Johann Wolfgang von Goethe

PART III

A new Approach for the Development of a Hybrid SAT Solver

As we have stated in the summary of the previous part, the combination of DPLL and SLS paradigms look, at least on the first glance, beneficial. The idea is to create a hybrid SAT solver, that inherits the best of both worlds while not suffering from their drawbacks. Ideally, a hybrid SAT solver would be complete, as well as more robust and much faster than its components left to themselves.

This is why considerable effort has been undertaken to create hybrid SAT solvers for more than a decade now. These approaches include the use of a SLS solver to acquire information that is supposed to support a DPLL solver [Cra, MSG98, FF04, HD04, FH07, LMS08], the use of a DPLL solver (and other complete search techniques) to acquire no-goods in order to improve the local search of a SLS solver [JL02, FR04, HLDV07], peer-like approaches where SLS and DPLL algorithms are supposed to equally benefit from each other [CBS04], or even the combination of conflict-driven and look-ahead techniques within a single DPLL algorithm [MvVW06].

However, the anticipated goals we have outlined above have not yet been met. One of the most promising approaches for hybridization so far is an incremental algorithm called `hinotos` [LMS08]. Empirical studies of this algorithm (also provided in [LMS08]) have shown it to be competitive with (but not consistently better than) its DPLL component. As a positive result from this empirical study stands the ability of `hinotos` to be able to solve about 5% more instances than its DPLL component, which means that `hinotos` is slightly more robust.

Since the intuitive approaches mentioned above all lack a decent improvement of solver robustness and/or performance gain, we have decided that it is necessary to undertake a more profound study on how a hybridization can be done. This study is to reveal a new and efficient approach for the combination of DPLL and SLS algorithms. In order to find such an approach, one is to study the search behavior of SAT solvers and eventually link them to properties that can be used for the creation of a hybrid SAT solver.

The following part of the work at hand is presenting our efforts to undertake such a study. We will first investigate the behavior of gNovelty+ and present numerous properties that have been identified for its search behavior. We will then evaluate a new idea (based on these properties), on how to create a hybrid SAT solver consisting of gNovelty+ and March_ks. Finally, we perform empirical tests to verify the usefulness of the proposed approach. The part is concluded by a brief discussion of the presented results and a presentation of the final idea for a new hybrid SAT solver.

Chapter 5

Properties of the Search Behavior of gNovelty+

A profound study is undertaken in this section, in which we try to identify properties of the search behavior of gNovelty+. These properties are then supposed to aid us in identifying exploitable information on an instance gNovelty+ is searching on. The information is then used in order to improve the search of the DPLL solver March_ks on the same instance. All together, this is supposed to yield an approach for a hybrid SAT solver. Before we perform such a study, we must first define the testbed and terms to deal with the acquired data.

5.1 The Testbed

Before we start a study, we must be clear about *what* we study and with what *means* we undertake this study. As already stated, we will perform a study on the search behavior of gNovelty+. However, gNovelty+ is a SLS solver capable of performing search on many different SAT formulas. Because a comprehensive analysis of all instance types is very time consuming, we had to abridge the types of formulas on which we study the search behavior of gNovelty+.

5.1.1 Used Formulas

We decided, that it would be interesting to first study uniform random 3-SAT formulas of different sizes. We selected a few formulas from two benchmark sets: the SATLIB benchmark[1], and the SAT 2007 Competition random benchmark[2].

The complete list is presented in table 5.1.

All the formulas are uniform random 3-SAT and belong to the hard region where

[1]downloadable from http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/RND3SAT/uf250-1065.tar.gz
[2]downloadable from http://www.satcompetition.org/2007/random.tar.bz2

Formula	Benchmark	Var.	Cls.	Ratio
uf250-011.cnf	Satlib	250	1065	4.26
unif-k3-r4.26-v360-c1533-S1806915801-08.SAT.shuffled.cnf	Satcompc	360	1533	4.26
unif-k3-r4.26-v360-c1533-S1422217194-16.SAT.shuffled.cnf	Satcomp.	360	1533	4.26
unif-k3-r4.26-v400-c1704-S1031412672-05.SAT.shuffled.cnf	Satcomp.	400	1704	4.26
unif-k3-r4.26-v400-c1704-S141590207-13.SAT.shuffled.cnf	Satcomp.	400	1704	4.26
unif-k3-r4.25-v450-c1912-S1113882813-11.SAT.shuffled.cnf	Satcomp.	450	1912	4.25
unif-k3-r4.25-v450-c1912-S216896591-15.SAT.shuffled.cnf	Satcomp.	450	1912	4.25
unif-k3-r4.26-v500-c2130-S1490740667-05.SAT.shuffled.cnf	Satcomp.	500	2130	4.26
unif-k3-r4.26-v500-c2130-S44928635-12.SAT.shuffled.cnf	Satcomp.	500	2130	4.26

Table 5.1: Selected Formulas to study gNovelty+'s search behavior.

the clauses to variables ratio is ≈4.26. These formulas differ in their sizes, ranging from 250 variables to 500 variables. All of these formulas are satisfiable.

Furthermore, what we study is not the only thing, that must be clear. It must also be clear, with what (hard- and software) means this study is performed. Since our study contains empirical results, one might need this information in order to verify or reproduce these results.

5.1.2 System and Hardware

All tests are performed on a computer with the following specification.

- Operating System: Windows XP Professional (Version 2002), 32Bit, Service Pack 3
- CPU: AMD Athlon 64 X2 Dual Core 4600+, 2.39 Ghz
- Memory: 1.87 GB

5.1.3 Software

As SLS solver, we use gNovelty+ in the version that was submitted to the SAT 2007 Competition. The sources for gNovelty+ can be obtained from the SAT 2007 Competition homepage[3]. When we speak of gNovelty+ in the rest of this work, we mean this exact version of it.

As DPLL solver, we use March_ks in a version that was submitted to the SAT 2007 Competition. However, since this version contains a bug, one should download the bugfixed version[4]. When we speak of March_ks in the rest of this work, we mean this bug-fixed version.

[3] http://www.satcompetition.org/2007/solvers-sat07-src.tgz
[4] http://www.st.ewi.tudelft.nl/sat/Sources/sat2007/march_ks.zip

5.2 Terms used in the Empirical Study

The functioning of gNovelty+ and its application to SAT formulas necessitate some additional terms. Given a formula F, that contains n variables and m clauses, we define additional terms as follows.

5.2.1 Runs

Definition 32 *Given a formula F, an assignment A_0, and $w \in \mathbb{N}, w < \infty$. We define a* `run` *$\mathcal{R}$ = gNovelty+(F, A_0, w) as a single attempt of gNovelty+ to find a solution for F (*`MAX-TRIES` *= 1). Thereby, A_0 is the starting assignment for gNovelty+'s search (usually randomly generated by gNovelty+ itself). The parameter w is called* `cutoff`*, and defines the maximum number of flips that gNovelty+ is allowed to make for the attempt (*`MAX-FLIPS` *$= w$).*

The cutoff w then defines the *runlength* for such a run $\mathcal{R}$. This runlength can also be measured in real-time. On the computer specified in section 5.1.2, gNovelty+ performs about 500000 flips per second. The cutoff is needed, because gNovelty+ is an incomplete solver.

5.2.2 Trajectories

During a run of gNovelty+, it visits numerous assignments in the search space that belongs to F. In order to address these assignments, we give the following definition.

Definition 33 *Given a run of gNovelty+ on a certain formula F, the* `search trajectory` *(or* `trajectory` *for short), is a finite sequence of assignments, that gNovelty+ visited during the run. We denote this $\mathcal{T} = (A_b)_{b=0}^{w}$.*

The first element in the trajectory is always the randomly generated starting assignment for this run. With the number of flips w for a given run, the number of assignments within the trajectory of that run is always $1 + w$ (the starting assignment plus an assignment for every flip made). This means, that the trajectory could contain a certain assignment multiple times.

5.3 Identified Properties

We will now present several experiments on the search behavior of gNovelty+ as well as certain identified properties for its search.

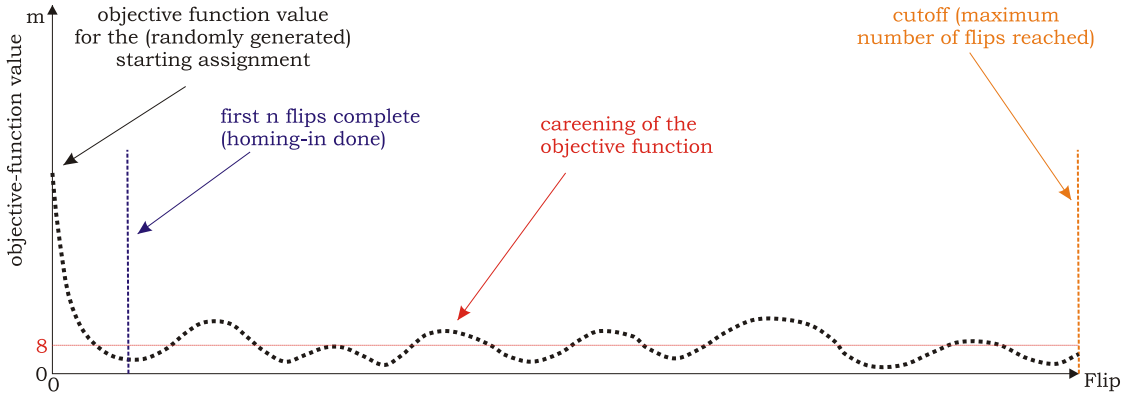


Figure 5.1: A schematic graphic of the development of the objective function values of a trajectory.

5.3.1 Homing-in and Careening

As stated during the review of gNovelty+ (see section 4.3 on page 115), the applied local search paradigm needs an objective function that measures the quality of an assignment in the search space. Recall, that the objective function in gNovelty+ counts the number of unsatisfied clauses for such an assignment in a weighted fashion. When ignoring weights, one will receive a simple objective function that simply counts the number of unsatisfied clauses.

Our first experiment was to find out, how the values of this simplified objective function evolve during a run of gNovelty+. The question was, if any regularities can be found for these values.

We therefore modified gNovelty+ in a way that it saves the starting assignment and all performed flips to a file. We then performed several runs of gNovelty+ on the formulas mentioned in table 5.1.1. The files containing the starting assignment and flips were used to retrace the search-trajectory of gNovelty+. While retracing is performed, one can check for each assignment what the number of unsatisfied clauses is (for the formula that belongs to the investigated run).

With these values given, one can easily create a graphical representation of the objective function values for the complete trajectory. Since the trajectory contains the assignments in a chronologically ordered manner, these values then represent the evolution of the objective function values. Our empirical tests (not reported here in detail) indicate a graphical representation like the one given in figure 5.1. This is only a schematic figure, not linked to any existing formula (the actual plots look a bit more erratic). However, it visualizes two important properties that hold for almost all trajectories.

- **Homing-in property:** The objective function values at the beginning of the run (during the first n flips), usually have a much larger value. These values are shrinking rapidly. This property of trajectories is linked to the stochastic local search mechanisms, whereas the first (randomly created) assignment leaves

Formula	Median Careening Value
uf250-011.cnf	7.89
unif-k3-r4.26-v360-c1533-S1806915801-08.SAT.shuffled.cnf	8.82
unif-k3-r4.26-v360-c1533-S1422217194-16.SAT.shuffled.cnf	8.15
unif-k3-r4.26-v400-c1704-S1031412672-05.SAT.shuffled.cnf	6.46
unif-k3-r4.26-v400-c1704-S141590207-13.SAT.shuffled.cnf	7.84
unif-k3-r4.25-v450-c1912-S1113882813-11.SAT.shuffled.cnf	6.80
unif-k3-r4.25-v450-c1912-S216896591-15.SAT.shuffled.cnf	6.99
unif-k3-r4.26-v500-c2130-S1490740667-05.SAT.shuffled.cnf	8.07
unif-k3-r4.26-v500-c2130-S44928635-12.SAT.shuffled.cnf	10.98

Table 5.2: The median careening values over six runs of 9 different formulas. The median of the median careening value over all formulas is 8.00.

quite a number of clauses unsatisfied. The solver then needs about n flips to "repair" this assignment by applying local search, until it reaches an area in the search space with assignments, that satisfy a much larger amount of clauses.

- **Careening property:** As soon as the first n flips have been made, the objective function values in the trajectory seem to careen around a certain value. Table 5.2 gives an overview of the median careening values for six runs on all examined formulas. As we can see, the median careening value for all six formulas is 8. The objective function value of 0 is reached, as soon as a solution has been found. Since a run of gNovelty+ is finished as soon as this happens, an assignment having an objective function value of 0 must always be at the end of the trajectory.

The properties mentioned above do not hold, in case either the randomly created starting assignment is a solution (for which the probability to happen is very small), or the run created by gNovelty+ has less than n flips. This can happen when the cutoff is very small or the randomly generated starting assignment is very close to a solution that gNovelty+ is able to find.

Since the first property was quite clear, we did not pay any more attention to it. The question then was to find out what causes the careening property to occur. Our intuition told us, that careening occurs because gNovelty+ is entering and escaping from local minima. It was yet unclear, if the same local minima were visited over and over again, or if the careening property occurred because gNovelty+ was visiting one (different) local minimum after another.

5.3.2 Variable Tendencies and Plateau Connection Graphs

Variable Tendencies

The idea then was to take a more detailed look at the variables and how they changed their values during the search. This was supposed to give us an idea on how good

Run	0.95**-tendentiously set to 0**	0.95**-tendentiously set to 1**
0	18 66 112 126 194	9 22 99 130 133 180 235
1	17 18 66 112 126 194	9 22 29 99 118 122 130 133 153 180 235
2	17 18 66 112 126 194	9 22 99 118 130 133 153 235
3	17 18 66 112 126 194 246	9 22 29 51 99 118 122 130 133 153 235
4	17 66 112 126 194	9 22 29 51 99 118 180 235
5	18 21 66 112 126 177 194 246	9 22 29 51 99 109 118 130 133 153 180 235

Table 5.3: Numbers of the 0.95-tendentiously set variables within six different trajectories created by gNovelty+ during 100000 flips on the `uf250-011.cnf` instance. The variables are divided into 0.95-tendentiously set to zero and 0.95-tendentiously set to one, so one can see similarities between the six trajectories.

gNovelty+ diversifies its search. One might say, that if the number of flips for all variables are quite similar, gNovelty+ is doing a very diversified search, trying to equally use all the variables to find a solution.

However, since local search is involved, it was more likely that gNovelty+ flips certain variables more often than others, since it is to explore local minima (and their surroundings) with more attention due to the local search mechanisms.

We performed an experiment, using six different runs of gNovelty+ on a formula F. We then analyzed the corresponding trajectories and simply counted how often a variable was flipped.

The (exemplary) result for the `uf250-011.cnf` formula is presented in figure 5.2. The x-axis gives the index of the variable, however, the indexes of the variables have been sorted according to their y-axis value to better reflect the tendency behind these values.

The y-axis gives the number of times a variable has been flipped. The triangles refer to values for a single trajectory. The crosses represent the median over all six trajectories. As we can see, there obviously are some variables that get flipped less often than others.

In order to be able to address these variables, we will define an new term for them as follows.

Definition 34 *Given a sequence of assignments (i.e. a trajectory of gNovelty+) and the corresponding formula F, we call variables from F, that have received the same truth value in at least the fraction $t \in [0, 1]$ of the total number of assignments, a t-`tendentiously set variable`.*

For a trajectory containing 100001 assignments (for 100000 performed flips), a 0.95-tendentiously set variable would have kept the same value in at least 95000 assignments. Table 5.3 gives a list of 0.95-tendentiously set variables of all the six trajectories created in the above mentioned experiment. Not only can we see, that t-tendentiously set variables do exist, we can also see, that these variables seem to be

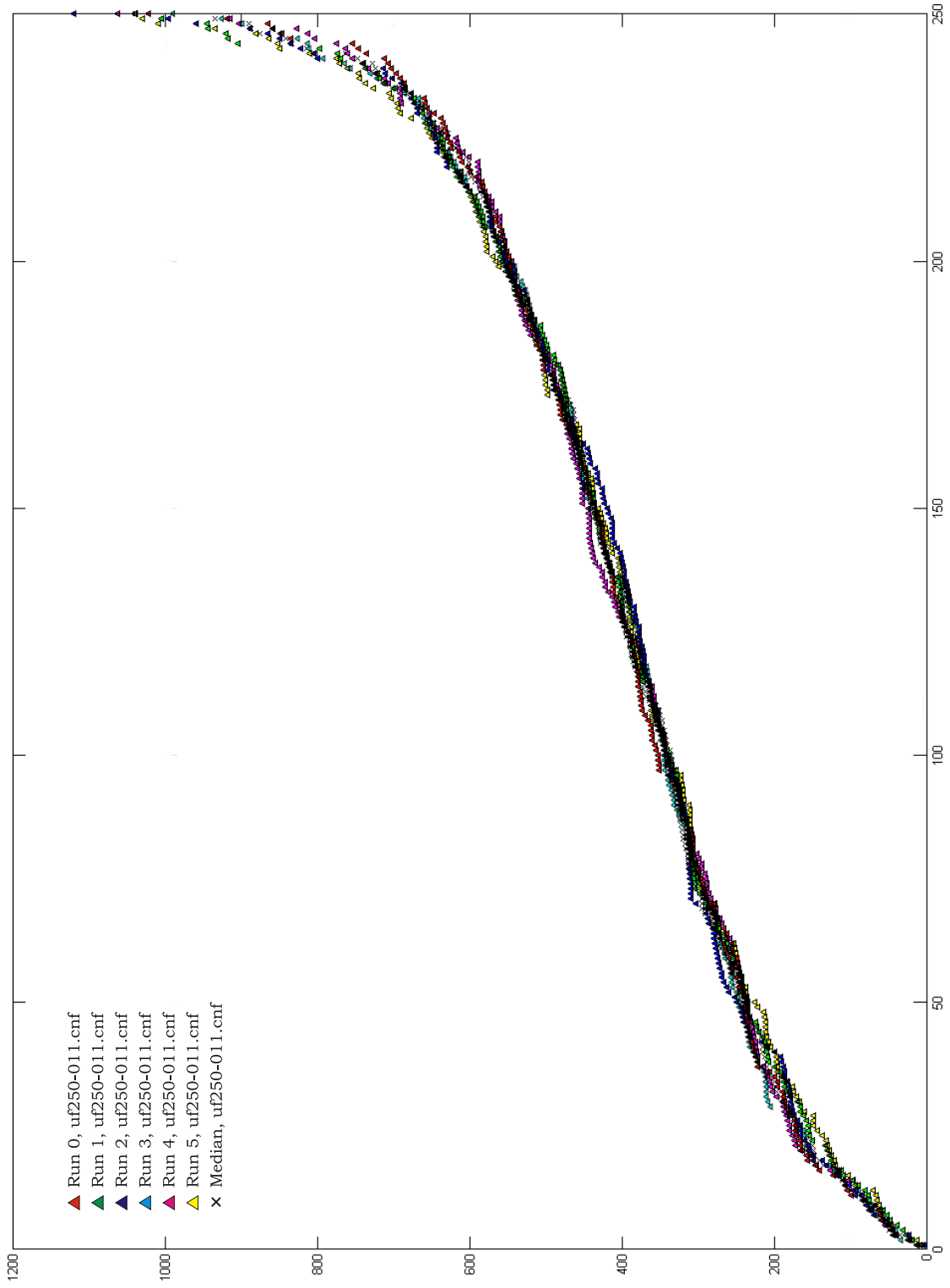


Figure 5.2: A plot of variable flip count for six different trajectories, represented as triangles. The median value is represented as a cross. The figure is rotated 90 degrees. The x-axis gives the index of the variable. These indexes have been sorted to better reflect their y-axis values. The y-axis value represent the number of times a variable has been flipped.

quite the same for all the trajectories. All together, this represents a new property of the search behavior of gNovelty+.

- **Variable tendency property:** During a run of gNovelty+ (that must be large enough for the careening property to occur), the variables are not equally often flipped. Some variables have the same truth value assigned for more than 95% of the searched assignments.

We then had to identify a reason for the existence of tendentiously set variables, and believe to have found them within plateau connection graphs. These plateau connection graphs are explained in the next section.

Plateau Connection Graphs

Plateau connection graphs have been described in [HS04]. They are a tool to structure the search space for a given formula F.

A plateau connection graph is an undirected and weighted graph, for which the vertices represent plateaus in the search space. A plateau is a set of assignments for which two properties hold.

1. The objective function value is the same for all assignments within this plateau.
2. The assignments within a plateau are connected, meaning that one can reach every assignment in a plateau from another by performing flips, for which the created intermediate assignments have the same objective function value as well.

A SLS solver like gNovelty+ will (due to the local search mechanisms) try to improve the objective function value. It will therefore try to leave a plateau with a high objective function value towards a plateau with a smaller objective function value.

We call an assignment that has a higher objective function value an *exit* (from a plateau), when it has hamming distance one to an assignment that has a lower objective function value. The assignment with the lower objective function value is called an *entry* (to another plateau). These two assignments together are, figuratively spoken, a passage between two plateaus.

Edges within a plateau connection graph represent a passage between two plateaus. In short, such a passage is defined by two assignments that are part of two different plateaus and have hamming distance 1.

However, plateaus can have multiple passages towards several other plateaus. They can also have several passages towards a single other plateau. When gNovelty+ is searching for a solution, it will always try to leave plateaus with higher objective function values towards those, with smaller objective function values. Since sideway-steps (inside a plateau for with the assignments all have the same objective function

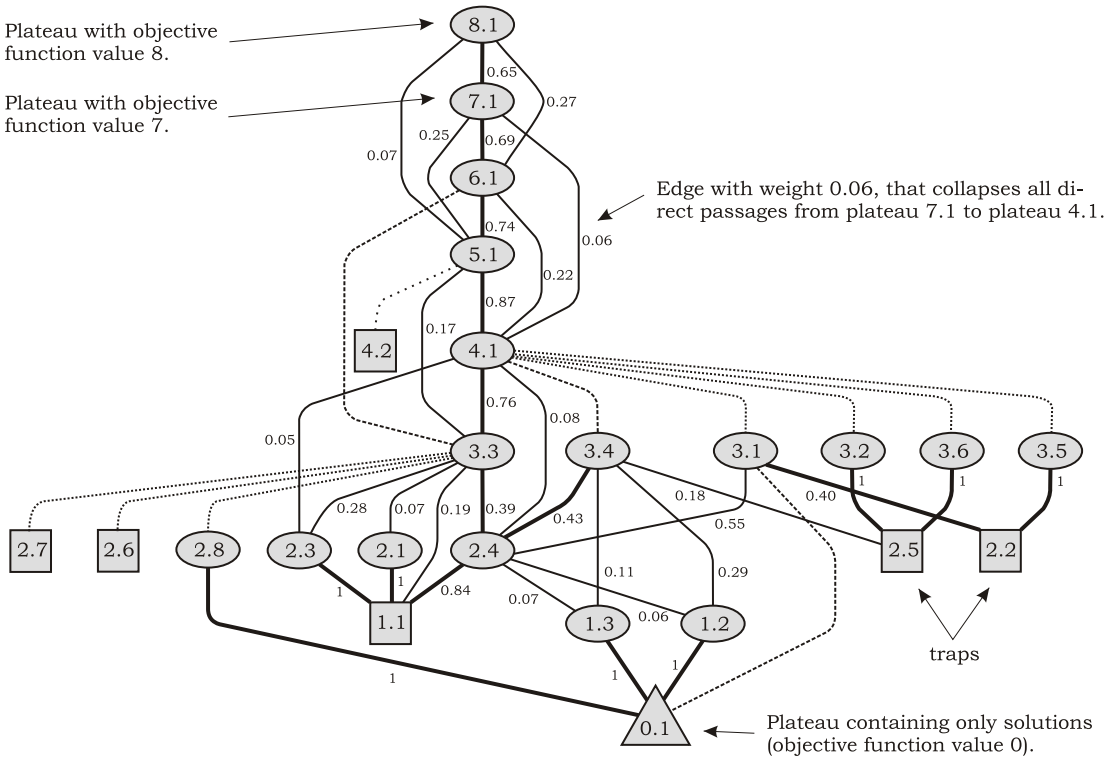


Figure 5.3: A (partial) plateau connection graph for a hard uniform random 3-SAT instance (20 variables, 91 clauses). See text for details.

value) are performed at random, there is a certain likelihood for gNovelty+ to find such a plateau with a smaller objective function value. This likelihood is linked to the number of passages from the plateau gNovelty+ is currently searching in towards the new plateau with smaller objective function value. In other words, the larger the number of passages from a high plateau A to a lower plateau B, the larger the probability that gNovelty+ will advance its search from A to B. These probabilities are represented as edge-weights in the plateau connection graph.

A plateau connection graph for a formula with 20 variables and 91 clauses is presented in figure 5.3 (taken from [HS04]). The higher the objective function value of assignments within a plateau, the more the vertice for this plateau is drawn towards the top of the image. The thicker the edged between two plateaus, the higher their weight, which means that the connected plateaus have more passages between them. Plateaus, that do not have a passage towards a plateau with smaller objective function value, are called traps and are drawn as rectangles. Such traps represent areas in the search space that gNovelty+ can only escape by the means to escape local minima. Plateaus drawn as triangles are those that contain only solutions, and the included assignments all have an objective function value of zero.

In order to understand why plateau connection graphs could hold the answer for the existence of the variable tendency property of gNovelty+'s search behavior, one

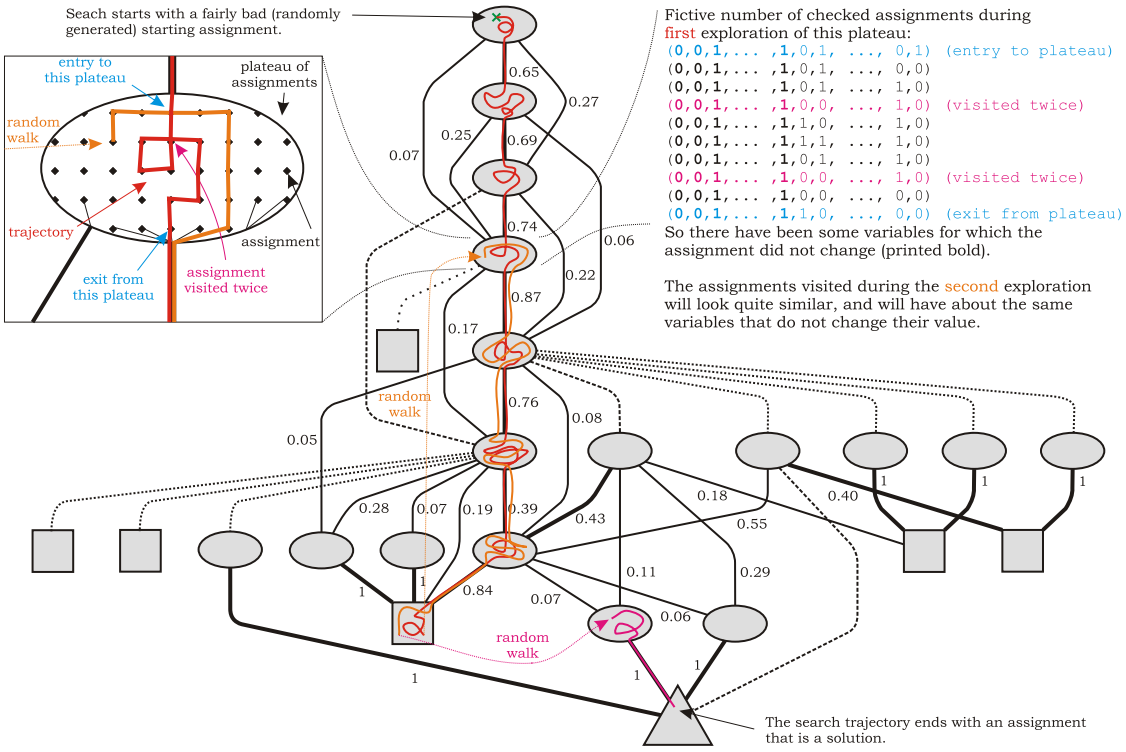


Figure 5.4: A search trajectory in the world of a partial plateau connection graph. See text for details.

must understand on how gNovelty+ moves through such a plateau connection graph in search of a solution. Figure 5.4 gives an image for such a search.

During the search for a solution (which is a trajectory represented as a line through the plateaus), gNovelty+ will explore plateaus in search for an exit assignment, so that it might continue towards a plateau with a smaller objective function value. This "plateau exploration" is visualized in form of squiggles of the trajectory within a plateau. During the exploration of a plateau, a number of variables never change their value, because such a flip is prohibited by the value of the objective function. In other words, some variables are not flipped, because they would deteriorate the objective function value. Variables that are flipped during the exploration are those, that represent sideway-steps.

When gNovelty+ finds an exit, it will follow its passage towards a plateau that has a smaller objective function value. In this plateau, a different set of variables is flipped for exploration. This scheme continues until gNovelty+ either finds a solution, or enters a trap. Escaping from a trap can only be achieved by the means that are used to escape from local minima (i.e. random walk, noise and clause weights).

When taking a look at gNovelty+'s trajectory through such a plateau graph, one realizes that there are certain routes gNovelty+ takes more often. This is because of the different likelihoods of entering one plateau from another. During the exploration

of plateaus, a certain number of variables is flipped more often. Furthermore, a certain number of plateaus is visited more often. Ergo, there are variables that get flipped more often for the complete trajectory. These are the variables needed for exploring all the visited plateaus.

On the other hand, there are variables that are almost never flipped. These are the variables (assigned to their respective truth value), that numerous plateaus visited have in common. These are the then t-tendentiously set variables, where t gives a hint on the number of the plateaus in which a t-tendentiously set variable has always the same value. The larger t is, the larger the number of plateaus that are visited under the assignment to this t-tendentiously set variable.

5.3.3 Heavy Tailed Search

Recall, that certain search algorithms have the property to be heavy tailed (see section 3.3.3 on page 85). The mean and the variance of the search cost for such search algorithms often behave erratic, and do not stabilize when search is conducted for an increasing number of problem instances.

Even though we do not provide empirical evidence here for this property to apply on gNovelty+, it is quite obvious that the solver suffers from this property. One can easily observe, that gNovelty+ is sometimes able to solve an instance in just a few thousand flips, while sometimes needing more than ten times as many flips.

All together, it is impossible to predict the runtime of gNovelty+ for a given instance, which gives us a new property of gNovelty+'s search behavior.

- **The heavy-tailed property.** The runtime of gNovelty+ on a given instance can not be predicted. It behaves heavy-tailed. For an instance with 250 variables (`uf250-011.cnf`), we observed various runtimes from about 6000 flips up to 100000 flips.

The reason for this property is, that the number of flips gNovelty+ needs in order to find a solution depends on the randomly generated starting assignment and the search landscape it starts in. A poor starting assignment within a very rugged portion of the search space landscape hinders gNovelty+ to rapidly advance its search towards a solution. On a new run, gNovelty+ might never enter such a rugged portion of the search space, and therefore, will be able to find a solution much faster.

5.4 Summary of the Identified Properties

We have identified four properties for the search behavior of gNovelty+.

1. **The homing-in property.** This property can be explained using the poor number of satisfied clauses for a randomly generated starting assignment and the then applied local search mechanisms.
2. **The careening property.** This property represents the inability of local search to further improve "near-solution" assignments, linked to the entering and escaping of local minima.
3. **The property of tendentiously set variables.** This property reflects the way in which gNovelty+ explores the search space and especially plateaus as well as areas of local minima.
4. **The heavy-tailed property.** The search of gNovelty+ is heavy-tailed, which means, that one cannot predict the search time gNovelty+ needs on a given instance.

We will now use these properties to create a new approach for the construction of a hybrid SAT solver.

Chapter 6

Partitions

Equipped with the knowledge gained in the previous chapter, we decided to investigate the careening property in more detail. We where interested in finding out, whether gNovelty+ was near a solution when searching assignments with an objective function value ≤ 8. If so, we planned to use this knowledge to support the DPLL solver March_ks by confining its search to assignments in the search space vicinity of assignments having an objective function value ≤ 8. For now, it is sufficient to intuitively understand the search space vicinity of assignments A_i as, for example, a set of assignments with hamming distance ≤ 3 to A_i for any i. Since all our investigated formulas are satisfiable, March_ks, confined to the smaller search space, should then be able to find a solution in much less time than it would have needed to find a solution in the unconfined search space of F.

However, before we could start our study, we needed to find an appropriate tool to confine the search space of a DPLL solver like March_ks. The easiest way to do so was to somehow create a new formula F_s from the formula F that gNovelty+ was searching on. The search space of this new formula was supposed to inherit the search space vicinity of the search space from assignments with objective function values ≤ 8.

Our hope was, that this search space vicinity already contains a solution, that March_ks is able to find fast.

We will now explain, how the creation of a sub-formula F_s from F is done by using the information that gNovelty+ gathered during a run. We will therefore define the terms of a *range*, a *partition* and the *search space vicinity*.

6.1 Ranges

Before we can address the search space vicinity of assignments with objective function value ≤ 8, we must be able to find numerous such assignments. To do so, we need a new term as given in the following definition.

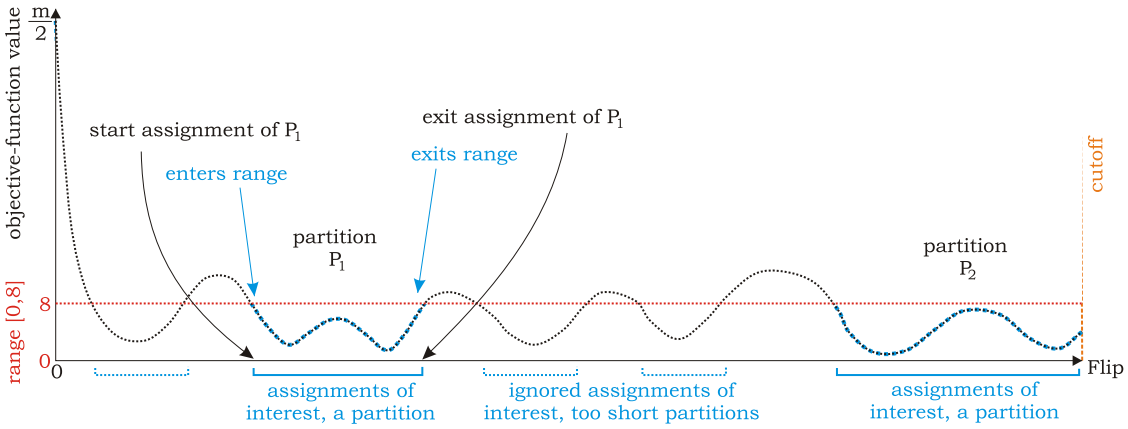


Figure 6.1: A schematic graphic of the development of the objective function values of a trajectory, and a graphical presentation for the terms of a *partition*, a *too short partition* and the *range*.

Definition 35 *Given a formula F with n variables and m clauses. Additionally, given two numbers $i, j \in [0, m]$ with $i < j$. We call the interval $[i, j]$ the* `objective function value range` *of interest (*`range` *for short).*

The range $[i, j]$ now addresses the assignments we are interested in as follows. An assignment with objective function value k is only of interest, iff $k \in [i, j]$. So when we speak of the range $[0, 8]$, we mean all assignments with objective function value in $[0, 8]$. Figure 6.1 gives a visual on a range for a trajectory and the assignments of interest identified by such a range. As one can see, the range somehow splits the trajectory into different areas of interest. It is these areas that represent assignments we want to investigate their search space vicinities of.

6.2 Defining the Term Partition

The previously mentioned areas of interest somehow group certain assignments. To be clear about what we mean with an "assignment group", we must first give a name to these groups in order to be able to address them. This leads us to the term of a partition.

Definition 36 *Given a trajectory $\mathcal{T} = (A_b)_{b=0}^{w}$ for a run on a formula F. We call $P = (A_j)_{j=s}^{e}$ with $s < e; s, e \in [0, w]$ and $A_j \in \mathcal{T} \forall j \in [s, e]$ a $[i, j]$-*`Partition`*, iff $\forall\ A_j \in P$: $objectiveFunctionValue(F,\ A_j) \in [i, j]$.*

A partition can be viewed as a sub-sequence of a trajectory. Since the trajectory is a finite sequence, a partition has a *start assignment* A_s and an *exit assignment* A_e (see figure 6.1).

Before we explain what we mean with the "search space vicinity" of assignments in a partition, we must explain how gNovelty+ can build such a partition when

searching, and that indeed, numerous of these partitions exist.

6.3 The Construction of Partitions

Since gNovelty+ is creating the trajectory in chronological order, it can create partitions "on-the-fly". As soon as gNovelty+ finds an assignment, for which the objective function value is in $[i, j]$, it saves this assignment in an array. The array (that will later be a partition) is considered complete, as soon as an assignment is found, that has an objective function value not in $[i, j]$. The next assignments within $[i, j]$ are saved in a new array and so on. The set of arrays is then the set of found partitions for a run and its trajectory.

The major advantage of this way to collect partitions is, that it does not interfere with the search of gNovelty+, since copying an assignment into an array is not considered to be heavy overhead. However, the memory that is used up by gNovelty+ is much larger when all such partitions are saved. To circumvent this, we furthermore filter out partitions, that are too short. A partition is considered to be too short, when it has less than n elements (see figure 6.1). These partitions are not saved.

Obviously, the ability of gNovelty+ to find partitions is somehow connected to the runlength of the search. We therefore have to provide a certain lower border for the runlength in order to be able to find numerous partitions to work with.

6.4 Sufficiently Large Runs of gNovelty+

The runlength of a run, in which enough partitions can be found, is connected to the number of variables a formula has. We have tested different runlengths on different formula sizes and came to the conclusion, that a run on a formula F with n variables should contain at least the following number of flips:

$$RL(n) = 100000 \cdot \left(\frac{n}{250}\right)^{\frac{n}{250}}.$$

Runs with length calculated via $RL(n)$ for different formula sizes yield about the same number of found partitions in total. Since this makes the comparison of results between different formulas easier, we call runs that have at least $RL(n)$ flips *sufficiently large runs* of gNovelty+.

Given such sufficiently large runs of gNovelty+, we had to explicitly confirm the existence of partitions for all formulas in table 5.1. Furthermore, we where interested in the distribution of partitions within a sufficiently large run. It was yet unclear, whether they are uniformly distributed within a trajectory, or if they follow some other scheme of arrangement.

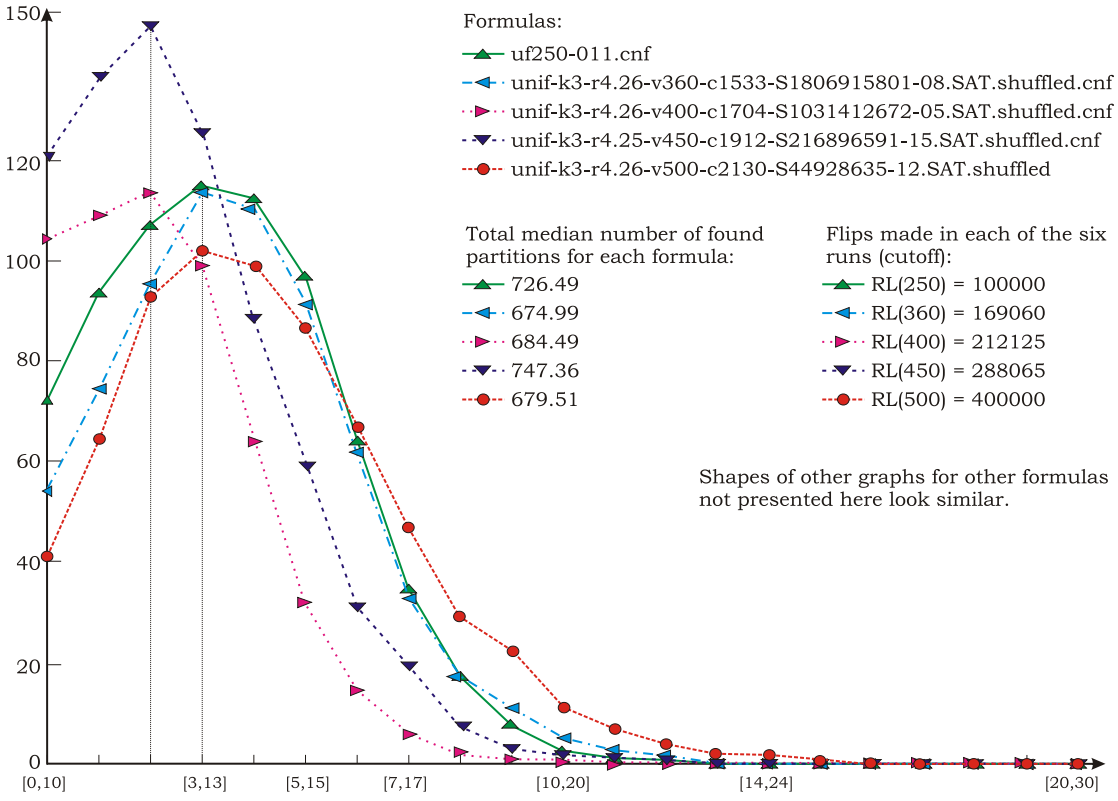


Figure 6.2: Figure of the number of partitions that have been found for a certain range and formula.

6.5 Existence and Distribution of Partitions

We performed two experiments. One to prove the existence of partitions, which will be explained next, and one to find out how partitions within a run are distributed, which will be explained afterwards.

6.5.1 Existence

In order to prove the existence of partitions, we created six different runs for every formula from table 5.1. We then counted the found partitions within different ranges for each trajectory of the six runs. The ranges go from $[0, 10]$ up to $[20, 30]$. In other words, we checked $[0, 10]$, then $[1, 11]$, $[2, 12]$ and so on. We then plotted the median value of found partitions over the six runs for each formula and range, as presented in figure 6.2. Recall, that we only use sufficiently large runs of gNovelty+, in which the number of flips is calculated via $RL(n)$. Furthermore, recall that we ignore too short partitions as explained in section 6.3.

As we can see, numerous partitions can be found within a sufficiently large run of gNovelty+. It does not surprise, that the number of found partitions depends on the range that is investigated. We already stated, that the median careening value

Figure 6.3: A Histogram of the exit assignments for each partitions within a trajectory in range $[3, 13]$. The x-axis (trajectory) is split into intervals of size 500. The y-axis represents, how many partitions had their exit assignment in such a part. Results for the other runs, ranges and formulas from table 5.1 look similar.

for all formulas and runs is 8. Therefore, the maximum number of partitions over all runs on all formulas is found within the range $[3, 13]$, that has 8 as its average.

6.5.2 Distribution

In order to determine the distribution of partitions over a trajectory within a given range, we performed the following experiment. Given a trajectory of a sufficiently large run of gNovelty+, and a range that is to be investigated. Intersected the trajectory into equally large parts and measure, how many partitions have their exit assignment inside such a part.

Figure 6.3 gives an exemplary result for the `uf250-011.cnf` formula for range $[3, 13]$ on a single trajectory with 100001 assignments.

As we can see, the found partitions in range $[3, 13]$ are spread quite uniformly over the trajectory.

Since we have now shown that numerous partitions exist within a sufficiently large run of gNovelty+, we can now explain what their use is and what we mean by the term of search space vicinity of assignments. This search space vicinity is, after all, the part of the search space of F, that March_ks is supposed to conduct search on.

6.6 Characterizing Partial Assignments of Partitions

With partitions, we are now able to find numerous assignments with a certain objective function value. We now need to define what we mean with the search space vicinity of such assignments. In order to do so, we will use the term of a characterizing partial assignment, defined as follows.

Definition 37 *Given a partition P of a trajectory, and the corresponding formula F. A* `characterizing partial assignment` *of P (denoted $CHAR(P)$), is a partial assignment that assigns all the* 1.0*-tendentiously set variables in P to their values, and leaves all other variables unassigned.*

That is, $CHAR(P)$ assigns all variables to a value, iff this value is the same for all assignments within P. The following example clarifies this.

Example 23 *Let $P = ((0,0,0),(0,0,1),(0,1,1))$ be a partition of three assignments. It is $CHAR(P) = (0,?,?)$, since the only variable that was 1.0-tendentiously set in P was the first variable. It receives the truth value that it had in all the assignments in P.*

Such a characterizing partial assignment can be extended to all the assignments in P. Furthermore, it can be extended to numerous more assignments that where not part of P, but have the same 1.0-tendentiously set variables. For the above given example, such an additional assignment would be $(0,1,0)$. These additional assignments are referred to as the search space vicinity of P. We give a more formal definition as follows.

Definition 38 *Given a formula F, its trajectory, and a partition P within this trajectory. We can calculate the characterizing partial assignment $CHAR(P)$. We say an assignment A, that $CHAR(P)$ can be extended to in the search space of F, is*

- *in the* `core` *of P, iff $A \in P$.*
- *in the* `search space vicinity` *of P, iff $A \notin P$.*

How characterizing partial assignments are supposed to be of use will be explained in the next section.

6.7 The Use of Characterizing Partial Assignments of Partitions

Given a formula F and a sufficiently large run of gNovelty+ on F, one can find a $[i,j]$-partition within the corresponding trajectory, and calculate its characterizing partial assignment. Since gNovelty+ performs flips very fast, it creates a sufficiently large run with a computation time that is negligible (less than a second). Therefore, the time that is consumed to find partitions is quite low as well, and will be ignored throughout the rest of this chapter.

Recall, that our initial idea was to search the search space vicinity of assignments with an objective function value ≤ 8. In order to do so, we first search for a $[0,8]$-partition. Let us call this partition P. Apply $CHAR(P)$ on F, which gives us $F_s = CHAR(P)(F)$, a sub-formula of F. The remaining search space for F_s is the set of assignments that $CHAR(P)$ can be extended to (in the search space of F). This includes all assignments in P, as well as the search space vicinity of P.

Furthermore, F_s consists of only those variables $x_i \in \mathcal{V}$ for which $Val_{CHAR(P)}(x_i) =?$, which means that the only variables in F_s are those, that do not receive a truth value by $CHAR(P)$. We denote these unassigned variables (the variables of F_s) with $\mathcal{V}_s$. It is $\mathcal{V}_s \subset \mathcal{V}$, and $\mathcal{V}_s \subsetneq \mathcal{V} \Longleftrightarrow \#(Val_{CHAR(P)}(x_i) =?) > 0$.

Recall, that a formula F in CNF can be viewed as a set of clauses $\mathcal{C}$. It is therefore equivalent to view the application of an assignment on a formula as the application of the assignment on its set of clauses. In short, it is $CHAR(P)(F) = F_s \mathrel{\widehat{=}} CHAR(P)(\mathcal{C})$.

We denote the set of clauses of F_s with $\mathcal{C}_s$. Therefore, $CHAR(P)(\mathcal{C}) = \mathcal{C}_s$. The question now is, what effect the application of the characterizing partial assignment of P has on the clauses in $\mathcal{C}$. That is, we want to know how the clauses in $\mathcal{C}_s$ look like.

The clauses in $\mathcal{C}_s$ can be grouped into four disjoint sets as follows. Let $c_i \in \mathcal{C}$. Let $c_i\text{`} = CHAR(P)(c_i)$.

- Iff $c_i\text{`} = c_i$, then no changes happened to the original clause from $\mathcal{C}$. We collect these unchanged clauses in $\mathcal{C}_{untouched}$.
- Iff $c_i\text{`} = \text{TRUE}$, i.e. if the clause c_i get satisfied under $CHAR(P)$, we put it in the set $\mathcal{C}_{sat}$.
- Iff $c_i\text{`}$ is the empty clause, we put it in the set $\mathcal{C}_{empty}$. However, since partitions have a minimum size of n, numerous variables in $\mathcal{V}$ will not be assigned by $CHAR(P)$. Therefore, the chance, that a clause c_i gets three of its literals assigned to values that make them FALSE, is practically non-existent (see Appendix on page 181). In fact, this never happened during our experiments, and therefore, $\mathcal{C}_{empty} = \emptyset$.
- Iff $c_i\text{`}$ is neither satisfied nor empty, but was affected by the application of $CHAR(P)$, it will contain less literals than its counterpart c_i. This is because all literals in c_i, for which the corresponding variable was assigned by $CHAR(P)$, must evaluate to FALSE. Otherwise, this clause would have been part of $\mathcal{C}_{sat}$. Therefore, the clause $c_i\text{`}$ subsumes $c_i$. We put these clauses $c_i\text{`}$ into the set $\mathcal{C}_{reduced}$.

As a result, we get $\mathcal{C}_s = \mathcal{C}_{untouched} \sqcup \mathcal{C}_{reduced} \sqcup \mathcal{C}_{empty}$[1] as the set of clauses for F_s. As already stated, the set $\mathcal{C}_{empty}$ was always the empty set during our tests. Therefore, the formula F_s is not unsatisfiable by design. Let us assume, that $\mathcal{C}_{empty} = \emptyset$ for the rest of this chapter. Then, $\mathcal{C}_s = \mathcal{C}_{untouched} \sqcup \mathcal{C}_{reduced}$.

All together, F_s will be a formula that is smaller than F in terms of variable and clause count. The creation of F_s can be done very efficient, since finding a partition and creating a sub-formula by applying its characterizing partial assignment is not a difficult computational task.

With such a sub-formula F_s given, we are able to call March_ks to see if it is able to find a solution for it. Recall, that all F we are investigating are satisfiable. Let us assume, that March_ks is indeed able to find a satisfying assignment A_s for F_s.

[1] The symbol $\sqcup$ refers to the disjoint union of two sets.

The question then is, how A_s can be of use in order to solve F. The answer to that question is as follows.

We have F, with its variables $\mathcal{V}$ and its clauses $\mathcal{C}$. Furthermore, we have $CHAR(P)$, that gives us $F_s = CHAR(P)(F)$ with its (remaining) variables $\mathcal{V}_s$ and its (remaining) clauses $\mathcal{C}_s$. Additionally, we have A_s, which is a model for F_s. It is important to understand, that $CHAR(P)$ and A_s are assigning disjoint sets of variables within $\mathcal{V}$, since the only variables left to be assigned by March_ks to yield A_s are those, that are in F_s. But variables left in F_s are exactly those, that did not get a value assigned to them by $CHAR(P)$.

$CHAR(P)$ together with A_s now assign all variables in $\mathcal{V}$, since

$$\mathcal{V} = \underbrace{(\mathcal{V} \backslash \mathcal{V}_s)}_{\text{assigned by } CHAR(P)} \sqcup \underbrace{\mathcal{V}_s}_{\text{assigned by } A_s} .$$

Additionally, we have satisfied all clauses in $\mathcal{C}$ as well. This is because, $CHAR(P)$ satisfies $\mathcal{C}_{sat}$. On the other hand, A_s satisfies $\mathcal{C}_{untouched}$ and $\mathcal{C}_{reduced}$. Since clauses in $\mathcal{C}_{reduced}$ subsume their counterparts in $\mathcal{C}$, these counterparts in $\mathcal{C}$ are now as well satisfied by A_s. The clauses in $\mathcal{C}_{untouched}$ are equal to their counterparts in $\mathcal{C}$. Since A_s satisfies clauses in $\mathcal{C}_{untouched}$, it also satisfies their counterparts in $\mathcal{C}$. Since $\mathcal{C}_{empty} = \emptyset$, no further clauses in $\mathcal{C}$ remain, that must be satisfied.

Therefore, we have satisfied all clauses in F by the application of the combined assignments of $CHAR(P)$ and A_s. We call this a *combined solution*. Since a partition P can be found fairly easy with gNovelty+, and the construction of $CHAR(P)$ costs almost no additional computation time, the construction of the first component of the combined solution can be done efficiently.

Furthermore, our tests show, that the remaining sub-formula F_s is a rather drastic reduction from F. It has about $n/2$ variables and a clauses-to-variables ratio of about 2.8. This in turn should enable March_ks to find a solution for F_s in much less time, than it would have needed to find a solution for F. Our hope was, that the combined runtimes of gNovelty+ and March_ks to acquire the combined solution would be much lower, than the time March_ks would need to find a solution for F, which in turn would result in an overall speed-up for our hybrid solver.

The question now is, if indeed March_ks is able to find a satisfying assignment A_s for F_s. We ran empirical tests, which showed that this is *not* the case. March_ks always returned "unsatisfiable" when searching for a model for F_s.

Despite the fact, that this result is not very beneficial for our work, it still is an interesting result, because it means that gNovelty+ does indeed perform a very thorough search around a local minimum, and does not accidentally miss solutions near them.

We where then compelled to find out about the reasons for the inability of March_ks to find a solution for F_s. A closer look at the sub-formulas F_s revealed, that

$CHAR(P)$ was assigning too many variables to a wrong truth value. In other words, the confinement of the search space of F by $CHAR(P)$ to yield F_s is so drastic, that it also eliminates all solutions. Most of the time, March_ks did not even have to conduct a search, because its preprocessor was already able to find a conflict via some unit clauses in $\mathcal{C}_s$ of the sub-formula F_s.

Our intuition then told us, that we must somehow reduce the number of assigned variables in $CHAR(P)$. However, one cannot simply unassign variables in $CHAR(P)$ at random, because all of the assigned variables in $CHAR(P)$ belong to "near-solution" assignments. Furthermore, the number of assigned variables in $CHAR(P)$ is about $n/2$, which makes it impractical to randomly unassign them in order to create and test a less confining characterizing partial assignment.

The means to identify variables within $CHAR(P)$, that can be unassigned in order to inherit more of F's search space (and hopefully some solutions) into F_s's search space, are described in the next chapter.

Chapter 7

Superpartitions

As we have presented in the previous chapter, the confinement of the search space of F via partitions to yield F_s, is so drastic, that no solutions remain in the search space of F_s. F_s is therefore an unsatisfiable formula. The scheme to assist March_ks, that was presented in the last chapter, can therefore not work, since a combined solution needs a solution for F_s.

We concluded, that we must somehow unassign some variables in $CHAR(P)$, before the construction of F_s. This will then inherit more of F's search space into the search space of F_s. Our hope was, that such an F_s will then be satisfiable and can be used in the construction of a combined solution.

The technique to unassigning variables in $CHAR(P)$ comes in the form of superpartitions. These will be described next.

7.1 Defining the Term Superpartition

Since we are only investigating sufficiently large runs of gNovelty+, and have already given evidence, that numerous partitions within such a run exist, we can assume the existence of enough such partitions for the rest of this work. A superpartition is then defined as follows.

Definition 39 *Given a trajectory $\mathcal{T} = (A_b)_{b=0}^{w}$ for a run on a formula F. Furthermore, given the set of **all** $[i,j]$-partitions $\{P_1, \ldots, P_l\}$ in $\mathcal{T}$. We call the sequence*

$$S_l = (P_k)_{k=1}^{l} = ((A_{j_1})_{j_1=s_1}^{e_1}, \ldots, (A_{j_k})_{j_k=s_l}^{e_l}) \quad e_i < s_{i+1} < e_l \leq w \ \forall i \in [1, l-1]$$

a $[i,j]$-l-`superpartition` of $\mathcal{T}$. We call $l = LENGTH(S_l)$ the length of the superpartition. When appropriate, we will omit the information on the index of the length l for this superpartition, and simply write S.

While a $[i,j]$-partition is a finite sequence of assignments in a trajectory, a superpartition is a finite sequence of partitions in a trajectory. One should note, that the

definition of superpartitions demands, that all partitions within the first assignment and the w-th assignment of the trajectory are included in this sequence. While the succession function of assignments in a partition is defined by gNovelty+'s flipping function, the succession function of partitions in a superpartition is defined by the sort they have in the trajectory. This sort is, however, also defined by the flipping function of gNovelty+.

Before we can use superpartitions in a hybrid SAT solver, we must be sure about the feasibility of the construction of superpartitions.

7.2 The Construction of Superpartitions

The construction of $[i, j]$-l-superpartitions by gNovelty+, as well as the construction of $[i, j]$-partitions, can be done "on-the-fly". Since gNovelty+ finds partitions one after another, and simply creates arrays of assignments for such partitions, a superpartition is then simply given by the set of these arrays. When the number of flips gNovelty+ performed reaches a certain cutoff (i.e. flip w is performed), this set of arrays then already represents the $[i, j]$-l-superpartition.

This means, that there is no additional computational overhead in the construction of a superpartition.

The existence of superpartitions can be deduced from the existence of partitions (see section 6.5.1) on sufficiently large runs of gNovelty+. We will now explain, how we make use of superpartitions.

7.3 Characterizing Partial Assignments of Superpartitions

Recall our idea of confining the search space of March_ks on F, by providing it with a reduced formula F_s. We have shown, that the confinement via characterizing partial assignments of partitions is too drastic for such an F_s to contain any solutions. We concluded, that we must somehow unassign certain variables in $CHAR(P)$, that is used to create F_s from F. However, this unassigning of variables in $CHAR(P)$ can not be done, since one does not know which of its variables to unassign.

We circumvent this problem, by using a different kind of characterizing partial assignment: the characterizing partial assignment of superpartitions. They are defined as follows.

Definition 40 *Given a formula F and a trajectory $\mathcal{T}$ for a sufficiently large run on F. One can find a $[i, j]$-l-superpartition $S_l = (P_k)_{k=1}^{l}$ in $\mathcal{T}$. A* `characterizing partial assignment` *of S_l (denoted $SUPER(S_l)$), is a partial assignment that assigns all the* 1.0*-tendentiously set variables in all assignments of all partitions P_i in S_l ($i \in [1, l]$).*

Example 24 *Given the two partitions*
$P_1 = ((0,0,0,0),(0,0,0,1))$ *and* $P_2 = ((1,0,1,0),(1,0,0,0),(1,0,0,1))$.
Let the superpartition be $S = (P_1, P_2)$. *It is* $SUPER(S) = (?,0,?,?)$, *since the second variable is the only one, that had the same value assigned to it in all assignments of all partitions in* S. *It is* $LENGTH(S) = 2$.

It is sometimes beneficial, to construct $SUPER(S)$ form the characterizing partial assignments of the partitions in S. We then write
$$SUPER(S) = SUPER(CHAR(P_1), \ldots, CHAR(P_l)).$$
The following example clarifies this.

Example 25 *Given the two partitions*
$P_1 = ((0,0,0,0),(0,0,0,1))$ *and* $P_2 = ((1,0,1,0),(1,0,0,0),(1,0,0,1))$
from the previous example. We get
$$CHAR(P_1) = (0,0,0,?) \text{ and } CHAR(P_2) = (1,0,?,?).$$
It then is
$$SUPER(CHAR(P_1), CHAR(P_2)) = SUPER((0,0,0,?),(1,0,?,?)) = (?,0,?,?),$$
as given in the previous example. As we can see, the characterizing partial assignment of S *assigns less variables, than the characterizing partial assignments of* P_1 *and* P_2.

We have already defined the term *search space vicinity* for a partition P. We will now define, what we mean with the search space vicinity of a superpartition.

Definition 41 *Given a formula* F, *a corresponding trajectory, and a superpartition* S *in this trajectory. We can calculate the characterizing partial assignment* $SUPER(S)$. *We say an assignment* A, *that* $SUPER(S)$ *can be extended to in the search space of* F, *is*

- *in the* `core` *of* S, *iff it is in a core of any of the partitions in* S.
- *in the* `search space vicinity` *of* S, *iff it is not in a core of any of the partitions in* S.

Example 26 *Again, given the partitions*
$P_1 = ((0,0,0,0),(0,0,0,1))$ *and* $P_2 = ((1,0,1,0),(1,0,0,0),(1,0,0,1))$,
and the superpartition $S = (P_1, P_2)$. *The assignment* $(0,0,0,0)$ *is in the core of* P_1, *and therefore, it is in the core of* S. *The assignment* $(1,0,0,0)$ *is in the core of* P_2, *and therefore, it is in the core of* S *as well. The assignment* $(1,0,1,1)$ *is neither in the core of* P_1, *nor is it in the core of* P_2. *But* $SUPER(S) = (?,0,?,?)$ *can also be extended to* $(1,0,1,1)$, *and therefore,* $(1,0,1,1)$ *is in the search space vicinity of* S.

This search space vicinity of a partition P was exactly the search space of the sub-formula $CHAR(P)(F) = F_s$. Our initial hope was to find a solution A_s for

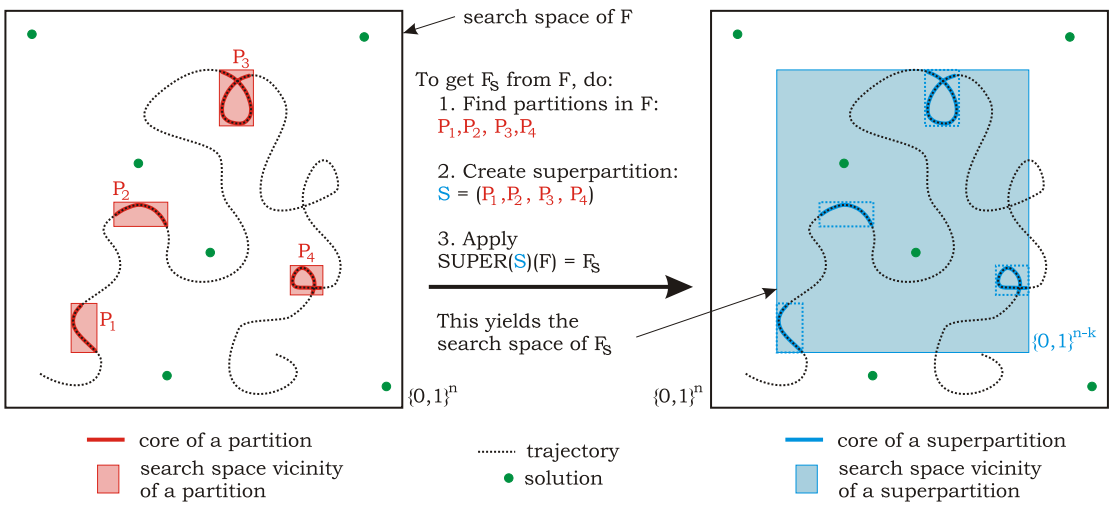


Figure 7.1: A visualization of the terms *partition*, *superpartition*, *core*, and search *space vicinity*.

F_s, so that we where able to obtain a combined solution from $CHAR(P)$ and A_s. However, this was not the case. All the sub-formulas F_s, that we have generated in our empirical tests, have been unsatisfiable.

How characterizing partial assignments of superpartitions are supposed to help us circumvent the too drastic reduction of F, in order to yield a satisfiable sub-formula, is explained in the next section.

7.4 The Use of Characterizing Partial Assignments of Superpartitions

The characterizing partial assignment of a superpartition S assigns a smaller number of variables than any characterizing partial assignment of a partition in S. Therefore, the search space vicinity of a superpartition is larger than the search space vicinity of a partition (see figure 7.1).

The question now is, how fast the search space vicinity of a superpartition grows with its length increasing. The answer is, that the search space vicinity for S grows rapidly with its length. In other words, the longer a superpartition, the larger the search space of the sub-formula we generate via $SUPER(S)$ (see Appendix on page 183). Therefore, with increasing l the chances raise, that we inherit a solution from the search space of the original formula into the search space of the sub-formula (see figure 7.1). On the other hand, an increase of l also means, that the search space of F_s grows, which in turn raises the solving time of March_ks for F_s.

As it was with partitions, the construction of $SUPER(S)$ and the corresponding sub-formula F_s can be done efficiently. The runtime for gNovelty+ for acquiring S and constructing F_s was negligible for all our tested formulas (less than a second).

This in turn means, that the time it needs to find a combined solution mostly depends on the time it needs to find a solution for F_s.

This leaves us with the question, what properties a superpartition must have (i.e. length, number of assigned variables, etc.) in order to yield an easily satisfiable F_s. This question is answered in the next section.

7.5 Promising Superpartitions

Many superpartitions (of different lengths) can be created from a trajectory of a run on F. However, not all of the superpartitions can be used to create a satisfiable sub-formula F_s. Furthermore, the solving times of March_ks differ for different sub-formulas F_s.

We need to identify superpartitions, for which the application of its characterizing partial assignment on F creates a satisfiable sub-formula F_s, for which the solving time of March_ks on F_s is smaller than the solving time on F. We call superpartitions, for which the above is true, *promising superpartitions*, as given in the following definition.

Definition 42 *Given a formula F, and a trajectory $\mathcal{T}$ from a sufficiently large run on F. A superpartition S is called* `promising`*, iff $F_s = SUPER(S)(F)$ is a satisfiable sub-formula of F, that can be solved by March_ks in less time than March_ks would need to solve F.*

Our goal now is to determine properties, that such promising superpartitions must have. In order to do so, we performed the following experiment.

7.5.1 Identifying the Properties of Promising Superpartitions

We create six runs $\mathcal{R}_i$ ($i \in [1, 6]$) for each of the formulas F from table 5.1 on page 132. Each of these runs contains a trajectory $\mathcal{T}_i \in \mathcal{R}_i$.

For each of these trajectories $\mathcal{T}_i$, we determine the median careening value c_i of the objective function. With such a c_i given, we can define the range of interest for the corresponding trajectory to be $[c_i - 5, c_i + 5]$, and find all m_i different partitions in that range for the $\mathcal{T}_i$. We will denote these partitions with $P^i = \{P^i_1, \ldots, P^i_{m_i}\}$.

With these partitions given, we create all possible superpartitions for the trajectory $\mathcal{T}_i$, denoted $S^i_{b_i} = (P^i_1, \ldots, P^i_{b_i})$ with $b_i \in [1, m_i]$.

With all the superpartitions $S^i_{b_i}$ for all six runs i on a formula F, we can create all possible sub-formulas $F^i_{b_i} = SUPER(S^i_{b_i})(F)$ for each run i. We can then check, whether

- the sub-formula $F^i_{b_i}$ is satisfiable (i.e. the superpartition $S^i_{b_i}$ suffices in creating a satisfiable sub-formula).
- the solving time of March_ks for $F^i_{b_i}$ is smaller than the solving time of March_ks for F (i.e. March_ks can solve the sub-formula faster than the original formula).

Any $S^i_{b_i}$, for which the above two properties hold, is a promising superpartition. Since we are interested in identifying promising superpartitions according to some properties, we must also collect additional data for each of the $S^i_{b_i}$. The set of additional data we want to acquire is as follows.

- The number of partitions in $S^i_{b_i}$, i.e. b_i.
- The number of assigned variables in $S^i_{b_i}$.
- The clauses-to-variables ratio of $F^i_{b_i} = SUPER(S^i_{b_i})(F)$.
- The number of reduced clauses in $F^i_{b_i} = SUPER(S^i_{b_i})(F)$. We call a clause in $F^i_{b_i}$ reduced, iff this clause is not satisfied and got at least one of its literals removed by the application of $SUPER(S^i_{b_i})$ on F. Removing a literal happens, iff the assignment to the corresponding variable of that literal makes it evaluate to FALSE. In other words, a clause in $F^i_{b_i}$ is called reduced, iff it is not yet satisfied an has less literals than its counterpart in F.

This set of data should then enable us to identify properties of superpartitions $S^i_{b_i}$, that all promising superpartitions have in common. An exemplary set of data of one run on the `unif-k3-r4.26-v360-c1533-S1422217194-16.SAT.shuffled.cnf` formula is presented in table 7.1.

The data now allows us to draw an important conclusion, even before determining properties to identify promising superpartitions: we can now always create a satisfiable sub-formula F_s. As can be seen in table 7.1, there is a certain border for the length of a superpartition (in this case $b_i = 3$), after which all superpartitions create a satisfiable $F^i_{b_i}$. For this $F^i_{b_i}$, we can now find a solution that can participate in the combined solution on F. This now solves the problem we had in the previous chapter when only using partitions for the simplification of F. In this case, all F_s created via a characterizing partial assignment of a partition had been unsatisfiable and no combined solution could be created.

When taking a look at the complete dataset (not presented here in detail), one realizes, that the number of reduced clauses in $F^i_{b_i}$ can be used to quite effectively identify promising superpartitions. The problem is, that this number varies drastically between different formulas, even if these formulas have the same size. Take a look at table 7.2. Whenever a sub-formula $F^i_{b_i}$ can be created, for which the number of reduced clauses is within the given interval, the sub-formula is satisfiable. However, with the data being so erratic, we are unable to provide a general rule for what the number of reduced clauses must be for a promising superpartition.

Run	$S^i_{b_i}$	Used partitions b_i	Assigned variables in $SUPER(S^i_{b_i})$	Clauses-to-variables ratio of $F^i_{b_i}$	Reduced clauses in $F^i_{b_i}$	$F^i_{b_i}$ satisfiable	March_ks speed-up for $F^i_{b_i}$
3	S^3_1	1	86	3.40	190	false	241.02
	S^3_2	2	55	3.73	126	false	90.62
	S^3_3	3	47	3.78	112	true	103.92
	S^3_4	4	41	3.83	98	true	90.62
	S^3_5	5	35	3.89	78	true	27.83
	S^3_6	6	31	3.95	65	true	11.69
	S^3_7	7	20	4.04	37	true	10.06
	S^3_8	8	14	4.10	27	true	2.34
	S^3_{12}	12	13	4.11	25	true	1.35
	S^3_{13}	13	9	4.18	11	true	1.67
	S^3_{15}	15	8	4.17	9	true	1.54
	S^3_{16}	16	6	4.18	8	true	1.54
	S^3_{17}	17	5	4.20	5	true	0.53
	S^3_{19}	19	4	4.22	3	true	3.87
	S^3_{49}	49	3	4.23	1	true	0.39

Table 7.1: Dataset for run number three on the `unif-k3-r4.26-v360-c1533-S1422217194-16.SAT.shuffled.cnf` instance (360 variables 1533 clauses). Superpartitions, that are not presented in the table (e.g. S^3_9), are those that are equal to their predecessor. These superpartitions can be ignored. The March_ks speed-up column presents a factor on how much faster March_ks is in finding a solution for $F^i_{b_i}$ in comparison to F.

Formula	Var.	Cls.	Interval of reduced clauses in $F^i_{b_i}$ for promising superpartitions
uf250-011.cnf	250	1065	$[1, 4]$
unif-k3-r4.26-v360-...-16.SAT.shuffled.cnf	360	1533	$[20, 46]$
unif-k3-r4.26-v360-...-08.SAT.shuffled.cnf	360	1533	$[73, 77]$
unif-k3-r4.26-v400-...-05.SAT.shuffled.cnf	400	1704	$[44, 72]$
unif-k3-r4.26-v400-...-13.SAT.shuffled.cnf	400	1704	$[45, 69]$
unif-k3-r4.25-v450-...-11.SAT.shuffled.cnf	450	1912	$[88, 93]$
unif-k3-r4.25-v450-...-15.SAT.shuffled.cnf	450	1912	$[6, 65]$
unif-k3-r4.26-v500-...-05.SAT.shuffled.cnf	500	2130	$[37, 40]$
unif-k3-r4.26-v500-...-12.SAT.shuffled.cnf	500	2130	$[5, 5]$

Table 7.2: The reduced clauses intervals of promising superpartitions for all tested formulas. The names of formulas have been truncated. As we can see, the intervals for the reduced clauses of promising superpartitions are quite unsteady.

While the necessity for an upper border for the number of reduced clauses is quite clear (an upper border for the reduction), we must explain why we provide intervals, i.e. a lower border as well. The reason for this is, that the solving times of March_ks to find a solution for one of these formulas is *almost* always smaller than the solving time for F. It is important to note, that March_ks sometimes needed longer to solve a sub-formula, than it needs to solve the original formula (see table 7.1 for superpartition S^3_{17}).

This is a bit of a curiosity, since all sub-formulas are smaller in terms of variable and clause count and have always a clauses-to-variables ratio smaller than 4.26. We strongly believe, that this has something to do with the number of solutions a sub-formula has in total. If a sub-formula has few solutions (in comparison with the original formula), March_ks has more trouble finding one of them. Therefore, one must not only give an upper border for the number of reduced clauses, but also a lower border.

This lower border then ensures, that the sub-formula is still reduced strong enough, such that March_ks can find a solution fast even if the number of solution for this sub-formula is comparatively low.

We abstain from presenting other properties of promising superpartitions, because these are as well not usable for creating a rule that safely identifies promising superpartitions for all formulas. Instead, we want to give a few reasons, why the identification of promising superpartitions has so far been unsuccessful.

7.5.2 Current Flaws in the Creation of Superpartitions

Superpartitions are created from partitions. However, the partitions are found by gNovelty+ according to the range of interest that the user specifies. The range we used in the above mentioned experiments is $[c_i + d_{lower}, c_i + d_{upper}]$, whereas c_i is the median careening value for the corresponding trajectory, and the d_{lower} and d_{upper} parameters have been held fixed to 5 for all the formulas. We strongly believe, that this setting is not optimal.

Using the median careening value for positioning the range around it is a start, since this is the area in which most partitions can be found. Hence, many options for creating a superpartition exist. Additionally, the width of the range affects the number and properties of found partitions as well. We think, that d_{lower} and d_{upper} should reflect the deviation of the objective function values around c_i. This would then enable gNovelty+ to find partitions, that better reflect the shape of the search space it was investigating. With such partitions given, one might be able to create superpartitions that will better suffice in creating an easily satisfiable F_s. However, we have not yet investigated this.

Furthermore, we have mentioned, that too short partitions are ignored. A partition is called to short, when it contains less than n assignments. This lower border for

the length of a partition is somewhat arbitrary. When the objective function values are fluctuating drastically, gNovelty+ might be able to find numerous partitions, but most of them will be too short. Therefore, a lot of the search-time of gNovelty+ will not yield any usable partitions. This in turn reduces the options on creating superpartitions, and therefore, the options on creating an easily satisfiable F_s.

We believe, that the lower border for the length of a partition should somehow reflect the ruggedness of the search space. More exactly, when the fluctuation of the objective function values is high, the lower border for the length of partitions should be reduced. However, we have not yet been able to investigate this in more detail.

At the moment, partitions are a finite sequence of assignments, and superpartitions are a finite sequence of partitions. We create superpartitions by simply appending the found partitions chronologically. It is yet unclear, if this scheme for the creation of superpartitions is optimal.

One could, for example, define a distance measure for partitions, and create a superpartition from only those partitions, that have certain distance properties. This could ensure, that the growth of the search space of F_s is as steady as possible. Thereby, it could be easier to detect promising superpartitions.[1]

Furthermore, other schemes of combining partitions are thinkable. Given the set of partitions $\{P_1, \ldots, P_k\}$, one could create a superpartition by selecting the first partition at random, lets say $S_1 = (P_1)$. Then, one measures the distance of S_1 to all remaining partitions $P_i, i \in [2, k]$, and append the P_i to S_1, for which the distance is the smallest. Let us use P_3 now. We then have $S_2 = (P_1, P_3)$. After that, one measures the distance between S_2 to all the remaining partitions, and again selects the one with the smallest distance.

With an appropriate distance measure, the growth of the search space of $F_b = SUPER(S_b)(F)$ is as slow as possible. The properties of the S_b and F_b will then change quite steadily, which makes the identification of promising superpartitions easier. However, we are not yet sure of what kind of distance measure and combining schemes are beneficial.

In summary, we have not yet been able to identify promising superpartitions, since it is yet unclear in which way to best construct partitions and superpartitions. However, we want to provide a proof of concept for the parameters we already used to hybridize gNovelty+ and March_ks. This proof of concept is presented in the next section.

7.5.3 Proof Of Concept for the Usage of Superpartitions

Recall the experiment from section 7.5 (on page 157). With the gathered data on all formulas F and all runs $\mathcal{R}_i$ on F, we can check, if an improvement for the solving

[1] Adrian Balint was the first to come up with such an idea, and we want to thank him for sharing it with us.

time of F is always possible. In other words, we want to check, if there is at least one promising superpartition per run, that can be used to find a combined solution in less time, than it would take March_ks to find a solution for F.

Table 7.3 presents the run-times of March_ks on all sub-formulas F_b^i, for which the optimal superpartition S_b^i has been used for creation. A superpartition is called optimal, if it creates a sub-formula for this run, for which the solving time for March_ks is the smallest of all available sub-formulas for this run.

Formula	**Run-time of March_ks on F**	**Run**	S_b^i	**Runtime of March_ks on F_b^i**	**Speedup**
uf250-011.cnf	968	1	S_{13}^1	125	7.74
		2	S_7^2	235	4.11
		3	S_9^3	469	2.06
		4	S_8^4	204	4.71
		5	S_{12}^5	718	1.34
		6	S_{21}^6	344	2.81
unif-...-v360-...-16.SAT.shuffled.cnf	11328	1	S_6^1	4265	2.65
		2	S_8^2	125	90.62
		3	S_3^3	109	103.92
		4	S_{22}^4	657	17.24
		5	S_4^5	93	121.80
		6	S_5^6	63	179.80
unif-...-v360-...-08.SAT.shuffled.cnf	1203	1	S_4^1	46	26.15
		2	S_5^2	47	25.59
		3	S_4^3	31	38.80
		4	S_7^4	94	12.79
		5	S_3^5	31	38.80
		6	S_{18}^6	62	19.40
unif-...-v400-...-05.SAT.shuffled.cnf	12667	1	S_{61}^1	344	36,82
		2	S_6^2	94	134.75
		3	S_{10}^3	31	408.61
		4	S_4^4	31	408.61
		5	S_4^5	78	162.39
		6	S_3^6	47	269.51
unif-...-v400-...-13.SAT.shuffled.cnf	35969	1	S_3^1	63	570.93
		2	S_4^2	93	386.76
		3	S_4^3	94	382.64
		4	S_9^4	94	382.64
		5	S_{26}^5	5782	6.22
		6	S_{18}^6	5312	6.77
unif-...-v450-...-11.SAT.shuffled.cnf	4328	1	S_{11}^1	734	5.89
		2	S_{10}^2	110	39.34
		3	S_4^3	203	21.32
		4	S_{12}^4	625	6.92
		5	S_9^5	484	8.94
		6	S_4^6	109	39.70
unif-...-v450-...-15.SAT.shuffled.cnf	418281	1	S_{24}^1	172	2431.86
		2	S_{14}^2	531	787.72
		3	S_{42}^3	2640	158.43
		4	S_{20}^4	281	1488.54
		5	S_6^5	47	8899.59

		6	S^6_{36}	1078	388.01
unif-...-v500-...-05.SAT.shuffled.cnf	14141	1	S^1_{16}	9609	1.47
		2	S^2_{14}	3422	4.13
		3	S^3_{44}	3219	4.39
		4	S^4_{47}	1140	12.40
		5	S^5_{17}	1469	9.62
		6	S^6_{16}	359	39.38
unif-...-v500-...-12.SAT.shuffled.cnf	369890	1	S^1_{32}	321093	1.15
		2	S^2_{81}	169656	2.18
		3	S^3_{15}	73375	5.04
		4	S^4_{62}	220891	1.67
		5	S^5_{47}	137953	2.68
		6	S^6_{40}	1047000	0.35

Table 7.3: The run-time of March_ks for finding a solution for a sub-formula, that has been created with the optimal superpartition (i.e. the superpartition, that creates a sub-formula, for which the run-time of March_ks is the smallest). The names of formulas have been truncated. Run-times are given in milliseconds.

As we can see from table 7.3, the construction of a promising superpartition was almost always possible. The only exception is run six on the last formula. We therefore conclude, that promising superpartitions do exist, even with the sub-optimal parameter setting used in the experiment to gather the above data.

Even though we are not yet able to safely identify promising superpartitions, we take the above data as a proof of concept for our approach to hybridize gNovelty+ and March_ks via superpartitions. We will now discuss the quality of the presented results of our empirical study.

7.6 Discussion of the Empirical Results

We performed numerous empirical studies, in order to create a new approach to hybridize gNovelty+ and March_ks. Since a substantial empirical study is very time consuming, we had to abridge the set of used formulas.

We used nine satisfiable uniform random 3-SAT formulas, as presented in section 5.1.1 (see page 132). Since nine formulas are hardly representative for all uniform random 3-SAT formulas, one must exercise caution when interpreting empirical results.

During some of the experiments, a number of runs had to be performed in order to gain a significant data set for one formula. We performed at most six such runs. Here as well, the number of runs might not create a data set that can be seen as representative for the respective formula.

All together, the conclusions we can draw from our tests so far are considered to be preliminary. We feel that there is a great need for repeating the empirical studies using a broader set of formulas and a larger set of runs.

We have now finished our presentation of the studies that led to our new approach to create a hybrid SAT solver with gNovelty+ and March_ks. We will summarize this approach in the next section.

7.7 Using Superpartitions to Create a new Hybrid SAT Solver with gNovelty+ and March_ks

Will will now summarize the approach to combine gNovelty+ and March_ks, that has been developed in the last three chapters. We will call the corresponding hybrid SAT solver *HYBRIZZL*.

Given a uniform random 3-SAT formula F, with n variables in $\mathcal{V}$ and $4.26n$ clauses in $\mathcal{C}$. In order to conduct search, HYBRIZZL will first call for gNovelty+, to collect data, that is needed to create a sub-formula F_s. Collecting this data works as follows.

gNovelty+ will start to perform a run on F. During the first n flips (homing-in), no additional data is collected. During the n-th flip and the $10n$-th flip, data is gathered to calculate the median careening value c_i for this run. c_i represents the median of all objective function values for the assignments checked up to the $10n$-th flip. With the median careening value given, we define the range of interest to be $[c_i-5, c_i+5]$. From the $10n$-th flip onwards, find partitions $\{P_1, \ldots, P_l\}$ as explained in section 6.3 (see page 145). If gNovelty+ is able to find a solution for F during its calculations, we return this as a model for F and HYBRIZZL terminates.

As soon as a promising superpartition $S_l = (P_1, \ldots, P_l)$ can be created from the found partitions, the search of gNovelty+ is put on hold. As already stated, it is not yet clear on how promising superpartitions can be identified for all formulas. Anyway, we use S_l to create $F_s = SUPER(S_l)(F)$, a sub-formula of F. By design, March_ks should be able to find a solution for F_s faster than for F.

Therefore, we call March_ks on F_s, and wait for it to return a model A_s for it. In section 6.7 (see page 148), we have already explained, that $CHAR(P)$ and A_s assign different variable in $\mathcal{V}$. The same is true for $SUPER(S_l)$ and A_s. Hence, we can create a combined solution $A_c = SUPER(S_l) \sqcup A_s$, as soon as March_ks returns a model for F_s. We return A_c as a solution for F.

In case March_ks returns "unsatisfiable" for F_s, which can happen if F is unsatisfiable or S_l was not sufficient in creating a satisfiable F_s, we must continue the search with gNovelty+. We therefore continue the search with gNovelty+ at the point we have put it on hold. We continue to find b_1 additional partitions with gNovelty+, and as soon as a new superpartition S_{l+b_1} can be created (that differs to S_l with respect to the characterizing partial assignments), we create a new sub-formula F_{s+b_1}. We again call March_ks to search a model for this sub-formula.

We continue the above scheme, until either a combined solution can be found, or a superpartition $S_{l+b_1+\ldots+b_r}$ is created, for which the characterizing partial assignment assigns no variables at all. This will be the case at some point, because the diversified

search of gNovelty+ will eventually flip all variables in assignments within the range, and thus, no more 1.0-tendentiously set variables will exist to be assigned in the superpartition. In this case, the created sub-formula $F_{s+b_1+\ldots+b_r} = F$.

When March_ks is called on $F_{s+b_1+\ldots+b_r} = F$, it will return a distinct answer. Iff F is unsatisfiable, March_ks will report so, and HYBRIZZL can return this result. Iff March_ks is able to find a solution for F, HYBRIZZL can return this as a model. The just explained functioning of HYBRIZZL is presented in listing 7.1.

Listing 7.1: HYBRIZZL.

```
HYBRIZZL(F){
    flipcount = 0; //the current flip of gNovelty+
    A = create random starting assignment;
    gNovelty+(F, A){
        if (A is a model for F){
            //HYBRIZZL terminates
            return A;
        }
        while (flipcount ∈ [0, n − 1]){
            //no additional task for gNovelty+
        }
        while (flipcount ∈ [n, 10n]){
            //collect data for calculating the c_i.
            if (flipcount == 10n){
                //calculate and set c_i, set range
                range = [c_i − 5, c_i + 5];
            }
        }
        while (flipcount ∈ [n, ∞)){
            //collect partitions P for range in array S
            //S symbolizes the superpartition that is created so far
            if (S is promising){
                //create sub−formula and call March_ks with it
                F_s = SUPER(S)(F);
                A_s = March_ks(F_s);
                if (SUPER(S) assigns no variables){
                    //it is F_s = F
                    //HYBRIZZL terminates
                    return A_s;
                } else {
                    if (A_s == unsatisfiable){
                        //no combined solution can be created
                        //HYBRIZZL continues search with gNovelty+
                        continue;
                    } else {
                        //return combined solution
                        //HYBRIZZL terminates
                        return SUPER(S) ⊔ A_s;
                    }
                }
            }
        }
        A = flip(A);
        flipcount++;
    }
}
```

As soon as we are able to safely identify promising superpartitions, HYBRIZZL should be able to outperform March_ks on satisfiable instances. Furthermore, HYBRIZZL should be quite competitive with gNovelty+, because we did not alter its search mechanics. Additionally, HYBRIZZL is a complete SAT solver, and is therefore able to prove the unsatisfiability of formulas. This in turn will give HYBRIZZL an advantage over gNovelty+.

With the above scheme given, we are finished in developing a new approach for creating a Hybrid SAT solver. We will now summarize our efforts in the following section.

Summary of Part III

The efforts presented in this part have been twofold. First, we performed a study on the search behavior of gNovelty+ in order to gain an idea on how gNovelty+ can be used in a hybrid SAT solver. This study revealed several properties of gNovelty+'s search behavior, like the existence of the careening property and the existence of t-tendentiously set variables (see chapter 5 on page 131).

We investigated these properties to gain an understanding for why they exist. We concluded, that careening and the t-tendentiously set variables exist, because of the local search mechanics gNovelty+ uses to advance the search. We used plateau connection graphs to explain these mechanics and effects (see section 5.3.2 on page 138).

Second, we tried to make use of the properties of gNovelty+'s search behavior. Therefore, we introduced the term of a partition (see chapter 6 on page 143). Partitions where then used to support the solver March_ks by confining the search space it had to work on. The general idea was to create a combined solution with the work done by gNovelty+ and March_ks. As we have explained, partitions themselves have not been sufficient to find such a combined solution (see section 6.7 on page 148).

In chapter 7 (see page 153), we introduced the term of a superpartition. With superpartitions, we where able to create combined solutions from the combined search efforts of gNovelty+ and March_ks. In order to gain a speed-up with respect to the search time of March_ks on the original formula, only a subset of all superpartitions can be used. Superpartitions that yielded a gain in the search time where called promising superpartitions (see section 7.5 on page 157).

In order to use promising superpartitions, one must be able to identify them. Our first efforts in identifying promising superpartitions have been presented in section 7.5.1 (on page 157). However, our efforts have not yet been successful. A few reasons why we have not been able to identify promising superpartitions are given in section 7.5.2 (see page 160).

In order to provide at least a proof of concept, that the combined solution from gNovelty+ and March_ks can be found in less time than March_ks would need to find a solution on the formula alone, we presented a study in section 7.5.3 (on page 161). This study provided evidence, that promising superpartitions do exist for almost all runs on all formulas, and hence, a speed-up can be gained in almost all cases.

Since we believe, that our empirical studies have not yet been very significant, we shortly discussed the need for acting with caution when interpreting their results (see section 7.6 on page 163).

Finally, we summarized our efforts and presented HYBRIZZL, the approach for hybridizing gNovelty+ and March_ks (see section 7.7 on page 164). We presented pseudo-code to clarify the general idea of HYBRIZZL (see listing 7.1 on page 165).

We are now finished with presenting our new approach for creating a hybrid SAT solver. The next part will conclude this diploma thesis and gives various ideas for future work.

So that I may perceive whatever holds
The world together in its inmost folds.
from "Faust: The Tragedy Part One"
– Johann Wolfgang von Goethe

PART IV

Conclusions and Future Work

Our work on this diploma thesis is coming to an end, and it is time to recapitulate our efforts. We will therefore give a summary of the most important details, as well as provide numerous conclusions in chapter 8. We will furthermore use the opportunity to present several ideas for future work in chapter 9.

Chapter 8

Conclusions

The main topic of this diploma thesis was the propositional satisfiability problem. Two tasks have been performed in this work. First, we gave a comprehensive overview of the state-of-the-art in SAT solving these days. Therefore, we have presented the two main SAT solver paradigms DPLL and SLS, and explained several modern SAT solvers. Second, we have studied the possibilities of developing a hybrid SAT solver, and presented a new approach to hybridize gNovelty+ and March_ks.

In Part I, we laid the technical foundation for our work by explaining the propositional logic and providing definitions for the most important terms used in connection with SAT.

We used these terms to give widely accepted definitions for SAT and k-SAT. Based on these definitions, we motivated the research on SAT by presenting its fundamental relevance to several science domains, like Engineering, Computer Science, and Math. We also presented a scheme for the application of SAT solvers to problem solving in general.

We furthermore outlined the different categories of SAT formulas: industrial, handmade, and random, and used uniform random k-SAT formulas to explain the term of problem hardness. Additionally, we presented empirical results, that not all random formulas are equally difficult to solve. Using these results, we explained, that uniform random formulas with a clauses-to-variables ratio of 4.26 are the hardest. These formulas have then been declared to be of special interest for the rest of this work.

We finished Part I by explaining, that the practical relevance of SAT increases with the number and types of formulas that can be solved. We therefore concluded, that further research on SAT solvers is worthwhile.

In Part II, we provided a survey of the state-of-the-art in SAT solving, by explaining the two major SAT solver paradigms: DPLL and SLS.

We started this survey by explaining the DPLL paradigm. Therefore, we first pre-

sented DLL, the basic algorithm for many modern DPLL SAT solvers. We then presented two DPLL solvers: March_ks, as a representative for look-ahead solvers, and RSat, as as a representative for conflict-driven solvers.

The description of March_ks included a detailed explanation of its key-features, like the look-ahead mechanisms, as well as variable and selection heuristics, branching heuristics and the functioning of its preprocessor.

For RSat, we presented the iterative SAT procedure, as well as the mechanisms to learn conflict clauses and the means to perform clause database maintenance, restarts, and progress saving.

We used March_ks and RSat to point out the general advantages and disadvantages of DPLL SAT solvers. Advantages of DPLL solvers are their completeness, their ability to find all solutions for a formula, and their ability to provide unsatisfiability proofs. Disadvantages of DPLL solvers are, that they lack a decent scalability and are quite complex in their design.

We then continued the SAT solver survey with the SLS paradigm. Therefore, we first presented GSAT to explain the basic idea behind local search for SAT. Two modern SLS solvers have then been explained, in order to point out several of the ideas used by todays state-of-the-art SLS solvers. The first of the presented SLS solvers was adaptG^2WSAT$_0$, a representative for random walk solvers. Based on the description of adaptG^2WSAT$_0$, we explained the functioning of gNovelty+, which is a representative for clause weighting solvers.

The description of adaptG^2WSAT$_0$ included the explanation of its several key-features, like random walks, the adaptive parameter tuning scheme for noise, and the gradient-based variable score update scheme. We also explained several variable selection heuristics, that are used to determine the next variable to flip.

The description of gNovelty+ included the explanation of clause weights, as well as further variable selection heuristics.

We used adaptG^2WSAT$_0$ and gNovelty+ to present the advantages and disadvantages of SLS SAT solvers. While their major advantages are their good scalability, fast solving times on satisfiable formulas, and their comparatively simple design, these solvers are susceptible to local minima and are unable to detect the unsatisfiability of a formula.

We finished Part II with a comparison of modern DPLL and SLS solvers. One of the major conclusions from this comparison was, that DPLL and SLS solvers complement each other very well. Therefore, we concluded that a combination of both paradigms would be worthwhile.

In Part III, we developed a new approach on how such a combination can be performed in the form of a hybrid SAT solver. Therefore, we first performed a study to better understand the search behavior of gNovelty+. We discovered several proper-

ties of gNovelty+'s search behavior, like the careening property and the existence of t-tendentiously set variables. We then investigated why these properties exist, and explained them using plateau connection graphs.

In order to make use of these properties in a hybrid SAT solver, we developed partitions. These partitions where then used to present a preliminary idea for the combination of gNovelty+ and March_ks. In this idea, gNovelty+ and March_ks were supposed to work together, in order to find a combined solution.

Our empirical test, however, uncovered a basic flaw in this initial idea. In order to overcome this flaw, we further developed superpartitions. With these superpartitions, we where able to create combined solutions as originally intended. Further empirical studies revealed, that combined solutions can only be created with a gain in solving time (with respect to the solving times of March_ks when left to itself), when a certain type of superpartitions is used. These superpartitions were called promising superpartitions.

In order to make use of promising superpartitions in a hybrid SAT solver, one must be able to safely identify them. We conducted several tests to find properties of promising superpartitions. We hoped, that we could use these properties to provide a general rule for identifying them. Unfortunately, our efforts to find such a rule have not yet been successful.

We were then compelled to provide at least a proof of concept for using superpartitions in a hybrid SAT solver. We did so by providing empirical evidence, that promising superpartitions exist. In fact, we were able to show the existence of at least one promising superpartition for 53 of the 54 created runs on the nine studied formulas. This study further revealed, that there is a reasonable increase in the solving time, when the optimal promising superpartition is used to create a combined solution.

We concluded Part III by providing a summary of our idea in the form of the HYBRIZZL hybrid SAT solver, that combines gNovelty+ and March_ks via superpartitions. As we have stated in this summary, HYBRIZZL is a complete SAT solver, that is able to outperform March_ks on satisfiable instances. Furthermore, HYBRIZZL is able to detect the unsatisfiability of formulas, and therefore, has an advantage over gNovelty+ as well.

Since the identification of promising superpartitions is not yet possible, HYBRIZZL is considered to be a preliminary idea for creating a hybrid SAT solver. However, we feel that further research on this idea would be worthwhile, and therefore, we provide opportunities for further research in the next chapter.

Chapter 9

Future Work

The major question that remains when it comes to the feasibility of HYBRIZZL is, how promising superpartitions can be safely identified. We believe, that our inability to provide a rule for such an identification comes from the hardly optimal parameter settings we used, when performing our preliminary empirical tests.

Promising superpartitions are, as any superpartition, created from partitions. Partitions and their properties are, however, determined by the range of interest for gNovelty+'s search. We therefore propose, that a study must be conducted in order to determine the impact of range properties on the properties of a (super)partition.

We have already provided the first study to uncover the whereabouts of partitions according to certain range properties. For example, we have seen, that most of the partitions can be found, when the range is positioned around the median careening value c_i of the objective function. However, it is yet unclear, what other properties the range must have. To be more exact, it must be clarified, what the border settings d_{lower} and d_{upper} for a range $[c_i - d_{lower}, c_i + d_{upper}]$ must look like.

We believe, that these border settings must somehow reflect the deviation of the objective function values according to c_i. The border d_{lower} should be larger, if the deviation below c_i is larger. The same goes for the upper border d_{upper}. Additionally, we think, that these values must be calculated independently from each other. This would then define non-symmetrical ranges, i.e. ranges, that are adapted to the non-symmetric development of the objective function values around c_i. Furthermore, it is yet unclear, if the range parameters should be adapted dynamically during the search of gNovelty+.

Adjusting the range parameters is not the only option to affect the number and properties of found partitions. We have already explained, that too short partitions must be ignored in order to prevent a memory explosion. We have defined a partition to be too short, when it has less elements than the number of variables n in the investigated formula. However, we are not yet sure, whether this lower border is useful.

We believe, that the lower border for partitions should somehow reflect the ruggedness of the search space. In other words, when the deviation of the objective function values is high, this lower border should be smaller. This would ensure, that enough partitions for the creation of superpartitions are found.

Altogether, with optimal parameter settings for the range and the lower border for the length of a partition, we would be able to collect an optimal number of partitions. This in turn would give us a larger number of opportunities to create a superpartition, and therefore, a larger number of opportunities to create promising superpartitions.

The combination of partitions into superpartitions in HYBRIZZL is done in a chronological manner. While this scheme is fairly easy to use, we are not yet sure if this is actually an optimal way to create superpartitions. In order to clarify this, we propose that a study must be conducted in order to develop and test alternative schemes for combining partitions.

Such alternative schemes could, for example, make use of additional properties of partitions, that have yet to be determined. One of the additional property could be the distance between partitions. With a distance measure for partitions, one could be able to re-arrange the combination, such that certain partitions are used first, and other partitions might be ignored. Therefore, one could be able to alter superpartitions and their properties in a steady way.

In case superpartitions can be created, with their properties changing smoothly, one would be able to better identify promising superpartitions, because the collected data of the properties of superpartitions for different formulas would not be as erratic, as it is at the moment.

When it comes to the collected data, we believe, that the set of formulas we tested in our study is hardly representative. So far, we have only investigated nine satisfiable uniform random 3-SAT instances. For a representative study, that creates significant results, one should use a much broader set of formulas.

Finally, we have presented several properties of HYBRIZZL, for example its completeness. HYBRIZZL will therefore be able to determine the unsatisfiability of a formula. This completeness, however, is achieved by iteratively creating an empty characterizing partial assignment for a superpartition of sufficient length. Until this superpartition is created, the solver will have to determine the unsatisfiability of several sub-formulas, that inherit the search space of their predeceasing sub-formulas. In other words, HYBRIZZL shows the unsatisfiability for several parts of the original formula several times. Since this needs additional computation time, HYBRIZZL will always be slower to determine the unsatisfiability of a formula in comparison to March_ks.

It would be of advantage, if the information of the unsatisfiability of a sub-formula would be used by gNovelty+, in order to not again visit certain portions of the search

space. In other words, gNovelty+ should learn from the results that March_ks provides. Superpartitions could then be used to intersect the search space of the original formula, and March_ks would have to show the unsatisfiability for each of these sub-formulas only once. As soon as March_ks provided this information for all intersections of the search space of the original formula, one would have proved its unsatisfiability.

We are, however, not yet sure how such a learning mechanism for gNovelty+ could look like. Additionally, it is yet unclear, if this learning would result in a speed-up for showing the unsatisfiability of a formula in comparison to the time March_ks would need to provide this information on its own.

We therefore propose to further investigate the possibilities on how gNovelty+ could be extended to a learning solver, without deteriorating its formidable solving times on satisfiable instances.

The message well I hear,
my faith alone is weak.
from "Faust: The Tragedy Part One"
– Johann Wolfgang von Goethe

PART V

Appendix

Some of the statements in Part III have not yet been fortified. We will catch up with these statements in the following part.

Chapter 10

Appendix

10.1 Falsifying Clauses in Uniform Random k-SAT Formulas using Random Partial Assignments

We will now fortify our statement (on page 149), that the probability to falsify a clause in a random formula by applying a random assignment, that assigns exactly $n/2$ variables, is practically non-existent.

Given a formula F with n variables and $4.26n$ clauses. The clauses $\mathcal{C}$ of F are all different. Let A be a partial assignment (applicable to F), that assigns exactly g variables ($g \in [1, n]$) at random.

We are interested in the probability, that $A(\mathcal{C})$ yields at least one empty clause (i.e. falsifies at least one clause).

First of all, we calculate the probability, that one given clause $c \in \mathcal{C}$ is falsified by a randomly generated partial assignment A, that assigns exactly g variables. The number of partial assignments A, that assign exactly g variables, is

$$\binom{n}{g}.$$

The number of partial assignments A, that assign at least the variables in clause c, is

$$\binom{n-g}{g-k}.$$

Only $\frac{1}{2^k}$ of these partial assignments can assign exactly the right values to the variables in c in order to falsify all k literals in c. Therefore, the probability to falsify a single clause c with A is

$$\frac{\binom{n-g}{g-k}}{2^k \cdot \binom{n}{g}},$$

n	$PROB(n,1)$
10	0.1949
20	0.0070
30	$4.7224 \cdot 10^{-5}$
40	$1.7719 \cdot 10^{-7}$
50	$4.8670 \cdot 10^{-10}$
60	$1.1110 \cdot 10^{-12}$

Table 10.1: The probability to falsify a single 3-SAT clause in F for various n.

and therefore, the probability to not falsify a given clause c with such an A is

$$1 - \frac{\binom{n-g}{g-k}}{2^k \cdot \binom{n}{g}}.$$

The set of all possible different k-SAT clauses (over n variables) has size

$$2^k \cdot \prod_{l=0}^{k-1}(n-l).$$

When now drawing $\lceil 4.26n \rceil$ clauses from this set (with replacement), the chance, that we never draw a clause we already have drawn, is

$$\prod_{i=0}^{\lceil 4.26n \rceil} \left(\frac{(2^k \cdot \prod_{l=0}^{k-1}(n-l)) - i}{2^k \cdot \prod_{l=0}^{k-1}(n-l)} \right)$$

Therefore, the probability, that one draws two same clauses when randomly generating a formula F, is rather small (with n having a large value, i.e. $n > 200$). By extension, this means that we can treat the clauses in $\mathcal{C}$, as if they have been drawn independently. The (rather small) error we introduce by this simplification is ignored. We then get the probability

$$\left(1 - \frac{\binom{n-g}{g-k}}{2^k \cdot \binom{n}{g}} \right)^{\lceil 4.26n \rceil + \epsilon}$$

to not falsify any clauses in $\mathcal{C}$. Finally, the probability to falsify at least one clause in $\mathcal{C}$, with a randomly generated partial assignment A that assigns g variables, is

$$1 - \left(1 - \frac{\binom{n-g}{g-k}}{2^k \cdot \binom{n}{g}} \right)^{\lceil 4.26n \rceil + \epsilon}.$$

For 3-SAT and $g = \frac{n}{2}$, this is equal to $PROB(n, \epsilon) = 1 - \left(1 - \frac{\binom{\frac{n}{2}}{\frac{n}{2}-3}}{8 \cdot \binom{n}{\frac{n}{2}}}\right)^{\lceil 4.26n \rceil + \epsilon}$.

Table 10.1 presents the probabilities for falsifying a single clause in a uniform random 3-SAT formula by a randomly generated partial assignment with $n/2$ assigned variables. As we can see, the probabilities are shrinking rapidly with increasing n, and therefore, the chance to falsify a single clause with such an assignment is practically non-existent.

10.2 The Growth of the Search Space Vicinity of Superpartitions

We will now fortify our statement (on page 156), that a sub-formula has a larger search space, when the length of a superpartition, that is used to create that sub-formula, increases.

A superpartition is a sequence of partitions. The number of free variables in the characterizing partial assignment for such a superpartition is depending on the length l of the superpartition. Therefore, the search space of the sub-formula $SUPER(S)(F) = F_s$ grows, if the length of the superpartition is increased. It will eventually be large enough, to inherit a solution from the original formula F.

In order to get an idea on how fast the search space grows with growing l, we have performed the following calculations.

Definition 43 *Given a partition P. With $FREE(P)$, we denote the number of unassigned variables in $CHAR(P)$. Given a superpartition S. With $FREE(S)$, we denote the number of unassigned variables in $SUPER(S)$. Let $LENGTH(S)$ denote the number of partitions within S.*

Given a formula F with n variables. Let $\{P_1, \ldots, P_l\}$ be a set of partitions in a trajectory from a run on F. We define $P_i \neq P_j$ for $i \neq j, i, j \in [1, l]$, i.e. they are pairwise different. We furthermore constrain the P_i to have disjoint sets of search space vicinities. Let $F_s^i = CHAR(P_i)(F)$. Let $S = (P_1, \ldots, P_l)$. Let $F_s = SUPER(S)$. With

$$d = \frac{2^{FREE(S)}}{\sum_{i=1}^{l} 2^{FREE(P_i)}} \tag{a}$$

we calculate the factor d of how much bigger the search space of F_s is in contrast to the combined search spaces of F_s^i. In order to simplify this, we will consider the case, where the $CHAR(P)$ only differ in one variable (which is the worst case for

pairwise different partitions). We have

$$\begin{aligned} &r = max(FREE(P_1), \ldots, FREE(P_l)) = FREE(P_{max}) \text{ for some } max \in [1, l] \\ &\Rightarrow FREE(P_i) \leq k \ \forall i \in [1, l]. \end{aligned} \tag{1}$$

Furthermore, we know that the superpartition must have at least r unassigned variables, since at least one partition (P_{max}) has this number of unassigned variables. Additionally, each pair of the l partitions differs in exactly one variable. Therefore, $SUPER(S)$ must have at least $l-1$ more unassigned variables than $CHAR(P_{max})$. Therefore, we know that

$$FREE(S) \geq r + l - 1. \tag{2}$$

We can now use (1) and (2), to simplify (a). We get

$$\begin{aligned} d &= \frac{2^{FREE(S)}}{\sum_{i=1}^{l} 2^{FREE(P_i)}} \\ &\overset{(1)}{\geq} \frac{2^{r+l-1}}{\sum_{i=1}^{l} 2^{FREE(P_i)}} \\ &\overset{(2)}{\geq} \frac{2^{r+l-1}}{\sum_{i=1}^{l} 2^{r}} \\ &= \frac{2^{r+l-1}}{l \cdot 2^r} \end{aligned} \tag{b}$$

This means, that the chances are increasing drastically that $SUPER(S)(F) = F_s$‘ inherits a solution from the search space of F, when l increases. Ergo, our chances to find a combined solution for F (which is $SUPER(S)$ and A_s‘) increase with increasing l.

It is, however, important to understand, that (b) only represents the growth as long as the search space vicinities of the partitions do not overlap. When numerous partitions are used, the chance increases, that this is no longer true. Therefore, (b) can only give an idea on the growth of the search space of the sub-formula.

Bibliography

[APSS05] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern sls. *Proceedings of the 20th National Conference of the American Association for Artificial Intelligence*, pages 354–359, 2005.

[APT79] Bengt Aspvall, Michael Plass, and Robert Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[Bac02] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. *Proceedings of National Conference on Artificial Intelligence (AAAI-2002)*, 2002.

[Bra04] Ronen Brafman. Simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics*, 34(1):52–59, 2004.

[BW04] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. *In Theory and Applications of Satisfiability Testing 2003*, 2919 of Lecture Notes in Computer Science, Berlin:341–355, 2004.

[CBS04] Konstantino Chatzikokolakis, George Boukeas, and Panagiotis Stamatopoulos. Construction and repair: A hybrid approach to search in csps. *Lecture Notes in Computer Science*, 3025:342–351, 2004.

[Coo71] Stephen Cook. The complexity of theorem-proving procedures. *Proceedings of the 3rd ACM Symposium on Theory of Computing*, 1:151–158, 1971.

[Coo00] Stephen Cook. The p versus np problem. In *Clay Mathematical Institute; The Millennium Prize Problem*, 2000.

[Cra] James Crawford. Solving satisfiability problems using a combination of systematic and local search. Rutgers University, http://www.cirl.uoregon.edu/crawford/papers/dimacs93.ps.

[CW03] Wahid Chrabakh and Rich Wolski. Gridsat: A chaff-based distributed sat solver for the grid. *ACM/IEEE Supercomputing 2003 Conference*, page 37, 2003.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] Martin Davis and Hillary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[EB05] Niklas Een and Armin Biere. Effective preprocessing in sat through variable and clause elimination. *Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05), Lecture Notes in Computer Science (LNCS)*, 3569, 2005.

[FF04] Brian Ferris and Jan Fröhlich. Walksat as an informed heuristic to dpll in sat solving. *http://www.cs.washington.edu/homes/bdferris/papers/WalkSAT-DPLL.pdf*, 2004.

[FH07] Lei Fang and Michael Hsiao. A new hybrid solution to boost sat solver performance. *Design, Automation, and Test in Europe*, pages 1307–1313, 2007.

[FR04] Hai Fang and Wheeler Ruml. Complete local search for propositional satisfiability. *Association for the Advancement of Artificial Intelligence*, pages 161–166, 2004.

[GPFW97] Jun Gu, Paul Purdom, John Franco, and Benjamin Wah. Algorithms for the satisfiability problem: a survey. *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 35:19–151, 1997.

[GSC97] Carla Gomez, Bart Selman, and Nuno Crato. Heavy-tailed probability distributions in combinatorial search. *Proceedings of CP-97*, pages 121–135, 1997.

[GW93] Ian Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for sat. *National Conference on Artificial Intelligence*, pages 28–33, 1993.

[HD04] William Havens and Bistra Dilkina. A hybrid schema for systematic local search. *LNCS, Advances in Artificial Intelligence*, 3060/2004:248–260, 2004.

[HLDV07] Djamal Habert, Chu Min Li, Laure Devendeville, and Michel Vasquez. A hybrid approach for sat. *Lecture Notes In Computer Science*, 2470:172 –184, 2007.

[Hoo98] Holger Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, Technische Universitaet Darmstadt, 1998.

[Hoo99] Holger Hoos. On the run-time behaviour of stochastic local search algorithms for sat. *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 661–666, 1999.

[Hoo02] Holger Hoos. An adaptive noise mechanism for walksat. *Eighteenth national conference on Artificial Intelligence*, pages 655–660, 2002.

[HS00] Holger Hoos and Thomas Stützle. Local search algorithms for sat: An empirical evaluation. *Journal of Automated Reasoning*, 24:421–481, 2000.

[HS04] Holger Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.

[HTH02] Frank Hutter, Dave Tompkins, and Holger Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. *Lecture Notes in Computer Science*, 2470:233–248, 2002.

[Hua06] Jinbo Huang. The effect of restarts on the efficiency of clause learning. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2006.

[HvM04] Marijn Heule and Hans van Maaren. Aligning cnf- and equivalence-reasoning. *International Conference on Theory and Applications of Satisfiability Testing*, pages 174–181, 2004.

[HvM06] Marijn Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.

[HvM07a] Marijn Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. *Lecture Notes in Computer Science*, 4501:258–271, 2007.

[HvM07b] Marijn Heule and Hans van Maaren. march_ks. *http://www.satcompetition.org/2007/march_ks.pdf*, 2007.

[HvM08] Marijn Heule and Hans van Maaren. Whose side are you on? finding solutions in a biased search-tree. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:117–148, 2008.

[HvZDvM04] Marijn Heule, Joris van Zwieten, Mark Dufour, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient lookahead sat solver. *Theory and Applications of Satisfiability Testing*, 1:345–359, 2004.

[JL02] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Seventh National Conference on Artificial Intelligence*, pages 169–174, 2002.

[LA97] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. *Principles and Practice of Constraint Programming*, pages 341–355, 1997.

[LH05] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. *Lecture Notes in Computer Science*, 3569:158–172, 2005.

[Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):75–80, 1999.

[LMS08] Florian Letombe and Joao Marques-Silva. Improvements to hybrid incremental sat algorithms. *International Conference on Theory and Applications of Satisfiability Testing*, pages 168–181, 2008.

[LSB04] Matthew Lewis, Tobias Schubert, and Bernd Becker. Early conflict detection based bcp for sat solving. *th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[LWZ06] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for sat. *Proceedings of LSCS-2006*, pages 2–16, 2006.

[LWZ07] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and promising decreasing variables in local search for sat. *Sat-Competition Homepage: www.satcompetition.org*, 2007.

[MFM05] Yogesh Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. *Lecture Notes in Computer Science*, 3542:360–375, 2005.

[MMZ+01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. *38th DAC*, 2001.

[MSG98] Bertrand Mazure, Lakhdar Sais, and Eric Gregoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319 – 331, 1998.

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, 1997.

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

[MvVW06] Dimosthenis Mpekas, Michiel van Vlaardingen, and Siert Wieringa. The first steps to a hybrid sat solver. *Technische Universiteit Delft, www.st.ewi.tudelft.nl/sat/reports.php*, 2006.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. *LNCS, Theory and Applications of Satisfiability Testing – SAT 2007*, 4501/2007:294–299, 2007.

[PD08] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 229–234, 2008.

[PG07] Duc Nghia Pham and Charles Gretton. gnovelty+. *SatCompetition Homepage: www.satcompetition.org*, 2007.

[Pha06] Duc Nghia Pham. *Modelling and Exploiting Structures in Solving Propositional Satisfiability Problems*. PhD thesis, Griffith University, 2006.

[Poo84] David Poole. Making 'clausal' theorem provers 'non-clausal'. *Proc. Canadian Society for Computational Studies of Intelligence National Conference*, pages 124–125, 1984.

[PW96] Andrew Parkes and Joachim Walser. Tuning local search for satisfiability testing. *Proceedings of AAAI-96*, pages 356–362, 1996.

[Qui55] Willard Van Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62:627–631, 1955.

[Rob65a] John Alan Robinson. Automated deduction with hyper-resolution. *Journal of Computer Mathematics*, 1(3):227–234, 1965.

[Rob65b] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Sat02] Satcompetition.org. The dimacs cnf input and output format. http://www.satcompetition.org/2004/format-solvers2004.html, January 2002.

[SKC94] Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, 1994.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI Press*, pages 440–446, 1992.

[SP05] Sathiamoorthy Subbarayan and Dhiraj Pradhan. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. *SAT 2005, Lecture Notes in Computer Science*, 3542:276–291, 2005.

[SS01] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete sat procedures. *Artificial Intelligence*, 132:121–150, 2001.

[SSH01] Dale Schuurmans, Finnegan Southey, and Robert Holte. The exponentiated subgradient algorithm for heuristic boolean programming. *International Joint Conference on Artificial Intelligence*, pages 334–341, 2001.

[TPBF04] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira. Additive versus multiplicative clause weighting for sat. *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'04)*, pages 191–196, 2004.

[WvM98] Joost Warners and Hans van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23:81–88, 1998.

[ZMMM01] Lintao Zhang, Conor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. *Proceedings of the International Conference on Computer Aided Design (ICCAD), IEEE Press*, 1:279–285, 2001.