Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

SAT SOLVING ALGORITHMS
A SURVEY AND COMPARISON

Bachelor's thesis

2012

Martin Šrámek

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

SAT Solving Algorithms
A survey and comparison

Bachelor's thesis

Study program: Informatics
Field of study: 2508 Informatics
Department: Department of Computer Science
Supervisor: RNDr. Tomáš Kulich, PhD.

Bratislava, 2012

Martin Šrámek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Martin Šrámek
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický

Názov: Algortimy na riešenie SAT

Cieľ: Urobiť podrobný prehľad existujúcich algoritmov na riešenie SAT a ich benchmarkov.

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 10.10.2011

Dátum schválenia: 12.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Martin Šrámek
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English

Title: SAT solving algorithms

Aim: Make a thorough survey of existing SAT solving algorithms and existing benchmarks.

Supervisor: RNDr. Tomáš Kulich, PhD.
Department: FMFI.KI - Department of Computer Science

Assigned: 10.10.2011

Approved: 12.10.2011 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgement

I want to thank my supervisor RNDr. Tomáš Kulich, PhD. for his valuable counsel and guidance during the process of writing this thesis.

Abstract

Author:	Martin Šrámek
Title:	SAT Solving Algorithms
University:	Comenius University in Bratislava
Faculty:	Faculty of Mathematics, Physics and Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Tomáš Kulich, PhD.
Number of pages:	69

The boolean formula satisfiability problem (SAT) is interesting not only in theory as the canonical NP-complete problem, but also for its importance in practice: Various problems from different areas of computing are expressed in terms of SAT and then solved using SAT solving algorithms. In this thesis, we present a survey and comparison of such algorithms. We examine the complete algorithms, the incomplete but very fast heuristics and even some special-case algorithms applicable only to formulae in a particular form. We also review the benchmarking methods for SAT algorithms and compare the surveyed algorithms using these benchmarks. Our intention was to create a thorough overview that focuses not only on the algorithms' theoretical principles, but also their practical implementation. We therefore also discuss various implementation details and other practical aspects of SAT solvers, such as the parsing and representation of formulae.

Keywords: SAT, SAT algorithms, SAT benchmarking

Abstrakt

Autor:	Martin Šrámek
Názov práce:	SAT Solving Algorithms (Algoritmy na riešenie SAT)
Univerzita:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci práce:	RNDr. Tomáš Kulich, PhD.
Rozsah:	69 strán

Problém splniteľnosti booleovskej formuly (SAT) je zaujímavý nielen z teoretického hľadiska ako kanonický NP-úplný problém, ale aj pre jeho použitie v praxi: Mnohé problémy z rôznych oblastí informatiky sa vyjadrujú vo forme inštancie SAT a následne sa riešia algoritmami pre SAT. V tejto práci prinášame prehľad týchto algoritmov. Skúmame kompletné algoritmy, nekompletné ale rýchle heuristiky, aj niektoré algoritmy určené pre formuly v špeciálnom tvare. Pozeráme sa aj na metódy benchmarkovania SAT algoritmov a porovnávame skúmané algoritmy pomocou týchto benchmarkov. Naším cieľom bolo zostaviť podrobný prehľad, ktorý sa zameriava nielen na teoretické princípy skúmaných algoritmov, ale aj ich praktickú implementáciu. Preto sa zaoberáme aj rôznymi implementačnými detailami a ďalšími aspektami SAT solverov, ako napríklad parsovanie a reprezentácia booleovských formúl.

Kľúčové slová: SAT, SAT algoritmy, SAT benchmarking

Contents

Introduction

The boolean formula satisfiability problem, commonly abbreviated as "SAT" is the problem of finding an assignment of variables of a propositional (i.e. boolean) formula, such that the formula evaluates to true. It has been shown that this problem is expressive enough to encode the fact whether a given non-deterministic Turing machine accepts a given input. Therefore, any problem in NP can be reduced to SAT. As this reduction can be done in polynomial time, it means that SAT is NP-complete, as was proven in the Cook-Levin theorem in early 1970s [5] [23].

This alone is enough for SAT to be an interesting problem in computer science. However, the reduction of other problems to SAT is not only a matter of theory; SAT solvers are nowadays used in various areas of computing, such as design debugging, AI planning, software testing, bioinformatics [13] or cryptography [19].

In this thesis, we would like to make a survey and comparison of a variety of algorithms that are used to solve SAT. Some of them, the complete ones, search the whole solution space and determine whether there is or is not a satisfying assignment for the given formula. Others, the heuristic ones, are designed to find a solution as quickly as possible, but may never finish if there is not any. Finally, there are special case algorithms that only solve a sub-problem of SAT, but do so much more effectively than the general algorithms. We will explore the different kinds of algorithms and then compare them by benchmarking their functioning on a large set of formulae.

It is an interesting fact that although the heuristic algorithms tend to optimise the solution locally, what does not always lead to the globally optimal solution (in this case any satisfying assignment), the industrial practice shows that this approach often works for SAT instances generated by "real-life" problems. Different heuristic approaches may prove to be effective for different kinds of applications [1] [17]. We will therefore also explore the various types of benchmarks used to evaluate SAT solvers today.

We hope that this work be a comprehensive study of the repertoire of algorithms used and often combined together in modern SAT-solvers. The detailness

of the survey should be thorough enough for it to serve as a textbook for any reader willing to be introduced into the area of SAT solvers. This will be done by both throughly explaining the theoretical concepts of the algorithms and actually measuring the algorithms' effectiveness with a real implementation and on real problems.

Our objective is to make the reader capable of implementing their own SAT solver. This is why our descriptions of the algorithms are not limited to high level sketches, but we also often discuss the details of their implementation. With this objective in mind, we also included a chapter on parsing and representation of boolean formulae. This is important to consider as different algorithms may expect different representations of formulae. Furthermore, parsing the input formula is important in the practice of implementing one's own SAT solver, although this task has been much simplified by the DIMACS formats [3].

For the purpose of benchmarking the algorithms, we have made our own implementations thereof, as well as simple benchmarking tools. We have published the source code at `https://sourceforge.net/projects/satbench/`, from where it is freely available. We encourage the reader to use this code for learning purposes or to devise their own SAT benchmarks.

Chapter 1

Boolean formula

In the first chapter, we are going to introduce the key terms and theorems of the propositional (boolean) logic needed to understand the SAT problem. We will then describe this problem. Before we go on to the next chapters and discuss various algorithms to solve SAT, however, we must discuss their common basis first. Thus, this chapter will also address the representations of a boolean formula and methods of how to parse the formula out of a program's text input. Our formalism and understanding of the propositional logic is based on [21] and [20].

1.1 Definition

Definition 1.1 (Boolean domain)**.** *The boolean domain is the two-valued domain* $\{0, 1\}$*.*

The elements of boolean domain are sometimes named $true$ (1) and $false$ (0). We will use the symbol $\mathbb{B}$ to denote the boolean domain. Note that $\mathbb{B} = \mathbb{Z}_2$, but we will use the symbol $\mathbb{B}$ to emphasize the logical semantic of this set instead the algebraic one.

Definition 1.2 (Boolean variable)**.** *A boolean variable is a variable over the boolean domain.*

We usually use the capital letters of latin alphabet to denote the variables. For example, $A, B, C, \ldots$.

Definition 1.3 (Boolean formula)**.**

1. *Every boolean variable is a boolean formula.*
2. *If ϕ is a boolean formula, then $\neg\phi$ is a boolean formula. If ϕ and ψ are boolean formulae, then $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi)$ are boolean formulae as well.*

3. *Every boolean formula can be constructed by applying rules* 1 *and* 2 *a finite number of times.*

A boolean formula is also referred to as a well-formed formula of the propositional logic. According to this definition, formulae are built recursively from the boolean variables. Therefore, we also refer to the variables as atomic formulae.

For convenience, we usually let out the outermost pair of parentheses, e.g. we write $A \wedge B$ instead of $(A \wedge B)$. It is also notable that no parentheses are used with the negation operator ($\neg$). This is due to the fact that the negation operator, as the only unary operator we use, has naturally higher precedence than the binary operators.

The definition we use may be extended to incorporate other boolean operators. In fact, there are 2^{2^2} mappings from $\mathbb{B}^2$ to $\mathbb{B}$, thus we can use a total of 16 binary operators. Apart from those we used in the definition, the most prominent ones are the exclusive disjunction (also called nonequivalence; XOR), nonconjunction (NAND) and nondisjunction (NOR). We can easily see there are three other unary operators as well - the assertion (unary identity), the constant truth and the constant falsehood. Apparently, none of these is really useful.

1.2 Valuations and satisfiability

Definition 1.4 (Valuation)**.** *A* valuation *is any function v from the domain of boolean formulae to $\mathbb{B}$ such that for any two formulae ϕ and ψ it satisfies the following criteria:*

1. $v(\phi \wedge \psi) = 1$ *iff both* $v(\phi) = 1$ *and* $v(\psi) = 1$
2. $v(\phi \vee \psi) = 1$ *iff* $v(\phi) = 1$ *or* $v(\psi) = 1$
3. $v(\phi \rightarrow \psi) = 1$ *iff* $v(\phi) = 0$ *or* $v(\psi) = 1$
4. $v(\phi \leftrightarrow \psi) = 1$ *iff* $v(\phi) = v(\psi)$
5. $v(\neg\phi) = 1$ *iff* $v(\phi) = 0$

Sometimes we also use the term *variable assignment*. The valuation of a non-atomic formula is a function of the valuations of its components. Therefore, due to the recursive definition of the boolean formula, we can easily see that the valuations of variables (atomic formulae) uniquely determine the valuation of the whole formula. It follows that for a formula containing n distinct variables, there is a total of 2^n possible valuations.

Definition 1.5 (Logical equivalence)**.** *If, for two formulae ϕ and ψ, it holds that $v(\phi) = v(\psi)$ for every valuation v, we say that ϕ and ψ are (logically) equivalent and write $\phi \equiv \psi$.*

Logical equivalence is usually defined as a syntactic property - in other words, the logical equivalence of two formulae ϕ and ψ can be proven by proving $\vdash \phi \leftrightarrow \psi$ in propositional calculus. However, such a definition is provably equivalent to ours.

It can be shown that

$$((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))$$

$$((A \vee B) \vee C) \equiv (A \vee (B \vee C))$$

which means that $\wedge$ and $\vee$ operators are associative. This makes it possible for us to adopt a convention of omitting even the parentheses around chains of conjunctions or disjunctions, as this will not introduce ambiguity to the formula. Furthemore, it can be shown that

$$(A \rightarrow B) \equiv (\neg A \vee B)$$

$$(A \leftrightarrow B) \equiv ((A \wedge B) \vee (\neg A \wedge \neg B))$$

We mentioned above that we could expand the definition of boolean formula to include more operators. This, on the other hand, allows us to actually use *fewer* operators without losing expressivity, as $\rightarrow$ and $\leftrightarrow$ (and, in fact, all the operators) can be defined in terms of $\neg$, $\wedge$ and $\vee$.

Definition 1.6 (Satisfaction)**.** *We say that a valuation v* satisfies *a formula ϕ iff $v(\phi) = 1$.*

Definition 1.7 (Satisfiability)**.** *We say that a formula ϕ is* satisfiable *iff there exists a valuation v such that v satisfies ϕ. If ϕ is not satisfiable, we say it is* unsatisfiable *or* contradictory*.*

Definition 1.8 (Tautology)**.** *If a formula ϕ is satisfied by every possible valuation v, we say that ϕ is a* tautology*.*

In propositional logic, the semantic aspect of tautologiness is matched by the syntactic aspect of logical validness. That means that proving a formula to be tautology is the same as proving that the formula is a theorem of the propositional logic. However, the proof of this equivalence is beyond the scope of this thesis.

Note that satisfiability and tautologiness are related in terms of negation. If ϕ is a tautology, then $\neg\phi$ is unsatisfiable and vice versa. If ϕ is satisfiable but not a tautology, then the same holds for $\neg\phi$. We consider the proof of these statements to be trivial and omit it.

1.3 Normal forms

We mentioned earlier that we will leave out parentheses around chains of associative operators, and thus, for instance, will write $(A \wedge B \wedge C)$ instead of $((A \wedge B) \wedge C)$. In this manner, we may redefine conjunction and disjunction as n-ary operators. It is not difficult to extend the definition of valuation to match these new definitions of $\wedge$ and $\vee$:

- $v(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n) = 1$ iff $v(\phi_i) = 1$ for all $i = 1 \dots n$.
- $v(\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n) = 1$ iff $v(\phi_i) = 1$ for at least one i of $1 \dots n$.

This definition can be proven to be consistent with the old one, i.e. it gives the formula $(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$ the same valuation as the old one did for $(\phi_1 \wedge (\phi_2 \wedge (\cdots \wedge \phi_n)) \dots)$ or any other chain of binary conjunctions (as there are multiple ways to write parentheses around such a chain, but they are, due to the associativity of conjunction, equivalent). The same holds for the disjunction.

We will use the convention of writing $\bigwedge_{i=1}^{n} \phi_i$ for $(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$.

Note that the above definition also allows for unary conjunction and disjunction and it trivially holds that $\bigwedge A \equiv \bigvee A \equiv A$.

Definition 1.9 (Literal)**.** *A* literal *is a boolean variable, or a negation thereof.*

For example, if A, B are variables, than $A, \neg A, B, \neg B$ are literals, but $(A \wedge B), \neg\neg A$ are not literals. We will call literals that consist solely of a variable *positive* and those that contain a negation *negative*.

Definition 1.10 (Clause)**.** *A* clause *is a disjunction of literals.*

For example, $(A \vee \neg B \vee \neg C)$ is a clause but $(A \vee \neg B \vee \neg(C \to D))$ and $(A \vee B) \vee (C \vee D))$ are not.

Definition 1.11 (Conjunctive normal form)**.** *We say that a formula ϕ is in the* conjunctive normal form *(abbreviated* CNF*) if ϕ is a conjunction of clauses.*

E.g. $(A \vee B) \wedge (\neg A \vee C \vee \neg C)$ is a formula in CNF. However, $(A \vee B) \wedge (C \vee (B \wedge \neg C))$ is not in CNF, because $(C \vee (B \wedge \neg C))$ is not a clause.

Definition 1.12 (Disjunctive normal form)**.** *We say that a formula ϕ is in the* disjunctive normal form *(abbreviated as* DNF*), if it can be written as $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$, where $\phi_1, \phi_2, \dots \phi_n$ are conjunctions of literals.*

The formula $(A \wedge B \wedge C) \vee (\neg A \wedge \neg C)$ is an example of a formula in DNF.

It can be seen that CNF and DNF are very similar. The CNF is a conjunction of disjunctions of literals, while the DNF is a disjunction of conjunctions of literals. The term *normal form* itself means that it is a form that every formula has. This idea is further explored in the following theorem. From now on, we will write "ϕ_c is the CNF of ϕ" meaning "ϕ_c is in CNF and $\phi_c \equiv \phi$" to facilitate our expression.

Lemma 1.1 (Distributive laws)**.** *Let $k_1, k_2, \ldots k_n$ be natural numbers. Let $\phi_{1,1}, \phi_{1,2}, \ldots, \phi_{1,k_1}$ up to $\phi_{n,1}, \phi_{n,2}, \ldots, \phi_{n,k_n}$ be sequences of boolean formulae, the i-th of them having k_i members. Then it holds that*

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_i} \phi_{i,j} \equiv \bigvee_{1 \leq i_1 \leq k_1,\ 1 \leq i_2 \leq k_2,\ \ldots,\ 1 \leq i_n \leq k_n} \bigwedge_{j=1}^{n} \phi_{j,i_j}$$

$$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} \phi_{i,j} \equiv \bigwedge_{1 \leq i_1 \leq k_1,\ 1 \leq i_2 \leq k_2,\ \ldots,\ 1 \leq i_n \leq k_n} \bigvee_{j=1}^{n} \phi_{j,i_j}$$

It will be easier to illustrate the distributive laws for $n = 2, k_1 = k_2 = 2$. It then holds that

$$(\phi_{1,1} \vee \phi_{1,2}) \wedge (\phi_{2,1} \vee \phi_{2,2}) \equiv (\phi_{1,1} \vee \phi_{2,1}) \wedge (\phi_{1,1} \vee \phi_{2,2}) \wedge (\phi_{1,2} \vee \phi_{2,1}) \wedge (\phi_{1,2} \vee \phi_{2,2})$$

$$(\phi_{1,1} \wedge \phi_{1,2}) \vee (\phi_{2,1} \wedge \phi_{2,2}) \equiv (\phi_{1,1} \wedge \phi_{2,1}) \vee (\phi_{1,1} \wedge \phi_{2,2}) \vee (\phi_{1,2} \wedge \phi_{2,1}) \vee (\phi_{1,2} \wedge \phi_{2,2})$$

Informally, the distributive laws allow us to swap the conjunction and disjunction if one of them is the 'outer' operator and the other one is 'inner'.

Lemma 1.2 (De Morgan's laws)**.** *Let $\phi_1, \phi_2, \ldots, \phi_n$ be boolean formulae. It then holds that*

$$\neg \bigwedge_{i=1}^{n} \phi_i \equiv \bigvee_{i=1}^{n} \neg \phi_i$$

$$\neg \bigvee_{i=1}^{n} \phi_i \equiv \bigwedge_{i=1}^{n} \neg \phi_i$$

A common verbal formulation of the De Morgan's laws is "conjunction and disjunction interchange under negation". The reader will kindly forgive that we again leave out the proofs of the distributive laws and the De Morgan's laws. The proofs are not difficult, but this thesis does not focus on thorough explanation of the basics of propositional logic.

Theorem 1.1 (CNF and DNF)**.** *For every formula ϕ, there exist formulae ϕ_c and ϕ_d such that ϕ_c is in CNF, ϕ_d is in DNF and $\phi \equiv \phi_c \equiv \phi_d$.*

Proof. Induction on the composition of ϕ. As we have mentioned earlier, $\rightarrow$ and $\leftrightarrow$ can be rewritten in terms of $\neg$, $\wedge$ and $\vee$, so it is sufficient to consider that ϕ only contains the latter three operators. This will allow us to omit the $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ cases of the definition's inductive step.

1. If ϕ is an atomic formula, it is inherently in CNF as it can be written in the form $\bigwedge \bigvee \phi$, i.e. as an unary conjunction of a single-member clause. Similarily, it holds that $\phi \equiv \bigvee \bigwedge \phi$. Therefore, in this case ϕ_c and ϕ_d are identical to ϕ.

2. Let ϕ have the form $\neg\psi$, where ψ has CNF ψ_c and DNF ψ_d. Let us construct ψ'_c by applying the De Morgan's laws on $\neg\psi_c$. Inspecting ψ'_c, we can see that it is in DNF. As $\psi'_c \equiv \neg\psi_c$ (it has been constructed in such a way), apparently $\psi'_c \equiv \neg\psi \equiv \phi$, therefore ψ'_c is the DNF of ϕ. We can construct the CNF of ψ analogously, by applying the De Morgan's laws on $\neg\psi_d$.

3. Let ϕ have the form $\psi \wedge \chi$, where ψ_c and χ_c are the conjunctive normal forms of ψ and χ, respectively. Apparently, $\psi_c \wedge \chi_c$ is the CNF of ψ. Now, if we apply the distributive law to $\psi_c \wedge \chi_c$ we get a formula in DNF equivalent to ϕ, i.e. the DNF of ϕ.

4. The case where ϕ is of the form $\psi \vee \chi$ is analogous to the previous case. We just use the disjunctive normal forms of ψ and χ instead.

□

1.4 The SAT problem

The conjunctive normal form has a special importance in computer science. In early 1970s, Cook [5] and Levin [23] showed that for any given nondeterministic Turing machine and given input we can construct a formula in CNF that describes the fact that the machine accepts the input and processes it, only being satisfiable if the machine halts and accepts the input. This reduction can be done in deterministic polynomial time. This means that for every problem L in $\mathcal{NP}$, represented by its Turing machine A, considering the input size n, we can construct a formula in CNF of size $p(n)$ for some polynom p. In yet other words, any problem in $\mathcal{NP}$ can be polynomially reduced to the problem of finding a satisfying valuation of a formula in CNF. If we consider the fact that validating such a valuation can be done in deterministic polynomial time, we see it is possible to solve this problem in nondeterministic polynomial time, meaning it is in $\mathcal{NP}$. This makes the problem of deciding whether a formula in CNF is satisfiable an $\mathcal{NP}$-complete problem; in fact, the first problem to have been to have this property.

Definition 1.13 (SAT)**.** SAT *is the problem of deciding whether a given boolean formula in conjunctive normal form is satisfiable.*

We often put constraints on the size of a clause in an instance of a SAT problem. For example, the problem of solving SAT for only those formulae in CNF that have 2 variables per clause is called 2-SAT, for 3 variables per clause it is 3-SAT etc.

The constrained conjunctive normal form considered in n-SAT is called n-CNF. Note that unlike CNF, n-CNF for any n is not a normal form, in the sense that it does not exist for every formula. Apparently, the formula $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n \vee \phi_{n+1}$ is already in CNF and there is no way to transform it

to n-CNF. The size of clauses influences the "difficultness" of SAT. While 2-SAT is solvable in linear time, 3-SAT is $\mathcal{NP}$-complete (the proof of the Cook-Levin theorem actually generates an instance of 3-SAT).

Note the interesting fact that as 3-SAT is $\mathcal{NP}$-complete, every problem in $\mathcal{NP}$ - *including SAT* - can be reduced to 3-SAT. This means that while not every formula in CNF can be transformed to an *equivalent* formula in 3-CNF, every formula in CNF can be polynomially transformed to an *equisatisfiable* formula in 3-CNF (where the term *equisatisfiable* means that the given formulae are either both satisfiable or both unsatisfiable).

The SAT problem in its most common formulation is only considered for boolean formulae in the conjunctive normal form. However, in this thesis we will also examine the techniques of solving the satisfiability problem for a formula in the general form. Naturally, one of the possibilities is transforming the formula to the conjunctive normal form, as this exists for every formula. There are, however, other possibilites.

1.5 Parsing and representation

In this section, we will ponder the representations of a boolean formula in the general and the conjunctive normal forms. We will describe how to parse a boolean formula and represent it in computer. First, let us introduce some new terms we will need in the following text.

Definition 1.14 (Subformula)**.** *Considering a boolean formula ϕ as a string over the alphabet of boolean variables, operators and parentheses, its subformulae are all substrings that are formulae themselves.*

For example, $(A \wedge (B \vee \neg C))$ is a formula and its subformulae are: $(A \wedge (B \vee \neg C))$ (i.e. itself), $(B \vee \neg C)$, $\neg C$, A, B, C. One may notice that these subformulae are exactly the steps encountered when one builds the formula formula according to the recursive definition. In particular, note that every subexpression bounded by a matching pair of parentheses is a subformula. Such subexpressions will be of high importance later in our algorithm. We will refer to them by the abbreviation *SBMPP*. For the formula given above, only $(A \wedge (B \vee \neg C))$ and $(B \vee \neg C)$ are SBMPPs.

Definition 1.15 (Depth, nesting and level)**.**

- *Given a SBMPP ϕ and a SBMPP ψ, if ψ is a substring of ϕ, we say ϕ is* nested *in ϕ.*
- *The* depth *of ϕ is the number of the longest chain of SBMPPs starting with ϕ such that every SBMPP in the chain is nested inside the previous one.*

- *The* level *of ψ (with reference to ϕ) is the depth of ϕ minus the depth of ψ plus one.*

Let $\phi \equiv (A \wedge (B \vee \neg C))$ and $\psi \equiv (B \vee \neg C)$. Then ψ is nested in ϕ. The depth of ϕ is 2. With reference to ϕ, ψ is on level 2 and ϕ itself is on level 1.

Now, let's consider a boolean formula in the general form. We propose the following parsing algorithm, consisting of two steps: checking the corectness of input and constructing a parse tree.

1.5.1 Checking the corectness of input

First, we will scan the input and analyse its corectness. Of course, it would be possible to immediately start parsing recursively as any error in input, such as mismatched parentheses, would occur to us sooner or later. However, we choose to scan the input before parsing for two reasons: first, having guaranteed the corectness of input makes the parsing itself easier and second, this approach allows us to output a meaningful error message to the user who has entered a wrong input and possibly does not know where the problem is. We will check that the following rules hold:

1. An opening parenthesis, (unary or binary) operator and the beginning of the string must be succeeded by a variable, an unary operator or an opening parenthesis.
2. A variable and a closing parenthesis must be succeeded by a a binary operator, a closing parenthesis or the end of string.
3. The subsequence of input consisting of all the parentheses must be a well-formed parentheses expression.
4. For every SBMPP, there must be a single corresponding binary operator on the outermost level and vice versa.

Properties 1 and 2 are regular and can be thus checked in a single sweep over the input, in linear time and constant memory. A counter is needed to check rule 3 and a stack for 4, but the check can still be done in a single linear sweep. For rule 3, we have to regard each opening bracket as going deeper into the expression, thus incrementing the counter, while each closing bracket means going "upwards" and therefore decrementing the counter. Starting at 0, the counter must be 0 again at the end of the expression but must never reach a negative number in between. We may check the rule 4 by remembering the operator seen in each SBMPP in a stack. As the depth of the formula may be indefinitely high, we need the stack to remember whether we have encountered a binary operator in each level of a nested set of SBMPPs. To illustrate this point better, consider the input

$$((A \wedge ((B \vee C) \wedge \neg A) \rightarrow D) \vee \neg C)$$

where the boldfaced $\rightarrow$ is the current character being processed by the above algorithm. At this point, we need to remember that we have already encountered another operator ($\wedge$) on this level in this SBMPP. However, we need to store this information in a stack, because the same property had to be checked before in the nested SBMPPs $((B \vee C) \wedge \neg A)$ and $(B \vee C)$ and at the time we encountered the $\wedge$, which we later identified to be conflicting with $\rightarrow$, we did not know how deep would these nested SBMPPs go.

Note that the rule 4 in its current formulation adheres to the former, strict definition of the boolean formula we stated in the first section of this chapter. It may be slightly modified to allow multi-operand conjunction and disjunction by replacing "there must be exactly one binary operator"" with "there must not be a pair of different binary operators or two occurences of the same non-associative binary operator"". For example, for an expression like

$$(A \wedge B \wedge C \vee \neg D)$$

when we first encounter the $\wedge$ operator, we will remember it in the stack. Then, encountering the second $\wedge$ operator no rule is violated, because $\wedge$ is an associative operator and it may occur more than once in the same SBMPP and on the same level. However, when we reach $\vee$, the rule 4 is broken because $\wedge$ and $\vee$ are a pair of different binary operators. Similarily, the rule would be broken for

$$(A \rightarrow B \rightarrow C)$$

as there are two occurences of the same, but non-associative binary operator. If we so wish, the fourth rule then may or may not include the requirement for at least one operator to be present on each level of a SBMPP. We then initialize the new value in stack to "undefined" with each opening bracket, i.e. when going "deeper" into nested expression. Then we check wheter this value is still undefined when encountering the closing bracket, i.e. when leaving the current subexpression - thus checking whether there was any operator present on that level. On the other hand, by not including this requirement, we are allowing the existence of a SBMPP without any operators on the outermost level. Then, according to rules 1 and 2 there can not be any variable either. Basically, we are allowing multiple parentheses around a SBMPP, for example

$$(((A \wedge B)))$$

which has virtually no effect, so we may as well include such possibility in the definition. Note that this weakening of the fourth rule still does not make the expression () a valid formula from our algorithm's point of view, as the first rule prohibits this. Finally, note that the fourth rule states that not only must each SBMPP have its corresponding operator, but also vice versa. This is not difficult to check; if an operator is on a nonzero level, it must have a corresponding SBMPP on that level. This disallows expressions such as

$$A \wedge B$$

because the $\wedge$ does not have its corresponding SBMPP. The correct expression would be

$$(A \wedge B)$$

.

If we wish to follow the convention of omitting the outermost pair of parentheses, it suffices to leave out the "vice-versa" requirement from the rule 4. Then even the former of the last two expressions will be accepted by this checking algorithm.

Now, let us prove that the described algorithm indeed recognizes a well-formed boolean formula. In order to make the proof simpler, we will adhere to the strict definition, disallowing multi-operand conjunction and disjunction, multiple-parenthesised SBMPPs and the omitment of the outermost pair of parentheses. However, considering our proof and taking the above notes on rules modification into account, one may get a more general proof that holds even for a looser definition of a boolean formula.

Theorem 1.2. *Consider a string ϕ over the alphabet of boolean variables, boolean operators and parentheses. Then ϕ is a boolean formula iff rules* 1 *to* 4 *hold for ϕ.*

Proof.

$\boxed{\Longrightarrow}$

If ϕ is a boolean formula, then rules 1 *to* 4 *hold for ϕ.*

At first, we should rewrite the definition of boolean formula in terms of the theory of formal languages:

i)	<formula>	::=	<variable>
ii)	<formula>	::=	<unary operator><formula>
iii)	<formula>	::=	(<formula><binary operator><formula>)

Let us take a look at these transcription rules. Clearly, a formula can only begin with a variable (*i*), an unary operator (*ii*) or the opening parenthesis (*iii*). It is therefore true that the beginning of the string must be followed by one of these. Furthermore, the operators and the opening parenthesis are always succeeded by a formula, which we already know must begin with a variable, an unary operator, or the opening parenthesis. These observations are exactly what the rule 1 states.

Let us examine where the variables and the closing parenthesis may occur. Apparently, they may only occur at the end of the formula (i, iii), or at the end

of a subformula (ii, iii); we can see that a subformula is either positioned at the end of the formula it is nested in (ii), or before a binary operator (iii) or a closing parenthesis (iii). Therefore, the variables and the closing parenthesis may only appear just before the end of the formula, a binary operator or the closing parenthesis. This is what the rule 2 states.

If we are to examine the structure of the substring of a formula consisting of all the parentheses, we can take the above formal grammar and leave out all terminal symbols except the parentheses. Clearly, we will get a grammar that will generate the desired substrings.

i') <formula> ::= ε
ii') <formula> ::= <formula>
iii') <formula> ::= (<formula><formula>)

Let us compare this with the grammar of the well-formed parentheses expressions.

i^*) S ::= SS
ii^*) S ::= (S)
iii^*) S ::= ε

Apparently, i') is the same as iii^*). Then, ii') is unnecessary and may be left out without influencing the generated language. Finally, iii') can be constructed by applying ii^*) and i^*) in that order. Therefore, a substring consisting of all the parentheses of any formula can be generated by the grammar of well-formed parentheses expressions. This was needed to show the validness of the rule 3.

Again referring to the grammar of boolean formulae, we can see that only iii) generates a binary operator; however, it also generates a SBMPP and vice versa: only iii) generates a SBMPP and it also generates a binary operator. If any other operator was to be in the same SBMPP, it would have to be inside the subformulae neighbouring the considered operator; however, these subformulae would have to be SBMPPs themselves, implying they would be a level deeper. It follows that there is exactly one operator per level in each SBMPP, as stated in the rule 4.

$\boxed{\Longrightarrow}$
If rules 1 *to* 4 *hold for an expression* ϕ*, then* ϕ *is a boolean formula.*

The reader may have noticed by now that we defined SBMPP as an *expression*, not a *formula* bounded by a matching pair of parentheses. If we defined it as a *formula*, the rule 4 would not make any sense, as we do not yet know whether ϕ - or any part of it - is a formula; we are yet to prove it.

Let us examine ϕ. By the rule 1, its first symbol must be a variable, an unary operator or the opening parenthesis.

- If it is a variable, let us designate it as A, then ϕ is identical to A. Suppose not - than A, as a variable, must be followed by either the closing parenthesis or a binary operator. If it were the closing parenthesis, this would violate the rule 3, as the first parenthesis in a well-formed parentheses expression must be an opening one. If it were a binary operator, this would violate the rule 4 as there are no matching parentheses to this operator (there would have to be an opening parenthesis preceding it). Both options lead to contradiction. Therefore, ϕ is indeed identical to A, therefore ϕ is a boolean formula.

- If it is a negation operator, it must be followed by a variable, another unary operator or the opening parenthesis. If it is followed by another negation operator, we are getting to this same situation. There may be a chain of negation operators, but by the rule 2 an expression cannot end with an unary operator. Thus, after the chain of negation operators ends, there must be a variable or an opening parenthesis. If it is a variable, we can easily see that we are in the same situation as in the previous paragraph. On the other hand, if it is an opening parenthesis, we get to the situation described in the following paragraph.

- If it is an opening parenthesis, by the rule 3 there must be a matching closing one. Therefore, ϕ is of the form $(s)t$, where s, t are unknown strings. As (s) is a SBMPP, there must be a corresponding binary operator. WLOG, let it be $\wedge$. Then $s = s_1 \wedge s_2$, where s_1, s_2 are unknown strings and ϕ is of the form $(s_1 \wedge s_2)t$. As t follows a closing parenthesis, it must begin with another closing parenthesis or a binary operator. However, t cannot begin with a closing parenthesis; its matching opening parenthesis would have to be before the SBMPP enclosing s, which is not possible. If t began with a binary operator, there would have to be a corresponding SBMPP for it, which is impossible for the same reason. The only possibility is that t is empty and now we know that ϕ is of the form $(s_1 \wedge s_2)$. Let us examine s_1. It could begin with a variable, a negation operator or an opening parenthesis. We can easily see that now we are solving the same problem recursively:

  - If it starts with a variable, then it is identical to that variable.
  - If it starts with a negation operator, there may be a chain of negation operators, after that one of the remaining two options occur.
  - If there is an opening parenthesis, than the matching closing parenthesis must be before the closing parenthesis bounding s, because t is empty. It must even be before the $\wedge$ operator. If it were not, that would mean $\wedge$ is one level deeper than the SBMPP (s), what is a contradiction as $\wedge$ is the operator corresponding to that SBMPP. Therefore we get that $s_1 = (s_3)s_4$. For the same reasons t was proven to be empty, s_4 can be proven to be empty too. Therefore in this case $s_1 = (s_3)$.

One can recognize that the same arguments hold for s_2 as well.

It follows that ϕ is of the form A (where A stands for any variable), $\neg\neg\ldots\neg\psi$, or $(\psi \wedge \chi)$. We chose $\wedge$ to stand for any binary operator to express the idea more clearly. This was done without the loss of generality, therefore ϕ can also be of the form $(\psi \vee \chi)$, $(\psi \rightarrow \chi)$ or $(\psi \leftrightarrow \chi)$. The most important thing to add here is that ψ and χ have the same structure as ϕ – they can too take on any of the forms listed herein.

Now we can finish the proof by induction on the complexity of ϕ.

1. ϕ is of the form A, then ϕ is clearly a boolean formula (step 1 of the definition).

2. ϕ is of the form $\neg\neg\ldots\neg\psi$ and by the induction hypothesis, ψ is a boolean formula. Then, by the second step of the definition, $\neg\phi$ is a boolean formula too. Inductively, $\neg\neg\ldots\neg\psi$ is a boolean formula as well.

3. ϕ is of the form $(\psi * \chi)$, where $*$ stands for any binary operator and by the induction hypothesis, ψ and χ are both boolean formulae. Then, by the step two of the definition, $(\psi * \chi)$ is a boolean formula as well.

□

We have proposed an algorithm to check whether a given input is a well-formed boolean formula and proven the algorithm's corectness. Now that we can guarantee the validness of the input, let us move on to the parsing itself.

1.5.2 The parse tree

As it has been suggested in the previous text, the language of boolean formulae is a context-free language. As such, it is possible to construct a parse tree (also known as expression tree). Given a formula ϕ, we regard each of its subformulae as a node in such a tree. The atomic formulae correspond to leaves, while the non-atomic, operator-containing subformulae correspond to an internal node which we will mark with that operator. The ancestor – descendant relationship in the tree correspond to the "is a subformula of" relation between two subformulae of ϕ. It follows that if ψ is a subformula of χ and there is no ξ such that ξ is a subformula of χ and ψ is a subformula of ψ, then the node for ψ is a child node of the one for ξ. Finally, the root node stands for the whole formula. This may be better illustrated by an example:

It can be further seen that assuming the strict definition of boolean formula, we can categorize the nodes by their degree. It has already been stated that the leaves correspond to the atomic formulae. The nodes with a single child then correspond to formulae in the form $\neg\phi$ and the ones with two children correspond to SBMPPs. No higher count of children is possible. Note that these are exactly the three kinds of subformulae that appear in the process of recursive construction of a boolean formula. This observation gives us a clue

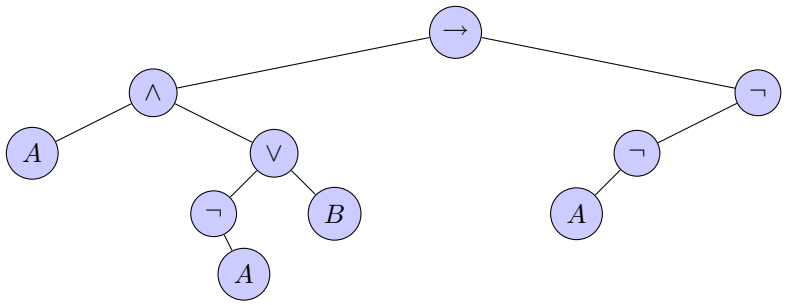

Figure 1.1: The parse tree of $(A \wedge (\neg A \vee B)) \rightarrow \neg\neg A$

on how to parse the formula. Given an input string and having verified its correctness using the algorithm discussed before, let us parse it. We will do another linear scan of the input string and build the parse tree along the way. We start with the parse tree having only a single (root) node. We will keep a pointer into tree, marking the currently considered node, on which operations will be done. Naturally, the pointer is initially set to point to the root node. Now, reading the input, for each character we decide:

- The character is an opening parenthesis, meaning the beginning of a SBMPP in the form $(\phi \circ \psi)$, $\circ$ standing for any binary operator. The current node will therefore have two children. Create them. As the next character is the first character of ϕ, which corresponds to the left child node, move the pointer there.
- The character is a binary operator. This again occurs in SBMPPs of the form $(\phi \circ \psi)$. It means that we are done parsing ϕ and the next character is the first character of ψ. As nodes for ϕ and ψ are siblings, move the pointer to the right sibling. Also, mark the parent with this operator.
- The character is an unary (negation) operator. We have encountered a subformula in the form $\neg\phi$. Its node only has one child, which corresponds to ϕ. Mark the current node with negation operator, then create that child node and move the pointer there.
- The character is a variable name or a closing parenthesis. In the former case, mark the current node with that variable. Both cases, however, mean that this is the last character of the subformula associated with this node. To continue parsing, move to the parent node. If the parent node is neither the root of the tree nor a node with two children, move up again. Keep moving up the tree until you reach a node that satisfies this condition.

This algorithhm constructs a parsing tree, marking each node with its meaning - a variable for the leaf nodes and an operator for the internal nodes. Let us take a look at the three kinds of subformulae we can encounter. For a variable,

we do not make any movement in the tree. For a SBMPP in the form $(\phi \circ \psi)$, we move down a level when reading the opening parenthesis. After parsing ϕ, we move to its right sibling, the node for ψ. After parsing ψ, on the closing parenthesis) we move back up again at least one level. Whether we move more than one level depends on what is $(\phi \circ \psi)$ a subformula of. Considering the last type of subformulae that are neither variables nor SBMPPs, they are in the form $\neg\neg \ldots \neg\phi$. We move down the tree for every negation. After parsing ϕ, we move up the tree until reaching the root or at least a node corresponding to a SBMPP. Apparently, we have to move up a level for each of the negation operators. This means that for a correct input, the pointer moves downwards the same amount of times as it moves upwards during the parsing, returning to the root as it finishes reading the whole formula. By similar pattern of thoughts, tracking the movement of the pointer, one can also see that every node gets marked. With this in mind, the correctness of the above algorithm may be proven by induction on the structure of the formula.

As usual, let us examine the modifications that have to be made to this algorithm to account for a less strict definition of boolean formula. If the outermost pair of parentheses is missing, we just add them prior to parsing. If we allow multi-operand operators, the SBMPPs will be represented by nodes with at least two children (and not just exactly two children as before), so while parsing them, we will move to the right sibling several times. If we allow chains of parentheses around a single SBMPP (i.e. $((\ldots(\phi))\ldots)$), we may view the extra pairs of parentheses as an assertion operator (i.e. unary identity on $\mathbb{B}$). A similar idea is to regard the extra pairs of parentheses as SBMPPs enclosing unary conjunctions or disjunctions.

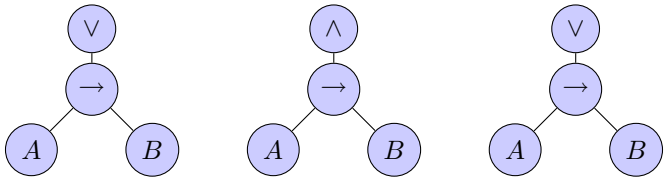

Figure 1.2: Various possibilities to represent the formula $((A \to B))$

It is also possible to optimalise the representation of chains of negations, as it holds that $\neg\neg..\neg\phi$ is equivalent to either ϕ or $\neg\phi$, depending on whether the count of negation operator in the chain is even or odd, respectively. Upon encountering a negation operator, the algorithm would scan forward and count the number of operators in the chain. It would then interpret the chain as a single operator or ignore it completely.

1.5.3 CNF - an array of clauses

As mentioned before, the SAT algorithms mostly focus on formulae in CNF. If we only work with formulae in CNF, the parse tree representation becomes rather cumbersome. The structure of parse tree would be invariant - a $\wedge$ operator in the topmost tier, $\vee$ operators in the second tier and finally literals in the last tier. Therefore we can represent a formula in CNF using a much simpler structure - an array of arrays of literals. Clearly, such structure is isomorphic to CNF: Given a formula in CNF, we can represent it as an array of clauses, while every clause is represented as an array of literals. The only information about the formula we do not store in this structure is what the operators are. However, as we know the literals in a clause are connected by a disjunction and the clauses themselves form a multi-operand conjunction, we can easily reconstruct the formula from such an array. Note that this structure is especially convenient when dealing with an instance of n-CNF. In this case, the second dimension of the two-dimensional array, representing a clause has a constant size, thus making it easier to implement.

If our algorithm is expecting the input formula in CNF, not only its representation, but also verifying the input correctness and parsing becomes a simpler task. We expect the input to be in the form

$$(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$$

where $\phi_1, \phi_2, \ldots, \phi_n$ are clauses. Therefore, as the first step of the parsing algorithm, we check whether the input starts with an opening parenthesis and ends with a closing one, then divide the rest into several parts, using the $\wedge$ operators as delimiters. Now each of the substrings of the input acquired by the division must be a clause. We have to check if it has the form

$$(\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_k)$$

where λ_i is a literal for every i in the range. Thus, these substrings must too be bounded by a pair of parentheses. Stripping those, the rest of the substring will be divided according to the $\vee$ operators into even smaller parts. Apparently, those must be literals, so we only check if they are of the form A or $\neg A$. One way to implement this algorithm would be to have three functions: `check_cnf()`, `check_clause()`, `check_literal()`. Each of the functions would take one argument - the index to the input string. They would check the substring beginning on that index and decide whether this substring is a formula in CNF, a clause or a literal, respectively. The `check_cnf()` function would call `check_clause()` several times, for each substring bounded by two $\wedge$ operators (or a $\wedge$ operator and a parenthesis). Similarily, the `check_clause()` function would call `check_literal()` to test the substring bounded by a pair of $\vee$ operators. As there are three "tiers" of calls, each of them having a total running time $O(length\ of\ input)$, we can say that this algorithm has a linear

time complexity.

One thing we must not forget about when considering the parsing algorithms is dealing with the variables' names. In the previous text, we assumed that each variable is represented by a single letter (i.e. symbol of the Latin alphabet). We may need to name the variables using longer than one-later names, especially if there are more variables than letters in the alphabet. It is not difficult to modify the parsing algorithm to reckon with this situation - when encountering a letter, we keep scanning the input string until a nonalphabetic symbol appears; we then parse the whole substring of alphabetic symbols a single token. Furthermore, we must consider the fact that the SAT-solving algorithms will need to store information about the variables - most importantly, the valuation. For this purpose, we need the parsing algorithm to build an associative array mapping the variable names to structures storing these information.

The associative array may be implemented using a binary search tree, giving us $\Theta(logN)$ access time for both the average and worst-case, where N is the size of input, i.e. the number of items in the associative array. Another option is to use a hashtable, which has $O(1)$ expected access time, but $O(N)$ in the worst case. We can even get guaranteed $O(1)$ access time if we only use numbers to name the variables, so we can store them in a regular array. In this case, however, N stands for the highest number used, not the variable count. This access time must be taken into account when analysing the time complexity of the studied algorithms. For simplicity, in our analyses we will consider that the array access time is $O(1)$, which means that neither storing nor reading any information about variables takes any additional time.

Chapter 2

The complete algorithms

In many applications, we want to find the solution for a given SAT instance as quickly as possible. In practice, this is important. However, in theory, when studying the complexity of this canonical NP-complete problem, we are interested in algorithms that can not only find a solution, but also find out *if* there is one. We call those algorithms complete. Unlike the heuristics that somehow focus on those valuations with higher probability to become solutions, these algorithms make a thorough search of the space of all valuations, considering all possibilities. This means they are generally also much slower. However, as we will see later, the ability to prove that a given formula is unsatisfiable has applications itself.

2.1 Brute-force search

The most trivial algorithm that comes to one's mind is the brute-force search. This means to iterate over all the possible valuations and check if any of them satisfies the given formula. If there are N different variables in the formula, having two possible values per variable, we get a total of 2^N valuations. We can search over these valuations recursively. In two branches, we set the value of the first variable to 1 and 0, respectively. In each of these branches, we branch again to consider the two options for the second variable etc. After branching for N times, when we reach the end of the recursion tree, we have constructed a valuation, having set the value of all variables. Now we have to check whether the given valuation satisfies the formula. If the formula is in CNF, this can be done in a single linear sweep over the input, checking for every clause if at least one of its literals evaluates to 1. If L is the length of input, this operation takes $O(L)$ time. If the formula is represented by a parse tree, the value can be obtained by a recursive call, where every internal node's value depends upon its children, the values of leaves are determined by the valuation and the value of the whole formula, which we seek, can be found as the value of the root node. As the size of graph is apperently $O(L)$, this 's running time is also $O(L)$.

Thus, we get $O(L \cdot 2^N)$ time complexity for the brute-force search. Note that an unsatisfiable formula is the worst case input for this algorithm. It means the algorithm will have to iterate through all the 2^N valuations, not finding any solution. Let us consider all the valuations to be equiprobable in terms of being satisfying. Then, if there exists a single satisfying valuation, its expected position is in the middle of valuations we iterate through. Therefore, in this case the expected running time of the algorithm is a half of the worst-case time. Furthermore, if there are multiple satisfying valuations, we are bound to find one of them even sooner. This means that although the time complexity of brute-force is exponential, if the formula is "easily satisfiable", the actual performance of the algorithm will be notably better than the worst-case.

2.2 DPLL

The DPLL algorithm was proposed in 1962 by Davis, Logemann and Loveland as a refinement of older Davis-Putnam algorithm, hence receiving its name as an abbreviation of its inventors' initials. We will follow its description in [7]. Unlike the brute-force search, DPLL is not destined for any formula, but only for formulae in CNF. Similarily to the brute-force, it is based on a recursive building of a valuation variable by variable. However, DPLL takes advantage of the fact that the input is in CNF and comes with a few observations that allow us to see in advance if some particular subtrees of the recursion tree contain a solution. Then we can skip those subtrees that do not, thus making our search consider fewer valuations when looking for a solution. Therefore, we can regard DPLL as a brute-force for CNF with a few improvements that are simple, but dramatically lower the runtime of the algorithm.

The three observations, or rules, that DPLL utilizes are:

- *The splitting rule.* For every variable, we split the problem into two sub-problems. In the first one, we assume that a given variable A is valued 1 and in the second one we assume the opposite. Therefore, we partition the set of all valuations into two mutually exclusive yet exhaustive subsets $\{v \mid v(A) = 1\}$ and $\{v \mid v(A) = 0\}$, each of the sub-problems considering one of the sets. The reader has surely noticed that we actually already described the splitting rule above to illustrate that DPLL can be thought of as an extension of the brute-force algorithm (that is, if we implement the brute-force recursively). The difference in DPLL is that both assumptions (that a given variable A is valued 1 or 0) lead to a certain simplification of the formula. Namely, if $v(A) = 1$, then all clauses containing the literal A are satisfied by the valuation v. We can easily see that the rest of the formula, which is a conjunction of the clauses not containing positive A, is equisatisfiable to the whole formula. Thus, we can strip the formula of all the clauses containing A, not interfering with the solvability of the

SAT problem. Analogously, the same is true for $v(A) = 0$ and clauses with an occurence of $\neg A$. Furthermore, if $v(A) = 1$ holds, we can remove all occurences of $\neg A$ from every clause for the same reason - this does not change the clauses' satisfiability. Needless to say, the situation for $v(A) = 0$ and occurences of A is analogous.

- *The one-literal clause rule* In the previous rule, we introduced the possibility of removing individual literals from clauses. Therefore, it may occur that there is a clause containing only a single literal (a unit clause) - WLOG, let it be A. It is apparent that if we set $v(A) = 0$, v will not satisfy the formula. It is apparent that v satisfies the formula iff $v(A) = 1$. At some point in the algorithm, adhering to the splitting rule, we would have to divide the problem into two branches, according to the value of A. Now we know there is no sense in considering A to be valued 0. We can therefore state that $v(A) = 1$ and simplify the formula again according to this fact, as we would do after aplying the splitting rule - only this time, there is no actual splitting as the other branch is never considered.

- *The pure literal elimination rule* It can occur, in the initial formula or in some of its simplifications, that there is no occurence of a certain literal (WLOG let it be $\neg A$). We then call the literal A a "pure literal". Now consider two valuations v_1 and v_2. Let $v_1(B) = v_2(B)$ for every variable B different from A. Let $v_1(A) = 0$ and $v_2(A) = 1$. If v_1 satisfies the formula, it means that in every clause there it at least one literal true under v_1. That literal can not be $\neg A$, because we assumed there is no such literal. It can not be A either, because $v_1(A) = 0$. The variable A has therefore no effect on whether the valuation satisfies the formula. This means that v_1 satisfies it only if v_2 does, as A is the only variable where they differ. Therefore, whenever a pure literal A appears, we may disregard the branch where $v(A) = 0$, because a solution exists in that branch only iff it exists in the one where $v(A) = 1$. Just like in the previous rule, we can set $v(A) = 1$ and immediately simplify the formula according to this fact.

Where not mentioned explicitely, it should be obvious that the rules are symmetrical in the sense that what holds for positive literals also (analogously) holds for the negative ones.

If at any point of the algorithm the simplification process removes all the literals from some of the clauses, i.e. creates an empty clause, it means that the current branch does not lead to any satisfying valuation. There is no need for further branching - we have to return from the recursion instead. On the other hand, if we reach the bottom of the recursion tree, setting a value for all the variables and ultimately removing all the clauses from the formula, we have found a solution.

Now let us explain the algorithm flow in detail. Let us consider that there are n variables named A_1 through A_n. First, we set $v(A_i)$ to undefined for

every i. Then we define a recursive function `solve()` taking one argument, the index of the variable it has to consider. The body of the algorithm then consists of a singe call `solve(1)`. So let us describe the `solve()` function. We will reference the function's argument simply as i. The function has two options to consider - that $v(A_i) = 1$ and that $v(A_i) = 0$. If $v(A_i)$ has been set earlier in the recursion as a result of the one-literal clause or pure literal rule, there is no need to consider the option that the value $v(A_i)$ is opposite. But otherwise, if $v(A)$ is undefined, we will consider both options. Now, for each considered option we do the following:

1. Simplify the formula according to the choice of $v(A_i)$.
2. Check if the one-literal clause or the pure literal rules are applicable. If they are, apply them and return to step #1. Check if there exists an empty clause.
3. If there is no empty clause and $i < n$, call $solve(i + 1)$.
4. After returning from the recursion, undo all the simplifications that resulted from this choice of $v(A_i)$. This means returning the removed clauses and literals back to the formula - it may be therefore more convenient to never actually remove them, just mark them as unused. Moreover, we have to revert any values of $v(A_j)$ for $j > i$ that were determined in advance as a result of simplification back to undefined.

The rule checks (step #2) can be done in a single linear sweep over the formula. For every variable, we remember if we have seen it in an positive and in a negative literal. After scanning the whole formula, we check if there exists a pure literal. For every clause, we check how many literals are marked as removed - if exactly one is remaining, this is an occurence of the single-literal clause rule. If all are marked as removed, this is an empty clause and we know this branch does not lead to a solution. In (step #3), if there is no empty clause and $i = n$, we must have defined $v(A_i)$ for every $1 \leq i \leq n$ and marked all clauses as already satisfied and removed. Therefore, v is the satisfying valuation and our solution. If this never occurs and empty clauses always force us to return from the recursion before reaching the bottom of the recursion tree, the algorithm halts, having checked all the possible valuations and found none. In this case, the given formula is unsatisfiable.

For the purposes of explanation, it was convenient to regard DPLL as an improvement for the brute-force algorithm. However, the improvements speed up the runtime so dramatically, that one can regard the comparison as dehonesting. In fact, DPLL is the foundation of many modern SAT-solvers [14].

2.3 DNF conversion

The DPLL algorithm, like many other algorithms we are going to examine, targets formulae in CNF. We know that SAT is $\mathcal{NP}$-complete and that its subset,

CNF-SAT is $\mathcal{NP}$-complete as well. This makes formulae in CNF the most difficult class of formulae in terms of inspecting their satisfiability. Previously, we have shown that for every formula, there exists an equivalent formula in CNF as well as DNF. Furthermore, the methods of constructing such formulae are quite analogous. The interesting thing is that formulae in DNF, on the other hand, are the easiest to determine satisfiability of. In fact, this can be done in linear time (with respect to the length of the formula). First, let us take a look at how this is done. A formula in DNF looks like this:

$$(\lambda_{1,1} \wedge \lambda_{1,2} \dots \lambda_{1,k_1}) \vee (\lambda_{2,1} \wedge \lambda_{2,2} \dots \lambda_{2,k_2}) \vee \dots \vee (\lambda_{n,1} \wedge \lambda_{n,2} \dots \lambda_{n,k_n})$$

where n is the number of conjunctions and $\lambda_{i,j}$ is the j-th literal of k_i total in the i-th conjunction. To satisfy a disjunction, we only need to satisfy one of its operands. In our case, which is DNF, those operands are conjunctions of literals. Suppose that for every variable A, only one of the literals $A, \neg A$ occurs in the conjunction. Then let v be a valuation such that $v(A)$ is random if A does not occur in the conjunction, $v(A) = 1$ if the literal A occurs in the conjunction and $v(A) = 0$ if $\neg A$ occurs there. Clearly, v satisfies all the literals in the conjunction, therefore it satisfies the conjunction itself. On the other hand, if there is a pair of complementary literals $A, \neg A$ for some variable A, the conjunction is clearly unsatisfiable as for every valuation at least one of the operands will be valued 0. It follows that a conjunction of literals is unsatisfiable iff it contains a pair of complementary literals. Therefore, a formula in DNF is unsatisfiable iff *every* its conjunction contains a pair of complementary literals. Apparently, this can be checked easily in a single linear sweep through the input. In each conjunction, for every of its literals we have to remember what variable it belongs to, so we can find out about the occurence of two opposite literals of the same variable.

Usually, the SAT problem instances are stated in CNF. This is probably due to the fact that CNF-SAT is somewhat more expressive of a problem, which is related to the fact that CNF formulae are the most difficult class of formulae for SAT. Let us examine how would solving CNF-SAT by conversion to DNF look like. We cannot expect the time complexity of this process any less complex than $\mathcal{NP}$ - if it was, it would make up for a faster solution of CNF-SAT. Consider the formula

$$(A_1 \vee A_2) \wedge (A_3 \vee B_4) \wedge \dots \wedge (A_{2n-1} \vee A_{2n})$$

which is in 2-CNF and has $2n$ literals. It is satisfied whenever for every i at least one of the pair A_i, B_i is satisfied. In DNF, we are basically listing the possibilities how to satisfy the formula. Every operand of disjunction is one such possibility. Every possibility is described as a conjunction of literals, where A stands for $v(A) = 1$ and $\neg A$ stands for $v(A) = 0$. Therefore, to construct a DNF from the above formula we must account for every possibility of how to satisfy it, so the resulting DNF can be no shorter than

$$\bigvee_{x_1,x_2,\ldots,x_n\in\{0,1\}} (A_{1+x_1} \wedge A_{3+x_2} \wedge \cdots \wedge A_{2n-1+x_n})$$

which has $n \cdot 2^n$ literals, exponentially more than the original formula. Apparently, conversion from CNF to DNF will lead to an exponential prolongation of formula, therefore also (at least) exponential time complexity and will be no better than the brute-force. Now let the formula in CNF have a total of V variables, C clauses and P literals per clause (assuming this number is constant; it is not a problem to extrapolate the following thoughts to the case where it is not constant, however, this assumption results in simpler expressions). We can easily see that the above construction leads to $C \cdot P^C$ literals. We do not have to construct the whole formula in place (as it is of exponential size, this would cause a memory insufficiency for our program), just construct conjunction after conjunction (as every conjunction only has size C). Every conjunction is obtained easily in the form $\lambda_{1,i_1} \wedge \lambda_{2,i_2} \wedge \cdots \wedge \lambda_{C,x_C}$ where for every $1 \leq i \leq C$ we have $1 \leq x_i \leq P$. Thus, we can generate all the conjunctions by generating the sequence of all at most C-digit numbers in base P and interpreting their digits as $x_1 x_2 \ldots x_C$. As will be seen later, converting DNF to CNF does not outperform the brute-force. It can be seen from the fact that in brute-force, we examine 2^V valuations, while in DNF conversion we construct P^C clauses. Usually, we have $P > 2$; for $P \leq 2$ we are dealing with 2-SAT, which is a lot simpler problem than general SAT as it is in P. We also usually have $C > V$ - again, the opposite would make up for a very easy problem, as we will show later on heuristics performance. Therefore, $2^V << P^C$, what makes the brute-force much faster.

We have shown that it is irrational to convert a formula stated in CNF into DNF. However, how about other formulae in more general forms? If a formula is in such a form that converting it to CNF and DNF is equally difficult, then DNF would be a better choice as the subsequent satisfiability test is faster. Apparently, even if the input is a bit "farther" from DNF than CNF, it could still be faster to convert it to DNF and then test it in linear time. As CNF and DNF are in some kind of a symmetry, we would expect that for a random formula, there is $\frac{1}{2}$ probability that it is more similar to CNF than DNF (and $\frac{1}{2}$ probability of the opposite). It follows that solving SAT on a random formula by converting it to DNF first would yield good results (in terms of speed) with probability higher than $\frac{1}{2}$. We will test this hypothesis later. Now, let us take a look on how to do the conversion. Unlike most SAT algorithms, which deal primarily with CNF-SAT, we can not use the set of clauses representation of a formula, as it is not applicable. The DNF transformation will therefore be done as a manipulation of vertices of the parse tree.

In the proof of Theorem 1.1, we have seen that the only operations we need in such a transformation are:

1. the definitions of $\rightarrow$ and $\leftrightarrow$ by $\wedge, \vee$ and $\neg$

2. the distributive laws
3. the De Morgan's laws
4. the law of double negation ($\phi \equiv \neg\neg\phi$)
5. joining several conjunctions (disjunctions) into one larger conjunction (disjunction)

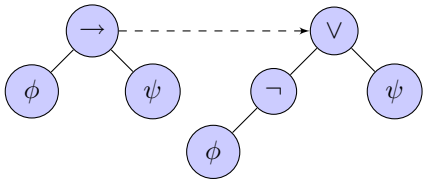

Figure 2.1: The definition of $\rightarrow$

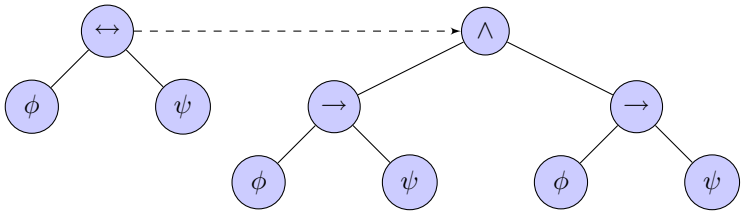

Figure 2.2: The definition of $\leftrightarrow$

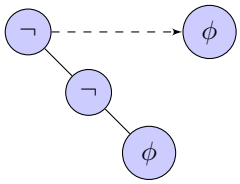

Figure 2.3: The law of double negation

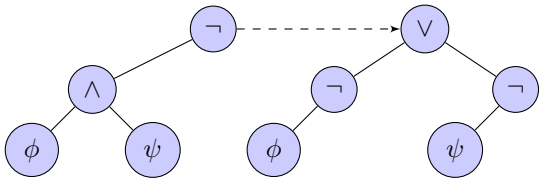

Figure 2.4: De Morgan's laws

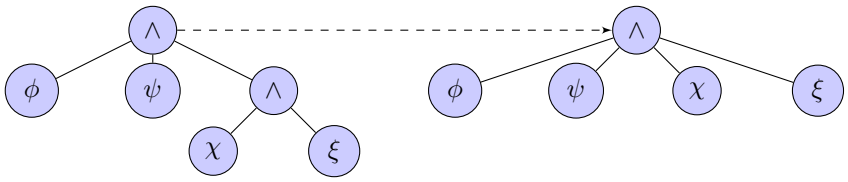

Figure 2.5: Merging together of a conjunction and its operand, which is also a conjunction

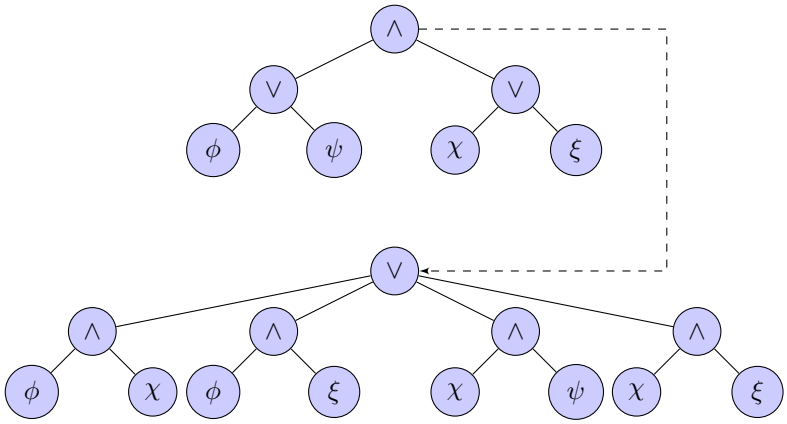

Figure 2.6: The distributive laws

First, we substitute all the $\rightarrow$ and $\leftrightarrow$ operators with their definitions by $\wedge, \vee$ and $\neg$. Now the parsing tree consists of $\wedge, \vee$ and $\neg$ as inner nodes and variables as leaves. If we are dealing with a special case formula like a single conjunction (e.g. $(A \wedge \neg B)$), we just add the disjunction (i.e. $\bigvee(A \wedge \neg B)$). Analogously, another special case is a single disjunction, where we add the conjunctions (e.g. we transform $(A \vee \neg B)$ into $(\bigwedge(A) \vee \bigwedge(\neg B))$). Now let us assume that there is at least one conjunction and at least one disjunction. This means the depth of our tree is at least three: one level for the root operator, whichever it is; the another operator must be in another level; as operators are always inner nodes, there must be at least one more level for the leaves. The target of our transformation is the DNF. The parse tree of a formula in DNF has the following properties:

1. On the 1st level, there is a single $\vee$ node
2. On the 2nd level, there are several $\wedge$ nodes
3. On the 3rd level, there are either variables (of positive literals) or $\neg$ nodes (of negative literals)
4. On the 4rd level (if there is any), there are variables of negative literals

We can see that the parse tree of a formula in DNF also has three or four levels. Therefore, generally, the main objective of this transformation is to decrease the depth of the parse tree. Then we only have to make sure that the operators are in correct order (because the CNF parse tree is similar, just with $\wedge$ and $\vee$ exchanged).

First, if we have a $\wedge$ node that is a parent of another $\wedge$ node, it means there is a conjunction with another conjunction as one of its parameters. Thus, we can merge the inner conjunction into the outer by replacing it with its operands. Thus, instead of $(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n \wedge \psi)$ where $\psi \equiv (\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_k)$, we get $(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n \wedge \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_k)$. The same can be done with $\vee$ and it assures that there will not be two adjacent nodes of the same binary operator.

Now, every path from the tree root to the leaves only contains $\wedge$, $\vee$ and $\neg$ operators, while $\wedge$ and $\vee$ are never adjacent. Let us deal with $\neg$ operators now. In Figure 2.4, we can see how the $\neg$ nodes can be moved downwards by being exchanged with $\wedge$ or $\vee$. After moving all $\neg$ nodes downwards, every root-to-leaf path will begin with a chain of $\wedge$ and $\vee$ nodes, followed by a chain of $\neg$ nodes and end with a variable node. By the law of double negation, each chain of $\neg$ nodes may be replaced by a single $\neg$ node or left out entirely, according to the parity of its length. In the chain of $\wedge$ and $\vee$ nodes, we may have caused another adjacencies of same operator nodes, so we must merge those too.

The previous operations left us with every root-to-leaf path in the form of an *alternating* chain of $\wedge$ and $\vee$ nodes ending either with a single variable node or a $\neg$ node followed by a variable node. This is very similar to our objective. All we have to do is make the $\wedge/\vee$ chain start with a $\vee$ node and shorten the length of the chain to two. The former can be done (if the chain actually starts with a $\wedge$) by using the distributive laws on the root node (see Figure 2.6). The latter is similar: we use the distributive laws on a $\wedge$ (or $\vee$) node to exchange its position in the chain with the following $\vee$ (or $\wedge$) node. As the chain consisted of *alternating* $\wedge$ and $\vee$ nodes, this would result in an adjacency of two $\wedge$ and another adjacency of two $\vee$ nodes (or only one adjacency if this is done on the end of the chain). By merging the conjunctions and disjunctions respectively, we would shorten the chain.

As mentioned earlier, by transforming the parse tree into this form, we transform the given formula to DNF. Now, to check its satisfiability, we have to check if there is a conjunction that does not contain two opposite literals. This correspond to checking if there is a $\wedge$ node that is parent to both a variable node A and a $\neg$ node followed by a variable node A, for some variable A. This can be done easily in a single traversal of the tree in a linear time.

2.4 The resolution method

The resolution method did not arise as a SAT solving algorithm, but could be used as such. It was introduced as early as 1965 by Robinson in [15]. It is used in first-order logic to prove that a given CNF formula is a theorem of the first-order logic. However, as boolean (propositional) logic is a subset of the first-order logic, the method is applicable there, too. In the first-order logic as well as in the propositional logic, theorems are the same tautologies. Therefore, TFAE:

- ϕ is a theorem
- ϕ is a tautology
- $\neg\phi$ is unsatisfiable

The resolution method takes advantage of the above equivalence. Instead of directly proving that the given formula is a theorem, we prove that its negation is unsatisfiable. The resolution method is complete - it can also prove that a given formula is not a theorem, and thus its negation is satisfiable. Therefore, we may list the resolution method as a complete SAT-solving algorithm.

The core of the resolution method is the resolution principle. Let $\phi \equiv (\phi_1 \vee \phi_2 \vee \cdots \vee \phi_i \vee \cdots \vee \phi_n)$, $\psi \equiv (\psi_1 \vee \psi_2 \vee \cdots \vee \psi_j \vee \cdots \vee \psi_k)$ be clauses and ϕ_i and ψ_j be opposite literals. Then for every valuation v such that ϕ and ψ are satisfied by v, the following clause is also satisfied by v.

$$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_{i-1} \vee \phi_{i+1} \vee \cdots \vee \phi_n \vee \psi_1 \vee \psi_2 \vee \cdots \vee \psi_{j-1} \vee \psi_{j+1} \vee \cdots \vee \psi_k$$

The latter clause is called the *resolvent* of the former two and is also their logical consequence. The resolvent method consists of generating a sequence of sets of clauses $\{S_i\}_{i=0}^{\infty}$, such that

- $S_0 = \{\phi \mid \phi \textit{ is a clause in the input formula}\}$
- $S_{i+1} = S_i \cup \{\chi \mid \chi \textit{ is a resolvent of } \phi \textit{ and } \psi \wedge \phi, \psi \in S_i\}$

If there are V variables in the formula, there only exist $2 \cdot V$ different literals, so the longest possible clause is $2 \cdot V$ literals long. This length constraint also constraints the number of different clauses. As clauses can be simply regarded as subsets of the set of all literals, there are only of $2^{2 \cdot V}$ of them. Furthermore, not all of them must be resolvable from S_0. As $\forall i : |S_i| \leq const$ and apparently $S_i \subseteq S_{i+1}$, there must be i_0 such that $\forall i \leq i_0 : S_i = S_{i_0}$. This means the algorithm will always halt, not being able to resolve more clauses.

Suppose there was the empty clause $\bigvee \emptyset$ resolved in the set S_j. Furthermore, suppose there is a valuation v that satisfies the input formula. It follows that

v satisfies all its clauses, which constitute the set S_0. As S_1 consists of clauses resolved from S_0, v also satisfies S_1. By induction, v satisfies even S_j and the empty clause $\bigvee \emptyset \in S_j$. This is a contradiction, because an empty clause is unsatisfiable (remember that disjunction is satisfied iff there exists an operand which is satisfied; this is not the case, as there are no operands at all). It therefore cannot hold that v satisfies S_0, for *any* valuation v. This means that whenever we resolve an empty clause, the algorithm may halt and output that the given formula is not satisfiable. An example will illustrate this:

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B)$$

$$\begin{aligned} S_0 &= \{\ (A \vee B),\ (A \vee \neg B),\ (\neg A \vee B),\ (\neg A \vee \neg B)\ \} \\ S_1 &= S_0 \cup \{\ \bigvee(A),\ \bigvee(B),\ \bigvee(\neg A),\ \bigvee(\neg B)\ \} \\ S_2 &= S_1 \cup \{\ \bigvee \emptyset\ \} \end{aligned}$$

The opposite also holds - if the algorithm halts by not being able to resolve more clauses and there has not been an empty clause resolved, the input formula is satisfiable. The proof of this fact, however, is long and is beyond the scope of this thesis. The proof can be found in [15]. Note that the resolution method decides whether the given formula is satisfiable, but does not actually find a satisfying valuation. Note also that this is one of the few algorithms that actually have better running time for unsatisfiable clauses, as it focuses on finding the proof of unsatisfiability (which is a resolved empty clause) rather than a sastisfying valuation.

2.5 Benchmarks

In this section, we are going to examine the performance of the described algorithms. We have implemented all four of the algorithms and measured their runtime over a large number of runs. We will use the following notation to describe the benchmarks:

- V - number of variables
- C - number of clauses
- S - size of a clause (if constant)
- R - number of runs per value of C

Each of the benchmarks has the same scheme. We iterate with V over a certain range of values (naturally, the faster the algorithm is, the larger the range can be for the benchmark to be still feasible). For every value of V, we run the algorithm on a set of R randomly generated formulae. These formulae are in S-CNF with V variables and C clauses; usually, S is a constant and C is a function of V.

We use the following process to generate the formulae. First, we generate a sequence $n_1, n_2, \ldots, n_{C \cdot S}$ by selecting each n_i independently with a uniform probability distribution from the interval $\{1, 2, \ldots, V\}$. If $n_i = n_j$ for any j such that $S \cdot \lfloor \frac{n}{S} \rfloor < j < i$, we do not use the value and re-generate it, until such a condition does not hold. Now, consider the variables to be named $A_1, A_2, \ldots, A_V$. We then construct the literals as a sequence $L_1, L_2, \ldots, L_{C \cdot S}$, where

$$L_i = \begin{cases} A_{n_i} & \textit{with } p = \frac{1}{2} \\ \\ \neg A_{n_i} & \textit{with } p = \frac{1}{2} \end{cases}$$

Finally, the formula takes the form

$$(L_1 \vee L_2 \vee \cdots \vee L_S) \wedge (L_{S+1} \vee L_{S+2} \vee \cdots \vee L_{2 \cdot S}) \wedge \cdots \wedge (L_{C-S+1} \vee L_{C-S+2} \vee \cdots \vee L_{C \cdot S})$$

2.5.1 Brute-force search

Let us start with the brute-force search benchmark results. For every value of V, we show the average of running times of all R runs of the algorithm.

$\boldsymbol{V}$	$\boldsymbol{C}$	$\boldsymbol{S}$	$\boldsymbol{R}$
$3 \ldots 30$	$\lceil 4.3 \cdot V \rceil$	3	1000

Table 2.1: Brute-force search benchmark #1 parameters

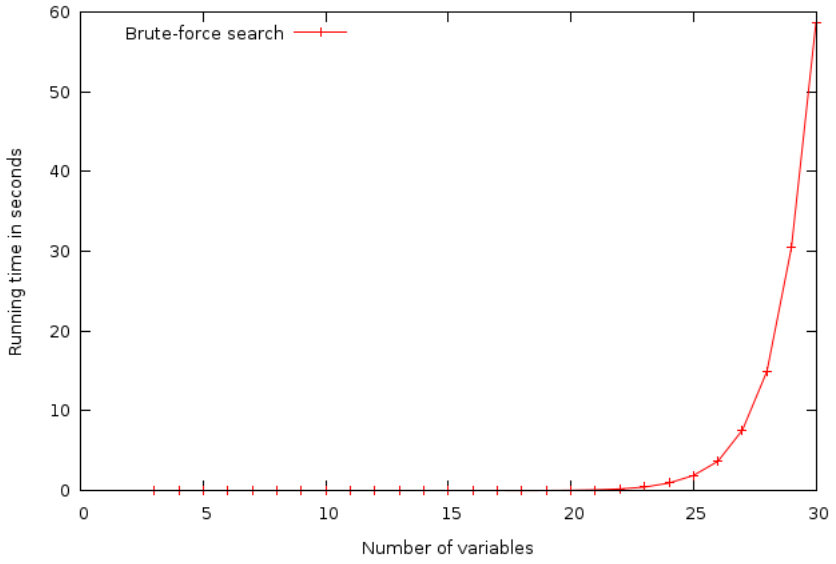

Figure 2.7: Running time of the brute-force search for increasing V (linear scale)

The curve seen in Figure 2.5.1 seems to be an exponential one, which conforms to the analysis which stated the brute-force search time complexity to be $O(2^N)$. This can be better seen if we reproject the curve on a logarithmic scale with the basis 2.

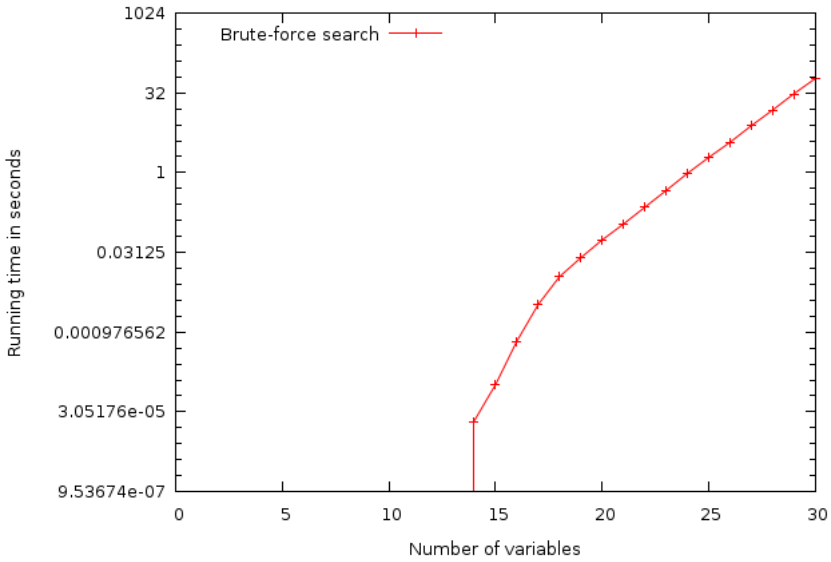

Figure 2.8: Running time of the brute-force search for increasing V (logarithmic scale)

On the logarithmic scale with basis 2, we get a line whose slope is 1, what suggests that the running time is $O(2^V)$. We know that the time complexity of brute-force search is $O(L \cdot 2^V)$. In this case, $L = S \cdot C = 3 \cdot 4.3 \cdot V$, so $L = O(V)$ and thus we may express the time complexity as $O(V \cdot 2^V)$. However, the influence of V factor is insignificant compared to the 2^V factor, so the curve on the graph is visually indistinguishable from 2^V. Of course, these estimations hold for large enough values of V. For small values of V, the running time is unmeasurably short and thus not even visible in the graph.

We also noted that an unsatisfiable formula is the worst case for the brute-force search. The running time for such a formula should be higher than that for any satisfiable formula of the same length and the same number of variables. Let us examine the running times for a set of equally-sized formulae. We will sort the times for individual runs in ascending order for a better visualization.

V	C	S	R
30	129	3	4000

Table 2.2: Brute-force search benchmark #2 parameters

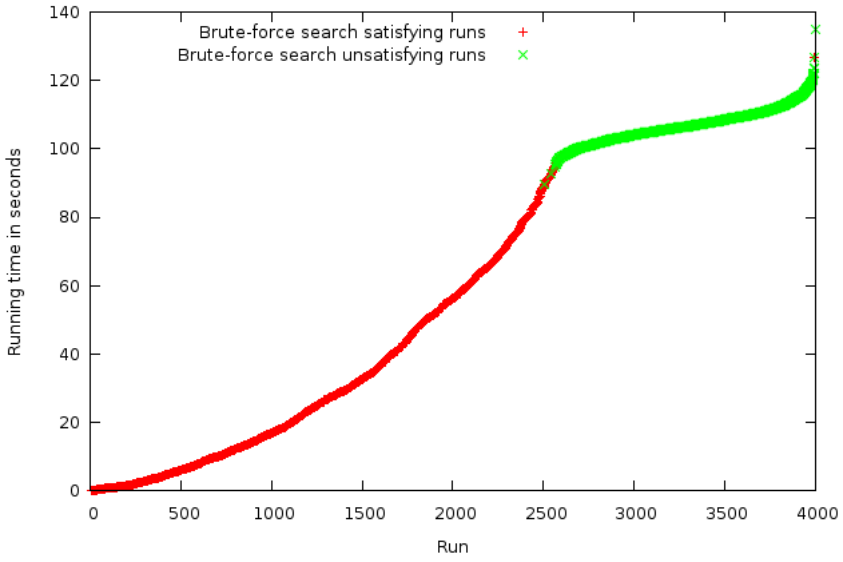

Figure 2.9: Running time of the brute-force search for different runs with constant V

The curve we see in this figure consists of a monotonically increasing segment and a constant segment (although the second segment is not exactly constant, this can be explained by small deviations of the clock ticks and processor performance of the benchmarking system). The constant segment comprises of the largest values and represents those runs when the input formula was unsatisfiable. The increasing segment, on the other hand, stands for the satisfying runs. While the running time for unsatisfiable formulae is always approximately the same, for satisfiable formulae it may be any value between that of an unsatisfying run and zero.

We can also see that for the given parameters, our method of random selection yields a higher probability of a formula being satisfiable than it being unsatisfiable. According to [17], this probability approaches $\frac{1}{2}$ as $V \to \infty$, if $S = 3$ and $C \approx 4.3 \cdot V$. This is the actual reason for our choice of S and C in the benchmarks. We chose $S = 3$ (i.e. 3-SAT) as the simplest NP-complete subclass of SAT and then set C accordingly. By having an approximately half probability of a formula being satisfiable, we can better examine the algorithm's performance in both cases without the curve being too skewed either way.

2.5.2 DPLL

We will continue by examining the DPLL algorithm. As it is much faster than the brute-force search, we can iterate V over a much larger range.

V	C	S	R
$3 \ldots 118$	$\lceil 4.3 \cdot V \rceil$	3	1000

Table 2.3: DPLL benchmark #1 parameters

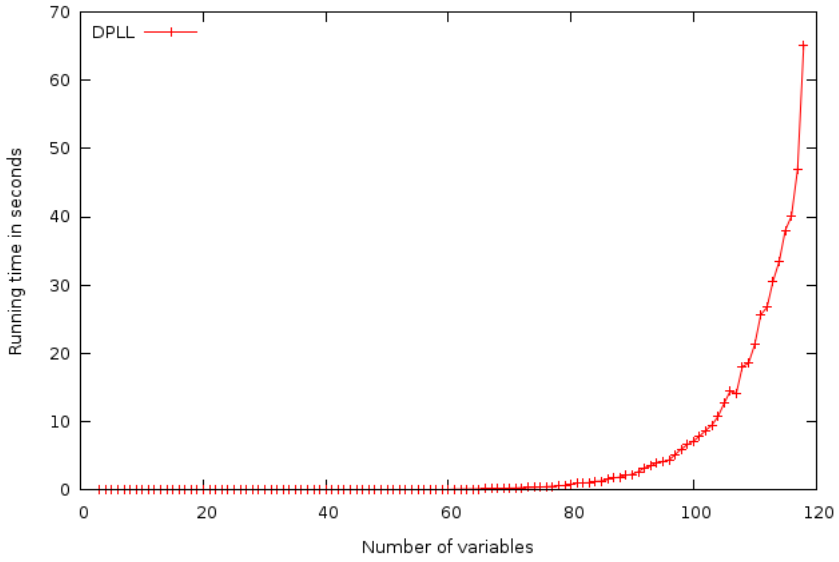

Figure 2.10: Running time of DPLL for increasing V (linear scale)

Although the splitting rule of DPLL generates an exponential number of cases, the other rules made it difficult for us to analyse the time complexity of DPLL. This benchmark suggests that the time complexity is indeed exponential. Let us prove that by reprojecting the curve to a logarithmic scale.

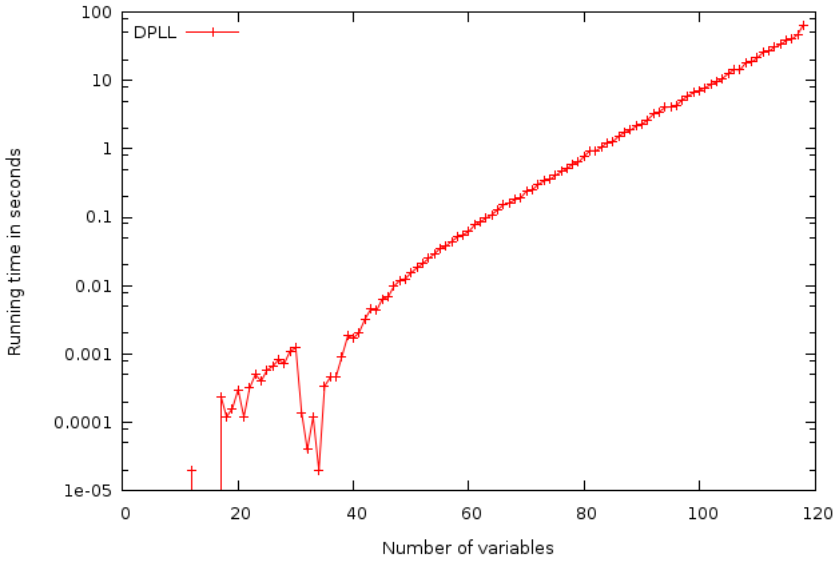

Figure 2.11: Running time of DPLL for increasing V (logarithmic scale)

Again, we get a linear curve, thus proving the time complexity of DPLL to be exponential. Note, however, the interesting fact that the base of this exponential curve is clearly less than 2. Let us now examine the differences in running time of DPLL for a set of equally-sized formulae.

$\boldsymbol{V}$	$\boldsymbol{C}$	$\boldsymbol{S}$	$\boldsymbol{R}$
100	430	3	10000

Table 2.4: DPLL benchmark #2 parameters

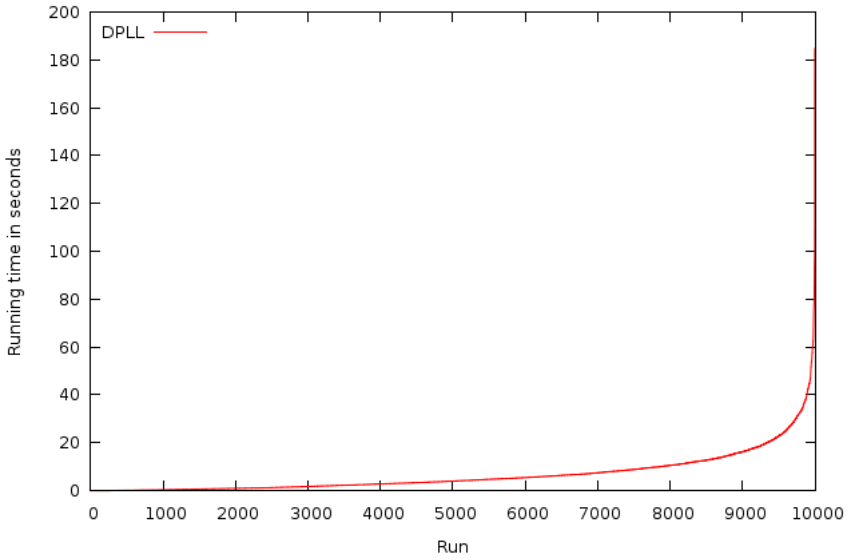

Figure 2.12: Running time of DPLL for different runs with constant V

Although in this case the ratio of unsatisfiable clauses to the satisfiable ones should be larger (closer to $\frac{1}{2}$ than in the brute-force search benchmark #2, there is no apparent constant segment in this case. It follows that although an unsatisfiable formula is theoretically the worst case for DPLL, even in such a case the running time may vary greatly. This benchmark resulted in 5320 formulae being satisfiable and 4680 formulae being unsatisfiable. Unlike in the brute-force algorithm benchmark, there is no apparent distribution of values such that the satisfying running times would all precede the running times of unsatysfing runs. They are well mixed together, although the extreme values at the end of the curve all belong to the unsatisfiable runs. We did not discern the satisfying and unsatisfying runs in the graph, as that would only lead to the curve being rendered as a melange of two colours.

2.5.3 Resolution method

We will now have a look at the performance of the resolution method. As this algorithm is very time and memory consuming, we only tested the algorithm on small formulae. We made two benchmarks. The first one had the usual parameters, while the second one used different parameters that allowed for slightly larger formulae. As the running time is marginal for a small number of variables but increases rapidly with increasing number of variables, it is not useful to visualize the results on a linear scale. We will therefore move straight to the logarithmic scale graphs.

$\boldsymbol{V}$	$\boldsymbol{C}$	$\boldsymbol{S}$	$\boldsymbol{R}$
$3 \ldots 8$	$\lceil 4.3 \cdot V \rceil$	3	1000

Table 2.5: Resolution method benchmark #1 parameters

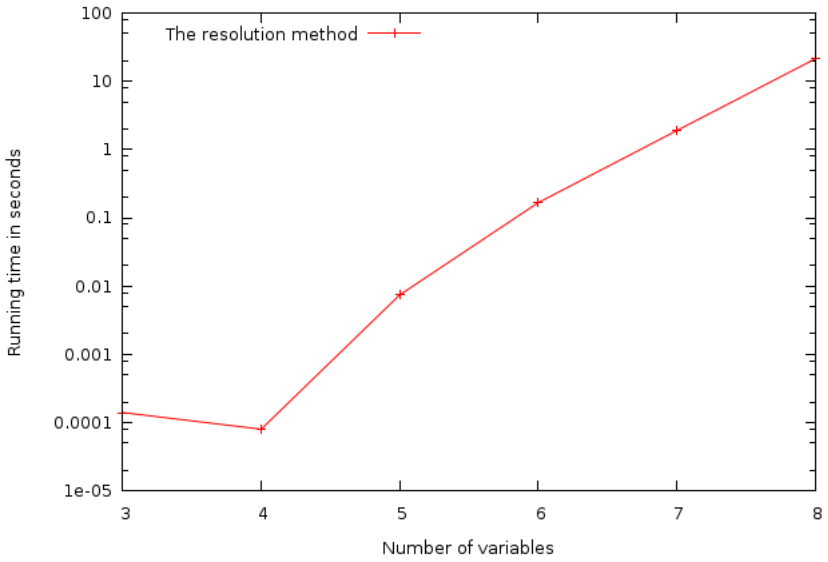

Figure 2.13: Running time of the resolution method for increasing V (benchmark #1, logarithmic scale)

$\boldsymbol{V}$	$\boldsymbol{C}$	$\boldsymbol{S}$	$\boldsymbol{R}$
$5 \ldots 10$	$2 \cdot V$	5	1000

Table 2.6: Resolution method benchmark #2 parameters

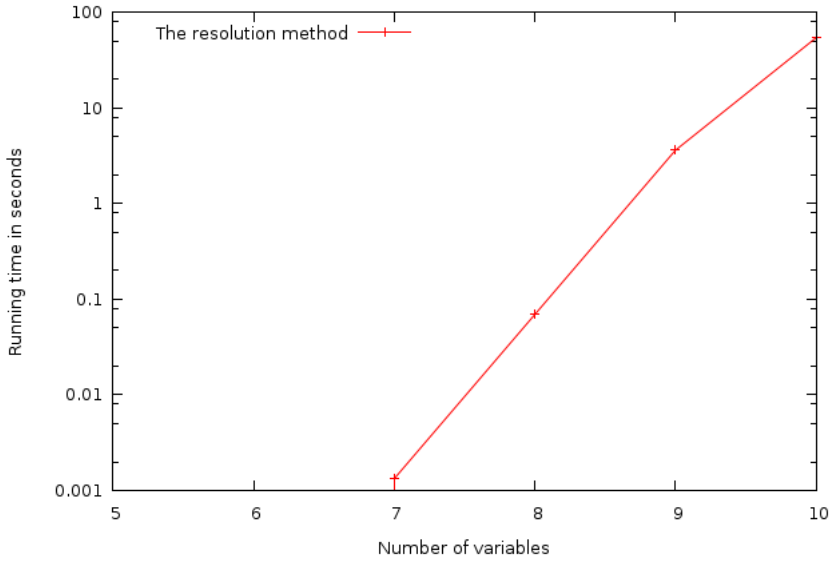

Figure 2.14: Running time of the resolution method for increasing V (benchmark #2, logarithmic scale)

At the first points of the first curve, the running time is marginally small, so even small fluctuations of timer and processor activity may cause a significant relative error. The first values of the second curve even came out as unmeasurably small. Later, however, both curves become approximately linear. We therefore postulate that the time complexity of the resolution method, in average case, is approximately exponential. However, as the data set is very small, it is statistically too insignificant for the task of verifying this hypothesis. Note that in [10], an exponential lower bound is given to the time complexity of the resolution method.

Let us again take a look at the differences of running times between a large number of runs with constant V.

V	C	S	R
8	35	3	1000

Table 2.7: Resolution method benchmark #3 parameters

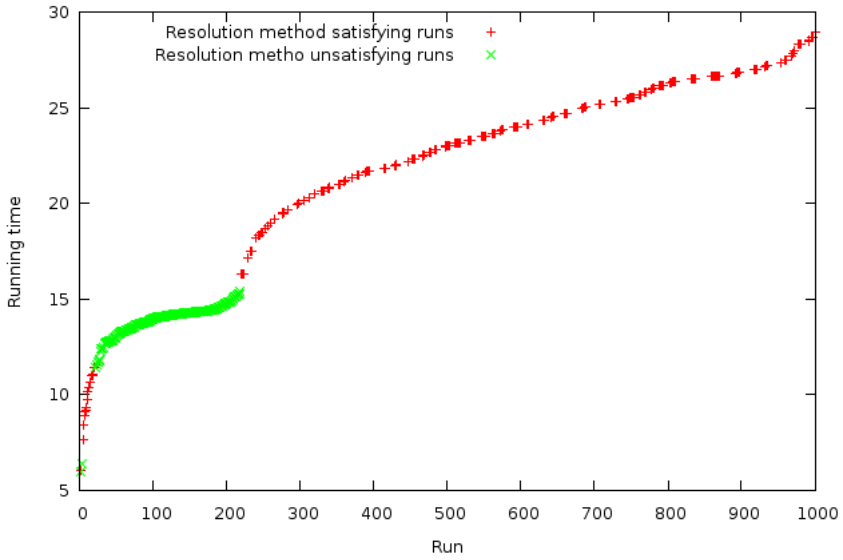

Figure 2.15: Running time of the resolution method for different runs with constant V

For $V = 8$, the probability of randomly generating a satisfiable formula is quite larger than that of generating an unsatisfiable one. In our graph,

- The short steeply increasing segment at the beginning of the curve consists of satisfying runs
- The following almost constant segment consists of unsatisfying runs
- The second steeply increasing segment and all the values beyond belong to satisfying runs

Apart from the first segment, this graph clearly illustrates the point that the resolution method (unlike most other algorithms) is faster for unsatisfiable formula than for the satisfying runs. This is because an usatisfying run of the algorithm only has to resolve an empty clause, while a satisfying run must exhaust all the possibilites of resolving the clauses. The latter, naturally, takes at least as much time as the former. The few cases in our graph that were declared satisfiable unusually quickly probably stand for degenerate cases that had little possibilities of resolution right from the beginning.

2.5.4 DNF conversion

The benchmarking methodology for DNF conversion is different from the previous cases, as this algorithm is concerned with formulae in a general form rather than CNF. We therefore generated the formulae by generating random parse

trees first. The process we used is as follows.

1. Create the root node, which is a binary operator. Create its two child nodes.
2. Starting at the root node, descend into the tree recursively. Every node encountered may be marked as
   - A binary operator with uniform probability of becoming $\wedge, \vee, \rightarrow$ or $\leftrightarrow$. Two child nodes are also created for this type of node.
   - A variable node. This node is a leaf. If there are no variable nodes present in the tree yet, a new variable is added to the variable list and used for this node. If the variable list is nonempty, a new variable is added with probability p. Otherwise, the variable used to mark this node is chosen uniformly among the existing variables.

   If the depth h of a node satisfies the condition $h \leq t$ for a given threshold constant t, the node is marked as a binary operator node. If $h > t$, the node is marked as a binary operator node with probability $b^{(h-t)}$ for a given constant b.
3. Finally, the tree is serialized by an in-order search and output. During this process, the output of every node may be prepended with the $\neg$ operator with probability q.

The process is to be run R times, generating a large number of formulae. For every generated formula, we will

1. transform it do DNF and evaluate its satisfiability in linear time.
2. transform it to CNF and evaluate its satisfiability by DPLL.

The parameters that we have chosen for this benchmarks are presented in the table below. The values were chosen experimentally in order for the formulae to be diverse enough, but not too large. Although at the first glance the values may seem very small, one must have in mind that they generate formulae complex enough to become very long upon transformation to CNF or DNF. In fact, our benchmarking system went out of memory during many such transformations.

t	p	q	b	R
2	0.4	0.5	0.85	30000

Table 2.8: DNF conversion benchmark parameters

The results of this benchmark are summarized below.

Total formulae generated	30000	(100.00%)
Both CNF and DNF conversions successful	18336	(61.12%)
Only CNF conversion successful	3478	(11.59%)
Only DNF conversion successful	3516	(11.72%)
Neither conversion successful	4670	(15.57%)

Table 2.9: DNF conversion benchmark successfulness results

As we can see, the ratio of formulae that went out of memory and were not successfully converted to at least one form was almost as high as 40%. Note that formulae that were successfully converted to CNF, but could not be converted to DNF are those, whose structure was relatively similar to CNF but very dissimilar to DNF (and vice-versa). The fact that the numbers of formulae that were only successfully converted to one of the forms is approximately equal. This supports our expectation that a randomly generated formula has equal probabilities of being similar to CNF and DNF, respectively. If this claim really holds, the expected time of both conversions would be the same. As the subsequent satisfiability analysis is faster for DNF, it would follow that the DNF conversion approach is faster than the CNF conversion one, what we in fact wanted to demonstrate by this benchmark. Note, however, that the results of this benchmark only hint, not prove, that the claim is true. Let us have a look at the actual benchmark times.

Total successful runs	18336	(100.00%)
CNF conversion slower than DNF conversion	9282	(50.62%)
CNF conversion faster than DNF conversion	8763	(47.79%)
Both conversions have the same running time	292	(1.59%)

Table 2.10: DNF conversion benchmark timing results #1

Average running time of CNF conversion in seconds	
only when DNF conversion was successful too	1.660273
all runs	2.422667
Average running time of DNF conversion in seconds	
only when CNF conversion was successful too	1.598815
all runs	2.392131

Table 2.11: DNF conversion benchmark timing results #2

As expected, DNF conversion was faster than CNF conversion in more times than vice-versa. The average running time for DNF conversion is lower than that of CNF conversion. Unfortunately, the differences are not large enough to be conclusive. One must have in mind that only relatively small formulae were successfully converted and such formulae are small enough for DPLL to process

them quickly. This disallowed us to see the difference between the $O(L)$ and $O(L \cdot 2^V)$ running times. However, our reasoning from the previous paragraph suggests that the superiority of the DNF conversion approach in our benchmark was not merely a statistical error, but rather a result that could be proven using a larger benchmark.

Chapter 3

The heuristics

In the previous chapter, we studied complete algorithms that could find a solution (or at least prove one exists) or prove that the formula is unsatisfiable. But the SAT problem can also be regarded as an optimization problem. We want to find a valuation which yields the highest amount of satisfied clauses. Unlike other optimization problems, however, we do not accept a suboptimal solution. If the given formula is satisfiable, finding a valuation that satisfies all clauses but one is just as good as finding one that satisfies none of them. If we do not reach such a solution, we do not know whether there is no satisfying valuation, or it is just our algorithm that searched the wrong portion of the ever vast solution space. The approach to SAT as an optimization problem is therefore applicable mainly if we **know** there **is** a solution and we want to find it.

The heuristic algorithms usually search the solution space with use of greediness or randomness, usually both. We always start with a random valuation and we want to make changes in it that improve the number of clauses it satisfies. If the optimal valuation satisfying all the clauses is not reached within a given time (or a given amount of steps), we restart the search with another random initialization. If a solution is not found after a given number of tries, we usually give up. This way, we only search a small portion of the vast solution space. This makes the heuristics considerably faster than the complete algorithms. Therefore, heuristics can be used on formulas of a size so large that it is unfathomable even for relatively fast algorithms such as DPLL. On the other hand, searching only a part of the solution space makes it possible for us to simply miss an optimal solution. This is the disadvantage of heuristics in comparison with the complete algorithms - if the search ends unsuccessfully, we do not know if the formula was satisfiable or not. What we know, however, is that the formula is somewhat "hard to satisfy". Maybe it has a negligible amount of satisfying valuations, so there was only a little chance to find one after several random restarts. Or it may have such a structure, that the valuations are not findable by a greedy approach.

Usual optimization algorithms, such as hill-climbing or simulated annealing are applicable for SAT as well. However, we are going to discuss some heuristic algorithms that are specific to SAT. These algorithms are GSAT and its later improvement WalkSat, presented by Bart Selman, Henry Kautz and Bram Cohen in [17].

3.1 GSAT

GSAT is a hillclimbing-like heuristic. The algorithm starts by generating a random valuation v. We are going to improve this valuation by *flipping* variables. We say that valuation v' is acquired from v by flipping the variable A, if $v'(B) = v(B)$ for every variable B except A and $v'(A) \neq v(A)$.

In every step, we evaluate how many clauses are satisfied by the current valuation v. Then, we make a list of all variables that are present in the unsatisfied clauses. This list is to be without duplicates - if a variable occurs in several unsatisfied clauses, it is still only listed once. Now for any variable A on the list, some of the unsatisfied clauses must contain either A while $v(A) = 0$, or $\neg A$ while $v(A) = 1$. In both cases, flipping A will make that literal - and therefore also aforementioned clauses - satisfied. This means that flipping any variable A on the list will satisfy all the previously unsatisfied clauses that contain A. However, there is also an opposite effect. If there is a clause such that A (or $\neg A$) is the only satisfied literal in that clause, then flipping A would unsatisfy it. Thus, for every variable A on the list, we can compute its "net gain" - the number of clauses that will become satisfied by flipping A decreased by the number of those clauses that will become unsatisfied by doing so. We will then flip the variable with the highest net gain. If there are more variables with the same net gain, we will choose randomly. We acquire a new valuation which is either a solution, if it satisfies all clauses, or, if it does not, we will proceed to the next step.

As has been already mentioned in the introduction of this chapter, we may limit the number of steps to be executed and if solution is not found until this limit is reached, we restart the algorithm with another randomly chosen valuation v. In practice, the success of heuristics is often determined by good tweaking of various constants they are parametrized by. This limit is also such an important parameter. The usual behaviour of GSAT is that in the first steps, it rapidly decreases the number of unsatisfied variables, until it reaches a so-called *plateau*. This means that the algorithm stagnates, as the highest net gain is zero. It may be therefore reasonable to estimate how quickly this happens on average (as a function of the number of variables, clauses etc.) and set the limit accordingly, so that if the algorithm is unable to descend from a plateau after a large number of steps, we end its fruitless effort. However, it must be noted at this point that it **is** possible to escape a plateau. Even a step that brings a zero net gain does change the valuation and therefore results in a

different list of variables occuring in unsatisfied clauses. In this new list, there may be a variable with nonzero gain. Even if a certain valuation occurs twice while plateauing, it does not necessarily mean an infinite cycle, because the process of choosing the to-be-flipped variable is not deterministic - the variable is chosen randomly among all the candidates that share the optimal net gain. Sometimes, it may even occur that the highest net gain is negative. This, too, is no exception to the rule - flip the variable with the highest net gain. Although such move does not improve the valuation, it is still a way of escaping a plateau and, after all, if we have to do so, there is no better possibility in the first place.

A common problem with greedy optimalization algorithms is that they are unable to get out of local minima (if we regard the problem as one of minimalizing the number of unsatisfied clauses). The plateaux are exactly this same phenomenon, though they got their name from the fact that the number of unsatisfied clauses remain constant for a long time, making a "plateau" appear in the graph of unsatisfied clauses over the algorithm's running time. Additional methods of escaping these minima can be used to improve GSAT. One such improvement is *random walk*. In every step, with a given probability p, we flip a random variable from the list instead of the locally optimal one. This means we still work towards satisfying the unsatisfied clauses, but allowing a choice that is not locally optimal makes it possible to satisfy clauses that would maybe never be chosen with greedy approach. For example, it would be those clauses whose satisfaction requires a negative net gain. However, if being "stuck" in a local minimum, even such negative moves are necessary to get out.

Another interesting modification to GSAT is specifying a *weight* for clauses. We initialize all weights to 0. In each step, we increment the weight of every unsatisfied clause. Then, instead of flipping a variable that results in the least unsatisfied clauses, we flip the one that results in the lowest net weight of unsatisfied clauses. This forces us to focus on satisfying the "heaviest" clauses first. The heaviest clauses are those whose weights have been incremented the most, which means they have been unsatisfied for the longest time throughout the algorithm's run. Consider, what is the effect of this in terms of escaping the minima. At first, the number of unsatisfied clauses drops rapidly like in regular GSAT, as all weights are low and do not do much difference. Usually, the vast majority of variables will already be satisfied until GSAT starts plateauing. Although the algorithm may consider many valuations during plateauing, there is always only one-variable difference between two consecutive valuations. For this, there will probably be large intersections between the sets of satisfied clauses of all the valuations considered during the plateauing. It follows that there will exist some clauses that will rarely become satisfied during the plateauing and their weights will increase. As soon as their weights exceed a certain threshold, weighted GSAT will focus on satisfying them first, even if it means unsatisfying a large number of previously satisfied clauses. Thus, they will serve as a motivation to abandon the plateau and even increase the number of unsatisfied clauses, if it means satisfying a higher net weight. We can

say that GSAT somewhat helps us identify the clauses that are "hard to satisfy".

3.2 WalkSat

The WalkSat algorithm is the result of GSAT authors' experimentation with random walk on GSAT. WalkSat has the same overall scheme as GSAT. We start with a randomly initialized valuation and try to improve it by flipping variables, one by one, in a sequence of steps. We may do this for a limited amount of steps, then we restart the algorithm with a new random initialization. We may also set a limited amount of restarts. The difference, however, is how these algorithms choose the variable to be flipped in every step.

While GSAT is rather greedy algorithm that may use random walk to mediate its greediness, WalkSat has somewhat of an opposite approach: In every step, we use the randomness first and the greediness second. We choose a random clause unsatisfied by the current valuation, then pick a variable from within that clause whose flipping will results in the least number of unsatisfied clauses. A fully random modification is also used - in that case, even the variable from the selected clause is picked at random.

An interesting fact, which has been emphasized in [17], is that the fully-random WalkSat is not equivalent to GSAT with random walk probability $p = 1$. The similarity is, that in both cases, only those variables that occur in the unsatisfied clauses may be chosen. In GSAT, this is ensured by building the list of such variables, while in WalkSat it is done by first choosing an unsatisfied clause. However, as the variables list in GSAT is without duplicates, every variable in the list has equal chance of being chosen (rembember, that for $p = 1$ we never evaluate their net gain). In WalkSat, though, the random process consists of two parts - choosing a clause, then a variable within - thus variables occuring in many clauses have higher probability of being chosen.

Note that a single step in WalkSat is faster than in GSAT, as it consists of less operations. Both of them keep track of satisfied and unsatisfied clauses, but GSAT must also make the list of candidates to flip. The elimination of duplicates would require an implementation of set. This brings search, insert and delete time either $O(log\ n)$ with a binary search tree, or an expected $O(1)$ with a hashtable. Although the latter may be rather effective with a large enough hashtable to avoid collisions, it is still an overhead in number of simple operations per step and will probably manifest itself when running thousands of steps. It is also possible to represent a set using an array of boolean values. This method is simple to implement, but may be impractical if the list is short, as the array takes linear time to both search and clean.

WalkSat may be used in tandem with DPLL as described in [9]. This is a

way to combine a fast heuristic with a fast complete algorithm into an advanced hybrid algorithm that may profit from the advantages of both its components.

3.3 Benchmarks

The International SAT Solver Competitions [12] are an anually held event for implementers of SAT solvers. The competition is a satellite event of the annual SAT conference and helps to mark the progress of modern approaches to SAT in practice. The performance of competing solvers is measured on benchmarks that can be submitted to the competition. The benchmarks are divided into three categories, as described in [12].

- *Random.* Randomly generated uniform k-SAT formulae. Note that these resemble our benchmarks in Section 2.5.
- *Crafted.* Formulae designed to be especially difficult for SAT solvers, or representing difficult problems like puzzle games.
- *Application (a.k.a. industrial).* Often very large formulae representing application problems.

Existing benchmarks used in past competitions can be found at the SAT Competitions webpage [1] or the Satisfiability Library [11].

The random benchmarks are usually sets of formulae randomly generated with given parameters, like the number of variables V, the ratio of the number of clauses and the number of variables R and sometimes the *backbone* size. The backbone of a formula ϕ is the set of all literals L such that $\phi \wedge \neg L$ is unsatisfiable [11]. Equivalently, it can be defined as the set of all variables whose value is constant over all the satisifying valuations. For WalkSat, it has been found out that formulae with smaller backbone are easier to solve [18].

The crafted benchmarks may include formulae representing problems coming from games such as Sudoku, Battleship or Towers of Hanoi. Others represent CNF-encoded hard instances of other computational problems, like graph coloring, or the high degree subgraph isomorphism problem. Some crafted benchmarks are generated randomly - however, not with uniform distribution, but rather selecting the more difficult ones (e.g. the backbone size could be one criterion for this) [11] [1].

The application benchmarks may comprise of instances from real world problems such as attacking AES cipher, program verification, logistic planning, AI planning and many others [1].

We have used Bart Selman and Henry Kautz's implementation of WalkSAT [16] with the benchmark instances from the 2011 SAT competition [1] to

demonstrate the performance of modern heuristics. Here, we present the results. The result for each benchmark will be formatted into a table. Every table will include

1. Benchmark name and type.
2. The characteristics of formulae in that benchmark. This includes the numbers of variables and clauses, and also clause sizes. As these parameters may vary greatly, usually an interval will be given. This does not mean that there is a formula in the set for every parameter value from within the interval.
3. Total number of formulae per benchmark.
4. Number of formulae that were successfully solved in at least one try.
5. Number of tries WalkSat makes for each formula. Some formulae are easy and will be satisfied in every try. Others are difficult and are only satisfied if a favorable random valuation initialization is generated at the beginning of the try.
6. The average number of satisfying tries over all the satisfying runs from the whole benchmark.
7. The minimum and maximum running times of a satisfying run. This is a sum of running times of all tries the run consists of. Note that the total running time divided by a number of tries is not an equivalent information, as satisfying tries take less time than the unsatisfying ones. This is because the former ones halt as soon as they find a satisfying valuation, while the latter only do so when they reach the maximum amount of flips.

Let us proceed with the results themselves.

Benchmark name and type: *medium* (random)	
Number of variables (V)	$50 \ldots 800$
Number of clauses (C)	$1065 \ldots 8010$
Clause sizes (S)	$3 \ldots 7$
Total number of formulae	400
Number of formulae solved	170
Number of tries per formula	1000
Average number of satisfying tries	260.98
Minimum running time of a satisfying try in seconds	4.84
Maximum running time of a satisfying try in seconds	731.96

Table 3.1: WalkSat random benchmark #1 results

Having 1000 runs at its disposal, WalkSAT solved a large number of the medium-sized randomly generated formulae. The number of satisfying tries

ranged from under 10 for large values of V through $100 \ldots 300$ for most cases to 1000 for the simplest ones. Considering the DPLL benchmark in subsection 2.5.2, we can see that there are SAT instances which WalkSat can solve in a few seconds, while using DPLL would take an enormous amount of time. Also note that solving only 170 of 400 instances does not mean that WalkSat failed for 230 formulae. As the formulae are generated randomly, many of them are actually unsatisfiable and WalkSat is simply not designed to find that out.

Benchmark name and type: *large* (random)	
Number of variables (V)	$150 \ldots 50000$
Number of clauses (C)	$10500 \ldots 210000$
Clause sizes (S)	$3 \ldots 7$
Total number of formulae	200
Number of formulae solved	1
Number of tries per formula	1000
Average number of satisfying tries	17
Minimum running time of a satisfying try in seconds	67.820004
Maximum running time of a satisfying try in seconds	67.820004

Table 3.2: WalkSat random benchmark #2 results

As we can see, 1000 tries per formula was not enough for larger formulae. This time, only a single formula was satisfied and it was a relatively small one. Its parameters were $V = 2500, C = 10500, S = 3$. This formula must have been easy in its structure, as it was satisfied in 17 tries of 1000, while other formulae of the same size were not at all.

Benchmark name and type: *skvortsov/battleship* (crafted)	
Number of variables (V)	$40 \ldots 1368$
Number of clauses (C)	$105 \ldots 16308$
Clause sizes (S)	$2 \ldots 57$
Total number of formulae	24
Number of formulae solved	14
Number of tries per formula	1000
Average number of satisfying tries	617.642857
Minimum running time of a satisfying try in seconds	0.10
Maximum running time of a satisfying try in seconds	105.46

Table 3.3: WalkSat crafted benchmark #1 results

Let us now have a look at this example of a crafted benchmark. Unlike the random benchmarks, the clause sizes in this case are not uniform. In every formula, most of the clauses have two members, while several of them are

much larger (up to 57). As 10 formulae are known to be unsatisfiable, WalkSat succeeded for 100% of the satisfiable ones. Furthermore, the average number of satisfying tries is well over 50%, so we can say that this benchmark was very easy for WalkSat.

Benchmark name and type: *spence/sgen* (crafted)	
Number of variables (V)	$120 \dots 300$
Number of clauses (C)	$252 \dots 720$
Clause sizes (S)	$2 \dots 5$
Total number of formulae	19
Number of formulae solved	3
Number of tries per formula	1000
Average number of satisfying tries	11.33
Minimum running time of a satisfying try in seconds	35.02
Maximum running time of a satisfying try in seconds	35.36

Table 3.4: WalkSat crafted benchmark #2 results

This crafted benchmark consists of a 19 formulae generated in such a way that they would be difficult. 10 of them are known to be satisfiable, while WalkSat only solved the smallest 3 for $V = 120, 130, 140$ respectively. For comparison, in the previous benchmark the running time for a formula with $V = 120$ was 0.18 seconds, what is about 200 times faster. The average number of satisfying tries is also considerably lower. We can thus conclude that this benchmark was much more difficult.

Benchmark name and type: *skvortsov/automata-synchronization* (crafted)	
Number of variables (V)	$1287 \dots 63882$
Number of clauses (C)	$2332 \dots 123981$
Clause sizes (S)	$2 \dots 3$
Total number of formulae	12
Number of formulae solved	0
Number of tries per formula	1000
Average number of satisfying tries	–
Minimum running time of a satisfying try in seconds	–
Maximum running time of a satisfying try in seconds	–

Table 3.5: WalkSat crafted benchmark #3 results

In this benchmark, we deal with formulae that are almost in 3-CNF, but contain several two-member clauses as well. 7 formulae were satisfiable, but the 1000 runs WalkSat had at its disposal were not enough to solve any. Note well that this benchmark was much larger in terms of V, C than the previous ones.

Benchmark name and type: *anton/SRHD-SGI* (crafted)	
Number of variables (V)	$545 \ldots 3672$
Number of clauses (C)	$29734 \ldots 1163952$
Clause sizes (S)	$2 \ldots 62$
Total number of formulae	28
Number of formulae solved	4
Number of tries per formula	1000
Average number of satisfying tries	1.75
Minimum running time of a satisfying try in seconds	373.04
Maximum running time of a satisfying try in seconds	758.00

Table 3.6: WalkSat crafted benchmark #4 results

The last of the benchmarks we tested comes from the high degree subgraph isomorphism problem. The formulae had structure similar to those in the **battleship** benchmark (Table 3.3) - most of the clauses have two members and the remaining few are much larger. Having allowed 1000 tries, WalkSat managed to satisfy the smallest formula in 4 tries, and 3 others among the several smallest ones in 1 try. Note also that even these smallest formulae took minutes to solve.

Benchmark name and type: *kullmann/AES/32* (crafted)	
Number of variables (V)	$300 \ldots 1116$
Number of clauses (C)	$1016 \ldots 4312$
Clause sizes (S)	$1 \ldots 5$
Total number of formulae	5
Number of formulae solved	1
Number of tries per formula	10000
Average number of satisfying tries	1
Minimum running time of a satisfying try in seconds	4409.92
Maximum running time of a satisfying try in seconds	4409.92

Table 3.7: WalkSat application benchmark #1 results

Finally, we present an application benchmark. This one is derived from the problem of finding the key in AES cipher. At first, we allowed 1000 tries as in the previous benchmarks, but the search was not successful. Among 10000 tries, one (but only one) succeeded. Such a result indicates that this benchmark is extremely difficult for WalkSat. Note that the values of V, C and S are relatively small compared to most other benchmarks. This means that the difficulty lies in the structure of the problem rather than in its size.

We experimented with larger application benchmarks, but to no effort. For difficult benchmarks with a large number of variables V and only a small number of solutions of scattered in the huge solution space, the stochastic approach of

WalkSat was not successful (at least not in the amount of time we were willing to allow). Note that these benchmarks are built for modern SAT solvers that are usually not heuristic-only, but are rather based on a modification DPLL (which is usually referred to as Conflict-Driven Clause Learning)[12]. The results of modern SAT solvers on the aforementioned benchmarks and others can be found at [1].

Chapter 4

Special case algorithms

In this last chapter, we are going to examine certain "easy" classes of boolean formulae. These formulae have a simple structure, taking advantage of which, we can design a *polynomial* complete SAT algorithm for them. Naturally, such special-case algorithms will fare better on their respective formula classes than the general complete algorithms. On the other hand, this is not so certain about heuristics. The special case algorithms base their effectivity on understanding the underlying structure of their respective class of formulae, while the heuristics do not have any such insight. However, the structure is usually so simple that the greedy approach which heuristics utilize may work very well for them.

4.1 2-SAT

In Chapter 1, we explained that 2-CNF is a constrained form of CNF, where every clause consists of two literals. We may or may not take this this definition strictly. If a clause has only one literal L, then we can rewrite it equivalently as a clause with two literals in the form $(L \vee L)$. Therefore, if we consider the two possible definitions of 2-CNF

1. A formula in CNF, where every clause has **exactly** two literals
2. A formula in CNF, where every clause has **at most** two literals

we have just shown that for every formula satisfying the second condition there exists an equivalent (and easily constructible) formula satisfying the first one. The opposite inclusion holds trivially. Therefore, the problems of satisfiability for these two classes of formulae are equivalent. In practice, it is convient to use the second definition. The reason is that a single-literal clause (L) already fixes the value of the variable contained inside. As any valuation satisfying the formula must adhere to this fixation, the clause (L) may never obtain another value. Thus, it may be taken off the formula, as it does not have to be considered anymore throughout the algorithm.

The problem of satisfiability of a formula in 2-CNF is called 2-SAT. Aspvall, Plass and Tarjan proposed an elegant algorithm to solve 2-SAT in linear time in [4]. We will describe their algorithm in the following paragraphs. For simplicity, we will adhere to the former definition of 2-CNF, as this one is more regular. However, only small changes have to be made in order for this algorithm to respect the latter one.

It can be easily proven that a two-literal clause $(L_1 \vee L_2)$ is equivalent to implication $(\neg L_1 \to L_2)$ as well as $(\neg L_2 \to L_1)$. Let us therefore represent a formula in 2-CNF as an oriented graph. For every variable, there will be two vertices standing for its positive and negative literal. For every clause, there will be two directed edges corresponding to the two implications. For example, consider the formula

$$(A \vee \neg B) \wedge (B \vee C) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee \neg D) \wedge (D \vee \neg E) \wedge$$
$$\wedge (F \vee \neg F) \wedge (D \vee F) \wedge (\neg E \vee F) \wedge (G \vee G)$$

which is represented by the following graph.

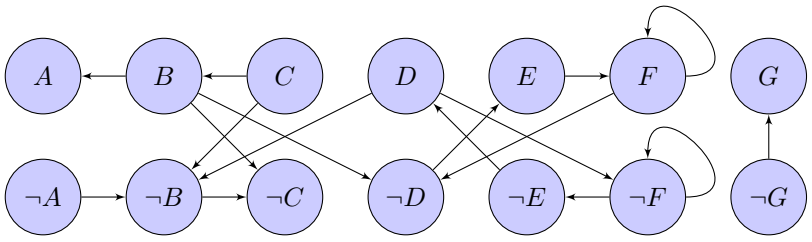

Figure 4.1: An example of a 2-SAT graph

Note that this graph is almost symmetrical. The only exceptions are clauses containing two literals of the same variable. While the clause $(F \vee \neg F)$ manifests as two loops, the clause $(G \vee G)$ only produces one edge (in other words, the two edges it would produce are identical). We will see later that this is exactly in accordance with the fact that the former of these two clauses is satisfied regardless of $v(F)$ and the latter, on the other hand, enforces that $v(G) = 1$.

We now have to find the strongly connected components (SCC) of this graph.

Definition 4.1 (Strong connection)**.** *We say that a graph $G(V, E)$ is strongly connected, if for every $v_1, v_2 \in V$ there exists a path in G that starts in v_1 and ends in v_2.*

Definition 4.2 (Strongly connected component)**.** *A* strongly connected component *(abbreviated* SCC*) of a graph G is its every maximal (with respect to inclusion) strongly connected subgraph.*

It can be easily seen that the membership of two vertices to the same SCC is a relation of equivalence.

Definition 4.3 (Induced SCC)**.** *Let ϕ be a formula in 2-CNF. Let G be the graph of ϕ. Let L be a literal in ϕ and v_L its representation in G. We then say that a SCC of G that contains v_L is an SCC* induced *by v_L.*

We will write $[v]$ to denote the SCC of a vertex v.

Definition 4.4 (Dual SCCs)**.** *We say that $[v_L]$ and $[v_{\neg L}]$ are* dual *SCCs.*

Theorem 4.1. *The relation of dual SCCs is well defined. I.e. if $[v_L]$ and $[v_{\neg L}]$ are dual and $[v_{L'}] = [v_L]$, then $[v_{L'}]$ and $[v_{\neg L}]$ are dual as well.*

Proof. If $[v_{L'}] = [v_L]$, it means that v_L and $v_{L'}$ are in the same SCC. If there exists an edge from v_{L_1} to v_{L_2} for some literals L_1, L_2, then by definition $L_1 \to L_2$. However, the $\to$ operator is transitive. A path is in fact an element of the transitive closure of the oriented edges, as a relation on the vertices of the graph. As v_L and $v_{L'}$ are in the same SCC, they are connected by an oriented path in both ways. It follows that $L \leftrightarrow L'$ and equivalently $\neg L \leftrightarrow \neg L'$. This means that $v_{\neg L}$ and $v_{\neg L'}$ are in the same SCC. Thus, $[v_{\neg L}] = [v_{\neg L'}]$. This entails that $[v_{L'}]$ and $[v_{\neg L}]$ are dual. □

After we find the SCCs, we contract them to construct a new graph G_{SCC}. This means the vertices of G_{SCC} will stand for the SCCs of the original graph G, and there will be an edge from $[v_L]$ to $[v'_L]$ for every edge from L to L' in G. Now, suppose there was a cycle $C_1, C_2, \ldots, C_N, C_1$ of vertices in G_{SCC} (they are vertices in G_{SCC}, but SCCs in G; that is why we will use capital C instead of the usual v to denote them). Then $C_1 \cup C_2 \cup C_n$ would be a strongly connected subgraph of G as well, what contradicts the maximality of SCCs $C_1, C_2, \ldots, C_n$. Therefore, G_{SCC} is a directed acyclic graph (DAG). Let us continue with our example.

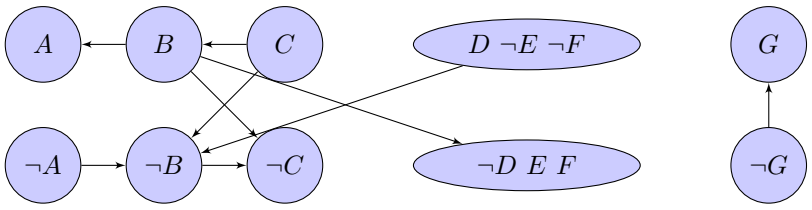

Figure 4.2: An example of a 2-SAT graph with SCCs contracted

If for any literal L it holds that $[v_L] = [v_{\neg L}]$, the formula is unsatisfiable, because $L \leftrightarrow \neg L$ is a contradiction. Otherwise, the formula is satisfiable and we will show how to find a satisfying valuation. Let us now consider G again. For every its component (not strongly connected, only connected), one of the two possibilites holds.

- If this component contains both vertices $v_L, v_{\neg L}$ for some literal L, then every vertex in this component has the opposite literal vertex in it. This can be easily seen from the fact that if there exists an unoriented path between two vertices v_A, v_B, then an analogous path exists between $v_{\neg A}, v_{\neg B}$, only with opposite vertices and edge orientations reversed.
- If the previous does not hold, then from the same observation about unoriented paths it holds that if some vertex v_A is connected with v_L, then $v_{\neg A}$ is connected with $v_{\neg L}$. Thus, the components of v_L and $v_{\neg L}$ can be obtained from each other by replacing the vertices by their opposites and reorienting the edges.

In the first case, every pair of opposite literal vertices is in the same component. The edges in this component therefore represent a set of clauses that does not share variables with the rest of the clauses in the formula, so it is unrelated to them and may be processed separately when constructing a valuation. In the second case, the set of all vertices opposite to those in one component forms another component. However, those clauses represented by the two components are again unrelated to the rest of the formula. By constructing a valuation for the literals in one of these components, we also construct the valuation for the another, as they contain the same variables. It follows that we may proceed component by component when constructing the valuation.

It is not difficult to construct the valuation in the second case, where v_L and $v_{\neg L}$ are in different components. For example, we may set $v(L') = 1$ if $v_{L'}$ is connected with v_L (i.e. $v(A) = 1$ if literal L is of the form A and $v(A) = 0$ if it is of the form $\neg A$) and $v(L') = 0$ otherwise. This way, all vertices in one component will have their respective literal valued 1, and in the other one, 0. Therefore, all oriented edges in these components would connect vertices with literal valuations $1 \to 1$ or $0 \to 0$. However, the oriented edges are representations of implications equivalent with individual clauses of the formula. As implication $\phi \to \psi$, where $v(\phi) = v(\psi)$ is satisfied, all clauses associated with the edges in these components are satisfied. Furthermore, as we already pointed out, setting values in a pair of opposite components does not interfere with satisfiability of the rest of the formula.

Constructing a valuation for the other case case, where the same component always contains both L and $\neg L$, is more difficult. In the previous case, we could assign all the literals of vertices in one component the same value. This is impossible here, as $v(L)$ and $v(\neg L)$ are in the same component and they must have opposite values. However, we can observe that all vertices in one SCC must be

assigned the same value for their respective literals. If $v(L_1) = 1$, $v(L_2) = 0$ and they are in the same SCC, then there is a path from v_{L_1} to v_{L_2} that must, at some point, contain an edge that leads from 1-valued to 0-valued vertex. This, however, is a contradiction, as oriented edges stand for implications and such an implication would be unsatisfied. Ultimately, the clause that is equivalent with this implication would be unsatisfied as well. This means that instead of assigning a value to individual vertices, we will rather assign values to whole SCCs. We will therefore have to examine G_{SCC} again. The component we were considering in G maps to a component of G_{SCC}, as contracting edges does not introduce new connections.

Definition 4.5 (Topological ordering)**.** *Let G be a directed acyclic graph, V the set of its vertices and E the set of its edges. Let $\leq_T$ be any ordering on V, such that for every edge $e = (u, v) \in E$ it holds that $u \leq_T v$.*

In other words, if $C_1, C_2, \ldots, C_n$ is a list of all vertices of a given component of G_{SCC}, then there exists no edge (C_i, C_j) such that $j < i$. A topological ordering exists for every directed acyclic graph (and, trivially, does not exist for cyclic ones).

Once the considered component of G_{SCC} is topologically sorted, we can process its vertices (i.e. the SCCs of G) in the reverse topological order. For every vertex $[v_L]$ whose literals have not yet been assigned a value, we set $v(L) = 1$. Note that this assignment is also reflected on the vertex $[v_{\neg L}]$. Due to the symmetricity of the graph, every assignment on side is accompanied by an opposite assignment on the other side. One can easily show that assigning the vertices this way will result in all edges leading between vertices valued $0 \to 0$, $1 \to 1$ and $0 \to 1$, again satisfying their respective clauses. Naturally, as we assign values to whole SCCs, all edges that have been contracted by construction of G_{SCC} will lead between two vertices of the same value, so their respective clauses are satisfied as well.

For discovering the strongly connected components, one can use Tarjan's algorithm (description can be found in [22]). A description of topological sort can be found in [6]. Both of these graph algorithms have time complexity $O(V + E)$, where V is the number of vertices and E the number of edges in the graph. In our case, V is twice the number of variables in the input formula and E is twice the number of its clauses. Therefore, both are linear in the length of formula L. Furthermore, the construction of graph in the beginning of the algorithm and the values assignment in its final stage both require only a single linear sweep over the formula. This entails that the whole algorithm has a linear time complexity.

4.2 HornSAT

A Horn clause, named after Alfred Horn who researched their properties, is a clause that has at most one positive literal. Such a literal is then called the head of the clause, while the negative literals form the body thereof. A Horn formula is a formula in CNF such that every its clause is a Horn clause. The problem of satisfiability of horn formulae is called HornSAT.

Just like for 2-SAT, there exists an algorithm with linear running time form HornSAT. In this case, the algorithm is very straightforward and much simpler than for 2-SAT. It is called the unit-propagation algorithm, as it takes advantage of the unit clauses, which are clauses that only have head, i.e. consist of a single positive literal. First, assume there is no pathological case of an empty clause (which would immediately entail an unsatisfiable formula). We can then determine the satisfiability of the given formula ϕ using the following steps.

First, let us determine if there is a unit clause (let it consist of a literal A). If there is one, then it must hold that $v(A) = 1$, or this clause would not be satisfied. Furthermore,

- all other clauses with A as their head become satisfied too, so they can be removed from the consideration. In other words, we can remove these clauses from the formula, thus receiving a new formula ϕ' that is equisatisfiable to the original ϕ. If it occurs that ϕ' is an empty formula, it means we have satisfied all the clauses and thus ϕ is satisfiable. Other explanation is that ϕ' is satisfied per se, as an empty conjunction always evaluates to 1.
- all clauses containing $\neg A$ in their body must be satisfied by other literal than $\neg A$, as we have already set $v(A) = 0$. Therefore, we may remove each occurence of $\neg A$ in all the clauses, constructing a new formula ϕ'. The reason for this is the same as above - the new clauses are equisatisfiable to the original ones and thus ϕ' is equisatisfiable to ϕ. The removal of literals may result in an empty clause being constructed. As empty clause is unsatisfiable, in this case even ϕ' and its equivalent ϕ would be unsatisfiable as well.

We will repeat this step while the condition holds, i.e. until there is no unit clause. Note that in every step, whenever we find the value $v(A)$ for any variable, we immediately remove all occurences of its literals from the formula. For negative literals, we only remove them from the body, while for the positive literals, we remove the whole clause with them. This way, after each step there remain only such variables that we have not yet determined the value of.

When we reach the point that there are no unit clauses, it means that every clause has a body, i.e. at least one negative literal. However, we also know all variables currently present in the formula have not yet been assigned a value.

Therefore, we may assigning $v(V) = 0$ for each remaining variable V, thus satisfying all the clauses, as every one of them contains a negative literal.

The above algorithm is clearly polynomial. By simply scanning the formula for unit clauses and then removing appropriate literals/clauses, we will have to do at most $O(C)$ scans, each of them $O(L)$ long, where C is the number of clauses and L is the length of formula. This results in time complexity $O(C \cdot L)$. As stated above, this can actually be improved into a linear time algorithm. What we have to do is to represent the unit propagation in a directed graph.

First, we scan a formula and construct a vertex for every clause and for every variable that occurs as a head. Then, we scan the formula again, adding

- variable-vertex→clause-vertex oriented edge for every variable that occurs in a negative literal in the body of that clause
- clause-vertex→variable-vertex oriented edge for every variable that occurs as a head of that clause

We also construct a queue (FIFO) containing all the unit clause vertices. Now, let us process the items in the queue by pulling them out one by one. Every item we pull out is a unit clause vertex. As such, the variable V that is the head of this unit clause must be assigned the value $v(V) = 1$. We then travel along the edge going from this vertex into the variable-vertex representing V. Then, from this vertex multiple edges may lead into the clause-vertices representing all the clauses containing $\neg V$ in their body. By the unit propagation principle, we may remove $\neg V$ from body of each of these clauses. In the graph, this is done by removing the respective edge along which we came into that clause-vertex. If removing such an edge results in the clause-vertex having an in-degree of zero, we have just created a clause without a body.

- If there is an outbound edge from this vertex, it represents the head of this clause. Thus, we have just constructed a new unit clause. We therefore insert it into the back of the queue, so it will be processed later.
- If there is no outbound edge, the newly created clause is empty. This implies that the formula is unsatisfiable.

If we never encounter an empty clause, manage to process all the items in the queue and empty it, the algorithm halts. Apparently, in this case the formula is satisfiable. Every variable V that was processed as a head of the clause was already assigned the value $v(V) = 1$, so to construct a valuation we only have to assign the value 0 to the rest of the variables.

Note that the graph was constructed during linear sweeps over the formula, in $O(L)$ time. Each clause-vertex→variable-vertex edge is traversed only once, if the clause is discovered to be a unit clause and not traversed at all otherwise.

Similarily, all variable-vertex→clause-vertex edges are traversed at most once, because they are removed immediately after the traversal. If the graph was constructed in $O(L)$ time and no edge is traversed more than once, it means that all the travelling also took $O(L)$ time. Finally, as every variable-vertex only occurs in the queue at most once, the queue manipulation is done in $O(L)$ time as well. Therefore, the time complexity of this algorithm is $O(L)$. Thus, we indeed have a linear time algorithm for HornSAT.

4.3 Benchmarks

In this section, we are going to compare the performance of heuristics with that of the special-case algorithms on their respective classes of formulae. We will use the same methodology of generating random formulae and the same notation of their parameters as in Section 2.5. We have implemented the both special-case algorithms and are going to compare their performance with WalkSat. For this comparison, we have used Bart Selman and Henry Kautz's WalkSat implementation [16]. Our implementation of Tarjan's SCC algorithm for 2-SAT was based on the one in [2].

As WalkSat will never find out if a given formula is unsatisfiable, the comparison is only meaningful for satisfiable formulae. Therefore, we will test each randomly generated formula using the following scheme.

1. Generate a new formula ϕ.
2. Test ϕ using the special-case algorithm. If ϕ is unsatisfiable, disregard it and return to step 1. Otherwise, continue.
3. Test ϕ using WalkSat.

Instead of limiting WalkSat by setting the maximum number of tries, we will a set a time limit. It will be 10 times the running time of the special case algorithm. Note that this time limit should be in place, because heuristics may take potentially unlimited time solving the problem, if the given formula is hard to satisfy. If during any run WalkSat does not manage to find a solution in time limit, the results for this run are not included in the result set. As we will soon witness, this did not happen too often and therefore it did not disrupt our statistical analyses of the results.

Let us now have a look at the parameters and outcomes of 2-SAT and HornSAT benchmarks.

4.3.1 2-SAT

V	C	S	R
$20,000 \ldots 100,000$	V	2	1

Table 4.1: WalkSat benchmark parameters

As we chose a very large range for V, we had to compensate with the small value $R = 1$. Due to the randomized nature of WalkSat, this can lead to quite dissimilar running times even for adjacent values of V. In the previous benchmarks, we used $R > 1$ and computed the average time of R runs for every value of V. In case of $R = 1$, this is not possible. However, we may smoothen the curve by replacing the running time for each value of V by the average of running times in the interval $V - \varepsilon \ldots V + \varepsilon$. As V ranges in order of 10^4, by choosing $\varepsilon << 10^4$ we may consider the formulae generated in the interval $V - \varepsilon \ldots V + \varepsilon$ similar enough to be averaged. Note also that we chose the dependency $C = 1 \cdot V$ experimentally, as this again led to not-too-easy and not-too-difficult formulae.

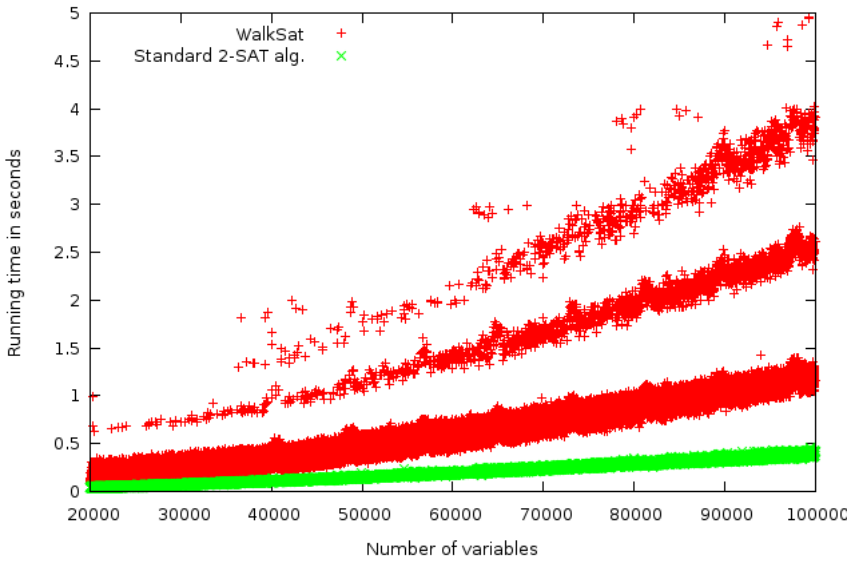

Figure 4.3: WalkSat benchmark results (not smoothened)

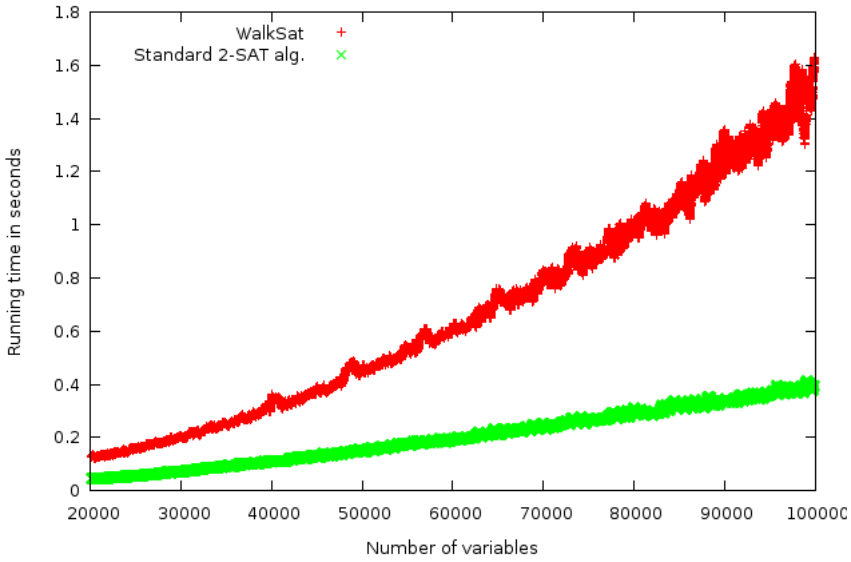

Figure 4.4: WalkSat benchmark results (smoothened with $\varepsilon = 100$)

We can see that WalkSat is generally slower than the 2-SAT algorithm. However, this does not yet mean that $t_{WalkSat} = \omega(t_{2-SAT})$, where $t_{WalkSat}$ is the expected time complexity of WalkSat and t_{2-SAT} is the time complexity of the standard 2-SAT algorithm. Let us examine the ratio

$$\frac{t_{WalkSat}}{t_{2-SAT}}$$

for every value of V of the softened curve.

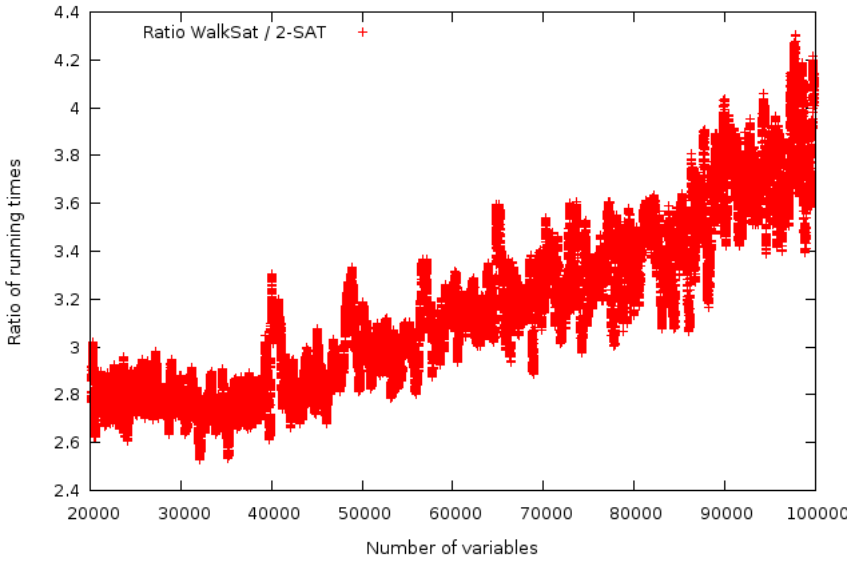

Figure 4.5: The ratio of WalkSat running time to the standard 2-SAT algorithm running time

Although the result is again not a smooth curve, we can see a steady increasing tendency. By generalizing this observation, we can say that

$$\lim_{V \to \infty} \frac{t_{WalkSat}(V)}{t_{2-SAT}(V)} = \infty \qquad \Rightarrow \qquad t_{WalkSat} = \omega(t_{2-SAT})$$

Thus, it can be concluded that in the case of 2-CNF, it is better to use the standard 2-SAT algorithm than WalkSat. Note also that since the time complexity of the 2-SAT algorithm is $O(L)$, in this case that is also $O(V)$, because $L = O(C) = O(V)$. This fact is well visible in the graph.

4.3.2 HornSAT

For HornSAT formulae, we cannot set $S = const.$, as we need to have both unit and non-unit clauses in the formula to properly examine the algorithm's functioning. If none of the clauses were a unit one, the formula would be always satisfiable (by an all-zero valuation). If all of them were, the solving process would resemble that of solving SAT for formulae in DNF, as we only have to check for pairs of opposite literals.

We will therefore use the following generating process. The clauses will be organized into "tiers", numbered 1 to T. The i-th tier will consist of C clauses of the size i. A clause of size S will be generated as follows.

1. Generate an S-tuple $[n_1, n_2, \ldots, n_S] \in \{1, 2, \ldots, V\}^S$ such that $n_i \neq n_j$ for $i \neq j$ with a uniform probability distribution

2. Select a random number $h \in \mathbb{Z}_{S+1}$. With probability $\frac{1}{3}$, choose $h = 0$ and with probability $\frac{2}{3}$, choose h uniformly among $\{1, 2, \ldots, V\}$.

3. Considering the set of variables to be $A_1, A_2, \ldots, A_V$, we will construct the clause $(L_1 \vee L_2 \vee \cdots \vee L_S)$ in such a way that

$$L_i = \begin{cases} A_{n_i} & \text{if } h = i \\ \neg A_{n_i} & \text{if } h \neq i \end{cases}$$

Thus, every clause will have a head with $\frac{2}{3}$ probability and all variables have equal probability of becoming one. Let us go on to the benchmarking itself. Note that C was again chosen experimentally.

$\boldsymbol{V}$	$\boldsymbol{T}$	$\boldsymbol{C}$	$\boldsymbol{R}$
$20 \ldots 500$	V	$\lfloor \frac{V}{20} \rfloor$	10

Table 4.2: HornSAT benchmark parameters

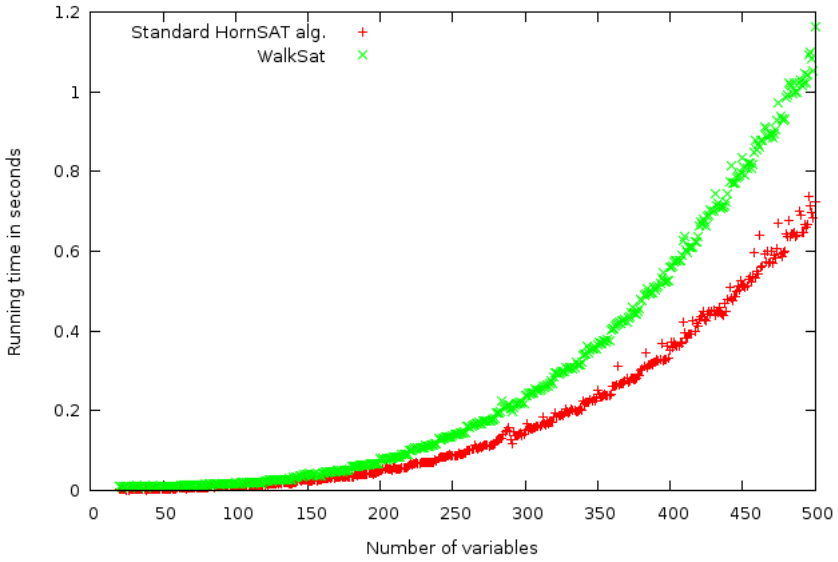

Figure 4.6: HornSAT benchmark results (based on V)

We know that the time complexity of the HornSAT algorithm is $O(L)$. In our case, the length of formula, which is equivalent to the number of literals, can be expressed as

$$L = \sum_{i=1}^{V} \left\lfloor \frac{i}{20} \right\rfloor \cdot i = O(V^3)$$

so the resulting curve should be cubic. We can prove that by replacing each value of V on the x-axis by its corresponding value of L computed using the above formula, which should result in a linear curve.

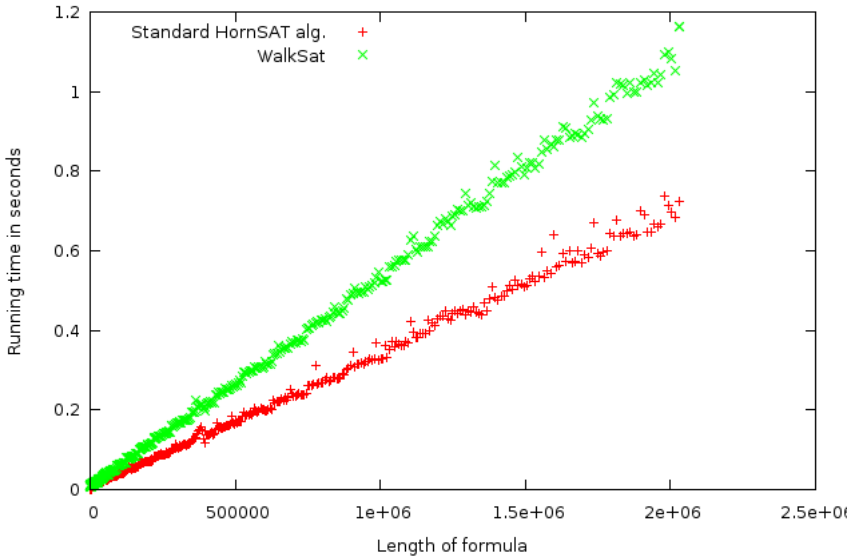

Figure 4.7: HornSAT benchmark results (based on L)

As in the case of 2-SAT, even here the special-case algorithm outperforms the heuristic. To see if the standard HornSAT algorithm is also asymptotically faster than WalkSAT, let us examine the ratio of their running times. As computing this ratio for small running times ($V < 100$) yields numerically inaccurate results, we are only showing the results for $100 \leq V \leq 500$).

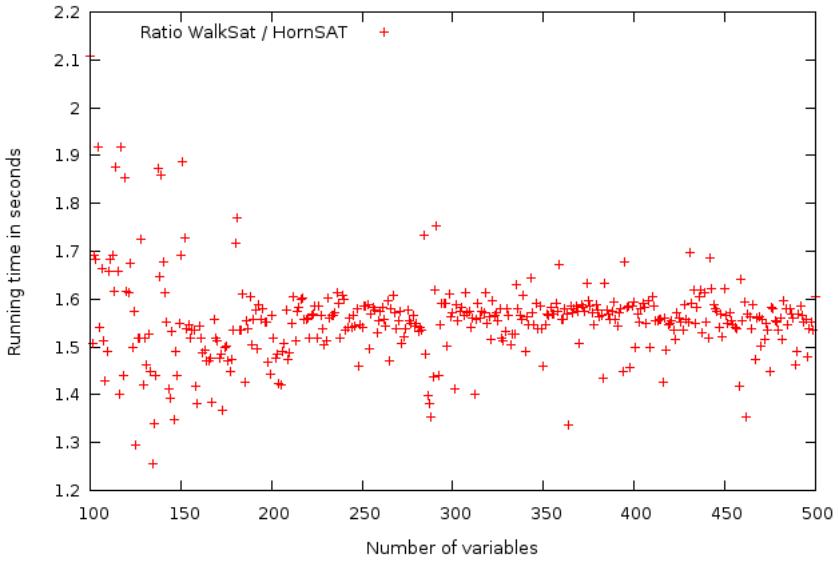

Figure 4.8: The ratio of WalkSat running time to the standard HornSAT algorithm running time

The majority of results is located in the belt between 1.5 and 1.6. If we use $t_{WalkSat}$ to denote the expected running time of WalkSat and $t_{HornSAT}$ to denote the running time of the standard HornSAT algorithm, we can say that $t_{WalkSat}(V) \approx 1.55 \cdot t_{HornSAT}(V)$. This means that $t_{WalkSat} = \Theta(t_{HornSAT})$. The conclusion is that it is reasonable to use WalkSAT for solving SAT on Horn CNF formulae. The reason why this is possible could be that it is relatively easy to construct a satisfiable valuation greedily. We have to start with all-zero valuation, then flip the conflicting variables that will probably not be many.

Conclusion

We have created a comprehensive survey of SAT solving algorithms. First of all, we discussed the methods of verifying the input to be a boolean formula, parsing it and representing it in the program. We then presented several SAT solving algorithms divided into three categories: the complete ones, the heuristics and the special-case ones. For each algorithm, we explained its theoretical foundations as well as the practical aspects of its implementation. In every category, we benchmarked the corresponding algorithms' performance. For the complete algorithms, we measured their time complexity and statistically analyzed their behaviour on a large set of formulae. We confronted the heuristics with benchmark sets from the International SAT Competitions. Finally, we examined whether the special-case algorithms outperform the heuristics on their respective classes of formulae.

Bibliography

[1] The international sat competitions web page. `http://www.satcompetition.org/`.

[2] Tarjan's strongly connected component algorithm. `http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm`.

[3] Satisfiability Suggested Format. Technical report, 1993.

[4] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121 – 123, 1979.

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

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[8] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267 – 284, 1984.

[9] B. Ferris and J. Froehlich. Walksat as an informed heuristic to dpll in sat solving. `http://www.cs.washington.edu/homes/bdferris/papers/WalkSAT-DPLL.pdf`.

[10] Z. Galil. The complexity of resolution procedures for theorem proving in the propositional calculus. Technical report, Ithaca, New York, 1975.

[11] H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In Walsh, editor, *SAT 2000*, pages 283–292. IOS press, 2000.

[12] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[13] J. Marques-Silva. Practical applications of boolean satisfiability. In *Workshop on Discrete Event Systems (WODES'08)*. IEEE Press, May 2008. Event Dates: May 2008.

[14] M. Nikolić, F. Marić, and P. Janičić. Instance-based selection of policies for sat solvers. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 326–340, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, January 1965.

[16] B. Selman and H. Kautz. Walksat algorithm implementation. `http://www.cs.rochester.edu/u/kautz/walksat/`, March 2012.

[17] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*, pages 521–532, 1995.

[18] J. Singer, I. P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

[19] M. Soos, K. Nohl, and C. Castelluccia. Extending sat solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 244–257, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] P. Štěpánek. *Predikátová logika*. Praha, 2000.

[21] V. Švejdar. *Logika: neúplnost, složitost a nutnost (Logic: Incompleteness, Complexity, and Necessity)*. Academia, Praha, 2002. ISBN 80-200-1005-X. In Czech. 464 pages. Section 5.2 on Gödel-Dummett logic, pp. 395–414, was written by Petr Hájek.

[22] R. Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114 – 121, oct. 1971.

[23] B. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing*, 6(4):384–400, oct. - dec. 1984.