INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Local Search for Unsatisfiable Propositional Formulae

David Miguel Neves Pereira

Dissertation for the degree of

Master of Science in Information Systems and Computer Engineering

Jury

President:	Prof^{a}. Ana Maria Severino de Almeida e Paiva
Research Advisor:	Prof^{a}. Maria Inês Camarate de Campos Lynce de Faria
Committee:	Prof. Vasco Miguel Gomes Nunes Manquinho
	Prof^{a}. Carla Maria Gonçalves Ferreira

October 2008

Acknowledgments

Since March 2007 until now, October 2008, this thesis has been a constant part of my life. Now that it is over, I would like to thank those that made it possible:

To my mother, father, sister, grandmother and grandfather who always supported me and were there when I most needed them.

To Prof. Inês Lynce for her excellent orientation of this dissertation, always ready to help.

To all my friends.

And to Caron Lim, whose love, affection and support made my life a wonderful place to be, and whose smile lightened even the darker of days.

I would also like to thank the Fundação para a Ciência e Tecnologia, which partially supported the elaboration of this dissertation within the scope of the project SATPot - POSC/EIA/61852/2004.

Abstract

Recently, we have seen a remarkable progress in propositional satisfiability (SAT), with theoretical and practical contributions. Even though the SAT solvers have exponential run time in the worst case, they can currently be used to solve hard problem instances. The ability to prove unsatisfiability has been barred from stochastic local search (SLS) procedures until recently.

This dissertation contributes to a better understanding of the two state of the art SLS solvers which prove unsatisfiability instead of satisfiability: RANGER and GUNSAT. We give a detailed description of each solver and provide some examples.

With the objective of improving RANGER, we add two methods used by GUNSAT to RANGER: unit propagation look-ahead and extended resolution. We then explain the advantages and disadvantages of each method when applied to RANGER and discovered some interesting results. We complement the description with several test sets using unsatisfiable instances.

Taking into account the experimental results and theoretical principles, we conclude that unit propagation look-ahead is a very promising technique to be added to RANGER in future versions, whereas extended resolution suffers from a heavy dependence on GUNSAT's powerful heuristics and scoring system, which ultimately are not compatible with RANGER's simplicity principles.

Keywords

Propositional Satisfiability, Proving Unsatisfiability, Local Search, RANGER, GUNSAT, Unit Propagation Look-Ahead, Extended Resolution

Resumo

Nos últimos anos assistimos a um progresso notável na área de satisfação proposicional (SAT), com contribuições tanto a nível teórico como prático. Apesar das ferramentas de SAT requererem tempo de execução exponencial no pior caso, estas conseguem actualmente resolver instâncias de problemas difíceis. A habilidade para provar a não satisfação de fórmulas proposicionais tem estado interdita a procedimentos de procura local estocástica (SLS) até recentemente.

Esta dissertação tem como objectivo contribuir para o estudo das duas ferramentas que provam a não satisfação em vez da satisfação: RANGER e GUNSAT. Dá-se uma descrição detalhada de cada ferramenta e apresentam-se vários exemplos.

Com o objectivo de melhorar o RANGER, adicionam-se-lhe dois métodos usados pelo GUNSAT: propagação unitária por antevisão e resolução estendida. Apresentam-se e explicam-se as vantagens e desvantagens de cada método quando aplicado ao RANGER e chega-se a resultados deveras interessantes. A descrição é complementada correndo a ferramenta com várias baterias de testes de instâncias não satisfazíveis.

Após a análise dos resultados obtidos e dos princípios teóricos aplicados, conclui-se que a propagação unitária por antevisão é uma técnica muito promissora e recomenda-se vivamente a sua adição a futuras versões do RANGER, enquanto a resolução estendida sofre de uma dependência muito forte das poderosas heurísticas do GUNSAT. Estas não são compatíveis com a simplicidade do RANGER.

Palavras Chave

Satisfação Proposicional, Provar a Não Satisfação, Procura Local, RANGER, GUNSAT, Propagação Unitária por Antevisão, Resolução Estendida

Contents

List of Tables

List of Algorithms

1

Introduction

Suppose you have to plan the assignment of some rooms in a conference center, in a very busy week, where a lot of guest speakers will make their speeches. You must plan the hours; who speaks in which room; some speakers have tight schedules and can only stay for a short time; the audience cannot be made to change rooms every hour, so the speakers of a particular issue must be grouped in the same room in an interval of time; and so on. All these constraints can be easily solved for a small number of variables: 2-3 speakers, 2 rooms, 1 day, among many others. But if the number of variables increase, even in a small scale, the problem easily becomes hard to solve.

This kind of problems can be encoded using propositional logic. Propositional logic encodes and represents propositions and their corresponding relations. We want to assume that a proposition (or condition) is either true or false, and so this logic can also be called Boolean logic because the propositions can be assigned truth values.

There are many problems that can be formulated as SAT (from SATisfiability), such as schedule planning [28] (the example given above), graph coloring [21], circuit planning and testing [7], blocks world [27], quasigroup problems [57], tower of Hanoi [28], robotics [12], database searching [53], model checking [1], cryptography [36], bioinformatics [35], etc.

Combinatorial problems can be found in various areas of computer science and other fields, such as artificial intelligence, bioinformatics, schedule planning, microprocessor planning and testing, etc. These problems usually involve finding groupings or assignments of a finite and discrete set of objects that satisfy certain constraints or conditions. The solution to combinatorial problems is formed by combinations of these solution components. The example above, about scheduling, can be viewed as a combinatorial problem in which the components that form the solution are the

events to be scheduled, and the values assigned to the events represent the time at which they occur. We can then define candidate solution as a set of assignments of Boolean values to the variables: this candidate solution must then be evaluated by some function to see if it is indeed a solution of the problem, such that all conditions are satisfied.

Many combinatorial problems can be said to be decision problems, in which the solutions to a given instance of the problem are specified by a logical set of conditions. This particular kind of combinatorial problems will be explored in more detail in the following chapter.

Another kind of problems is optimization problems. Many of the more practically relevant combinatorial problems are optimization problems rather than decision problems. For each optimization problem, there is a corresponding decision problem that asks whether there is a feasible solution for some particular set of assignments. The goal is to find the best possible solution, and an objective function is used to measure and evaluate all candidate solutions. Optimization problems can be either minimization problems or maximization problems.

There are two very important complexity classes: *P* and *NP*. The first is the class of problems that can be solved by a deterministic machine in polynomial time, whereas the second is the class of problems that can be solved by a nondeterministic machine in polynomial time. Every problem in *P* is contained in *NP*. Many hard problems from *NP* are closely related and can be translated into each other in polynomial deterministic time. A problem that is at least as hard as any other problem in *NP* is called *NP-hard*. But these problems do not have to belong to the *NP* class themselves, because their complexity may be higher. *NP-hard* problems that are contained in *NP* are called *NP-complete*.

SAT was the first decision problem to be proved to be a *NP-complete* problem. This was done by Stephen Cook in 1971 [9].

Propositional Satisfiability (SAT) can be viewed as a special case of finite-domain Constraint Satisfaction Problem (CSP), which determines whether a set of constraints over a set of discrete variables can be satisfied. In propositional satisfiability, every variable can only assume the values true or false. Problems that are often formulated as CSP or SAT for the purposes of benchmarking include, among others, graph coloring, the n-queens problem, the Tower of Hanoi, blocks world, quasigroup problems and many problems from real world application domains.

There has been an explosion in SAT research in recent years. The strong relationship between the theory, the algorithms and the applications motivated a growing interest and recent advances. It is still an open research area and nothing seems to announce an end to it.

Of course, one may wonder why now if these problems have always been there. But sum it with the advances in technology, which allowed solving some problems and centering the attention in

others more difficult to solve and therefore more challenging.

At first, those who had empirical understanding of a specific problem domain would solve the problems of that area. By treating them separately, it was difficult to identify their common characteristics.

However, an effort has been made to abstract from specific problem details, to formalize all of them into conventional formulas. In practice, propositional logic began to be used as a framework for knowledge representation and problem solving. Consequently, analyzing those problems as a whole made possible a unified perspective. Based on their structure, new techniques were proposed that benefit the entire domain.

Despite many years of effort, it remains an open question: "Can satisfiability problems be solved efficiently in practice?". The fact is that a first negative answer to a concrete problem can become affirmative later on. So we tend to propose a more ambitious formulation: "What has to be done in order to efficiently solve an instance of SAT?" To such a question we have different answers depending on the properties of the concrete problem instance.

Recent years have seen the proposal of several effective approaches for solving propositional satisfiability. These approaches include, among others, local search and its variations [45; 48], backtrack search improved with different pruning techniques [26; 25; 55], backtrack search with randomization and restarts [19], continuous formulations [49] and algebraic manipulation [20; 30; 50]. These different approaches have allowed efficiently solving different classes of SAT instances.

This document is organized as follows. Chapter 2 provides the underlying framework to understand the main topics focused on the rest of the document. We give some definitions and notations and briefly explain the propositional satisfiability problem. We also extensively discuss some simplification rules for propositional satisfiability at the end of this chapter.

In chapter 3 we give an overview of the two most important complete procedures: the Davis-Putnam algorithm and the Davis-Logemann-Loveland algorithm.

The fourth chapter focuses on local search algorithms. We explain what are local search procedures and present some of the earliest and most important algorithms: GSAT, WalkSAT and others. In this chapter we also give some insight on techniques to further improve local search procedures. Finally we explore some state-of-the-art SLS (Stochastic Local Search) algorithms.

In chapter 5 we discuss the two most prominent local search algorithms for proving unsatisfiability: RANGER and GUNSAT. At the end of this chapter we focus on the advantages and disadvantages of both of them.

In chapter 6 we present a detailed overview of the original RANGER solver as it was implemented

by Prestwich and Lynce: pseudo-code of the algorithm, constants, data structures and the most important functions.

We then follow, in chapter 7, with a detailed explanation of the techniques and methods used to improve the original RANGER, namely unit propagation look-ahead and extended resolution. We can find, in this chapter, a lengthy discussion about implementation issues and problems found, as well as the most important data structures and functions implemented.

In the eight chapter, we present the results achieved with the various methods experimented on RANGER and draw some conclusions about the usefulness of such procedures in a solver such as RANGER.

Finally in chapter 9 we draw conclusions about this thesis and suggest some future work.

2

Fundamental Concepts

This chapter will introduce the fundamental concepts used throughout this document. Some topics will be covered to allow the reader an easier understanding of the underlying notations and definitions used.

We start by presenting some definitions and notations that are of the utmost importance to the understanding of this thesis. We explain some of the basic concepts behind propositional satisfiability solving and modeling.

Next we will give a brief overview of the propositional satisfiability problem, SAT.

In the following section we will present some well known simplification techniques used in many of the current state of the art SAT solvers.

We then discuss some data structures to be used in the implementation of SAT algorithms.

Finally, we provide insights on some common methods to speed up the search, the algorithm used for some testing and the most widely accepted format in which instances of problems are encoded.

2.1 Propositional Satisfiability: Definitions and Notation

The most widely used notation to express a formula in propositional logic is the Conjunctive Normal Form, CNF. A formula is in conjunctive normal form if it is a conjunction of clauses, where a clause is a disjunction of literals. A literal represents either a complemented or uncomplemented Boolean variable. We sometimes refer to a formula as a clause database, since a formula is a set of clauses. We refer to a clause as a set of literals as well.

Let φ be a CNF formula on n Boolean variables $x_1, ..., x_n$ and m clauses $w_1, ..., w_m$. A clause w is a disjunction of literals l_i. In other words, a literal l represents a variable x or its negated form $\neg x$. For simplification, we will refer to a set of Boolean variables throughout this thesis as characters $a, b, ..., z$.

Example 2.1.1. *The CNF formula:*

$$\varphi = (a) \wedge (\neg a \vee b \vee \neg c) \wedge (b \vee d) \tag{2.1}$$

has three clauses

$$w_1 = (a) \tag{2.2}$$

$$w_2 = (\neg a \vee b \vee \neg c) \tag{2.3}$$

$$w_3 = (b \vee d) \tag{2.4}$$

and four variables a, b, c and d. The first clause has one literal, the second has three literals, and the third has two literals. Both the variables a and c in the second clause are negated.

Let Φ be the set of propositional formulae, v_i the set of propositional variables and ϕ the set of clauses. Φ can be defined through induction as follows:

- $v_i \in \Phi$, for each $i \in \mathbb{N}$,
- $(\neg\phi \in \Phi)$, if $\phi \in \Phi$,
- $(\phi_1 \vee \phi_2 \in \Phi)$, if $\phi_1, \phi_2 \in \Phi$,
- $(\phi_1 \wedge \phi_2 \in \Phi)$, if $\phi_1, \phi_2 \in \Phi$.

We can consider another operation concerning propositional formulae, albeit it is a higher level and free definition: the *implication*. It can be defined as a conjunction of two sets, as in $(\phi_1 \Rightarrow \phi_2) \Leftrightarrow (\neg\phi_1 \vee \phi_2)$.

We can define the assignment of variables as a function $A : X \rightarrow \{true|false\}$, where $X = \{x_1, ..., x_n\}$ is a set of n Boolean variables. We may as well represent an assignment as a list of pairs *(variable,value)*. We sometimes refer to truth value *true* as T or 1, and to truth value *false* as F or 0. An assignment to a formula φ with n variables is *complete* if all the variables have corresponding truth values, otherwise it is called *partial*.

If there are unassigned literals in a clause, they are called *free literals*. If an unresolved clause contains only one free literal it is called a *unit clause*. If it contains two free literals then it is a *binary clause*. With three free literals it is a *ternary clause*.

Example 2.1.2. *Let us consider the example above where* $\varphi = (a) \wedge (\neg a \vee b \vee \neg c) \wedge (b \vee d)$ *and such that no assignments have been made to the variables. Thus,* $w_1 = (a)$ *is a unit clause,* $w_3 = (b \vee d)$ *is a binary clause and* $w_2 = (\neg a \vee b \vee \neg c)$ *is a ternary clause. If we consider the following partial assignment* $A = \{(b, 0)\}$ *to the formula* φ *above then* w_3 *becomes a unit clause (with* d *as its free literal)and* w_2 *becomes a binary clause, with* $\neg a$ *and* $\neg c$ *as its free literals.*

Let ρ be an assignment of variables over the formula φ, such that $\rho : X \rightarrow \{true|false\}$. The satisfiability of the propositional formula φ is inductively defined as follows:

- $\rho \models x_i$ if $\rho\,(x_i) = 1$,
- $\rho \models (\not\varphi)$ if $\rho\neg \models \varphi$,
- $\rho \models (\varphi_1 \vee \varphi_2)$ if $\rho \models \varphi_1$ or $\rho \models \varphi_2$,
- $\rho \models (\varphi_1 \wedge \varphi_2)$ if $\rho \models \varphi_1$ and $\rho \models \varphi_2$.

Let us consider the formula φ such that $X = vars(\varphi)$ is the set of variables under φ, and let w be a clause in φ. A truth assignment A for X satisfies w if and only if at least one literal l in w is true under A, i.e., a clause w is satisfied if at least one of its literals assumes the value 1. On the other hand, a clause w is unsatisfied if all of its literals assume the value 0, and such a clause is called an *empty clause.*

As such, given a CNF formula $\varphi = (w_1 \wedge ... \wedge w_n)$, this formula is satisfiable if there exists an assignment ρ such that $\rho \models \{w_1, w_2, ..., w_n\}$, i.e., all clauses must be satisfied if the formula φ is to be satisfied. If the formula is satisfied we can also say that it evaluates to true. On the other hand, if the formula is unsatisfied after a particular assignment, we say that, for that assignment, the formula evaluates to false.

If a clause is neither satisfied nor unsatisfied then we call it *unresolved.* A clause that contains both a literal and its complement is called a *tautology* and is always satisfied.

Example 2.1.3. *Let us consider the example above where* $\varphi = (a) \wedge (\neg a \vee b \vee \neg c) \wedge (b \vee d)$*. If we now consider the following partial assignment* $A = \{(a, 1), (b, 0), (c, 1)\}$ *then we conclude that clause* w_1 *is satisfied, clause* w_2 *is unsatisfied and clause* w_3 *is unresolved.*

Even though CNF is the most widely used notation to express a formula in propositional logic, there is a variety of other notations that can be used. It is, then, useful to have a way to transform a formula in some other notation to CNF. Given a propositional formula φ, there exists an equivalent CNF formula γ, i.e., given an assignment ρ, if $\rho \models \varphi$ then $\rho \models \gamma$. It is also true that if $\rho \not\models \varphi$ then $\rho \not\models \gamma$.

The fact that satisfiability algorithms use CNF formulae is in no way impeditive, because even if a given formula is not in the CNF format, by the result above, there is a formula γ equivalent to φ which is in CNF. This formula can be easily obtained through polynomially bound algorithms [40].

2.2 The Propositional Satisfiability Problem

In the propositional satisfiability problem (SAT problem) it is given an input formula in propositional logic (usually in CNF) and it must be decided whether there is a set of Boolean values to be assigned to the variables in the formula such that it evaluates to *true*. Equally important is to determine whether no such assignments exist, which would imply that the formula is *false* for all possible variable assignments. In this latter case, we would say that the formula is unsatisfiable; otherwise it is satisfiable.

Algorithms for SAT can be divided into 2 basic subgroups:

- **Complete Algorithms**, which find one or more solutions if the formula is satisfiable, or prove that it is unsatisfiable. The most famous SAT complete algorithms are the Davis-Putnam procedure [11] and the Davis-Logemann-Loveland procedure [10] that, in spite of its age, is still inspiring most state-of-the-art algorithms.
- **Incomplete Algorithms**, which find one or more solutions to the formula if it is satisfiable, but do not end or return *Unknown* if the formula is unsatisfiable. That way, one cannot know if the formula is unsatisfiable or if the algorithm simply did not search long enough. Therefore, these algorithms cannot prove unsatisfiability. However, there has been recent research in the incomplete algorithms field, which resulted in the development of two procedures that can prove unsatisfiability but not satisfiability (RANGER [41] and GUNSAT [4]).

There has also been some development on *Hybrid Algorithms*, that is, local search algorithms that can prove satisfiability as well as unsatisfiability, and that are therefore complete local search algorithms. Refer to [18] and [43] for a more detailed description of these procedures.

2.3 Simplification Rules for Propositional Satisfiability

There is a wide range of simplification techniques that can be applied to a propositional formula such that the obtained formula becomes simpler to solve. These simplification rules can remove clauses in the original formula, remove variables, remove literals, make assignments, add clauses

that contain important information, among others, or they can be a combination of the rules above.

In this section we will explain some of the more common simplification rules that can be applied to formulas in CNF.

2.3.1 The Unit Clause Rule

A unit clause is an unresolved clause containing a single free literal. For example, $w = (l)$ is a unit clause, where l is an unassigned literal. The clause $w = (\neg a \vee b)$ is also a unit clause if the assignment is $A = \{(b, 0)\}$. The unit clause rule is used whenever a unit clause is identified. Then, two rules are applied:

- Every clause containing the free literal l is removed from the formula;
- The literal $\neg l$ is removed from every clause where it appears.

The application of this rule leads to a new set of clauses that is equivalent to the old one.

Let us illustrate this rule with the following example:

Example 2.3.1. *Consider the formula $\varphi = (\neg a \vee b) \wedge (c \vee d) \wedge (c) \wedge (\neg c \vee a)$, where no assignments have yet been done. We can easily see that the clause $w_3 = (c)$ is a unit clause, since it has a single literal, c. Since $w_2 = (c \vee d)$ contains the literal c, this clause can be removed. Clause w_3 will also be removed. Since $w_4 = (\neg c \vee a)$ contains the negation of the literal in the unit clause, literal $\neg c$ can be removed from the clause. The resulting set of clauses is: $(\neg a \vee b) \wedge (a)$, which is equivalent to the original formula φ. The new unit clause (a) that results from unit propagation can be used for a further application of the unit clause rule, which would transform $(\neg a \vee b)$ into (b) and eliminate the unit clause (a), and the updated formula would be (b). We can obviously remove clause (b) from the formula, and thus it becomes empty, which means the formula is satisfiable.*

2.3.2 The Pure Literal Rule

A pure literal is a literal that appears only with one polarity in a formula in CNF. All the occurrences of the literal are either complemented or uncomplemented. Thus, the value of the variable corresponding to the literal can be immediately determined.

The algorithm that applies the pure literal rule begins by detecting the literals whose occurrences are either all positive or all negative (the so called *pure literals*). The next step consists in assigning the variables with a truth value such that the pure literals become satisfied. Therefore, all the clauses containing pure literals become satisfied.

The pure literal rule was first mentioned in [13] as a simplification technique for formulas in CNF.

2.3.3 Resolution Rule

Resolution is an inference rule for propositional logic that maps a pair of clauses having certain properties to a third clause, called the *resolvent*. It is a single valid inference rule that produces a new clause implied by two clauses containing complementary literals. The resolution rule can be regarded as a general technique for deriving new clauses. Resolution was introduced in [44]. It can be formally defined as follows: if we have $w_i = \alpha_i \vee v$ and $w_j = \alpha_j \vee \neg v$, where w_i and w_j are clauses, α_i and α_j are disjunctions of literals and v is a literal, the resolution rule allows us to eliminate the variable v and derive the resolvent clause $w = \alpha_i \vee \alpha_j$.

Example 2.3.2. *Suppose we have the following clauses:* $w_1 = (\neg a \vee b \vee \neg c)$ *and* $w_2 = (b \vee d \vee c)$. *The resolution rule allows deriving the resolvent clause* $w_3 = (\neg a \vee b \vee d)$.

When the two clauses contain more than one pair of complementary literals, the resolution rule can be applied (independently) for each such pair. However, only the pair of literals that are resolved upon can be removed: all other pair of literals remain in the resolvent clause (this turns the clause into a tautology).

The resolution proof procedure consists of repeatedly generating resolvents from original clauses and resolvents that were previously generated until either an empty clause is derived or no more resolvents can be generated.

If an empty clause is derived by resolution then that formula is unsatisfiable. Clearly, if that is ever derived, the entire set of clauses from which it is derived is also unsatisfiable and the original formula is unsatisfiable. Thus, resolution provides a complete system for proof by refutation [44].

Unfortunately, this procedure generates an exponential number of clauses in the general case, and thus requires exponential space in general. The order in which clauses are derived can have a critical effect on the space and time complexity of the procedure, because it affects the size and number of the created clauses. Usually, the majority of clauses generated are redundant and irrelevant to the final result.

There is also an *extension rule* that, combined with the resolution rule, gives the *extended resolution rule* [52]. It adds a new variable e, not occurring in the original formula or previous definitions, such that $e \Leftrightarrow l_1 \vee l_2$, where l_1 and l_2 are literals from the formula. It then adds the new clauses $(\neg e \vee l_1 \vee l_2)$, $(e \vee \neg l_1)$ and $(e \vee \neg l_2)$, along with the new variable, to the formula.

2.3.4 Subsumption

Subsumption is another simplification rule. If we have two clauses and one is the subset of the other, then we can eliminate the larger one without changing the set of solutions. One clause is a subset of another when the literals in the first are a subset of those in the second. Therefore, the smaller clause is more specific than the larger one. Any assignments that satisfy the smaller clause also satisfy the larger clause. We say that the smaller clause *subsumes* the larger, or that the larger clause *is subsumed* by the smaller one.

Example 2.3.3. *Suppose we have the following clauses:* $w_1 = (\neg a \vee b \vee \neg c)$ *and* $w_2 = (b \vee \neg c)$*. Then* w_2 *subsumes* w_1 *and the clause* w_1 *can be eliminated from the formula.*

Subsumption can be very useful in resolution based algorithms, because resolution tends to produce large clauses. Subsumption reduces the number of clauses in those algorithms and keeps the formula small, eliminating unnecessary (subsumed) clauses.

2.4 Data Structures

State of the art SAT algorithms work with very large problems that can easily reach a million of variables and several millions of clauses. Furthermore, during the learning phase of some algorithms, many clauses are generated due to conflicts, which increases even more the number of clauses needed to be stored. Therefore, efficient data structures must be developed and used to wrestle with such an enormously amount of information.

2.4.1 Arrays

It is common practice to store clauses in a linear way, i.e., each clause occupies its own space and does not overlap other clauses' space. Using this method, the database of clauses grows in a linear way proportional to the number of literals in the set of clauses. Normal arrays are used to store information in such a way.

When this data structure is used it is important to keep a set of counters that store all the necessary information about the state of the formula. It is common to use counters for the number of clauses, number of variables, size of each clause, number of literals, both in its positive and negative phase, among others. With these counters one can at any time have complete information about the formula and each clause, particularly, and if needed, the location of each literal in each clause and its assignment.

2.4.2 Trie

Zhang and Stickel [56] proposed a new data structure for storing clauses dubbed *trie*. A trie, or prefix tree, is an ordered ternary tree data structure where each node represents a variable and each of its three branches represent a positive state, negative state or unknown/does-not-matter state. The leaves of this ternary tree will have the value 0 or 1. Each set of ramifications of a ternary tree until it reaches a leaf with the value 1 represents a clause. A trie is ordered if for each node v, the node at a superior level has a variable with an inferior index than the current node v.

A trie has the advantage of being able to easily and quickly detect duplicated and subsumed clauses.

2.4.3 Lazy Data Structures

Another mechanism used for the first time in the solver SATO [56] has two pointers associated with each clause, called the head and tail respectively. In SATO (as well as most other SAT solvers), a clause stores all its literals in an array. Initially, the head pointer points to the first literal of the clause (i.e. beginning of the array), and the tail pointer points to the last literal of the clause (i.e. end of the array). The solver maintains the invariant that for a given clause, all literals located before the head pointer are assigned the value 0, and all literals located after the tail pointer are assigned the value 0. Therefore, if a literal is not assigned the value 0, it must be located between the head and tail pointers.

In this scheme, each variable keeps four linked lists that contain pointers to clauses. The linked lists for the variable v are `clause_of_pos_head(v)`, `clause_of_neg_head(v)`, `clause_of_pos_tail(v)` and `clause_of_neg_tail(v)`. Each of these lists contains the pointers to the clauses that have their head/tail literal in positive/negative state of variable v. If v is assigned with the value 1, for each clause c in `clause_of_neg_head(v)`, the solver will search for a literal that does not evaluate to 1 from the head literal to the tail literal of c. Notice the head literal of c must be a literal of v in negative sate. During the search process, four cases may occur:

- If during the search a value 1 literal is encountered first, then the clause is satisfied; nothing more needs to be done for this clause;
- If during the search a free literal l is encountered that is not the tail literal, then the solver removes c from `clause_of_neg_head(v)` and adds c to the head list of the variable of l. This operation is referred to as moving the head literal because in essence the head pointer is moved from its original position to the position of l;

- If all literals in between these two pointers are assigned the value 0, but the tail literal is unassigned, then the clause is a unit clause, and the tail literal is the unit literal;
- If all literals in between these two pointers and the tail literal are assigned the value 0, then the clause is a conflicting clause.

This algorithm is also referred to as the *Head/Tail List* algorithm.

The authors of the solver chaff [37] proposed a slightly different algorithm, based on the head/tail list algorithm, called *2-literal watching*. The algorithm is based on the observation that for a given clause, as long as it contains two literals that are not assigned to value 0, then it will neither be unit nor conflicting. Therefore, the solver only needs to keep track of two non-zero valued literals for each clause in order to detect unit clauses and conflicting clauses in the clause database.

Similar to the head/tail list algorithm, 2-literal watching has two special literals for each clause called *watched literals*. Each variable has two lists containing pointers to all the watched literals corresponding to it for both states. The lists for variable v is denoted as `pos_watched(v)` and `neg_watched(v)`. In contrast to the head/tail list scheme in SATO, there is no imposed order on the two pointers within a clause, and a pointer can move in either direction, i.e. the 2-literal watching scheme does not have to maintain the invariant of the head/tail scheme. Initially the watched literals are free (i.e. unassigned). When a variable v is assigned value 1, for each literal p pointed to by a pointer in the list of `neg_watched(v)` (notice p must be a literal of v with negative state), the solver will search for a non-zero literal l in the clause containing p. There are four cases that may occur during the search:

- If such a literal l exists and it is not the other watched literal, then the solver removes the pointer to p from `neg_watched(v)`, and adds the pointer to l to the watched list of the variable of l. This operation is referred to as moving the watched literal because in essence one of the watched pointers is moved from its original position to the position of l.
- If the only such l is the other watched literal and it is free, then the clause is a unit clause, with the other watched literal being the unit literal.
- If the only such l is the other watched literal and it evaluates to 1, then nothing needs to be done for this clause.
- If all literals in the clause are assigned the value 0 and no such l exists, then the clause is a conflicting clause.
- During unassignment, there is no need to do anything: this is the advantage over arrays and tries.

2.5 Other Aspects

SAT algorithms are complex tools in constant evolution and are frequently updated. They are, nowadays, very competitive in the resolution and solving of difficult and hard problems.

2.5.1 Pre-processing

Pre-processing can also be a powerful tool when applied to SAT algorithms. The goal of this method is to simplify the formula before the SAT algorithm starts its computation. The pre-processing phase is applied once and not at each iteration of the algorithm, preventing a very costly overhead later on. Lynce and Marques da Silva [34] presented some pre-processing techniques and experimental results that proved its efficiency.

Some available tools for pre-processing techniques are, for example, SatElite, which is a CNF minimizer, intended to be used as a preprocessor to the SAT solver. It is designed to compress the CNF fast enough not to be a bottle neck, and is particularly aimed at improving SAT encodings resulting from translation of net lists (combinational boolean circuits) [14].

2.5.2 Random Restarts

The time that each algorithm takes to solve similar SAT instances varies greatly between each algorithm. Furthermore, two exact problems that only differ in the ordering of its variables can take completely different times to solve by the same algorithm. To prevent this problem [19] presents the notion of *random restart*. For each restart the current search tree is abandoned but the learned clauses are stored for future use, so that the information from the former search is not completely lost [6]

2.5.3 Phase Transitions

One particularly interesting property of uniform Random-3-SAT (problem instances in which each clause has exactly three variables) is the occurrence of a phase transition phenomenon [51], i.e., a rapid change in solubility which can be observed when systematically increasing (or decreasing) the number of k clauses for fixed problem size n. More precisely, for small k almost all formulae are satisfiable; at some critical $k = k'$, the probability of generating a satisfiable instance drops sharply to almost zero. Beyond k', almost all instances are unsatisfiable. Intuitively, k' characterises the transition between a region of underconstrained instances which are almost certainly soluble, to overconstrained instances which are mostly insoluble. For Random-3-SAT, this phase transition

occurs approximately at $k' = 4.26n$ for large n; for smaller n, the critical clauses/variable ratio $\frac{k'}{n}$ is slightly higher. Furthermore, for growing n the transition becomes increasingly sharp. The phase transition would not be very interesting in the context of evaluating SLS algorithms, but empirical analyses show that problems from the phase transition region of uniform Random-3-SAT tend to be particularly hard for both systematic SAT solvers and SLS algorithms. Striving for testing their algorithms on hard problem instances, many researchers used test-sets sampled from the phase transition region of uniform Random-3-SAT. Although, similar phase transition phenomena have been observed for other subclasses of SAT, including uniform Random-k-SAT with $k >= 4$, but these have never gained the popularity of uniform Random-3-SAT. Maybe one of the reasons for this is the prominent role of 3-SAT as a prototypical and syntactically particularly simple NP-complete problem.

2.5.4 DIMACS Format

Most of today's SAT solvers use an input format normalized by the community. A solver receives a `file.cnf` file in DIMACS format as an input.

This format is widely accepted as the standard format for boolean formulas in CNF. Benchmarks listed on `satlib.org` [24], for instance, are in the DIMACS CNF format.

An input file starts with comments (each line starts with c). The number of variables and the number of clauses is defined by the line

```
p cnf variables clauses
```

Each of the next lines specifies a clause (one line for each clause): a positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. The last number in a line should be zero.

Example 2.5.1. *An example of a cnf file follows:*

c A sample .cnf file.

c this example corresponds to the formula $\varphi = (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$

p cnf 2 3

1 -3 0

2 3 -1 0

This format is used in the SAT Competition [1] as well as in the more recent SAT Race [2].

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

[2] http://www-sr.informatik.uni-tuebingen.de/sat-race-2008

2.5.5 MiniSAT Solver

To test the satisfiability of our algorithms (whether they are SAT or UNSAT) we used a well known SAT algorithm called *MiniSAT*.

MiniSAT [3] was first made available in 2003 as an effort to help people get into the SAT community by providing a small, yet efficient, SAT solver with good documentation. It was developed by Niklas Eén and Niklas Sörensson [15; 16]. It was developed so that it would be a not too large/heavy tool, well documented and which can be easily customized by researchers and students.

MiniSAT implements many state of the art techniques and methods for SAT solving, including conflict-clause recording, conflict-driven backjumping, VSIDS dynamic variable order, two-literal watch scheme, and even extensions for incremental SAT and for non-clausal constraints over Boolean variables [16].

MiniSAT proved to be extremely efficient at the SAT Competition of 2005, taking the silver medal in the category of industrial instances (both in SAT+UNSAT and only SAT). In 2007 it proved again its worth by being placed in the podium in both industrial and handmade categories, this time in UNSAT+SAT, SAT and UNSAT subcategories.

The version 1.14 is used in this dissertation even though a more current version can be downloaded from their website. This version was chosen because a light version suits our purposes better. The version used can be obtained from
`http://minisat.se/downloads/MiniSat_v1.14_linux`.

[3] http://minisat.se/MiniSat.html

3

Complete SAT Algorithms

In this section we will introduce the two most important complete procedures for proving satisfiability that are the basis of many of the state-of-the-art complete algorithms today: the David-Putnam algorithm and the Davis-Logemann-Loveland algorithm.

3.1 Davis-Putnam Algorithm

The Davis-Putnam (DP) algorithm was developed by Martin Davis and Hilary Putnam for checking the satisfiability of propositional logic formulae in conjunctive normal form, CNF (see [11]). It is a form of resolution in which variables are iteratively chosen and the corresponding clauses removed by resolving every clause in which the variable is contained positively with any clause in which the variable is negated.

The DP procedure is based on the resolution rule explained above, where variables are eliminated one by one and all possible resolvents are added to the set of clauses. Resolution is applied iteratively to eliminate one variable at a time, i.e., the resolution rule is used between all pairs of clauses containing literals l and $\neg l$. Each iteration generates a smaller problem (sub-problem) with one less variable, but possibly quadratically more clauses, depending on the number of clauses where l and $\neg l$ appear. This procedure stops when either an empty clause is derived (the formula is unsatisfiable) or only satisfied clauses with pure literals are obtained, which means the formula is satisfiable. If φ is the formula, then we can resume the DP algorithm in the pseudo-code given in algorithm 1.

As a final remark, we should note that the name DP-algorithm is often confused and incorrectly

```
Input  : formula φ
Output: simplified formula φ OR UNSAT
for every variable v in φ do
    for every clause w_i such that v ∈ w_i AND every clause w_j such that ¬v ∈ w_j do
        φ = φ ∪ Resolve (w_i, w_j)
    end
    for every clause w_i such that v ∈ w_i AND every clause w_j such that ¬v ∈ w_j do
        φ = φ - w_i
        φ = φ - w_j
    end
    if φ contains an empty clause then
        return UNSAT
    end
    if φ only contains clauses with pure literals then
        return φ
    end
end
```

Algorithm 1: DP Algorithm

used to refer to the Davis-Logemann-Loveland (DLL) algorithm, although they are distinct algorithms.

3.2 Davis-Logemann-Loveland Algorithm

The Davis-Logemann-Loveland (DLL) algorithm is a complete, backtracking-based algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form (CNF) ([10]).

This algorithm was introduced in 1962 by Martin Davis, George Logemann and Donald W. Loveland (and thus dubbed as DLL, see [10]) and is a refinement of the earlier Davis-Putnam algorithm (DP algorithm, see [11]).

The DLL is a backtrack search procedure and is still the core of many of the most efficient complete SAT solvers of our time, more than 45 years later.

DLL builds upon depth-first search with backtracking. The most basic version of the backtracking algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If the simplified formula is satisfiable then the original formula is satisfiable as well; otherwise, the same recursive check step is done assigning to the literal the opposing truth value. This is known as the *splitting rule*, as it splits the problem into two simpler sub-problems. The simplification step essentially removes all clauses which become true under the assignment from the formula, and all literals that become false from the remaining clauses.

DLL enhances the basic backtracking algorithm by the use of the unit propagation and pure literal elimination rules (both explained in section 2.3) at each step.

If during the search one clause becomes empty then that assignment makes the formula unsatisfiable. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or if all clauses are satisfied. Unsatisfiability of the complete formula can only be detected after exhaustive search.

We can resume the DLL procedure in algorithm 2. In this pseudo code, unit-propagate(l, φ) and pure-literal-assign(l, φ) are functions that return the result of applying unit propagation and the pure literal rule, respectively, to the literal l in the formula φ. In other words, they replace every occurrence of l with true and every occurrence of $\neg l$ with false in the formula φ, and simplify the resulting formula. In practice, the algorithm chooses a variable and then branches the literal. The DLL function only returns whether there is an assignment that satisfies the formula. In a real implementation, the satisfying assignment is typically also returned on success.

```
    Input  : formula φ
    Output: true OR false
1.1 if φ is satisfied then
1.2 |  return true
1.3 end
1.4 if φ contains an empty clause then
1.5 |  return false
1.6 end
1.7 while exists unit clause (l) do
1.8 |  φ = unit-propagate(l, φ)
1.9 end
1.10 while exists pure literal (l) do
1.11 |  φ = pure-literal-assign(l, φ)
1.12 end
1.13 l = choose-literal(φ)
1.14 return DLL(φ∧ l) OR DLL(φ ∧ ¬ l)
2.15
```

Algorithm 2: DLL Algorithm

The DLL algorithm depends on the choice of the branching literal, which is the literal considered in the backtracking step. As a result, this is not exactly an algorithm, but rather a family of algorithms, one for each possible way of choosing the branching literal. Efficiency is strongly affected by the choice of the branching literal: there exist instances for which the running time can differ significantly depending on the choice of the branching literals [38].

4

Local Search for Satisfiability

The most common classification of search algorithms is based on the distinction between systematic and local search. *Systematic search algorithms* traverse the search space of the problem in a systematic manner which guarantees that either a solution is found or that, if no solution exists, this fact is determined with certainty.

On the other hand, *local search algorithms* start at some location in the given problem's search space and then move from the start location to a neighbouring location in the search space which is determined by a decision based on local knowledge only. These local search algorithms are typically incomplete, that is, there is no guarantee that an existing solution will be found, and if no solution exists that fact can never be determined with certainty. Furthermore, these search methods can visit the same location in the search space more than once and they can get stuck in a small number of locations from which they cannot get out: these are called *local minima*, which will be discussed shortly, and require special *escape strategies*.

Many widely known and high-performance local search algorithms make use of randomized choices when generating and/or selecting candidate solutions for a given problem [23]. These algorithms are called *stochastic local search (SLS) algorithms*, and they belong to the most powerful methods for practically solving large and hard satisfiable instances of SAT, and outperform the best systematic search methods on a number of domains. In SLS algorithms the initial position in the search space is chosen randomly, as are the decisions to move from a position to another.

Local search is typically applied to satisfiable problem instances, that is, instances where there is at least one solution. Local search aims to find an assignment of truth values to the variables of the problem, such that it evaluates to true.

We begin this chapter by introducing the most basic of SLS algorithms, the Uniformed Random Picking [23].

Then we give a brief theoretical overview about evaluation functions, local minima and escape strategies.

In the next sections we present SAT algorithms and respective architectures that were stepping stones in SAT solving: GSAT [48] and WalkSAT [46].

Finally, the last section of this chapter focuses on recent developments, following the solvers that stood out in last year's SAT Competition.

4.1 Uniformed Random Picking

The simplest SLS strategy is Uniformed Random Picking [23]. This procedure does not store information about previously visited locations. It starts the search in a randomly chosen location in the search space, i.e. makes a random assignment over all the variables and tests if a solution is found. If this assignment is indeed a solution, then it returns the variable assignment; if it is not a solution, then it randomly chooses a variable and flips its truth value, and checks again for a solution. The given problem instance is usually in CNF. This procedure is illustrated in algorithm 3, where φ is the given CNF formula and $var(\varphi)$ denotes the set of variables of φ.

```
    Input  : formula φ in CNF
    Output: assignment a over var(φ)
2.1 a = randomly chosen assignment over var(φ)
2.2 while true do
2.3     if a satisfies φ then
2.4         return a
2.5     end
2.6     v = randomly chosen variable in φ
2.7     a = a with v flipped
2.8 end
3.9
```

Algorithm 3: Uniformed Random Picking Algorithm

This is the most basic form of SLS algorithms for SAT. There is no objective/evaluation function, no memory is used and no escape strategies are implemented. But it has an important property common to all the other SLS algorithms: the local search steps modify at most the value assigned to one of the propositional variables appearing in the formula; such a move is called a *variable flip*.

Example 4.1.1. *To illustrate some steps of this algorithm let us consider the formula* $\varphi = (a \vee b) \wedge (c \vee \neg a) \wedge (d)$ *as the input for the algorithm. It begins by randomly choosing an assignment, and*

let us assume that assignment is $A = \{(a,1),(b,0),(c,0),(d,1)\}$. *A does not satisfy the formula* φ*, so a variable is randomly picked, b, and its truth value is flipped from 0 to 1, such that now we have* $A = \{(a,1),(b,1),(c,0),(d,1)\}$. *It still does not satisfy* φ*, so a variable is again randomly chosen and flipped: a is picked and now* $A = \{(a,0),(b,1),(c,0),(d,1)\}$. *This assignment satisfies* φ*, so A is returned as the output of the algorithm and it terminates.*

This strategy is quite ineffective, since it does not provide any mechanism to guide the search towards solutions. We will go through some strategies that prevent or overcome the stagnation of the search.

4.2 Evaluation Functions

To improve on the simple uniformed random picking strategy discussed above we need a mechanism to steer the search towards solutions. This can be achieved through an *evaluation function* that maps each search position in the search space into a real number such that the global optima of the given formula corresponds to its solution.

The evaluation function, also called an *heuristic*, is used to rank the candidate solutions in the neighbourhood of the current search position. This function is then integrated into the algorithm in such a way that it improves the search by indicating the best neighbour in the search space, with respect to the solution of the given instance.

Typically the evaluation function is problem specific, and its choice is dependent on the search space, solution set and neighbourhood underlying the SLS strategy being used. We can abruptly reduce the time taken to solve a given set of problems simply by choosing a better evaluation function.

4.3 Local Minima and Escape Strategies

Sometimes it can happen that, during the search, there are no candidate solutions that improve over the current search position. A candidate solution with this property corresponds to a *local minimum*. Local minima are positions in the search space from which no single step can improve over the current position. They are defined as a state whose local neighbourhood does not include a state that is strictly better. Local search algorithms that do not consider this issue can get stuck in such search positions.

In many cases, local minima are quite common, and techniques for avoiding or escaping from local minima are very important for a successful SLS algorithm. Many such methods have been

presented and discussed over the years. We present two of the most simple and popular methods for escaping from local minima.

- **Restart Strategy** [23]: one simple method for modifying the search procedure and escaping from local minima is to restart the search whenever a local minimum is encountered. This technique can work reasonably well when the number of local minima is small, but it can become very costly in many other cases. To implement this method we use *maxTries* and *maxSteps*. The first denotes the number of restarts allowed for an algorithm before it returns 'no solution found'; the second denotes the maximum number of iterations (steps) allowed for an algorithm before it randomly restarts with a new assignment.
- **Non-improving step** [23]: alternatively, one can relax the improvement criterion and, when a local minimum is encountered, perform a randomly chosen non-improving step. Thus, one can randomly choose between performing a normal improving step or randomly choosing a neighbouring position as the next position in the procedure.

Neither of the two mechanisms give any guarantee that the search procedure will escape from the local minima, because after this step it can happen that the only improving position is the previous one. Successful SLS algorithms must have this issue in consideration.

4.4 GSAT Architecture

The GSAT algorithm [48] was one of the first SLS algorithms for SAT and was very important on the development of a broad range of solvers, including many of the current state-of-the-art SLS algorithms for SAT.

GSAT performs a greedy local search for a satisfying assignment of a set of propositional clauses. The procedure starts with a randomly generated truth assignment. It then changes (flips) the assignment of the variable that leads to the greatest decrease in the total number of unsatisfied clauses. Note that the greatest decrease may be zero (sideways move, the number of unsatisfied clauses remains the same) or negative (uphill move, the number of unsatisfied clauses increases). Flips are repeated until either a satisfying assignment is found or a pre-set maximum number of flips is reached (using the restart strategy mentioned earlier). If the only possible move for GSAT is uphill, it will make such a move, but such forced uphill moves are quite rare and are not effective in escaping from local minima or plateaus.

As noted above, local minima are the primary obstacle to the application of local search methods. GSAT's use of sideways moves does not completely eliminate this problem, because the algorithm

can still become stuck on a plateau (a set of neighbouring states each with an equal number of unsatisfied clauses). Thus, it is useful to employ mechanisms that escape from local minima or plateaus by making uphill moves (flips that increase the number of unsatisfied clauses). For details, see algorithm 4.

```
    Input  : formula φ in CNF, maxTries and maxSteps
    Output: solution for φ or 'no solution found'
3.1 for try = 1 to maxTries do
3.2     a = randomly chosen assignment over var(φ)
3.3     for step = 1 to maxSteps do
3.4         if a satisfies φ then
3.5             return a
3.6         end
3.7         x = randomly selected variable flipping that minimizes the number of
            unsatisfied clauses
3.8         a = a with x flipped
3.9     end
3.10 end
3.11 return 'no solution found'
4.12
```

Algorithm 4: The basic GSAT algorithm

One can further improve the GSAT algorithm by introducing several extensions. One of the most successful introductions to GSAT is the *random walk strategy* [45] which results in the following:

- With probability p, pick a variable occurring in some unsatisfied clause and flip its truth assignment.
- With probability $1 - p$, follow the standard GSAT scheme, i.e., make the best possible local move.

Note that the walk moves can be uphill.

The new GWSAT algorithm is outlined in algorithm 5. Starting from a randomly chosen variable assignment, it repeatedly flips variables according to the following heuristic: with a fixed probability wp, a currently unsatisfied clause is randomly selected and one of the variables appearing in it (also randomly selected) is flipped; this is called a *random walk step*. In the remaining cases, one of the variables which, when flipped, achieve the maximal increase (or least decrease) in the total number of satisfied clauses is selected and flipped. If after $maxSteps$ no solution is found, the search is started from a new, randomly chosen assignment. If after $maxTries$ still no solution is found, the algorithm terminates unsuccessfully. GWSAT with $wp = 0$ corresponds to the original GSAT algorithm.

```
Input  : formula φ in CNF, maxTries, maxSteps and wp
Output: solution for φ or 'no solution found'
for try = 1 to maxTries do
    a = randomly chosen assignment over var(φ)
    for step = 1 to maxSteps do
        if a satisfies φ then
            return a
        end
        with probability wp
            w = randomly selected clause unsatisfied under a
            v = randomly selected variable appearing in w
        end
        otherwise
            v = randomly selected variable flip which minimizes the
            number of unsatisfied clauses
        end
        a = a with x flipped
    end
end
return 'no solution found'
```

Algorithm 5: The GWSAT algorithm

4.5 WalkSAT Architecture

The WalkSAT algorithm was first introduced in [46], and differs significantly from the other SLS algorithms in its evaluation function: it counts the number of currently satisfied clauses that will become unsatisfied by flipping a given variable x.

The basic WalkSAT procedure is outlined in algorithm 6. The WalkSAT algorithm starts from a randomly chosen variable assignment and repeatedly selects one of the clauses which is violated by the current assignment. Then, according to the given *slc* heuristic function, a variable occurring in this clause is flipped using a greedy move to increase the total number of satisfied clauses. The heuristic used in the basic WalkSAT algorithm is as follows: if, in the selected clause, variables can be flipped without violating other clause, then one of these variables is randomly chosen. Otherwise, with a fixed probability p, a variable is randomly chosen from the clause and with probability $1 - p$ a specific variable is picked. This variable minimizes the number of clauses that are currently satisfied but would become violated by the variable's flip.

WalkSAT is closely related to GSAT. However, there are a number of significant differences between both of them which accounts for the generally superior performance of WalkSAT.

```
Input  : formula φ in CNF, maxTries, maxSteps and heuristic
         function slc
Output: solution for φ or 'no solution found'
5.1  for try = 1 to maxTries do
5.2      a = randomly chosen assignment over var(φ)
5.3      for step = 1 to maxSteps do
5.4          if a satisfies φ then
5.5              return a
5.6          end
5.7          w = randomly selected clause unsatisfied under a
5.8          x = variable selected from w according to heuristic function slc
5.9          a = a with x flipped
5.10     end
5.11 end
5.12 return 'no solution found'
6.13
```

Algorithm 6: The WalkSAT algorithm

4.6 Recent Developments

SLS research is a very competitive field of study with new developments made every few months. There is a biannual SAT Competition[1] whose purpose is to identify new challenging benchmarks and to promote new solvers for the propositional satisfiability problem as well as to compare them with state-of-the-art solvers. In this section we will briefly analyze the three best solvers in the *Random+SAT* category that were present in the last competition in 2007. They did not compete in the *Random+UNSAT* or *Random+SAT+UNSAT* because these are SLS algorithms that are only viable options for solving satisfiable instances, not for proving unsatisfiability.

The solver that scored gold in the above mentioned category was *gNovelty+* [39]. It is based on the two SLS solvers that were placed first and second in the 2005 SAT competition, *R+AdaptNovelty+* [2] and *G2WSAT* [31]. The former makes *gNovelty+* effective in random $>$ 3-SAT problems (problem instances in which each clause has exactly three variables), although it makes no use of a resolution-based preprocessing step as *R+AdaptNovelty+* does. The latter makes *gNovelty+* effective in 3-SAT problems. In addition, *gNovelty+* uses clause weighting to gain more efficiency.

gNovelty+ is outlined in algorithm 7. At each step, gNovelty+ randomly selects a variable with probability wp that appears in an unsatisfied clause, or, with probability $1-wp$, selects a promising variable that is also the least recently flipped. If no such variable exists, it uses an heuristic based on *AdaptNovelty* to select a variable. It then updates the weights of all unsatisfied clauses and with probability sp smooths the weights of all clauses. Refer to [39] for a more detailed description of the algorithm.

[1] www.satcompetition.org

```
Input  : formula φ in CNF
Output: solution for φ or 'no solution found'
for try = 1 to maxTries do
    initialize the weight of each clause to 1
    a = randomly chosen assignment over var(φ)
    for step = 1 to maxSteps do
        if a satisfies φ then
            return a
        end
        if within a walking probability wp then
            x = randomly selected variable that appears in a false
            clause
        else if there exist promising variables then
            x = promising variable greedily selected, breaking ties
            by selecting the least recently flipped one
        else
            x = variable selected according to the weighted
            AdaptNovelty heuristic
            update the weights of false clauses
            with probability sp smooth the weights of all clauses
        end
        a = a with x flipped
    end
end
return'no solution found'
```

Algorithm 7: The gNovelty+ algorithm

The solver that scored silver in 2007 SAT Competition is *adaptg2wsat0* [33]. It combines the adaptive noise mechanism [22] with *G2WSAT* in the following way: if there are promising decreasing variables [31], flip the oldest one. Otherwise, use *Novelty++* to select the variable to flip. Refer to [33] for a more detailed description.

The solver that scored bronze is *adaptg2wsat+* [54]. It improves on *adaptg2wsat* in two aspects: when there is no promising decreasing variable [31] it uses *Novelty+* instead of *Novelty++* [31; 32] to select a variable to flip from a randomly chosen unsatisfied clause; and when promising decreasing variables exist, it no longer flips the promising decreasing variable with the highest score among all promising decreasing variables, but chooses the least recently flipped one. Refer to [54] for a more detailed description.

5

Local Search for Unsatisfiability

Selman, Kautz and McAllester posed an intriguing challenge in 1997 to use local search to prove unsatisfiability instead of satisfiability [47]. In 2006 and 2007 two different approaches were proposed in response to that challenge [41; 4]. These two algorithms use local search to prove unsatisfiability but not satisfiability, still being incomplete algorithms. They can, however, prove that a formula is satisfiable under certain conditions.

In the next sections we will give an extended overview of these two procedures, with explanation of both algorithms, examples and considerations.

We finish this chapter with a comparison between RANGER and GUNSAT.

5.1 RANGER

RANGER [41] stands for RANdomized GEneral Resolution and was presented in 2006 as the first SLS algorithm that can prove unsatisfiability instead of satisfiability. It explores a space of multisets of resolvents using general resolution and aims at deriving the empty clause non-systematically but greedily, and thus proving unsatisfiability. RANGER will eventually refute any unsatisfiable instance while using only bounded memory.

A theoretical result behind the exploration of local search on multisets of resolvents can be found in [17]. Given an unsatisfiable SAT formula φ with n variables and m clauses, a general resolution refutation can be represented by a serie of formulae φ_1,...,φ_s where φ_1 consists of some or all of the clauses in φ, and φ_s contains the empty clause. Each φ_i is obtained from φ_{i-1} by (optionally) deleting some clauses in φ_{i-1}, adding the resolvent of two remaining clauses in φ_{i-1}, and (option-

ally) adding clauses from φ. The *space* of a proof is defined as the minimum k such that each φ_i contains no more than k clauses.

Intuitively, each φ_i represents the set of *active* clauses at step i of the proof. Inactive clauses are not required for future resolution, and after they have been used as needed they can be deleted (such are the clauses that are subsumed by some other clauses, for example).

The *width* of a proof is the length (in literals) of the largest clause in the proof. Any non-tautologous clause must have length no greater than n, so this is a trivial upper bound for the *width* used in RANGER. However, in practice, it may succeed even if the resolvent length is restricted to a smaller value, which will save memory on large problems.

Each φ_i will be of the same constant size, and derived from φ_{i-1} by the application of resolution or the replacement of a clause by one taken from φ.

The architecture of RANGER is shown in algorithm 8. It has six parameters: the formula φ, three probabilities p_i, p_t and p_g, the width w and the size k of the formula φ_i.

The RANGER algorithm begins by choosing any k clauses from the formula φ into φ_1. It then performs iterations i, either replacing a φ_i clause with a φ clause (with probability p_i) or resolving two φ_i clauses and placing the result r into φ_i. In the latter case, if r is a tautology or contains more than w literals then it is discarded and $\varphi_{i+1} = \varphi_i$. Otherwise a φ_i clause must be removed to make room for r: either (with probability p_g) the removed clause is the longer of the two parents of r or it is randomly chosen. In the former case, if r is longer than the parent then r is discarded and $\varphi_{i+1} = \varphi_i$. With probability p_t any satisfiability-preserving transformation may be applied to φ, φ_i or both. One can apply subsumption and the pure literal rule in several ways as satisfiability-preserving transformations. If the empty clause has been derived then the algorithm returns the message UNSATISFIABLE, otherwise it may not terminate. A time-out condition may be added to restrict the CPU time that the algorithm is allowed to run.

In this algorithm the goal is to derive the empty clause, and as such φ_i must contain some small clauses. This is controlled by the level of greediness (probability p_g). A greedy local move is one that does not increase the number of literals in φ_i. So, increasing p_g will increase the greediness of the search, reducing the proliferation of large resolvents.

Example 5.1.1. *Let us illustrate RANGER's behaviour with an example taken from [42]. Consider the following CNF formula φ with five clauses:*

$$w_1 = (a \vee b \vee c)$$
$$w_2 = (\neg a \vee b \vee c)$$
$$w_3 = (a \vee \neg c)$$

```
Input  : formula φ in CNF, p_i, p_t, p_g, w, k
Output: UNSATISFIABLE or UNKNOWN
for try = 1 to maxTries do
    i = 1 and φ_1 ={any k clauses from φ}
    for step = 1 to maxSteps do
        if φ_i contains the empty clause then
            return UNSATISFIABLE
        end
        with probability p_i
            replace a random φ_i clause by a random φ clause
        end
        otherwise
            resolve random φ_i clauses c,c' giving r
            if r is non-tautologous and |r| ≤ w then
                with probability p_g
                    if |r| ≤ max(|c|, |c'|) then
                        replace the longer of c,c' by r
                    end
                end
                otherwise
                    replace a random φ_i clause by r
                end
            end
        end
        with probability p_t
            apply any satisfiability-preserving transformation to
            φ,φ_i
        end
        i = i + 1 and φ_{i+1} ={the new formula}
    end
end
return UNKNOWN
```

Algorithm 8: The RANGER algorithm

$$w_4 = (\neg a \vee \neg c)$$
$$w_5 = (\neg b \vee c)$$

This formula has three variables. Given that we must have $k \geq n+1$ we set $k = 4$. Suppose that φ_1 contains the following four clauses:

$$w_1 = (a \vee b \vee c)$$
$$w_2 = (\neg a \vee b \vee c)$$
$$w_4 = (\neg a \vee \neg c)$$
$$w_5 = (\neg b \vee c)$$

Now suppose that the condition in line 7 in algorithm 8 is not satisfied, so that we proceed to line 11 and generate a resolvent $w_6 = (b \vee c)$ from w_1 and w_2. Resolvent w_6 is non-tautologous and w_6 has less than three literals, so the conditions in line 12 of the algorithm are satisfied. Suppose that the condition in line 13 is not satisfied, so that the algorithm proceeds to line 19 and replaces clause w_4 by w_6. So far φ_1 has been transformed into the clauses:

$$w_1 = (a \vee b \vee c)$$
$$w_2 = (\neg a \vee b \vee c)$$
$$w_5 = (\neg b \vee c)$$
$$w_6 = (b \vee c)$$

Assuming that the condition in line 23 is satisfied, then both clauses w_1 and w_2 are deleted from φ and φ_1 because they are subsumed by w_6. Then we randomly choose w_3 and w_4 from φ to make φ_2 up to four clauses:

$$w_3 = (a \vee \neg c)$$
$$w_4 = (\neg a \vee \neg c)$$
$$w_5 = (\neg b \vee c)$$
$$w_6 = (b \vee c)$$

In subsequent iterations we may infer $w_7 = (c)$ from w_6 and w_5, and $w_8 = (\neg c)$ from w_3 and w_4. From w_7 and w_8 the empty clause is derived and the algorithm terminates.

RANGER has a useful convergence property: for any unsatisfiable SAT problem with n variables and m clauses, it finds a refutation if $p_i > 0$, $p_i, p_t, p_g < 1$, $w = n$ and $k \geq n+1$. For a proof, see [41]. The space complexity of RANGER is $O(n + m + kw)$. To guarantee convergence, it requires $w = n$ and $k \geq n+1$ so the space complexity becomes at least $O\left(m + n^2\right)$. In practice, it may require k to be several times larger, but a smaller value of w is usually sufficient.

It should be noted that RANGER performs very poorly on unsatisfiable random 3-SAT problems. It is an interesting asymmetry, given that local search performs well on satisfiable random problems. This may be because such refutations are almost certainly exponentially long [8].

5.2 GUNSAT

GUNSAT [4] proposes to make a greedy walk through the resolution search space in which, at each iteration of the algorithm, it tries to compute a better neighbouring set of clauses (a set of clauses similar to the previous one, with only minor variations on a few clauses), differing from the previous one by at most two clauses: one added by resolution and one that may have been removed. A score is given to all pairs of literals based on their frequency appearance in the formula. GUNSAT also makes use of higher reasoning mechanisms, based on *extended resolution* [44] and *unit propagation look-ahead* [29], that are key to make this approach effective.

The GUNSAT algorithm uses a method to score clauses such that it can perform greedy moves later on considering the appropriate clauses. A first measurement (at depth 0) estimates the number of filtered models considering clauses independently. However, it is clear that filtered models are not independent and this measure gives a very inaccurate indication of the value of the proof.

At depth 1 of the approximation, the granularity is fixed on literals, maintaining the estimation of the number of filtered models for each literal. Even though this is an improvement over depth 0 approximation, it is clear that if a literal l occurs in the two clauses $l \wedge q$ and $l \wedge \neg q$ then this scoring scheme is not appropriate. It is important to take the locality of common variables into account.

This is what approximation of depth 2 does. Here, the granularity is on pairs of literals. Let us consider a clause c_i of length n_i. Each pair (l_1, l_2) appearing in c_i is credited a weight of $w_2(n_i) = \frac{2^{n-1-n_1}}{n_i.(n_i-1)}$. The score of a pair of literals (l_1, l_2) is defined as the sum of its weights in all clauses and noted $S(l_1, l_2)$. The score $S(c)$ of a clause c is the sum of the scores of all the pairs of literals it contains.

The deeper the approximation, the higher the cost to maintain it. Triplets of literals could still be considered, but their representation in reality would quickly become unpracticable on realistic UNSAT benchmarks.

Before continuing our explanation of GUNSAT's algorithm, let us introduce a technique it uses and that will be useful in understanding GUNSAT's behaviour later on. The 2-saturation step ensures that, each time a new binary clause is found in φ, all resolutions between the set of binary

clauses are performed to saturate φ with it. In order to exploit their full power, an equivalency literal search is performed. While performing the binary clause saturation the algorithm may find new unary clauses (note that in a resolution step between two binary clauses the resolvent can have either one or two literals). The literal l of the unary clause is then propagated in the whole formula by unit propagation. An inconsistency may be found at this step and it returns UNSAT to the main algorithm, proving the unsatisfiability of the formula. Refer to [5] for the use of binary clause saturation for preprocessing purposes. Let us illustrate this 2-saturation step with an example.

Example 5.2.1. *Consider the same formula as presented in example 5.1.1:*

$$w_1 = (a \vee b \vee c)$$
$$w_2 = (\neg a \vee b \vee c)$$
$$w_3 = (a \vee \neg c)$$
$$w_4 = (\neg a \vee \neg c)$$
$$w_5 = (\neg b \vee c)$$

We start the GUNSAT algorithm with a call to 2-saturation, where the algorithm performs resolution between all pairs of binary clauses. Our binary clauses are w_3, w_4 and w_5. We have that

$$w_6 = w_3 \otimes w_4 = (\neg c)$$
$$w_7 = w_3 \otimes w_5 = (a \vee \neg b)$$
$$w_8 = w_4 \otimes w_5 = (\neg a \vee \neg b)$$

w_6 is a unit clause, and as such unit propagation is performed with the literal $\neg c$, such that, after this step, we have:

$$w_1 = (a \vee b)$$
$$w_2 = (\neg a \vee b)$$
$$w_5 = (\neg b)$$
$$w_7 = (a \vee \neg b)$$
$$w_8 = (\neg a \vee \neg b)$$

Clauses w_3, w_4 and w_6 have been removed through unit propagation. But now w_5 is also a unit clause, and as such unit propagation is performed again with the literal $\neg b$, such that:

$$w_1 = (a)$$
$$w_2 = (\neg a)$$

Clauses w_5, w_7 and w_8 have been removed through unit propagation. Now we have two unit clauses, and after the next unit propagation no clauses will remain. Therefore this call to 2-saturation returns UNSAT to the main algorithm, where it also returns UNSAT and terminates. This simple example shows that 2-saturation is sometimes enough to prove the unsatisfiability of a formula.

Instead of trying to improve a measurement over the whole proof, GUNSAT focuses on *quadruplets* of pairs only. In order to derive an empty clause, it needs a step where l and $\neg l$ are in the current formula. Because it performs 2-Saturation with unit clause propagation at each step of the proof, such a case never occurs. Thus, two literals l_1 and l_2 must be found such that the clauses $l_1 \vee l_2$, $l_1 \vee \neg l_2$, $\neg l_1 \vee l_2$ and $\neg l_1 \vee \neg l_2$ can be derived from the formula. It will try to improve on the scores of the four pairs that result from the combinations of the above literals, called *quadruplets* and noted $[x_1, x_2]$. The score of each quadruplet $S_q([x_1, x_2])$ is defined as the sum of the squares of the scores of its pairs $S_q([x_1, x_2]) = S(l_1, l_2)^2 + S(l_1, \neg l_2)^2 + S(\neg l_1, l_2)^2 + S(\neg l_1, \neg l_2)^2$. Any move that increases the score of one of the best scored quadruplets is therefore a greedy move.

Any explicit exploration of the neighbourhood is impossible in practice. However, if it is known in advance that the score of one of the best quadruplets needs to be improved, then the score of any one of its pairs may be tried to be improved.

Let us consider $[x_1, x_2]$ as the best scored quadruplet in the formula. Increasing the score of any one of its pairs (l_1, l_2) is the same as adding any new clause containing both l_1 and l_2. Short clauses will give (l_1, l_2) a higher score, and should be preferred.

When looking for a new clause with l_1 and l_2, a pivot variable must first be localized on which the resolution rule will be performed. If such a pivot p exists, then $S(l_1, p) > 0$ and $S(\neg p, l_2) > 0$. A clause c_p must then be chosen, containing l_1 and p, as well as a clause $c_{\neg p}$ containing $\neg p$ and l_2. To prevent the new clause from having a high score, only the two lowest scored clauses are considered to produce the new clause c. Because of this restriction, it is not always possible to generate a new clause c containing l_1 and l_2 (for example, c may be subsumed by some clause in the formula, or it may simply be a tautology). If one pair's score cannot be improved, then the other pairs of the same quadruplet are iteratively tried. If the algorithm fails on all pairs of the best quadruplet, then the second quadruplet is tried and so on.

It was very important to add three powerful mechanisms to GUNSAT in order to make it competitive with other resolution-based reasoning systems: binary clause saturation, unit propagation look-ahead with pairs of literals and extended resolution. The first but essential refinement concerns subsumptions. Before adding a new clause, the algorithm performs a forward/backward subsumption detection [58].

The extended resolution [52] is used when the algorithm has tried to increase the score of a given pair of literals too many times without success, and it uses extended resolution to artificially increase that score. The extended rule $e \Leftrightarrow l_1 \vee l_2$ is encoded by the three clauses $(\neg e \vee l_1 \vee l_2)$, $(e \vee \neg l_1)$ and $(e \vee \neg l_2)$. These three clauses and the new variable e can then be added to the formula.

Look-ahead techniques are used to detect equivalences between literals until an inconsistency is found. It uses a reformulation of the initial formula, based on a set of triplets ($p \Leftrightarrow q \Leftrightarrow r$ and $p \Leftrightarrow q \Rightarrow r$ only). GUNSAT uses look-ahead unit propagation on pairs of literals, such that the four possible pairs of values are propagated in φ, searching for more unit propagations.

Example 5.2.2. *To illustrate the look-ahead behaviour of GUNSAT suppose we have the following formula φ with the clauses:*

$$w_1 = (a \vee b \vee c)$$
$$w_2 = (a \vee \neg b \vee c)$$
$$w_3 = (\neg a \vee b \vee c)$$
$$w_4 = (\neg a \vee \neg b \vee c)$$
$$w_5 = (d \vee \neg e)$$

We will use the Stalmark method [50] to detect equivalencies between literals. Let us consider that if $a = 0$ and $b = 0$ then c must be 1. Furthermore, if $a = 0$ and $b = 1$, $a = 1$ and $b = 0$, $a = 1$ and $b = 1$ then, for these three cases, $c = 1$. Thus, we have that c must always be 1, regardless the assignment made to other variables, and the clause $w_6 = (c)$ can be added to the formula as a unit clause. We can then perform unit propagation, and thus the name unit propagation look-ahead.

Example 5.2.3. *The unit propagation look-ahead can also be used to derive an empty clause. Suppose we have the following formula φ:*

$$w_1 = (a \vee \neg b)$$
$$w_2 = (\neg a)$$
$$w_3 = (\neg c \vee d)$$

If we consider that $b = 1$, then, according to clause w_1, we have that a must be 1. But from clause w_2, if that clause is to become satisfied, a must be 0. Thus, we arrive at an inconsistency. Therefore, no assignment made to the literal a can make the formula satisfied and as such we conclude it is unsatisfied.

Intuitively, GUNSAT will add new clauses and remove existing ones to the formula φ, trying to derive the empty clause by using the resolution rule. This procedure is depicted in algorithm 9.

```
Input  : formula φ in CNF
Output: UNSAT or UNKNOWN
for try = 1 to maxTries do
    for step = 1 to maxSteps do
        if 2-saturation(φ) returns UNSAT then
            return UNSAT
        end
        if |φ| > MaxSize then
            remove-one-clause(φ)
        end
        add-one-clause(φ)
        add-extended-variables(φ)
        simplify-look-ahead(φ)
    end
    replace φ by all its vital clauses
end
return UNKNOWN
```

Algorithm 9: The GUNSAT algorithm

GUNSAT either proves that a problem instance is unsatisfiable or it runs indefinitely. It uses the random start strategy such that, if it does not derive an empty clause within *maxTries* then it returns UNKNOWN.

If the size of the current formula is greater than a fixed $MaxSize$ then a clause is removed by `remove-one-clause`. However it is essential to keep *vital clauses* in φ. They ensure the unsatisfiability of the formula is preserved. Vital clauses are initial clauses, or any clause that previously subsumed another vital clause. The algorithm also keeps all binary clauses. In each iteration there is a call to `add-one-clause`, which adds one clause to the current formula according to an heuristic that considers the scores of clauses and pairs of literals. Both `add-extended-variables` and `simplify-look-ahead` use high reasoning mechanisms (extended resolution in the former, unit propagation look-ahead in the latter) to improve the chances of deriving an empty clause in the next iteration. `Add-extended-variables` adds the three clauses generated through extended resolution to the formula. `Simplify-look-ahead` may use unit propagation on the formula. It may also conclude the formula is unsatisfied as seen in example 5.2.3. When GUNSAT fails to derive the empty clause after *maxSteps* a restart is performed. By then all clauses, except vital and binary clauses, are removed.

5.3 RANGER vs. GUNSAT

There are some few important differences between these two local search algorithms for proving unsatisfiability. RANGER generates a large number of the shortest possible clauses as fast as

possible, using non intelligent local moves, whereas GUNSAT takes longer to make more intelligent moves based on a more complex objective function. GUNSAT also uses higher reasoning techniques like extended resolution and unit propagation look-ahead (RANGER uses only general resolution). As such, GUNSAT has a more powerful reasoning mechanism and a more refined heuristic to guide moves, whereas RANGER is simpler but performs many more moves per second. Also, unlike GUNSAT, RANGER uses a mechanism to ensure bounded memory.

6

The Original RANGER

To support their paper, Local Search for Unsatisfiability [41], S. Prestwich and I. Lynce developed a program in C that, given a formula in CNF, computes whether that formula is unsatisfiable. This program is based on the algorithm presented for the first time in [41] that was dubbed RANGER, for RAndomized GEneral Resolution. Refer to section 5.1 - RANGER for a more detailed description of the algorithm and theoretic concepts associated.

6.1 Input/Output

As mentioned above, RANGER may prove that a formula is unsatisfiable. It receives a formula encoded in a CNF file as input and runs until either an empty clause is found (which means that the formula is unsatisfiable) or until a predetermined number of iterations has been reached. The latter case does not prove the unsatisfiability of the formula, because it may happen that the program has not ran long enough.

The program used is *ranger_release.c*. It has been developed and tuned to run on UNIX machines, and it is not guaranteed that it will run on other operating systems.

Before starting the program we must set the number of variables and clauses of the formula in the file *ranger_release.c*. To start the program we must first compile it with the simple command:

gcc ranger_release.c

The file *a.out* will be generated. To run the program with an input file one must do

./a.out < <file.cnf>

Let us assume we have a formula in CNF encoded in the file *formula.cnf*. To determine whether the formula is unsatisfiable let us run ranger with this file as input with the following command:

./a.out < formula.cnf

After the program finishes there are usually two possible outcomes:

- REFUTED, which means that the program found that the formula given as input is unsatisfiable;
- HALTED, which means that the program could not determine whether the formula is unsatisfiable or not. It could happen that the formula is indeed unsatisfiable but the program did not run long enough to prove it, or that the formula is satisfiable, in which case the program will never end.

If the program finishes with REFUTED, then it will also print the number of variables and clauses remaining in the formula (after all the transformations made to it). On the other hand, if it finishes with HALTED, it will print the final number of clauses, pure literals and variables that were not being considered in the current formulation (unmentioned variables).

Independently of the result reached, the program will always print seven columns, while it is running, which correspond to the following information:

- Iterations: the number of iterations performed so far;
- Resolutions: the number of resolutions performed;
- Subsumptions: the number of subsumptions applied;
- Literals removed: the number of literals removed through subsumption;
- Pure literals: the number of pure literals currently in the formula;
- Clauses: the number of clauses currently in the formula;
- Literals: the number of literals currently in the formula.

At the end, a file (named *processed.cnf* by default) is generated, containing the formula that was being processed when the program finished. This formula is, in most cases, different from the initial one due to the transformations performed by the algorithm.

An example of such a formula when RANGER proves the unsatisfiability of a given instance is as follows (this example was taken from a `uuf50` instance that was proved unsatisfiable by RANGER):

```
p cnf 50 4
24 0
-38 0
-43 38 0
43 -24 0
```

6.2 Data Structures

The program distinguishes between two instances of the initial formula: *problem* and *derived*. The first corresponds to the initial formula and can only be modified by the satisfiability-preserving transformations (see the RANGER algorithm, section 5.1, line 24) and is represented in the RANGER algorithm by φ. The second is equivalent to the various iterations of the algorithm, φ_i, and is potentially modified in each iteration.

The most relevant data structures implemented by ranger_release are presented bellow:

- *struct clause* is a structure that represents clauses, comprised of the length of the clause and an array with the literals contained in the clause.
- *problem0*, *Clauses0* and *literals0*. the first is used to store the clauses present in the *problem* formula and is an array of clauses which dimension is the number of clauses in *problem*; the second and third store the number of, respectively, clauses and literals in *problem*.
- *Poccs* and *Noccs*, which store the number of occurrences of each variable in its positive form (*Poccs*) or in its negative form (*Noccs*). Both are a simple array which dimension is the total number of variables in the formula.
- *LocationP* and *LocationN*, which store the location (in terms of clauses) where each variable appears in its positive (*LocationP*) or negative (*LocationN*) form. Both are composed by three arrays: the first has size two and determines whether the variable being considered is in the *derived* formula or the *problem* formula; the second position maps the variable, and its dimension is the total number of variables in the formula; and the third position indicates the actual clause where the variable is located and has dimension two, which means it can only store two clauses.

6.3 Parameters

The most relevant parameters used in ranger_release are:

- V, which corresponds to the number of variables in the input formula.
- C, which corresponds to the number of clauses in the input formula.
- *Space* is the space of the proof and is defined as the minimum k such that each φ_i contains no more than k clauses [41] (k and φ_i as used in the RANGER algorithm). In the original implementation it is defined as ten times the number of variables in the formula.
- *Width* of a proof is the length (in literals) of the largest clause in the proof [41]. It is defined as the total number of variables.
- *Pi* is the inertia probability, which defines the rate at which a clause in *problem* is inserted in the current *derived*. It is set to 10 (corresponding to 10%, see line 7 of the RANGER algorithm).
- *Pt* is the transformation probability, which defines the rate at which satisfiability-preserving transformations are performed. It is set to 90 (corresponding to 90%, see line 23 of the RANGER algorithm).
- *Pg* is the greed probability and defines the greediness of the program. It is set to 95 (corresponding to 95%, see line 13 of the RANGER algorithm).
- *Iterations* which defines the number of iterations performed by the program before it is halted (and therefore it cannot be determined if the formula is unsatisfiable or not). The number of iterations is set to one billion (1.000.000.000).

6.4 Functions

The most relevant functions implemented by ranger_release are:

- *i_step*, which replaces a clause in *derived* (φ_i) by a clause in *problem* (φ) (line 8 of the RANGER algorithm).
- *r_step*, which resolves a random φ_i clause and performs the greedy step in the algorithm (lines 11 to 19 of the RANGER algorithm).

The following are the functions that implement the satisfiability-preserving transformations performed by the program, as indicated in line 24 of the algorithm:

- *sd_step*: performs the subsumption rule. Chooses two clauses from the *derived* set of clauses with a common variable either in its positive or negative form and checks whether they can

be subsumed. If this is the case then it replaces the subsumed clause (in φ_i) by a random clause in φ.

- *d_step*: performs the subsumption rule. Randomly chooses two clauses from φ that either contain a variable appearing positively or negatively. If one of them can be subsumed by the other then the subsumed clause is removed from the problem.

- *p_step*: performs the pure literal rule. Randomly chooses a clause in φ and checks whether it contains a pure literal. If this is the case then it removes the clause from the formula.

- *pd_step*: performs the pure literal rule. Randomly picks a variable that either appears positively or negatively in a clause. If this variable is a pure literal in φ_i then the clause where it appears is replaced by a clause randomly taken from φ.

- *sp_step*: performs the subsumption rule. Randomly picks a variable and then picks two clauses (one from φ_i and the other from φ) where that variable appears either positively or negatively. If the clause from φ_i subsumes the clause from φ, then the latter clause is replaced by the φ_i clause.

7

Improving RANGER

7.1 Modifications to the original RANGER

The algorithm developed as a consequence of the research work done for this dissertation is based on the original solver by S. Prestwich and I. Lynce and dubbed ranger_release (refer to chapter 5.1 - RANGER for a more detailed description of the algorithm and theoretic concepts associated, and to chapter 6 for an overview of the solver itself).

It was kindly yielded to serve as a starting point for the achievement of the objectives proposed in this thesis: the development of a stochastic local search solver that could prove unsatisfiability and that combined the most prominent elements of both RANGER [41] and GUNSAT [4], the current state-of-the-art SLS solvers for unsatisfiability.

Before starting to implement new features into the solver (as we will see shortly, techniques like unit propagation look-ahead and extended resolution) we made a series of modifications to the original tool to better accommodate our needs. In the following sections we will present the changes made to the original ranger_release solver and state the reasons which made us choose to do so.

7.1.1 Constants to Global Variables

One of the first difficulties we came upon when we started to run tests with the original ranger program on different instances was the fact that, for each instance that we wanted to test, we had to manually set the number of variables and clauses that composed that instance. We had to hard-code it on the ranger_release code, which implied reading the instance manually to find out

the number of variables and clauses, then changing their number in the code, compile and only then run the program. And we had to do this for every instance which had a different number of variables and/or clauses from the previous one.

As we can see from chapter 6.3 - Constants, both the number of variables and the number of clauses were represented as constants in the code. We changed that to global variables which kept the same name, V for variables and C for clauses, and that indicated the number of variables and clauses in the formula being used in the current iteration of the algorithm.

When the program is reading the CNF file it will dynamically set the number of variables and clauses according to the formula being provided, without any need for external tinkering.

This modification also implied that two other constants had to be changed into global variables: $Width$ and $Space$, which correspond, respectively, to the length (in literals) of the largest clause in the proof, and as the minimum k such that each φ_i contains no more than k clauses. Again, refer to chapter 6.3 - Constants for a more complete explanation of these variables.

7.1.2 Addition of Seeds

A random seed (or seed state, or just seed) is a number used to initialize a pseudo random number generator. As a stochastic algorithm, ranger_release extensively utilizes pseudo random numbers for decisions.

As an example, let us refer again to chapter 6.3 - Constants, and to the probability P_t. It is defined as 90 which means that if it is used to decide whether a transformation will be tried on that iteration of the algorithm or not, it has a 90% chance that it will be tried. A number is pseudo randomly generated and the remainder of its division with 100 is computed. If that number is less than or equal to P_t, the transformation will take place, otherwise it will not.

The ranger_release solver uses the `srand` function to seed the pseudo random number generator with an integer seed, which determines what sequence of pseudo random numbers will be returned in future calls to the function `rand`. This can be used to help ensure the sequence is different at each run (with a random seed), or to generate the same sequence on a subsequent run (with the same seed), i.e., for reproducibly random results.

The original ranger_release uses time as a random seed through the instruction:

```
srand(time(NULL));
```

but it does not have a mechanism to reproduce random results. That means we could not reproduce the steps that the algorithm took for a certain instance, because the seed used to initialize the pseudo random number generator was always different.

We added such mechanism, where the user specifies whether he wants to use a random seed or a determined seed when he starts the program. Thus, to run the solver one must do the following:

./a.out <seed> < <file.cnf>

where *seed* represents the seed to be used, or one can type:

./a.out < <file.cnf>

to use a seed based on time.

Let us assume we have a formula in CNF coded in the file *formula.cnf*. If we want to reproduce the result later, we use a seed:

./a.out 1 < formula.cnf

In this case we used the seed 1. If we do not care about reproducing the results, we may simply type

./a.out < formula.cnf

and a seed will be taken according to the current time.

7.1.3 Dynamically Allocated Arrays

As a consequence of modifying the variables, clauses, space and width from constants to global variables, some arrays that depended on them become useless as they are. The length of the arrays $problem0$, $problem$, $derived$, $LocationP$, $LocationN$, $Poccs$ and $Noccs$ can no longer be initialized before the number of variables and clauses are read from the file. The original ranger_release initializes their length as in the following example:

```
struct clause problem[C]
```

where the length of the array *problem* is initialized with C positions. This approach worked when C was a constant, but it no longer does. Thus, we now have a method to dynamically allocate space for all variables that depend on uninitialized values to be allocated. This method, `init`, is called during the input phase of the algorithm (phase where it reads the CNF file) and after the values for V and C are determined. The above array would now be allocated dynamically, as in:

```
struct clause * problem = (struct clause *) malloc(sizeof(struct clause)*C);
```

The modified variables to be dynamically allocated are:

- $problem0$, which is an array of the type `struct clause` with length C and which stores the clauses from the original formula; it is never modified.

- *problem* is an array similar to *problem*0 but that may be modified during the search to accommodate for subsumptions, for example.

- *derived*, which is an array similar to the arrays above but with length equal to *Space*, and stores the clauses being used in each iteration; it is frequently modified.

- *Poccs* and *Noccs*, which store the number of occurrences of each variable in its positive form (*Poccs*) or in its negative form (*Noccs*). Both are a simple array which dimension is the total number of variables in the formula.

- *LocationP* and *LocationN*, which store the location (in terms of clauses) where each variable appears in its positive (*LocationP*) or negative (*LocationN*) form. Both are composed by three arrays: the first has size two and determines if the variable being considered is in the derived formula or the problem formula; the second position maps the variable, and its dimension is the total number of variables in the formula; and the third position indicates the actual clause where the variable is located and has dimension two, which means it can only store two clauses.

7.2 RANGER with Unit Propagation Look-Ahead

The solver GUNSAT successfully uses a method dubbed Unit Propagation Look-Ahead [4] to improve its basic algorithm (refer to chapter 5.2 - GUNSAT for a detailed description of this method). The literals of the formula under consideration are extensively checked to see if there are any conflicts arising from assignments.

7.2.1 Basic Unit Propagation Look-Ahead Method

In its most basic form, unit propagation look-ahead with one variable will assign a value to that variable and check whether the formula is satisfiable or not, and then assign the opposite value and recheck. Three cases may occur:

- The formula is unsatisfiable for both cases, which means it is unsatisfiable for any possible set of assignments.

- The formula is unsatisfiable for one assignment but satisfiable for the other assignment, which means that that variable can only be assigned one value (the one that did not make the formula unsatisfiable) and this is represented by a new unit clause; we then just have to apply the unit clause rule and thus remove the variable from the formula.

- The formula is satisfiable for both assignments, and nothing can be concluded off this variable.

7.2.2 Unit Propagation Look-Ahead with Two Variables

GUNSAT implements unit propagation look-ahead using two variables. These two variables are then assigned a value such that the four possible combinations are covered. If v_1 and v_2 are our variables, then the four possible combinations are:

- $v_1 = 0$ and $v_2 = 0$;
- $v_1 = 0$ and $v_2 = 1$;
- $v_1 = 1$ and $v_2 = 0$;
- $v_1 = 1$ and $v_2 = 1$.

Let us define a *conflict* as a situation where, given an assignment for both variables, the formula evaluates to false. For each iteration of this look-ahead method, i.e., for each combination of variable assignments, we store the value of each variable in the formula, only if that variable is forced to be assigned (note that these assignments are only temporary, done for each iteration of the look-ahead and stored only for the duration of the look-ahead for the two variables). Let us define the *intersection* over a variable as the set of assignments to that variable, after all the combinations of the look-ahead method are executed, but before the unit propagation phase begins.

Example 7.2.1. *Let us consider a variable v represented as an array of four positions, where each position corresponds to the assignment to that variable after each combination of the look-ahead algorithm. If, after the look-ahead method is complete, the array looks like:*

```
v[1,1,0,1]
```

then this means that on the first, second and fourth combinations, the variable was assigned to 1; on the third combination, the variable was assigned the value 0. In this case we can conclude nothing about this variable (if we consider all four combinations), because not all values in the array v are equal. On the other hand, if we only consider the first two combinations, then we conclude that this variable must be assigned to 1 for the formula to be satisfiable.

According to the number of conflicts after the application of the look-ahead technique, we have five possible scenarios for the unit propagation phase:

- **Zero conflicts**: if there are no conflicts, we will consider all four possible assignments when deciding the intersections; if a variable is assigned the same value through all combinations, then that variable will be propagated and the unit clause rule will be applied; this result is applied to all the following cases excluding the last one;
- **One conflict**:
  - the intersections will be calculated, but only considering the combinations that did not yield a conflict (three in this case);
  - a binary clause is added to the formula: this clause results from the negation of the assignments that yielded a conflict;
- **Two conflicts**:
  - as above, the intersections will be calculated, but only considering the combinations that did not yield a conflict (two in this case);
  - two binary clauses are added to the formula, resulting each one from the negation of the assignments that yielded a conflict; there is a special case where only a unit clause is added, which happens when both conflicts are due to an initial assignment to the same variable (the two binary clauses are resolved to yield the unit clause);
- **Three conflicts**:
  - the variables that were assigned in the only combination that did not yield a conflict will be propagated;
  - two unit clauses are added, each one with each variable assignment that did not yield a conflict;
- **Four conflicts**: the formula is unsatisfiable for all combinations, which means the formula is unsatisfiable.

Let us illustrate the unit propagation look-ahead method with two variables with the following examples:

Example 7.2.2. *Let us consider the following propositional formula φ in CNF:*

$$w_1 = (a \vee b \vee c)$$
$$w_2 = (\neg a \vee b \vee d)$$
$$w_3 = (a \vee \neg b \vee c)$$
$$w_4 = (\neg a \vee c \vee \neg d)$$
$$w_5 = (\neg a \vee \neg b \vee d)$$

and let us apply all four combinations of the look-ahead algorithm for the variables a and b:

- *If we start with $a = 0 \wedge b = 0$ we satisfy clauses w_2, w_3, w_4 and w_5. From clause w_1 we conclude that, for this specific assignment, we must assign $c = 1$ for the formula to be satisfiable; the set of assignments for this iteration is $\{a = 0, b = 0, c = 1\}$.*
- *With $a = 0 \wedge b = 1$ we satisfy clauses w_1, w_2, w_4 and w_5, and from clause w_3 we have that $c = 1$, with the assignment $\{a = 0, b = 1, c = 1\}$.*
- *With $a = 1 \wedge b = 0$ we satisfy clauses w_1, w_3 and w_5 and conclude from clause w_2 that we must assign $d = 1$. Clause w_4 is simplified to $w_4 = (c \vee \neg d)$. In the next iteration, clause w_2 becomes satisfied and clause w_4 implies $c = 1$; the set of assignments is $\{a = 1, b = 0, c = 1, d = 1\}$.*
- *Finally, for $a = 1 \wedge b = 1$, clauses w_1, w_2 and w_3 become satisfied and from clause w_5 we conclude that $d = 1$. In the next iteration, clause w_5 becomes satisfied and from clause w_4 we have that $c = 1$ with the assignment $\{a = 1, b = 1, c = 1, d = 1\}$.*

The vector values for a and b are `a[0,0,1,1]` *and* `b[0,1,0,1]`*. Thus, we have no conflicts and as such, according to the look-ahead technique discussed above, we will try to find out a variable that is assigned the same value through all the four combinations of assignments of the variables a and b. The array of intersections for the variables c looks like*

`c[1,1,1,1]` *or, following the sets of assignments above*

$\{a = 0, b = 0, c = 1\} \cap \{a = 0, b = 1, c = 1\} \cap \{a = 1, b = 0, c = 1, d = 1\} \cap \{a = 1, b = 1, c = 1, d = 1\} = \{c = 1\}$

All the values in the array are equal, and the intersection of all the set of assignments yields $\{c = 1\}$ so we may conclude that variable c will be propagated, with $c = 1$, and we apply the unit clause rule.

Example 7.2.3. *Let us now consider the following propositional formula φ:*

$$w_1 = (\neg a \vee \neg b \vee c)$$
$$w_2 = (a \vee \neg b \vee \neg d)$$
$$w_3 = (\neg a \vee \neg c)$$
$$w_4 = (a \vee d)$$

and let us apply all four combinations of the look-ahead algorithm for the variables a and b:

- *With $a = 0 \wedge b = 0$ clauses w_1, w_2 and w_3 become satisfied and w_4 simplifies to $w_4 = (d)$, i.e., $d = 1$;*

- *We now consider* $a = 0 \wedge b = 1$. *Clauses* w_1 *and* w_3 *become satisfied. Simplifying clause* w_2 *yields* $d = 0$ *and simplifying clause* w_4 *yields* $d = 1$, *and therefore we arrive at a conflict;*

- *With* $a = 1 \wedge b = 0$ *clauses* w_1, w_2 *and* w_4 *become satisfied, while with clause* w_3 *we conclude that* $c = 0$;

- *If we consider* $a = 1 \wedge b = 1$ *then clauses* w_2 *and* w_4 *become satisfied, while clause* w_1 *is simplified to* $w_1 = (c)$ *and clause* w_3 *to* $w_3 = (\neg c)$. *We thus arrive at a conflict, because the variable* c *cannot be assigned two opposite values.*

We now have two conflicts. According to the rules described above, we add two binary clauses to the formula, resulting each one from the negation of the assignments that yielded a conflict. The clauses added are:
$w_5 = (\neg a \vee \neg b)$
$w_6 = (a \vee \neg b)$

We can apply resolution to the above clauses and generate the clause
$w_7 = (\neg b)$

This is the special case referred above, where, if a variable assignment is repeated in both conflicts, then only a unit clause is added through resolution.

Example 7.2.4. *Finally, let us consider the following CNF formula* φ:

$$w_1 = (a \vee b)$$
$$w_2 = (\neg a \vee \neg b)$$
$$w_3 = (\neg a \vee b)$$
$$w_4 = (a \vee \neg b)$$

And let us run the look-ahead algorithm for the variables a *and* b:

- *With the assignment* $a = 0 \wedge b = 0$, *clauses* w_2, w_3 *and* w_4 *become satisfied but clause* w_1 *yields a conflict;*

- *With* $a = 0 \wedge b = 1$, *clauses* w_1, w_2 *and* w_3 *become satisfied but clause* w_4 *results in a conflict;*

- *With* $a = 1 \wedge b = 0$, *we have that clauses* w_1, w_2 *and* w_4 *become satisfied, while clause* w_3 *yields a conflict;*

- *Finally, with* $a = 1 \wedge b = 1$ *clauses* w_1, w_3 *and* w_4 *become satisfied, but clause* w_2 *results in a conflict.*

In this example we have four conflicts and, according to the look-ahead algorithm, because the formula is unsatisfiable for all possible assignments to the variables under consideration, the formula is unsatisfiable.

7.2.3 Implementation of the Unit Propagation Look-Ahead

One of the objectives of this dissertation is to successfully integrate features of the GUNSAT algorithm into the RANGER algorithm. We did not want to modify the most important properties of the original RANGER, nor alter its flow. The unit propagation look-ahead was, thus, added to the section of satisfiability-preserving transformations (refer to algorithm 8, lines 23-25, chapter 5.1 - RANGER). The probability, P_t, to execute these transformations is 90%, and like the other transformations, unit propagation will be executed, on average, in 90% of the iterations of the algorithm.

This addition is divided into two parts: unit propagation look-ahead and unit propagation only. The first is ran only once, during the first time that satisfiability-transformations are executed, due to the negative impact it has on the performance. The look-ahead with two variables, over all variables in the formula, can be very heavy and deteriorate the performance of the algorithm for very little gain in return.

The second part of this addition is unit propagation without look-ahead, and this is executed in every satisfiability-preserving transformation. The details of both methods will be discussed shortly.

Let it be noted that these methods can prove the unsatisfiability of a formula by themselves: if, during the look-ahead method, four conflicts arise, then the formula is unsatisfiable; likewise, if during the unit propagation phase, a conflict is detected, the function also returns UNSAT.

When, during the unit propagation, a unit clause is found, it will be computed according to the unit clause rule (refer to chapter 2.3.1 - The Unit Clause Rule). A free literal is found within that unit clause, l, and two rules are applied:

- Every clause containing the free literal l is removed from the formula;
- The literal $\neg l$ is removed from every clause where it appears.

Whenever we refer to propagated literals, or literals which are propagated, it means they are free literals in a unit clause and the unit clause rule will be applied.

7.2.4 New/Modified Data Structures

With the implementation of the unit propagation look-ahead into ranger_release, it was necessary to modify some data structures and to add several others to accommodate the new algorithms.

The concept of making assignments to variables in the formula had to be introduced because the base RANGER algorithm does not consider them. This concept will only be used within the unit propagation and look-ahead methods, though, and will not interfere with the rest of the program.

Similarly, it has now become important to know the specific location of each literal in the formula, i.e., we must have a complete understanding of where each literal is in each clause, and a method to quickly reach it. The arrays *LocationP* and *LocationN* were modified to address this problem.

It has now become more important to know the number of each literal occurrences in the working formula (*derived*), as well as in the base formula (*problem*). Previously, only the number of literals in *problem* were needed. Two new array variables were added to cope with this issue.

The modified data structures are shown below:

- `LocationP` and `LocationN`: These are the arrays that store the location of each literal in each clause in the following way:

  `LocationP[N,V,2]`

  where `N` is either 0 or 1, depending on whether we are considering the literals in *problem* or *derived*; `V` is the variable being considered, and its length is the number of variables in the formula; and the last position of the array is the number of clauses it stores, which is two in the original RANGER. If we want to know in which clause a certain positive v literal appears, in the working formula (*derived*), we would do:

  `LocationP[0,`v`,0]`

  The two clauses it stores, for each literal, are randomly chosen from any clause where that literal appears. As has been said above, we now need to know exactly in which clause each literal is, and not only two randomly chosen clauses. We addressed this problem by extending the two position array to the length of the number of clauses in the formula. Thus, the variable becomes:

  `LocationP[2,V,C]`

  where `C` is the number of clauses in the formula. This way, if we want to check if a negative literal v appears in the clause c in the base formula (*problem*) we just do:

  `LocationN[1,`v`,c]`

  Furthermore, we also added the location of literals in the first formula read from the CNF file, always unaltered, *problem*0. The final `LocationP` is:

  `LocationP[3,V,C]`

Note that we implemented the same changes to `LocationP` and to `LocationN`, where the first stores the location of positive literals and the second stores the locations of negative literals.

The new data structures added to the algorithm are presented below:

- `DPoccs` and `DNoccs`: in the original ranger_release we have two variables which store the number of occurrences of each variable in its positive form (`Poccs`) or in its negative form (`Noccs`). Both are a simple array which dimension is the total number of variables in the formula. These two variables consider only the literals of the base formula, *problem*. We added the variables `DPoccs` and `DNoccs`, which also store the number of occurrences of each literal, but now in the working formula (*derived*). If one wants to find out how many times the variable v_1 appears in the working formula, in its positive phase, one types:
  `DPoccs[`v_1`]`
  If, on the other hand, one is more interested in the number of occurrences of the negative phase of variable v_2 in the base formula, one types:
  `DNoccs[`v_2`]`

- `size`: to check the length of a clause in the original RANGER, we had to access the respective clause and then, through its structure, get its length. Note that the structure of a clause is:

  ```
  struct clause {
    int length;
    int *literal;
  };
  ```

  To simplify the method in which the size of clauses is accessed, we added a new variable, `size`, which is a simple array with the size of each clause, and with length equal to the number of clauses in the working formula. If one wants to find out the size of clause c, one will type:
  `size[`c`]`

- `upsize`: in the unit propagation look-ahead methods there is a need to know the size of clauses where a propagated literal has been computed (the propagated literal will either satisfy the clause, in which case its size will be zero, or decrease its length by one). This size is only temporary, and used only during the unit propagation look-ahead methods. Therefore, we added a new variable, `upsize`, which is the same as the variable `size` at the beginning but that will be modified according to the unit propagation.

- `upassigned`: indicates whether or not a variable is assigned. The original RANGER algorithm does not consider assignments, but they are needed for the unit propagation and look-ahead methods. The length of `upassigned` is equal to the number of variables in the working formula.

- `touched`: indicates the assignment of each variable for each combination of variable assignments during the look-ahead phase. We assign a pair of variables for the look-ahead, which means we have four combinations of assignments and therefore this array has four positions. Furthermore, it has the assignment for all variables in the working formula for each combination, and thus its structure is as follows:
  `touched[4][V]`
  where `V` is the number of variables in *derived*. The four combinations are ordered in the following way, considering the two variables a and b:

  - Combination 1 $\rightarrow a = 0 \wedge b = 0$;
  - Combination 2 $\rightarrow a = 0 \wedge b = 1$;
  - Combination 3 $\rightarrow a = 1 \wedge b = 0$;
  - Combination 4 $\rightarrow a = 1 \wedge b = 1$;

  Thus, if we want to find out the value assigned to the variable v (if any) during the look-ahead phase, in combination 2, we would do:
  `touched[1,`v`]`
  If we wanted to find out, after running the look-ahead method which yielded zero conflicts, the intersections of the variable v, we would do:

```
if(touched[0][v] == touched[1][v] == touched[2][v] == touched[3][v])
  to_propagate = TRUE;
else
  to_propagate = FALSE;
```

  Remember that, if the value assigned to a certain variable is the same through all combinations, then that variable is to be assigned to that value.

- `upla_count`: indicates the number of variables propagated through the unit propagation look-ahead methods. It is a statistical measure that may be used to assert the usefulness and efficiency of the algorithm, i.e., how many variables were removed from the formula during the search.

7.2.5 New Functions

In this section we will give an overview of some of the methods added to ranger_release to implement the unit propagation look-ahead, as well as some auxiliary functions that manipulate the addition and removal of clauses from the formula in a different way than done before. These new auxiliary functions are now used within functions of the original ranger_release, for ease of use and compatibility with the new data structures and global variables.

As a brief example, in the original RANGER one could not add clauses to the formula unless they substituted some other clause that was already there, with resolution or subsumption, for example. We altered this procedure such that, now, a clause can be added to the formula by expanding its search space. Note that a clause will be added only if it cannot be added as a substitute of an existing clause, because the algorithm can be deteriorated by increasing the number of clauses, and therefore the search space.

- `add_clause`: adds a given clause to the end of the working formula (*derived*). In the process, it extends all global variables that are directly related to the number of clauses in *derived*, such as `LocationP/LocationN`, `derived`, `size/upsize` and of course the number of clauses in the working formula, `Space`. This function is used every time a clause is to be added to the working formula, and not just replacing an existing clause, as it has been done until now;

- `remove_clause`: removes a given clause from either the working formula or the base formula, depending on the arguments given. The last clause in the formula will take its place, and the length of the array variables that depend on the number of clauses is decreased. The global variables modified are the same as for the function `add_clause`: `LocationP/LocationN`, `derived`, `size/upsize` and the number of clauses in the working formula, `Space`, or the base formula, `C`. This function is also used every time a clause is removed from the formula, both in the new functions and the functions already in use in the original RANGER;

- `unit_propagation`: this function will be called in every satisfiability-transformation step of the algorithm. It performs unit propagation in the working formula of the algorithm in the following way: it searches the formula for clauses of length one; if it finds any, it will propagate the literal within that clause throughout the whole working formula, with the function `propagate_lit`. Every time a unit clause is found, its free literal is stored and after having searched the formula, all free literals stored are propagated. This function can assert the unsatisfiability of the formula by itself;

- `propagate_lit`: this function is called during the unit propagation method, with a literal

as argument, and will remove every clause where that literals appears in its positive phase and removes said literal from every clause where it appears in its negative phase. This is the implementation of the unit clause rule discussed above (refer to chapter 2.3.1 - The Unit Clause Rule). The literal is removed both from the working formula and the base formula. If a conflict occurs, i.e., if by removing a literal from a clause that clause becomes empty, then the formula is unsatisfiable and the function returns UNSAT;

- `upla`: this function will implement the unit propagation look-ahead algorithm. It is called only once during the call to ranger_release, in the first satisfiability-transformation phase. It will iteratively assign truth values to pairs of literals in the working formula and then search for conflicts for those assignments. To search for conflicts it will use the function `find_conflicts`. If four conflicts are found for a certain assignment to a pair of variables, the function returns UNSAT and the formula is unsatisfiable. It will then, in every iteration of assignments, make a call to the function `analyse_upla` to analyse the number of conflicts and act accordingly;

- `find_conflicts`: receives two assigned variables as arguments and performs the look-ahead method on the working formula. It works in the same way as the function `unit_propagation` discussed above, but starts with two already assigned variables. It will look for unit clauses and will propagate its free literals, assigning values to those variables which will be stored in the global variable `touched` to be processed later by `analyse_upla`. Each time it encounters a conflict it will return to `upla`, indicating a conflict was found;

- `analyse_upla`: this function receives the number of conflicts found in a given iteration of `upla` and two assigned variables as arguments and performs the look-ahead technique for the five possible scenarios for the unit propagation phase, described above. It adds clauses according to the number of conflicts and considers the intersections as well. The clauses are added to the working formula.

7.3 RANGER with Extended Resolution

According to the authors of GUNSAT [4], extended resolution is a method which gives very good results in practice when added to their solver. When the algorithm has tried to increase the heuristic score of a given pair of literals too many times without any success, extended resolution is used to artificially increase the score of this pair of literals. Using the rule:

$$e \Leftrightarrow l_1 \vee l_2$$

encoded by the three clauses:

$$w_1 = (\neg e \vee l_1 \vee l_2)$$
$$w_2 = (e \vee \neg l_1)$$
$$w_3 = (e \vee \neg l_2)$$

we may add a fresh variable e to the formula by adding all of those three clauses. Refer to chapter 5.2 - GUNSAT for a more detailed description of the algorithm.

7.3.1 Implementation of the Extended Resolution

The way extended resolution is used in GUNSAT is intrinsically related to the algorithm itself, built to take advantage of its scoring system. Note that extended resolution is only used when the algorithm has tried to increase the score of a pair of literals too many times without success. It is very different from the way RANGER operates, where no scoring system for literals is used. Thus, we have to add GUNSAT's scoring system to RANGER if we want to implement extended resolution in the same way it was successfully used in GUNSAT.

But there is a problem with this approach. GUNSAT's scoring system for pairs of literals is part of the main heuristic of the solver. It was developed to be the backbone of the algorithm and to be a highly refined heuristic of scoring and choosing the best literals and clauses to resolve. Methods like extended resolution, unit propagation look-ahead or binary clause saturation are only meant to improve this heuristic.

In the previous section, we described the way in which unit propagation look-ahead, a method also used in GUNSAT and which proved to yield successful results, was added to the original ranger_release. But this method was not intrinsically linked to the base of GUNSAT, like extended resolution is. One could simply add it as a pre-processing technique, or run it as a satisfiability-transformation method to RANGER without loss of identity.

Even though adding extended resolution to RANGER did not seem to be neither more efficient nor an easy task, nor did it promise to integrate well with the already implemented algorithm and methods, we tried to make it work with RANGER and to improve on its results for unsatisfiable instances.

As has been said before, RANGER does not have a scoring scheme for literals like GUNSAT does, so that extended resolution could be applied directly to that scheme and be integrated seamlessly with the algorithm. Instead, we chose to adapt the scoring method of GUNSAT to RANGER and thus apply the extended resolution in the same way GUNSAT does.

Any instance of extended resolution is executed only during the satisfiability-preserving transformations phase of the RANGER algorithm, in the same way as unit propagation look-ahead, and for the same reasons. We did not want to alter RANGER's base concept and program flow. Furthermore, besides the probability P_t of this phase of the algorithm, we inserted another probability P_{er}, and the steps of extended resolution will only be executed according to this probability. We had some trouble finding an appropriate number for this probability, mostly because, in their paper about GUNSAT [4], its authors say that extended resolution will be used after the algorithm has tried to increase the score of a given pair of literals "too many times". Definitely not very specific about how much "too many times" is. We settled at $P_{er} = 5\%$, which means it will be executed in about 4.5% of the iterations (calculated by multiplying the 90% chance that the satisfiability-preserving transformations phase will be run and the 5% chance that extended resolution will be executed within that phase).

At the start of each phase of extended resolution, our algorithm will compute the scores for all the pairs of literals in the working formula, *derived*, in the same way that GUNSAT calculates its scores: each pair of literals (l_1, l_2) appearing in the formula is credited a score computed by adding the score of each literal in each clause c_i according to the following formula: $\frac{2^{n-1-n_1}}{n_i.(n_i-1)}$. We compute the score of each variable in each clause and add them for each pair of literals. The score of a clause is then computed by summing the scores of all the pairs of literals it contains. Finally, we calculate the score of each quadruplet by adding the sum of the squares of the scores of its pairs of literals.

After the scores have been computed, we continue to follow GUNSAT's heuristic to improve the score of a pair of literals (remember that extended resolution will only be applied if we cannot improve the score of a pair of literals after too many times). The best scored quadruplet in the formula is computed and found, containing the literals l_1 and l_2. We then try to find a new clause with both l_1 and l_2 by searching the working formula for two clauses: one containing l_1 and a pivot literal p, and the other containing l_2 and the complement of the pivot literal, $\neg p$, such that the resolution rule can be performed and the needed clause with l_1 and l_2 is generated. We only try to generate the new clause c from the two clauses having the lowest scores. Because of this restriction, it is not always possible to generate a new clause according to the specified conditions (the cause could simply be that the new clause may be subsumed by an already existing clause, or it may be a tautology). If one pair score of the highest scored quadruplet cannot be improved, the other scores of the same quadruplet are iteratively tried. If no pairs of literals in this quadruplet can be improved, the second best scored quadruplet is tried and so on.

Finally, the algorithm will check, for all pairs of literals, whether their score have been increased too many times without any success or not. We set this value at 20, i.e., we consider the score of

a pair of literals to be increased too many times when that value reaches 20. If it does, and for literals l_1 and l_2, three new clauses will be added along with a new variable, e. The clauses added are:

$$w_1 = (\neg e \vee l_1 \vee l_2)$$
$$w_2 = (e \vee \neg l_1)$$
$$w_3 = (e \vee \neg l_2)$$

This is the extended resolution rule added to the RANGER algorithm.

As we can see, there is a lot of overhead work needed to implement this method in RANGER, especially if we apply it in the same way GUNSAT does. Computing the scores and looking for the new clause is very demanding for the algorithm, and we are adding an additional layer of complexity to RANGER, whereas in GUNSAT this computation was already part of the algorithm itself and was not added to it so that extended resolution would work, like it is done in RANGER.

In the following sections we will present and describe the data structures and functions that were used to implement the extended resolution in ranger_release.

7.3.2 New Parameters

For the implementation of extended resolution, two new parameters were added to the program:

- `Per`: represents the probability that the steps of extended resolution will be executed, and is set to 5%. As has been said above, extended resolution is ran as a satisfiability-preserving transformation and is subject to the transformation probability, P_t, which is defined as 90%, and therefore the true probability that extended resolution may be ran is 4.5%;

- `ER_INCREASES`: represents the number of times that the score of a pair of literals tried to be increased without any success. This constant is currently set at 20 tries. It is used to apply extended resolution, i.e., add a new variable and three new clauses to the formula when the score of a pair of literals cannot be increased by normal means.

7.3.3 New Data Structures

With the implementation of extended resolution into the program, it was necessary to add new data structures to accommodate the new algorithm. To cope with the new concept of scores needed by extended resolution, four new global variables were added to ranger_release:

- `scoreL`: represents the score of each individual pair of literals in the formula. It is implemented as a matrix of the type $float$, because the scores are decimal values, which further increase the toll that this method and its variables have on the algorithm. It is used as the first step in the computation of the scores of each quadruplet in the formula;

- `scoreC`: represents the score of each clause in the working formula. Again, this variable is represented as a $float$, and has length equal to the number of clauses in *derived*. It is used in the same context as the variable above: during the computation of the score of each quadruplet;

- `scoreQ`: represents the score of each quadruplet, and its type is, again, $float$. It is used during the phase where the algorithm is trying to find a clause which results of the resolution of two other clauses that have literals l_1 and l_2, the same literals from the highest scored quadruplet (stored in this `scoreQ`). The values of the scores are stored, and not only calculated on the go, because the first pair of clauses tried to be resolved may not be valid, and as such a lower scored quadruplet needs to be used. If the score of the quadruplets is already computed, the algorithm needs only to access the array in the next position instead of having to recalculate;

- `scoreI`: this variable stores the number of times the score of a pair of literals tried to be increased without success. It has the same number of positions as the variable `scoreL`, but its type is *int* instead of $float$, which cuts in half the space it takes. It is used as a measure of the difficulty that the algorithm has in increasing the score of a particular pair of literals. It is directly related to the constant `ER_INCREASES` which is compared to the values of this variable when it must be decided whether or not the algorithm has tried to increase the score of a pair of literals too many times. Note that we set the value of `ER_INCREASES` to 20 times.

7.3.4 New Functions

In this section we will give an overview of some of the methods added to ranger_release to implement the extended resolution. There are three main functions which comprise the bulk of what was done to implement this method, and which are called inside the satisfiability-preserving transformations: `compute_scores`, `ER_add_one_clause` and `ER_add_one_variable`. The control flow of this method in the main part of the algorithm is depicted below:

```
if(rand()%100<Per) {
  compute_scores();
  ER_add_one_clause();
```

```
  ER_add_one_variable();
}
```

These functions will be described in the following lines:

- `compute_scores`: the extended resolution method in RANGER starts with this function, which begins by computing the score of each pair of literals in the formula (and storing them in the variable `scoreL`); it then calculates the score for each clause in the formula, simply by adding the score of each pair of literals in each clause, and saving the results in the variable `scoreC`; and finishes by computing the score of each quadruplet, again with the support of the variable `scoreL`, by summing the squares of the scores of its pairs. This value is then saved in the variable `scoreQ`. This function is executed every time extended resolution is ran, and computes fresh scores each time;

- `ER_add_one_clause`: after all scores are calculated, this function is called. It will find the best scored quadruplet, i.e., the quadruplet with the highest score according to the values in `scoreQ` and will randomly choose a pair within that quadruplet. Let us consider that in such pair, literals l_1 and l_2 are found. The following step is to compare all the clauses in which l_1 appears with those in which l_2 appears, and assert whether, for each pair of clauses, they have one literal p in one and $\neg p$ in the other. Let us call this literal p a complementary literal. The auxiliary function that does such computation is `has_complementar_lit`. For each pair of clauses that meets this requirement, a score is calculated which is the sum of the scores of both clauses. The final step of this function is to find out which pair of clauses had the lowest score and resolve the two clauses. Function `resolve` checks if the new clause is eligible to be added to the working formula. If that is the case, `ER_add_one_clause` terminates by adding the new clause; if not, the other pairs of the same quadruplet are iteratively tried. If the algorithm fails on all pairs of the best quadruplet, then the second best quadruplet is tried, and so on;

- `resolve`: this function is used by `ER_add_one_clause` to check whether or not a pair of variables is eligible to be resolved and to generate other variable. It checks if the new clause is a tautology, if it has repeated literals, if the clause already exists or if it is subsumed by an existing clause. If any of the above is true, then the the new clause is not eligible to be added to the formula;

- `has_complementar_lit`: this function is called to assert whether two clauses, where literal l_1 appears in one and literal l_2 in the other, have a common literal p which is on its positive phase in one clause and on its negative phase in the other; in other words, a complementary literal; the function is used in the `ER_add_one_clause` function;

- `ER_add_one_variable`: the last step in the execution of extended resolution within RANGER is this function, which adds three new clauses and a new variable to the formula if a score in `scoreI` is greater than `ER_INCREASES`, i.e., the score of a given pair of literals tried to be increased too many times (in this case, 20) without success.

8

Experimental Evaluation

In this chapter we will present the experimental results achieved with the implementation of the different methods used to improve RANGER. We will then draw some conclusions regarding the usability of the methods proposed.

8.1 Experimental Setup

Our results were obtained in an Intel Xeon 5160 server (3.0GHz, 1333Mhz, 4GB) running Red Hat Enterprise Linux WS 4. Each problem instance was run with 10 seeds and for a timeout of 1000 seconds.

8.2 Experimental Results

In the following sections we will present the experimental results achieved with our modifications of the original ranger_release algorithm in its different stages: results for the original RANGER, results for RANGER with unit propagation look-ahead and results for RANGER with unit propagation look-ahead and extended resolution. We will compare them with each other and with the solver GUNSAT, the other SLS solver that proves unsatisfiability instead of satisfiability. Finally, we will draw conclusions about the results achieved by each of the implementations and we will try to explain the results.

The set of problem instances used included the following types of instances:

- `uuf50`: this benchmark consists of uniformed random-3-SAT instances. Uniform Random-3-SAT is a family of SAT problems distributions obtained by randomly generating 3-CNF formulae in the following way: for an instance with n variables and k clauses, each of the k clauses is constructed from three literals which are randomly drawn from the $2n$ possible literals (the n variables and their negations) such that each possible literal is selected with the same probability of $\frac{1}{2}n$. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological (i.e., they contain a literal and its negation). Each choice of n and k thus induces a distribution of Random-3-SAT instances. Uniform Random-3-SAT is the union of these distributions over all n and k [51]. The instances were generated in an unforced way and were separated into test-sets of satisfiable and unsatisfiable instances. We will consider the test-set of unsatisfiable (uuf) instances with 50 variables and 218 clauses, and thus dubbed the test-set `uuf50-218`. We will use a benchmark of 100 of these instances, downloaded from www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

- `aim`: the aim instances are all generated with a particular Random-3-SAT instance generator [3]. The particularity is that it generates yes-instances and no-instances independently for wide ranges. So its primary role is to provide instances such that the conventional random generation can hardly generate. The generator runs in a randomized fashion, so that it is essentially different from those generated in some deterministic fashion or those translated from other problems. As a result, the benchmark set of instances includes:

  - no-instances at low clause/variable ratios, and
  - yes-instances at low and high clause/variable ratios that however have only one solution.

  We utilized four sets of unsatisfiable aim problem instances: each set has either 50, 100 or 200 variables, and within each set we have two groups: four instances where the ratio clause/variable is 1.6, and another group of four instances with a 2 ratio.

To illustrate the behaviour of RANGER in the above mentioned problem instances, we compiled a set of experimental results in the following tables. We utilized the number of seconds each instance takes to be solved by a given tool as a measurement (CPU time) as well as the number of iterations it takes to solve the problem. To better illustrate the usefulness of each component added to RANGER, each problem instance was run with different versions of RANGER: the original RANGER code as a basis for comparisons; the modified RANGER without unit propagation look-ahead or extended resolution; RANGER with unit propagation look-ahead; and RANGER with both unit propagation look-ahead and extended resolution. Results for GUNSAT were also collected.

	Characteristics of each problem set		
Instance	#instances	#vars	#cls
$aim-50-no-1\_6$	4	50	80
$aim-50-no-2\_0$	4	50	100
$aim-100-no-1\_6$	4	100	160
$aim-100-no-2\_0$	4	100	200
$aim-200-no-1\_6$	4	200	320
$aim-200-no-2\_0$	4	200	400
uuf_50	100	50	218

Table 8.1: Characteristics of each set of problem instances

	original RANGER			
Instance	#instances	avg_time (sec)	iterations	% Solved
$aim-50-no-1\_6$	4	0.015	39223	100
$aim-50-no-2\_0$	4	250.026	67105	75
$aim-100-no-1\_6$	4	0.834	433398	100
$aim-100-no-2\_0$	4	11.35	3266805	100
$aim-200-no-1\_6$	4	259.227	5902875	75
$aim-200-no-2\_0$	4	512.156	9802581	55
uuf_50	100	845.6	174752209	20

Table 8.2: Experimental results for the Original RANGER

Table 8.1 illustrates the characteristics of each set of problem instances. We did not use the test-set `uuf75` (the next version of `uuf50`, with 75 variables and 325 clauses) because the original RANGER could not solve any of the instances we tried (10 instances of this set ran 10 times for the same number of different seeds).

Table 8.2 through 8.6 illustrate the results of the various phases of RANGER as well as GUNSAT's in terms of time taken, iterations and the percentage of solved instances for each problem set. The time column is calculated by computing the median of the time taken when the solver is run with each seed, for each instance, and then calculating the mean of each instance's median. It is the mean of each median value. Let it be noted that, when looking at the time column, one must keep in mind the column of the percentage of solved instances so as to better understand the time result presented. As an example, let us consider table 8.2: in the time column we have that the problem set `aim_50_no_2_0` takes an average time of 250.026 seconds to solve. Looking at the previous and following rows in the same table, this result seems incongruent, but now let's look at the last column: this instance set was only solved 75% of the times, whereas the others considered are always solved (100%). In this case, one of the instances was not solved for any of the seeds and the default value of 1000 (the maximum number of seconds that any instance was allowed to run) was considered when the mean was computed, thus raising the value of the time taken. The number of iterations presented are taken only from successful runs.

	original RANGER modified without UPLA/ER			
Instance	#instances	avg_time (sec)	iterations	% Solved
$aim-50-no-1\_6$	4	0.126	24076	100
$aim-50-no-2\_0$	4	250.271	74452	75
$aim-100-no-1\_6$	4	5.75	414130	100
$aim-100-no-2\_0$	4	11.03	3717396	100
$aim-200-no-1\_6$	4	285	5320123	77.5
$aim-200-no-2\_0$	4	567.01	9420952	52.5

Table 8.3: Experimental results for the Original RANGER modified without UPLA or ER

	RANGER+UPLA			
Instance	#instances	avg_time (sec)	iterations	% Solved
$aim-50-no-1\_6$	4	0.096	4201	100
$aim-50-no-2\_0$	4	0.114	6240	100
$aim-100-no-1\_6$	4	0.91	134572	100
$aim-100-no-2\_0$	4	1.74	160971	100
$aim-200-no-1\_6$	4	24.56	1390580	77.5
$aim-200-no-2\_0$	4	238.7	3066713	62.5
uuf_50	100	521.6	43620471	39

Table 8.4: Experimental results for RANGER with unit propagation look-ahead

As we can see from table 8.2, the original RANGER solves problem instances with a low ratio of variables/clauses (`aim`) more easily than instances with a higher ratio (`uuf`). The ratio of variables/clauses in `aim` is 1.6 and 2.0, whereas the ratio in `uuf50` is 4.36. This corresponds to the phase transition of each test-set of instances (refer to section 2.5.3 - Phase Transitions for more information).

Table 8.3 illustrates the results of the modified RANGER without unit propagation look-ahead or extended resolution. The modifications made to accommodate these methods turned RANGER into a slightly slower algorithm, mostly due to the fact that the location of every literal in a clause must now be stored, instead of the lighter version of the original RANGER where only two clauses were randomly stored for each literal in the formula. We did not test this version of RANGER with the problem set `uuf50` because we only want to illustrate that this algorithm is slower, and the instances `aim` alone serve our purposes.

Results for RANGER with unit propagation look-ahead are presented in table 8.4. It can achieve far better results than the original RANGER does. For very small and easy instances (`aim-50`), RANGER+UPLA generates slightly worse results due to the initial overhead of the unit propagation look-ahead, but its results in harder instances compensate for that overhead and it shines over the original algorithm. Gains in terms of time taken to solve the instance can rise up to 85% for `aim-100-no-2_0` whereas for `aim-200-no-2_0` it rises by 54%.

	RANGER+UPLA+ER			
Instance	#instances	avg_time (sec)	iterations	% Solved
$aim-50-no-1\_6$	4	0.281	4958	100
$aim-50-no-2\_0$	4	0.384	7026	100
$aim-100-no-1\_6$	4	13.57	120122	100
$aim-100-no-2\_0$	4	24.09	174131	100
$aim-200-no-1\_6$	4	317.1	1788612	55
$aim-200-no-2\_0$	4	818.2	3762236	37.5
uuf_50	100	740.1	51390176	5.7

Table 8.5: Experimental results for RANGER with unit propagation look-ahead and extended resolution

	GUNSAT		
Instance	#instances	time (sec)	% Solved
$aim-50-no-1\_6$	4	1.21	100
$aim-50-no-2\_0$	4	1.41	100
$aim-100-no-1\_6$	4	36.96	97.5
$aim-100-no-2\_0$	4	3.27	100
$aim-200-no-1\_6$	4	16.72	80
$aim-200-no-2\_0$	4	379.8	80
uuf_50	100	67.39	59.3

Table 8.6: Experimental results for GUNSAT

But it is in the number of iterations that RANGER with unit propagation look-ahead far outshines the original RANGER. For all the `aim` instances, the gain is always over 69%, reaching 89%, 91% and 95% in some sets of problems. The reason why the gain in time taken to solve each instance is not the same as the gain in the number of iterations is the unit propagation that takes place in each iteration of the algorithm where there is a satisfiability-preserving transformation (95% of the iterations). This method and the pre-processing look-ahead are responsible for the sheer decrease in number of iterations needed to solve the algorithm. We believe that some tweaking and a more efficient implementation of the unit propagation look-ahead will provide better times, although the number of iterations will probably remain the same.

Table 8.5 illustrates the results when extended resolution is added to RANGER with unit propagation look-ahead: we verify a degradation in the performance of the algorithm. The number of iterations needed to solve a given problem instance remains largely the same, but the time used by each iteration increases substantially. Even though extended resolution is only executed in 4.5% of the iterations, it is still responsible for the longer time it takes the algorithm to solve the same instance, when compared to RANGER with only unit propagation look-ahead. Calculating scores adds an unnecessary overhead to the algorithm. If the iterations remain the same but the time taken increases, we can conclude that extended resolution does not add significant benefits to RANGER.

	Time (sec)				
Instance	original	modified	UPLA	UPLA+ER	GUNSAT
$aim-50-no-1\_6$	0.015	0.126	0.096	0.281	1.21
$aim-50-no-2\_0$	250.026	250.271	0.114	0.384	1.41
$aim-100-no-1\_6$	0.834	5.75	0.91	13.57	36.96
$aim-100-no-2\_0$	11.35	11.03	1.74	24.09	3.27
$aim-200-no-1\_6$	259.227	285	24.56	317.1	16.72
$aim-200-no-2\_0$	512.156	567.01	238.7	818.2	379.8
uuf_50	845.6	-	521.6	740.1	67.39

Table 8.7: Comparison of time results for the solvers tested

	Iterations			
Instance	original	modified	UPLA	UPLA+ER
$aim-50-no-1\_6$	39223	24076	4201	4958
$aim-50-no-2\_0$	67105	74452	6240	7026
$aim-100-no-1\_6$	433398	414130	134572	120122
$aim-100-no-2\_0$	3266805	3717396	160971	174131
$aim-200-no-1\_6$	5902875	5320123	1390580	1788612
$aim-200-no-2\_0$	9802581	9420952	3066713	3762236
uuf_50	174752209	-	43620471	51390176

Table 8.8: Comparison of the number of iterations for the solvers tested

Finally, we present results for GUNSAT over the same problem instances and in the same conditions as offered to RANGER. We only had access to a pre-compiled version of GUNSAT and not to its source code, so we could not modify the output to produce the number of iterations needed to solve a given problem instance, and as such that column is not present in table 8.6, which presents GUNSAT's results.

GUNSAT is the solver that takes longer to solve the easier instances, and it produces unexpected results on `aim-100-no-1_6`: it takes much longer to solve this problem than its version with a higher ratio of variables/clauses. It beats the original RANGER in the instances `aim-100-no-2_0`, `aim-200-no-1_6` and `aim-200-no-2_0`, and gives a far better result in the set of instances `uuf50`, where it has a gain of 92% in terms of time taken. It would be interesting to have the number of iterations it takes GUNSAT to solve a problem, because it would give us a better understanding of the underlying differences of both algorithms.

GUNSAT may have had some memory problems, and increasing the amount of memory it was allowed to use might improve its results. The version of the Java platform used is jre1.6_0_07.

Tables 8.7 and 8.8 show the results of, respectively, the time taken to solve each instance and the number of iterations needed by each solver.

From these tables, we conclude that the best solver to solve easier instances with few variables and a low ratio of variables/clauses is the original RANGER due to its simplicity. In all the other

instances of `aim`, RANGER with unit propagation look-ahead beats the other two solvers in both time taken and iterations (for the number of iterations, this refers only to the solvers that provide such information). For the `uuf50` set of instances, GUNSAT is far superior to RANGER in any of its forms.

8.3 Conclusions

This chapter evaluated the usefulness of applying methods native to GUNSAT to the other SLS solver RANGER. We tested both the original RANGER and GUNSAT in a number of unsatisfiable instance sets, in which GUNSAT proved to be faster in the harder instances by systematically beating RANGER in every such test. This is mostly due to the fact that GUNSAT has a more powerful reasoning mechanism and a finer heuristic to guide moves, whereas RANGER is simpler. Because of the use of powerful high reasoning mechanisms like unit propagation look-ahead and extended resolution, and a finer heuristic, GUNSAT's moves are slower but more intelligent and pondered, more than making up for RANGER's rather faster but blinder moves.

We then tested RANGER's behaviour with both unit propagation look-ahead and extended resolution implemented, and found that RANGER with unit propagation look-ahead is indeed faster than the original RANGER, due to its pre-processing look-ahead phase as well as to the rather light unit propagation that runs in almost every iteration of the algorithm.

Not surprisingly, extended resolution proved to be too heavy for RANGER's rather simplistic approach, and did not produce good results. The strength of RANGER lies in being simple enough to perform many moves per second and that makes up for the rather simple and unintelligent but fast heuristics used. On the other hand, extended resolution, while being a simple technique to be applied, is used in GUNSAT to improve its scoring system, which is a heavy and highly tweaked for performance feature of the algorithm. By implementing part of this scoring system in RANGER with the only goal of adding extended resolution to the algorithm, we actually went against RANGER's principles of simplicity, and thus the results achieved suggest not to use extended resolution when RANGER is concerned.

We can, thus, conclude that adding simple and fast paced techniques to RANGER is a viable option when trying to improve its base algorithm. These methods must not rely on scoring systems or depend too much on certain conditions to be met: they must be independent and simple. This is based on the fact that they will be added to RANGER on its satisfiability-preserving transformation phase, and will not alter the base algorithm significantly.

Unit propagation look-ahead can be included in this family of simple methods.

Extended resolution cannot.

It is also interesting to note that RANGER with unit propagation look-ahead will prove that some of the `uf20`[1] (satisfiable instances with 20 variables and 91 clauses) instances are satisfiable (i.e., the solver will return HALTED before its timeout condition is met, and thus concluding the instance is satisfiable), and some of the times it will do so with only the look-ahead pre-processing step. This is in fact an interesting result because it can open a window for RANGER to prove unsatisfiability as well as satisfiability.

[1] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

9

Conclusions and Future Work

Propositional Satisfiability is fundamental in solving many application problems in artificial intelligence and also in other fields of computer science and engineering. Recent years have have seen the proposal of several effective approaches for solving SAT instances. In particular, it is important to know whether a problem instance is satisfiable or not. In this thesis, we have presented new methods which draw on the strengths of two state of the art SLS solvers which prove unsatisfiability instead of satisfiability. Our approach was to use the RANGER solver as a base and extend it with the most effective methods used by the GUNSAT solver, and thus produce a new solver that is more than the sum of RANGER and GUNSAT.

To preserve RANGER's simplicity we did not alter its algorithmic flow and instead added the new techniques to a phase of the algorithm that already supported the so called satisfiability-preserving transformations. These new methods, which were used by GUNSAT to achieve very good results and improvements, are unit propagation look-ahead and extended resolution. Their implementation in RANGER produced interesting results.

We started by modifying the original RANGER's data structures and some functions to better prepare the solver for the implementation of the new techniques. This turned RANGER into a slightly slower algorithm, mostly due to the fact that the location of all the literals need to be stored, instead of only two randomly clauses where each literal appears. In the original RANGER there was no need to know extensively where each literal is contained, but it is a pre-condition for unit propagation look-ahead and extended resolution.

Next, we implemented unit propagation look-ahead into RANGER. It was divided into two parts: a pre-processing part which performed unit propagation look-ahead with every pair of variables, and

another part with only unit propagation, which ran in almost every iteration of the algorithm (it ran in 95% of its iterations). We decided to follow this direction because running unit propagation look-ahead with two pairs of variables in almost every iteration was very heavy and degradated the performance of the algorithm without adding any real usefulness: in most of the iterations no look-ahead was needed and only unit literals were found. Thus, a pre-processing phase prepares the search space of the problem instance for the algorithm by performing look-ahead, and unit propagation is then performed in almost every iteration from then on.

Unit propagation look-ahead achieved very good results in practice, beating both the original RANGER and GUNSAT in most of the instances tested (GUNSAT was only better in the test-set of `uuf50` problem instances). We thus recommend adding unit propagation look-ahead to future versions of RANGER, and deem it an interesting addition to future SLS algorithms that delve deeply into proving unsatisfiability.

The second technique did not work quite as well. GUNSAT utilizes extended resolution to improve its scoring system, and according to its authors it is an important part of the solver, which improves its results in practice. To implement such feature in RANGER in the same way that it is used in GUNSAT, we need to add a scoring system to RANGER. This goes against RANGER's policies of simplicity with fast but not so intelligent moves as GUNSAT's, which produces good results in practice. A scoring system similar to GUNSAT's was added to RANGER (we chose a similar scoring system because it is argued by GUNSAT's authors that extended resolution works well with that system, so there is no point in applying another one). As before, and with the purpose of not altering RANGER's flow, extended resolution was added to the satisfiability-preserving transformations phase in the same way as unit propagation.

The scoring system added computes the scores of each pair of literals, each clause in the formula and each quadruplet (every possible combination of two variables). It then tried to improve the formula by adding a low scored clause. If the algorithm tried to improve the score of a pair of literals too many times without any success, extended resolution was applied: a new variable and three new clauses were added to the formula. This worked well in GUNSAT because the algorithm depends on the scoring system to produce good results: it is its main heuristic. In RANGER, this scoring system was only added to implement extended resolution, and thus it can be regarded as a waste of resources, not achieving its true potential like it has been achieved in GUNSAT. RANGER's simplicity principles prevent that.

This step of extended resolution ran only in 4.5% of the iterations, but the overhead of computing and trying to improve scores proved to be too expensive. RANGER with UPLA+ER was consistently slower than RANGER with only UPLA, and it only beat GUNSAT on easier instances.

We can conclude that extended resolution is not a viable option to be added to RANGER in future

versions, or at least not in the same way it is used in GUNSAT.

Nonetheless, we recognize that the developed tool is mostly a prototype, and a re-implementation of the tool should be considered. Moreover, considering the promising results obtained, several interesting topics can be suggested for future work. Apart from implementation details, which can be decisive for reducing the expected computational effort, and therefore decisive for solving an instance in a reasonable amount of time, other aspects should be considered. Looking at extended resolution outside of GUNSAT's use of it could prove to be interesting, as it is a simple technique that could be applied to RANGER's own heuristic. We also expect to integrate into RANGER more simplification techniques, which are used successfully by other SLS solvers, mostly those that do not require high computations and therefore fit into RANGER's simplicity principles.

Extended resolution could be added to RANGER in a simpler manner. It could, for example, be executed after x iterations of the algorithm where no improvement could be done. It would be interesting to test this scenario of adding a new variable and three new clauses to the formula in such a simple manner, which better reflects RANGER's simplicity principles. The algorithm can sometimes have many consecutive non-improving steps, usually because no new clauses can be generated either by replacing or resolving clauses from the original formula into the working formula, nor by the unit propagation look-ahead and satisfiability-preserving transformations. Using extended resolution to add fresh clauses in such scenarios can prove to be a very useful technique, while keeping the methods used simple and fast (only a counter for the number of non-improving steps is used, as well as a simple method to add new clauses to the formula).

Bibliography

[1] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An Analysis of SAT-based Model Checking Techniques in an Industrial Environment. In *Conference on Correct Hardware Design and Verification Methods*, 2005.

[2] Anbulagan, D. N. Pham, J. K. Slaney, and A. Sattar. Old resolution meets modern sls. In *AAAI*, pages 354–359, 2005.

[3] Y. Asahiro, K. Iwama, and E. Miyano. Random generation of test instances with controlled attributes. In *D.S.Johnson and M.A.Trick, editors, Cliques, Coloring, and Satisfiability: The Second DIMACS Implementation Challenge. Vol. 26 of DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 377–394. American Mathematical Society, 1996.

[4] G. Audemard and L. Simon. Gunsat: A greedy local search algorithm for unsatisfiability. In *Twentieth International Joint Conference on Artificial Intelligence*, 2007.

[5] F. Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[6] L. Baptista and a. P. M. S. Jo˙ Using randomization and learning to solve hard real-world instances of satisfiability. In *CP '02: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 489–494, London, UK, 2000. Springer-Verlag.

[7] S. Chakradhar, V. Agrawal, M. Bushnell, and T. Truong. Neural net and boolean satisfiability models of logic circuits. *IEEE Design and Test of Computers*, 07(5):54–57, 1990.

[8] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.

[9] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[11] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[12] J. P. A. Deutsch. A Short Cut for certain Combinatorial Problems. In *British Joint Computer Conference*, 1966.

[13] B. Dunham, R. Fridshal, and G. L. Sward. A non-heuristic program for proving elementary logical theorems. In *Proceedings of the International Conference on Information Processing*, pages 282–285, 1959.

[14] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *In proc. SAT05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[15] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In J. Marques-Silva and K. A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.

[16] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[17] J. L. Esteban and J. Torán. Space bounds for resolution. *Information and Computation*, 171(1):84–97, 2001.

[18] H. Fang and W. Ruml. Complete local search for propositional satisfiability. *AAAI-2004*, 2004.

[19] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI '98/IAAI '98*, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[20] J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. *Journal of Automated Reasoning*, 24(1-2):101–125, 2000.

[21] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81(1):127–154, 1996.

[22] H. Hoos. An adaptive noise mechanism for walksat. *Proceedings of the AAAI02, AAAI Press*, 2002.

[23] H. Hoos and T. Stutzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[24] H. H. Hoos and T. Stützle. Local search algorithms for sat: An empirical evaluation. *J. Autom. Reason.*, 24(4):421–481, 2000.

[25] a. P. M. S. Jo and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[26] R. J. B. Jr. and R. Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208, 1997.

[27] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In H. Shrobe and T. Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.

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

[29] D. Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, 2001.

[30] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.

[31] C. M. Li and W. Q. Huang. Diversification and determinism in local search for satisfiability. In *Proceeding of the Eight International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, pages 158–172, 2005.

[32] C. M. Li, W. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *SAT*, pages 121–133, 2007.

[33] C. M. Li, W. Wei, and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. *SAT Competition 2007*, 2007.

[34] I. Lynce and J. P. Marques-Silva. The Puzzling Role of Simplification in Propositional Satisfiability. In *EPIA'01 Workshop on Constraint Satisfaction and Operational Resarch Techniques for Problem Solving*, 2001.

[35] I. Lynce and J. P. M. Silva. Efficient Haplotype Inference with Boolean Satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, 2006.

[36] I. Mironov and L. Zhang. Applications of SAT Solvers to Cryptanalysis of Hash Functions. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 102–115, 2006.

[37] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM.

[38] M. Ouyang. How good are branching rules in DPLL? *Discrete Applied Mathematics*, 89(1-3):281–286, 1998.

[39] D. N. Pham and C. Gretton. gnovelty+. *SAT Competition 2007*, 2007.

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

[41] S. Prestwich and I. Lynce. Local search for unsatisfiability. In *Proceedings of SAT*, pages 283–296, 2006.

[42] S. Prestwich and I. Lynce. Refutation by randomized general resolution. In *AAAI'07*, 2007.

[43] E. T. Richards and B. Richards. Nonsystematic search and no-good learning. *Journal of Automated Reasoning*, 24(4):483–533, 2000.

[44] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[45] B. Selman and H. A. Kautz. Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence(IJCAI-93)*, Chambéry, France, 1993.

[46] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.

[47] B. Selman, H. A. Kautz, and D. A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54, 1997.

[48] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[49] Y. Shang and B. W. Wah. A discrete lagrangian-based global-searchmethod for solving satisfiability problems. *J. of Global Optimization*, 12(1):61–99, 1998.

[50] G. Stalmark. A system for determining propositional logic theorem by applying values and rules to triplets that are generated from a formula. *US Patent 5,276,897; Canadian Patent 2,018,828; European Patent 0403 545; Swedish Patent 467 076*, 1994.

[51] W. M. Taylor, P. Cheeseman, P. Cheeseman, B. Kanefsky, and B. Kanefsky. Where the really hard problems are. In *J. Mylopoulos and R. Reiter (Eds.), Proceedings of 12th International Joint Conference on AI (IJCAI-91),Volume 1*, pages 331–337. Morgan Kauffman, 1991.

[52] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2:466–483, 1970.

[53] J. D. Ullman. *Principles of Database Systems.* W. H. Freeman & Co., 1983.

[54] W. Wei, C. M. Li, and H. Zhang. Deterministic and random selection of variables in local search for SAT. *SAT Competition 2007*, 2007.

[55] H. Zhang. Sato: An efficient propositional prover. In *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, London, UK, 1997. Springer-Verlag.

[56] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96), Fort Lauderdale (Florida USA)*, pages 166–169, 1996.

[57] H. Zhang and M. E. Stickel. Implementing the davis-putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.

[58] L. Zhang. On subsumption removal and on-the-fly cnf simplification. In *SAT*, pages 482–489, 2005.