UNIVERSIDAD DE GRANADA

UNDERGRADUATE THESIS

Consistency in Propositional Logic

Author:
Pedro Bonilla Nadal

Supervisor:
Dr. Serafín Moral Callejón

A thesis submitted in fulfillment of the requirements
for the degrees of Computer Engineering and Mathematics

in the

Department of Computer Science and Artificial Inteligence

June 4, 2020

Contents

Part I

Satisfiability Problem: Definition and Relevance

Chapter 1

Logic

TODO: Incluir introducción del problema.

INFO:
- Explicar que vamos a explicar en este capítulo.
- Handbook incluye una introducción histórica.
- Libro verde incluye una introducción intuitiva excelente.

In this chapter we present the bases of Logic and formal languages. Logic will provide us with a framework on which we will be able to define the Satisfiability Problem. We will present the area with formality, explaining only the things that will be necessary for our goal.

1.1 Boolean Algebra

The same way I started my journey on the university, we could have started this text right from the axioms, making a really romantic thesis. Nonetheless, given the goal we want to achieve, it seems excessive. We will refer to the commonly used *Zermelo-Fraenkel axioms*, in order to have a point of reference, and therefore we will work without more considerations with sets and sets operations. We will put *Zorn's lemma* to work, so the axiom of choice will also be needed, although in practice we will only work with finite sets of formulas.

Further on this section we will present Boolean Algebra in a classic lattice-based way that could be found in related literature. In particular we follow Introduction to mathematics of satisfiability[12] for the definition of Booblean Algebra and Propositional Logic. The definition of Lattice of Partitions is adapted from [18]

Definition 1.1.1. A partial ordered set, also poset, is a pair $\{X, \leq\}$ where X is a set and $\leq$ is a partial order of X. A chain Y of $\{X, \leq\}$ is a subset of X where $\leq$ is a total order.

Proposition 1.1.1 (Zorn lemma)**.** If every chain in a poset $\{X, \leq\}$ is bounded, then X possesses a maximal elements and for every x there is a maximal element y such that $x \leq y$.

Definition 1.1.2. A lattice is a partial ordered set $\{X, \leq\}$ where every pair of elements possesses a least upper bound and a greatest lower bound. A lattice has two new operations defined: given two elements $x, y \in X$

- $x \vee y$ denote the least upper bound.
- $x \wedge y$ denote the greatest lower bound.

A lattice is complete if every subset has a unique largest element and a unique lowest element. A lattice is presented generally as a duple $\{L, \leq\}$, a triple $\{X, \vee, \wedge\}$ and, whenever possible, is presented as a quintuple $\{X, \vee, \wedge, \top, \bot\}$ where $\top$ is the greatest element and $\bot$ the lowest element. A lattice is called distributive if $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

With the concept of lattice just included, we present the *Knaster and Tarski fixpoint theorem* fixpoint theorem. In order to do that we will introduce some notation. Given a function $f : \{L, \leq\} \to \{L, \leq\}$, a prefixpoint (resp. postfixpoint) is a point $x \in L$ such that $f(x) \leq x$ (resp. $f(x) \geq x$). A fixpoint is a point that is both prefixpoint and postfixpoint. Note that, given that they exists, $\top$ and *bot* are a prefixpoint and a postfixpoint of f respectively.

Theorem 1.1.1 (Knaster and Tarski fixpoint theorem [12])**.** *Let $f : \{L, \leq\} \to \{L, \leq\}$ be a monotone function in a complete lattice. Then:*

1. *f has a least prefixpoint l that is a fixpoint.*
2. *f has a largest postfixpoint l that is a fixpoint.*

Proof. 1. We know that there is at least a prefixpoint. Let

$$l = \bigwedge_{\{x \in X : x \text{ is a prefixpoint}\}} x$$

. Lets prove that l is a fixpoint. Le x be an arbitrary fixpoint, therefore, $l \leq x \leq f(x)$. Since x was arbitrary, $f(l) \leq l$. To show that it a fixpoint it suffices to see that $f(l)$ is a prefixpoint to, as f is monotone.

2. Apply the previous result on $f : \{L, \leq\} \to \{L, \leq\}$.

□

Definition 1.1.3. A *Boolean algebra* is a distributive lattice $\{X, \vee, \wedge, \top, \bot\}$ with an additional operation $\neg$, called complement or negation, such that for all x:

1. $x \wedge \neg x = \bot,\ x \vee \neg x = \top$
2. $\neg(x \vee y) = \neg x \wedge \neg y, \neg(x \wedge y) = \neg x \vee \neg y$
3. $\neg\neg x = x$

Definition 1.1.4 (Lattice of Partitions)**.** Given a set $X \neq \emptyset$, we denote as $\mathcal{P}(X)$ the partitions of X. Let $\pi, \pi' \in \mathcal{P}(X)$. We say that $\pi \leq_{\mathcal{P}} \pi'$ if for every $A \in \pi$ there exists $b \in \pi'$ such that $A \subset B$. The *lattice of partitions* of X is the lattice $\{\mathcal{P}(X), \leq_{\mathcal{P}}\}$.

For example given the lattice $\{\mathcal{P}(\{1,2,3,4\}), \leq_{\mathcal{P}}\}$ and two partitions:

$$\begin{aligned} \pi_1 &= \{\{1,2\},\{3,4\}\} \\ \pi_2 &= \{\{1,2,3\},\{4\}\} \end{aligned} \tag{1.1}$$

We have that:

$$\begin{aligned} \pi_1 \wedge \pi_2 &= \{\{1,2\},\{3\},\{4\}\} \\ \pi_1 \vee \pi_2 &= \{\{1,2,3,4\}\} \end{aligned} \tag{1.2}$$

1.2 Propositional Logic

Propositional logic is the framework that will allow us define the main topics of this text. Let's define some concepts:

- An alphabet A is an arbitrary non-empty set.
- A symbol a is an element of the alphabet.
- A word $w = \{a_i : i \in 1, .., n\}$ is a finite sequence of symbols.
- The collection of all possible words over an alphabet A is denoted by A^*.
- A language L over A is a subset of A^*.

For example, Spanish is a language with a well-known alphabet. Also, Spanish is a proper language over its alphabet as it is not empty, and it does not include all possible words.

When we talk about a logic system we are talking about a distinguished formal language. A formal language is defined by it syntax and its semantics. The syntax is the rules that define the language. They state what words over an alphabet are valid in the language. The semantics deal with the interpretations of the elements in the language. Usually this is achieved by assigning truth values to each word.

We will define now propositional logic, or zeroth-order-logic.

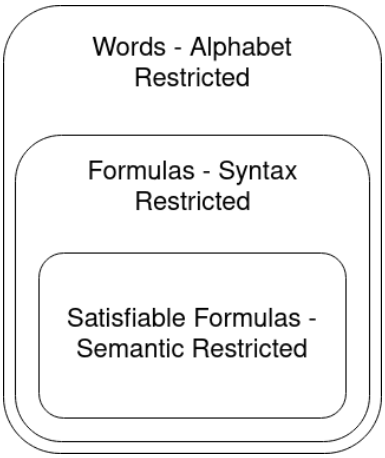

FIGURE 1.1: Diagram showing the different classes which are constructed on the formal language of Propositional Logic.

1.2.1 Syntax of Propositional Logic

We first start with the basic building blocks, which collectively form what is called the alphabet:

- Symbols x, y, z for variables. As more variables are necessary sub-indexes will be used.

- Unary operator $\neg$ (negation). A literal will refer to a variable or a negated variable. Thorough the text symbol l will denote a literal.
- Values 0 and 1. These values are often named as $\bot$ and $\top$ respectively.
- Binary operators: $\wedge, \vee, \rightarrow, \oplus, \iff$

The words of Propositional Logic are called formulas.

Definition 1.2.1. A Boolean formula is defined inductively:

- The constants 0 and 1 are formulas.
- Every variable is a formula.
- If F is a formula, then $\neg F$ is a formula.
- The concatenation with a binary operator of two formulas is a formula too.

Examples of formulas are $x \vee y$ or $x_1 \wedge x_2 \vee (x_4 \vee \neg x_3 \wedge (x_5 \rightarrow x_6) \vee 0)$.We should distinguish a special type of formula: the clauses. A clause is a formula with the form $l_1 \vee ... \vee l_n$ where $l_i, i \in 1, ..., n$ are literals. Clauses are will be often regarded as a finite set of literals. Example of a clause is $(x_1 \vee \neg x_4 \vee x_2)$. When regarded as a set every clause C has a cardinal $|C|$, that represents the number of literals contained.

1.2.2 Semantics of Propositional Logic

When facing a way to provide semantic meaning to formulas the use of function In this section we will discuss to ways of providing meaning to the formulas: two-valued logic and three valued logic.

In two valued logic define the truth value of a formula by assigning a truth value(1 for Truth and 0 for False) to each variable. Note that we assign a meaning of truth to the constants 1 and 0, that until now where meaningless. The truth value of the formulas that involve operators are provide by their truth table.

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \oplus q$	$p \rightarrow q$	$p \iff q$
0	0	1	0	0	0	1	1
0	1	1	1	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	1	1	0	1	1

TABLE 1.1: Truth tables of different operators in two valued logic.

The truth value of a formula is therefore obtained by replacing each variable by their assigned constant and propagating the value. The tool that we will use to assign a truth value to each variable is the assignments

Definition 1.2.2. An assignment is a function α from the $Form_{Var}$ to $Form_{Var}$, on which some variables $\{x_1, ..., x_n\}$ are replaced by predefined constants $\{a_1, ..., a_n\}$ respectively.

An assignment that assigns a value to a variable x is said to map the variable x. In two valued logic we will consider only assignment that maps all variables, and therefore all formulas are given a value by an assignment. We also see that any assignment generate a map from Var to $\{0,1\}$. Conversely, any map from Var to $\{0,1\}$ would uniquely represent a assignment $alpha$ over $Form_{Var}$. In practice when we talk about an assignment α we will refer indistinctly to either the function over $Form_{Var}$ or the mapping over Var.

One can then *apply* an assignment α to a formula F, denoting it by $F\alpha = \alpha(F)$. To describe an assignment we will use a set that pairs each variable to it value, i.e. $\alpha = \{x_1 \to 1, ..., x_n \to 0\}$. For example given an assignment $\alpha_0 = \{x_1 \to 1, x_2 \to 1, x_3 \to 0, x_4 \to 1\}$ and $F_0 = x_1 \to (x_2 \wedge x_4)$ then $F_0\alpha_0 = 1 \to (1 \wedge 1) = 1$.

Definition 1.2.3. An assignment is said to *satisfy* a formula F if $F\alpha = 1$ and in the case $F\alpha = 0$ it is said to *falsify* the statement. A formula F is called *satisfiable* if is exists an assignment that satisfy it. Otherwise it is called *unsatisfiable*.

Note that we have a really restrictive constraint on assignments: they should map all variables. This is so in order for an assignment to give a meaning to every formula. To ease this constraint we use three-valued logic. On three valued logic we have three significant: True or 1, False or 0, and unknown or v. Now the assignment will map every variable to one of these values. These new assignments will be called partial assignments, as they only map some variables to a truth value. We can propagate the previous values adding new rules.

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \oplus q$	$p \to q$	$p \iff q$
v	0	v	v	0	v	v	v
v	1	v	1	v	v	1	v
0	v	1	v	0	v	1	v
1	v	0	1	v	v	v	v

TABLE 1.2: Truth table of different operators in three valued logic.

In practice partial assignments will be only define by denoting only the variables that are mapped to either 0 or 1. We can see that the composition of assignments (seen as functions over $Form_{var}$ is also a partial assignment. Also, when applying a partial assignment to a formula, instead of mapping it to v we will avoid operating over the variables assigned to v. For example given a partial assignment $\alpha_0 = \{x_1 \to 1, x_2 \to 1, x_3 \to 0\}$ and $F_0 = x_1 \to (x_2 \wedge x_4)$ then $F_0\alpha_0 = 1 \to (1 \wedge x_4) = 1 \to (x_4)$. Although F_0 is mapped to another formula by α_0, α_0 is still providing a meaning to it (the unknown meaning).

Partial assignments will be also used to iteratively *expand* them: let $Var = \{x_i : i \in 1, ..., n\}$ the set of variables and let α_1 be partial assignment that map variables $[x_1 \to a_1, ..., x_j \to a_j]$ with $1 < j < n$ and $a_j \in \{0,1\}$ for every j, we can expand it by choosing a nonempty subset $A \subset \{a_k : k \in j+1, .., n\}$ and a value $c_x \in \{0,1\}$ for every $x \in A$. Then we can define:

$$\alpha_2(x) = \begin{cases} \alpha_1(x) & x \in \{x_i : i \in 1, ..., n\}, \\ c_x & x \in A, \\ v & \text{otherwise.} \end{cases}$$

We can see that α_2 expands α_1 in the sense that the truth value assigned to a formula by α_1 holds in α_2 if it where different that unknown. Therefore we are expanding the 'known' values of the formulas. Note that in the definition of α_1 it where not necessary to state what variables were mapped to v at it was implicit that every variable not listed were of unknown value.

In practice we will try to avoid refer to this process whenever is evidently enough what is being done. Nonetheless partial assignments will be a central part of later explained algorithms such as DPLL[5.1]. When context is clear enough, assignments will be used for both assignments and partial assignments.

Arguably, the most special case of partial assignment are autarks assignments[2.2.2]. An autark assignment is a partial assignment that simplify a formula in a sense latter explained.

Given an assignment or partial assignment α we will denote the set of variables mapped to either 0 or 1 by $Var(\alpha)$. Analogously, given a formula F, $Var(F)$ will denote the variables that appear in F. Note that if $F \in Form_{Var}$ then $Var(F) \in Var$ and it is not necessary that $Var(F) = Var$.

Two formulas F, G are said to be equal, represented as $F \sim G$, if for every two-valued assignment α maps $F\alpha = G\alpha$. It follows from the equivalently properties on constants that $\sim$ is an equal relationship. This definition is really intuitive, as it define as equal the formulas that has the same meaning in every possible situation.

With $\sim$ defined we can have what is called a *Lindenbaum algebra*, as a quotient space of $Form = Form_{Var}$ by the relation $\sim$, denoted as $Form/\sim$. It follows that every operator respect the quotient space structure, i.e., for every $[\phi_1], [\phi_2] \in Form/\sim$:

- $\neg[\phi_1] = [\neg\phi_1]$
- $[\phi_1] \vee [\phi_2] = [\phi_1 \vee \phi_2]$
- $[\phi_1] \wedge [\phi_2] = [\phi_1 \wedge \phi_2]$

The interest of Lindenbaum algebra resides in the fact that $\{Form, \vee, \wedge, [1], [0]\}$ is a Boolean algebra, providing therefore a nexus between the algebraic formulation of the problem an its semantics.

1.3 First order Logic

1.4 Modal Logic

Chapter 2

Definition of the problem

2.1 Satisfiability Problem

2.1.1 Decision Problems

Computability and complexity theory attempt to answer questions regarding how to efficiently solve real-world problems. In this section we provide a formal approach to the concept of problem, and its resolution.

We will study the complexity of functions. In order to standardize the approach we code the input of the function and the output of the functions using words over a finite alphabet. As for every finite alphabet A there is a bijective mapping from A^* to $\{0,1\}^*$ we can assume when its convenient that the alphabet is $\{0,1\}$. With this convention we are now ready to define what is a decision problem.

Definition 2.1.1 (Decision Problem[1])**.** Given a language L over an alphabet A, it has an associated decision problem that consist on, given a word $w \in A^*$ check whether w is in L.

When we have a named language, we refer indistinctly by this name to both the language and the associated decision problem. In order to define a decision problem is only needed to define a language over an alphabet. Therefore a decision problem may be defined implicitly, that is, as the set of the words in an alphabet that satisfy some condition. As semantics provide meaning to the languages, real world problems can be addressed as decision problems.

2.1.2 Definition

Given the previous definitions, we are now almost prepared to define the central part of this thesis: the satisfiability decision problem of propositional logic, SAT for short. To this end we define a special subset of formulas in Propositional Logic: the formulas in Conjunctive Normal Form.

Definition 2.1.2. A formula F is said to be in Conjunctive Normal Form (CNF) if F is written as:

$$F = C_1 \wedge ... \wedge C_n$$

Where C_i are clauses.

Note that every formula in CNF can be regarded as a set of clauses. This approach is really useful in some contexts and will be often used.

Definition 2.1.3. The Satisfiability Language of Propositional Logic (SAT) is the language over the alphabet of propositional logic that includes all formulas that are both satisfiable and in CNF.

We will refer with the acronym SAT to both the language and the associated decision problem. As checking if a formula is in CNF is a fairly simple syntax problem, we are only interested in asserting whether or not a formula in CNF is satisfiable.

Definition 2.1.4. A *SAT-Solver* is an algorithm that, being given a formula F in CNF as input, answer whether or not is satisfiable.

On chapter[II] we analyze the main SAT-solver developed in the literature. We will differentiate two types of SAT-Solver. The algorithms that, given enough time always output the correct result at the end are called *complete*. The SAT-solvers that doesn't guaranty its result are called *incomplete*. Of particular interest among incomplete SAT-solvers are the one-sided bounded error SAT-solvers. These are the called probabilistic algorithms, discussed on section[6]

2.1.3 Variations

The SAT decision problem quite a lot of variations, all of them of interest for certain complexity classes. We will list some of the most important, starting with two decision problems. The first of them is a natural generalization.

Definition 2.1.5. The Generalized Satisfiability Language of Propositional Logic (GSAT) is the language over the alphabet of propositional logic that includes all formulas that are Satisfiable.

With Tseitin's Theorem[3.3.2] we can see that these two problems are in fact fairly similar. More often than not GSAT will be solved by solving an equivalent SAT problem. Analogously a *GSAT-Solver* is a SAT-solver that also accepts as inputs formulas not in CNF. Further on, every new problem will have a associated *solver*, defined analogously.

Definition 2.1.6. Let F be a formula. F is said to be k-CNF formula (equivalently a formula in k-CNF) if it is in CNF and $\forall C \in F, |C| = k$. k-SAT is the language of the formulas that are both satisfiable and in k-CNF.

Other variations of SAT could be achieved by generalizing the concept of decision problem.

Definition 2.1.7 (Function Problem)**.** Let A, B be two sets. Given a relation $R \subset A \times B$, it has an associated function problem that consists on, given a word $a \in X$ find a word $b \in B$ such that $(a, b) \in R$.

Definition 2.1.8. Let CNF be the set of propositional formulas in CNF and B the set of assignments. The Satisfiability Function Problem of Propositional Logic (FSAT) is the function problem defined by the relation

$$R = \{(F, b) : F \in CNF, b \in B, Fb = 1\}.$$

That is, is the problem of finding an assignment that satisfy a formula. Most of SAT-solvers not only try to solve SAT but also to solve FSAT, i.e., try to find an assignment that satisfy the formula should it exists.

Definition 2.1.9. Let CNF be the set of propositional formulas in CNF and B the set of assignments. The Maximum Satisfiability Problem (MAXSAT) is the problem. function problem defined by the relation

$$R = \{(F, n) : F \in CNF, n = \max_{\alpha \in B}\{|\{C \in F : C\alpha = 1\}|\}.$$

That is, is the problem of finding the maximum number of assignments that can be satisfied simultaneously.

As we will see, most SAT-solvers are FSAT-solvers. In related literature the FSAT-solver are called constructive SAT-solvers, as they provide a constructive solution of the problem. Solvers that only solve sat are called non-constructive SAT-solvers. After presenting the concept of algorithmic complexity we will see that from a non constructive SAT-solver, a constructive SAT-solver can be made so that the latter is not much less efficient[3.3.3].

2.1.4 Constraint Satisfaction Problem

We want to introduce the notion of Constraint Satisfaction Problem (CSP) because it defines a new optic over the SAT problem. CSP is, in fact, a generalization of SAT. When dealing with CSP problem we want to find a solution with certain restriction. A example of what is a CSP is watching film with your family: each member impose its restrictions, and then we look for a film that satisfy them all. Should it happen that no film is found, we have other type of problem. This concept naturally translates into propositional logic formulation. Lets define CSP formally:

Definition 2.1.10 ([19]). A *Constraint Satisfaction Problem*(CSP) is a triple $\{X, D, C\}$ where:

- $X = \{x_1, ..., x_n\}$ is the set of variables occurring in the problem.
- $D = \{D_1, ..., D_n\}$ is the set of the domains. Each $D_i = d_{i,1}, .., d_{i,n_i}$ is the domain of the variable x_i.
- $C = \{C_1, ..., C_m\}$ is the set of constraints over the variables. For our intentions, these constraints must be written as:
  - An equality, for example: $(x_i, x_j) = (d_{i,k}, d_{j,k'})$
  - An inequality, for example: $(x_i, x_j) \neq (d_{i,k}, do_{j,k'})$
  - Concatenation with a Propositional Logic operator of two equalities or inequalities, for example: $((x_1) = (d_{1,1}) \vee (x_2 \neq d_{2,5}) \wedge \neg((x_8, x_9) = (d_{8,3}, d_{9,7}))$.

The goal of a CSP is to found a mapping

$$\alpha : X \to \cup_{i \in 1,...,n} D_i$$

such that every variable x_i is mapped to a value on its associated domain D_i and every constraint is satisfied. Such map will be called an *assignment*, and if this map satisfy all constraints it is said that α *satisfy* the CSP problem.

Note that we can use all our artillery from Propositional Logic as both an equalities and inequalities hold a binary truth value (True/False), therefore can be handled as Propositional Logic Variables.

The value in CSP resides on the simplicity of its formulations. One can easily define a CSP just by selecting the desired conditions of a solution and describing

its context. Moreover, a lot of real world problems can be defined in terms of constraints. Constraint programming is a programming paradigm that consists on solving problems by defining them as CSP and letting CSP-solvers do the work.

SAT could be seen as a CSP where every domain is $\{0,1\}$ and each clause is a constraint. Therefore if we know how to solve CSP we know how to solve SAT. Let see the reverse.

Proposition 2.1.1. Every CSP problem has an equivalent SAT problem.

Proof. Let $\{X, D, C\}$ be a CSP problem. To define a equivalent SAT problem we are going to define a SAT problem that can be solved if, and only if, the CSP problem can be solved. We will also request that from every assignment that satisfy the equivalent SAT problem, we can deduce an assignment that satisfy the CSP problem, and conversely. In order to define a SAT problem we are going to define a set of variable and a set of clauses to be satisfy.

Our set of variables consists on a variable $y_{i,j}$ for each variable $x_i \in X$, and each value $d_{i,j} \in D_i$ that represent whether or not $x_i = d_{i,j}$. Now we define the set of clauses. The first to group of clauses are added for consistency reason, and the later is added in order to maintain the constraints.

1. $(y_{i,1} \vee ... \vee y_{i,n_i})$ for all $i \in 1, ..., n$ that represents that every variable should take a value.

2. $(\neg y_{i,j} \vee \neg y_{i,j})$ for all $i \in 1, ..., n,\ j \in 1, ..., n_i$ that represents that a variable can not take more that one value.

3. $(y_{i,j})$ for every equality $x_i = d_{i,j}$ and $(\neg y_{i,j})$ for every inequality $x_i \neq d_{i,j}$. If two equalities or inequalities are expressed concatenated by a Propositional Logic operator we express the associated literals of the equalities and inequalities concatenated by the same Propositional Logic Operator. In order to express the resulting formulas as a CNF formula, we use Tseitin's Theorem. A proof of this theorem will be provided on [3.3.2].

If there is an assignment α that satisfy the associated SAT problem, then there is an assignment β that satisfy the CSP problem such that $\beta(x_i = d_{i,j}$ if $\alpha(y_{i,j}) = 1$. From the clauses generated in 1. and 2. we can assert that such mapping is well defined, and from the clauses generated by 3. follow that β satisfy all constraints.

Conversely we can define an assignment α that satisfy the SAT problem from an assignment β that satisfy the CSP problem by mapping $x_{i,j}$

$$\alpha(x_i) = \begin{cases} 1 & \beta(xi) = d_{i,j} \\ 0 & \text{otherwise.} \end{cases}$$

Therefore the CSP problem is solvable, if and only if, the SAT problem is satisfiable, and given a satisfying assignment of either the SAT or CSP problem we know how to generate a satisfying assignment of the other problem. □

In practice we will use CSP as a methodology of problem defining. It will provide easy solutions for complex problem, given that we solve SAT problem. More on this will be shown on [III]

2.2 Some Properties about SAT

In this section we explain some concepts of SAT that are interesting on that they have beautiful maths theories related and can be useful on resolving and analyzing complexity. In particular we will talk about:

- Symmetric Clauses.
- Autarks assignments.

The first of them are useful when modelling a problem in order to generate SAT problems on which we can work more efficiently. Then second of them is an useful technique that is used extensively on SAT solvers.

2.2.1 Symmetry

In this section we talk about symmetry groups and its application to SAT. The information and examples resemble the ones in [18]. We will start this section with a motivating example.

Example 2.2.1. Consider the boolean formula:

$$F = (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c)$$

It is not difficult to see that this functions remains invariant under some variations, namely:

- The trivial variation: the identity. For the example, we will denote this transformation by I.
- Swapping the inputs of a and b. It is equivalent to renaming a as b and b as a. For the example, we will denote this transformation as α:

$$\alpha(F) = (\neg b \wedge a \wedge c) \vee (b \wedge \neg a \wedge c) = (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) = F$$

- Swapping the inputs of a and $\neg b$. It is equivalent to renaming a as $\neg b$ and b as $\neg a$. For the example, we will denote this transformation as β:

$$\beta(F) = (\neg\neg b \wedge \neg a \wedge c) \vee (\neg b \wedge \neg\neg a \wedge c) = (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) = F$$

- Swapping a with $\neg a$ and b with $\neg b$. For the example, we will denote this transformation as γ:

$$\gamma(F) = (\neg\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge \neg\neg b \wedge c) = (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) = F$$

With these three invariants we can also see that the composition of each one of these invariants with each other produce another invariant. Moreover, each invariant is its own inverse, as $\varphi \circ \varphi = I$ for $\varphi \in \{I, \alpha, \beta, \gamma\}$.

We can see therefore that by invariant γ, it does not matter what value does we assign to a, as either both $F\{a = 1\}$ and $F\{a = 0\}$ are satisfiable or none are. Therefore we can solve$F\{a = 1\}$, and we have a simplified problem to examine. For the record, the formula is not satisfiable.

This is what is called *symmetry breaking*. And avid reader would have already recognized the group structure on the invariants. On this section we are going to explore the concepts related to define a symmetry, explore the group of negations and permutations, and develop some strategies to implement *symmetry breaking*. Now we are going to define a few concepts.

Definition 2.2.1. Let F be a formula and ϕ be a function $\alpha : Form_{Var(F)} \to Form_{Var(F)}$. We say that ϕ is an invariant of F if $\phi(F) = F$.

Definition 2.2.2 (Permutation)**.** Given a set of variables $X = \{x_1, ..., x_n\}$, a *permutation* of X is any injective mapping $\alpha : X \to X$. Each permutation induces a homonym function on $\alpha : Form_X \to Form_X$ that replaces every variable by its image by α.

For example given $X = \{a, b, c\}$, and a injective mapping $\alpha : X \to X$ such that:

$$\begin{aligned} &\alpha(a) \to c, \\ &\alpha(b) \to b, \\ &\alpha(c) \to a. \end{aligned} \tag{2.1}$$

We have a function $\alpha : Form_X \to Form_X$ that maps $F = (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c)$ to $\alpha(F) = (\neg c \wedge b \wedge a) \vee (c \wedge \neg b \wedge a)$.

We can see set of all permutations over a set X such that $|X| = n$ along with composition is the *permutation group* on n elements, P_n. This group is studied in algebra, and is usually introduced as the group of mappings of $X = \{1, ..., n\}$ and composition. Nonetheless we are more interested on calling those elements $\{x_1, ..., x_n\}$ in order to have it frame on satisfiability. We can use this without more regards as the definition of the permutation group is element-free. As this is a well known group we will not prove that is, in fact, a group.

Definition 2.2.3. Let X be an non-empty set of variables. Given $A \subset X$ a negation of A is a mapping $\sigma_A : Lit(X) \to Lit(X)$ defined by:

$$\sigma_A(x_i) = \begin{cases} \neg l & if l \in A, l \\ \text{otherwise.} \end{cases}$$

Where $Lit(X)$ is the set of literals over X. Each negation induces a homonym function $\sigma_A : Form_X \to Form_X$.

Proposition 2.2.1. The set of negations over a set X and composition is a group.

Proof.

- Closure: We can see that $\sigma_A \circ \sigma_B = \sigma_{(A \cup B) \setminus (A \cap B)}$.
- Associativity: Associativity is inherited from the general associativity of composition.
- Identity: We have an identity element $\sigma_\emptyset$.
- Inverse: For every $A \subset X$, $\sigma_A^2 = \sigma_\emptyset$.

□

We will denote the group of negations over n variables as N_n.

Proposition 2.2.2 (NP_n)**.** The set of negations and permutations on n variables (NP_n) along with the composition will be called

Proof. Note that every element $x \in NP_n$ can be expressed as $\alpha \circ \sigma$ where α is a permutation and σ is a negation. Also note that

- Closure: $\alpha \circ \sigma_A \circ \alpha' \circ \sigma_{A'} = \alpha \circ \alpha' \circ \sigma_{\alpha^{-1}A} \circ \sigma_{A'} = \alpha'' \circ \sigma_{A''}$, where α'' is a permutation and $\sigma_{A''}$ is a negation.
- Associativity: Associativity is inherited from the general associativity of composition.
- Identity: We have an identity element $\sigma_\emptyset = I$.
- Inverse: For every $\alpha \circ \sigma_A$ the function $\alpha^{-1} \circ \sigma_{\alpha(A)}$ is its inverse.

□

Proposition 2.2.3. NP_n s the semi-direct product of N_n and P_n. That is:

$$NP_n = N_n \rtimes P_n$$

Proof. Note that due to the property aforementioned, $NP_n = P_n N_n$ and $P_n \cap N_n = I$. Therefore NP_n is the semi direct product of P_n and N_n. □

For us, a symmetry of a formula F is any $\phi \in NP_{|Var(n)|}$ that is an invariant for F.

Definition 2.2.4 (Group Action)**.** An *action* of a group G on a set S is a map $G \times S \to S$ such that:

- $es = s$ for e the identity element of G and every $s \in S$.
- $(g_1 g_2)(s) = g_1(g_2 s)$ for all $s \in S$ and all $g_1, g_2 \in G$.

The concept of group action is important because it induces a partition over the elements of a set. Partitions will be the tool necessary in order to find symmetries.

Definition 2.2.5 (Group-Induced Equivalence Partition)**.** Let G be a group and X be a set. The action of the group G over X induces an equivalence relation such that, for $x_1, x_2 \in X$:

$$x_1 \sim x_2 \qquad \text{if} \qquad \exists g \in G \text{ such that } g x_1 = x_2$$

This relation induce a quotient space on X and, therefore, a partition, that could be seen as an element of the lattice of partitions of X. We will denote this partition as $P(X, G)$

Note that the inverse property on groups and the two properties of group actions imply that the equivalence relation is, in fact, an equivalence relation. We note a simple result in order to make this concept more manageable.

Proposition 2.2.4. Let G be a group, $\{g_1, ..., g_n\} \subset G$ be a set that generates G and X be a set. We have that:

$$P(X, G) = \vee_{i \in 1,...,n} P(X, < g_i >)$$

Where $< g_i >$ is the cyclic group generated by g_i.

We are going to search symmetries a CNF formulas F. Until now we have a naive method in order to do this: for every $\phi \in NP_n$ check whether ϕ is an invariant of F. Nonetheless the complexity of this process is as hard as solving SAT. Instead, we will in fact reduce the problem of symmetries detection to a colored graph automorphism problem[3.2].

Definition 2.2.6. Given a graph $G = (V, E)$ be a graph where V is the set of nodes and E the set of edges, represented as unordered pairs. A *coloring* of a graph its a partition $(V_1, ..., V_n)$ of V. Each V_i will be called a *color*. Also, let $v \in V$ we define

$$d(v, V_i) = |\{u \in V_i : (u, v) \in E\}|$$

We say that a coloring is *stable* if

$$d(u, V_i) = d(v, V_i), \qquad \forall u, v \in V_j,\ \forall i, j \in 1, ..., n$$

From every coloring π of G we can make a stable coloring π' by iteratively splitting colors with different vertex degree. We can see that $\pi' \leq_{\mathcal{P}} \pi$. If π' is a discrete partition, i.e., $|\pi'| = |V|$, G has no symmetries beyond the identity. Otherwise we have some candidates for symmetry.

2.2.2 Autarks assignments

Once we have defined what is a CNF formula and what is a problem we can proceed to define this anticipated concept.

Definition 2.2.7. An partial assignment α is called autark for a CNF formula F if for every clause $C \in F$ it happens that if $Var(C) \cap Var(\alpha) \neq \emptyset$ then $C\alpha = 1$.

An autark assignment α for a CNF formula F is an assignment that satisfies all clauses that it 'touches'. These assignments provide simplifications of the CNF Formulas in the context of satisfiability, as they generate a new CNF formula $F\alpha$ that are satisfiable if, and only if, F is satisfiable. In set notation we can state that α is autark for F if $F\alpha \subsetneq F$. Subsequently, trying to find simple autark assignment, i.e. assignment with not many variables,is a good praxis.

Should it happen that we have an algorithm for the Autarks Finding Problem, iterating it, we could find a satisfying assignment of any given formula if it exists such assignment, therefore solving FSAT. Let's define the problem formally:

Definition 2.2.8. Let CNF the set of formulas in CNF and $Part$ the set of partial assignment. The Autark Finding Problem is the function problem defined by the relation:

$$R = \{(F, \alpha) : F \in CNF \wedge \forall C \in F(Var(C) \cap Var(\alpha) \neq \emptyset \implies C\alpha = 1)\}$$

Proposition 2.2.5. There is a reduction from FSAT to the Autark-Finding problem.

Proof. Suppose that an algorithm such that if it exists any autark it return one of them, and end with an error code otherwise is given.

Given a formula F, if there is not an autark then there is no solution for the SAT problem. If it finds an Autark-assignment α then we apply the same algorithm to $\alpha(F)$. Also, as it happens that $|Var(\alpha(F))| < |Var(F)|$ so we would only apply the algorithm finitely many times. Also, F will be solvable if, and only if, $F\alpha$ is solvable. □

The most common autark assignment are the pure literal. A literal l is a pure literal for a formula F if there is no $\neg l$ in F. The partial-assignment that only maps

$u \rightarrow 1$ is an autark assignment for F. This type of autark are used on DPLL algorithm[5.1]. The MS algorithm[5.2.1] also uses an autark finding technique.

Chapter 3

Complexity Classes and Relevance of the Problem

3.1 Models of Computation

In this section we will discuss two computation models: Turing Machines and Circuits. We do not expect the text to be the first approximations to Turing machines, so we present a quick formal approach to the area.

3.1.1 Turing Machines

Turing machines are arguably the epicenter of models of computation. A Turing Machine represents a long mechanical tape on which we are going to operate. The tape is divided in to discrete positions, such that we can see the tape as a one-dimensional array. Operating on this tape we can focus on a cell, scan its contents, overwrite its contents or move to an adjacent cell. These operations try to resemble the process of human calculus, as done with paper and pencil when applying the long division method for example. Formally:

Definition 3.1.1 (Turing Machine [8])**.** We describe a Turing Machine as a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ whose components have the following meanings:

- Q the finite set of *states* of the finite control.
- Σ the finite set of *input symbols*.
- Γ the finite set of *tape symbols*. Σ is always a subset of Γ.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ the transition function.
- q_0 the *start state*.
- B the *blank symbol*.
- F the *accepting states*.

A configuration of a Turing machine is a triplet $C = (q, u, v)$ where $q \in Q$, $u, v \in \Gamma^*$. A configuration is accepting if $q \in F$.

A configuration should be understand as a state of the machine, where q is the current state, u the part of the tape left to the cell on which we focus and v is the part of the tape right to the cell we focus, starting on it.

As a brief note before going on, we have not define the empty word yet, as in propositional logic is not a valid formula so it lacks our interest until now. We will

note the empty word as ϵ and would consist of an empty sequence of symbols. We can now define a relation between configurations:

Definition 3.1.2. Let M be a Turing Machine and $C = (q, u, v)$, $C' = (q, u, v)$ be two configurations of M. We say that $C \vdash C'$ if there is a transition $\delta(q, v_1) = (q', b, D)$ with $D \in L, R$ and:

- if $D = L$, then if $u = u_1...u_n$ and $v = v_1...v_m$, it should happen that $u' = u_1...u_{n-1}$ and $v' = u_n b v_2...v_n$ with two exceptions:
  - if $u = \epsilon$ then $u' = \epsilon$ and $v' = bv_2...v_n$
  - if $v = v_1$ and $b = B$ then $u' = u_1...u_{n-1}$ and $v' = u_n$.
- if $D = R$, then if $u = u_1...u_n$ and $v = v_1...v_m$, it should happen that $u' = u_1...u_n b$ and $v' = v_2...v_n$ with two exceptions:
  - if $u = \epsilon$ then $u' = b$ and $v' = v_2...v_n$
  - if $v = v_1$ and $b = B$ then $u' = u_1...u_{n-1}$ and $v' = \epsilon$.

Note that on both cases the two exceptions can be given simultaneously (if $b = B$) and then give $(q, \epsilon, u_1) \vdash (q', \epsilon, B)$. We say $C \vdash^* C'$ if there exists a finite sequence $\{C_i\}_{i \in 1,...,n}$ such that $C_1 = C$, $C_n = C'$ and $C_i \vdash C_{i+1}$ for every $i \in 1, ..., n-1$.

A Turing machine solves a decision problem, i.e. having an alphabet Σ (its input alphabet) it takes a word in Σ^* and decides whether it belongs to a language or not.The intuitive idea is that we write the word in question on the tape and the machine tells us the results after some computations. In this way we will consider that a word belongs to the language decided by the Turing machine if and only if it is *accepted* by that machine. Let's define when this concept.

Definition 3.1.3. Let M be a Turing Machine. We say that $u \in \Sigma^*$ is *accepted* by M if there exists a final configuration C such that $(q_0, \epsilon, u) \vdash^* C$. The language *accepted* by M is the collection of all words accepted. We say that M *decides* a language L if L is the language accepted by M.

We can also consider a Turing Machine that can solve a function problem. The intuitive idea is that we write the input on the tape and after some computations we have written on the tape a word that is related to the input one. Formally:

Definition 3.1.4. Let $f : \Sigma^* \to \Sigma^*$. A Turing Machine M *compute* f if for every $u \in \Sigma^*$ there is an accepting configuration $C = (q', v, v')$ of M such that $(q, \epsilon, u) \vdash C$ and $f(u) = vv'$. A Turing Machine M computes a function problem defined by R if it computes a function $f_M : Sigma^* \to \Sigma^*$ such that $(u, f(u)) \in R$ for all $u \in \Sigma^*$.

3.1.2 Circuits

The circuit model provide another computation model that is naturally related to propositional logic. We define it briefly in order to prove some results on the area. Circuits are the computation model that is used in practice. One of the most important area of industrial application of SAT is circuit verification. This is going to be discussed further on AÑADIR CAPITULO.

1. TODO:
2. Moreover, as checking whether an assignment is autark is linear on the number of clauses, then this make the autark-finding problem NP-Complete(NP-C further on).

3.1.3 Reduction

3.2 Complexity Classes

In this section we are going to define what a complexity class is and then we are going to discuss some results of the complexity of the sat problem. This is for two reasons:

1. TODO:
2. To study the complexity of the SAT problem and its variants so far.
3. To give one more justification of the relevance of the problem.
4. Include graph automorphism complexity.
5.

3.2.1 Exponential Time Hypothesis

In this subsection we will introduce the result, shown first on [11]. This result states simply that no sub-exponential time algorithm can be found for 3-SAT. This hypothesis, although widely accepted, is still unproven. Formally:

Definition 3.2.1 (ETH)**.** For $k \geq 3$, lets define:

$$s_k = \inf\{\delta : \text{there exists} O(2^{\delta n})k - \text{SAT solver.}\}$$

It is claim that $s_k > 0$.

This result has some equivalent formulations.

Proposition 3.2.1 (theorem 1. [11])**.** The following statements are equivalent:

1. ETH: for all $k \geq 3$, $s_k > 0$.
2. For some k, $s_k \geq 0$.
3. $s_3 \geq 0$.

This theorem is prove making use of Sparsification Lemma, which is based in turn on the ideas of critical and forced variables. By the time of the publication of the article, Zane has already worked with these ideas for the development of the PPZ algorithm[16]. Although we are not going to prove this result, we are going to present this ideas when analyzing the Paturi-Pudlák-Zane Algorithm[6.1.1].

This claim is harder that $P \neq NP$, as it not only declares that SAT is not polynomial time, but neither is sub-exponential time.

3.3 Postponed results

In this section we will develop some result that where intentionally postponed, in order to talk about them once complexity classes has been introduced, although they belong thematically to previous sections.

3.3.1 Tseitin Theorem

Now that we are able to talk about efficiency is time to talk about an interesting, anticipated result. If we remember Lindenbaum algebra[1.2.2] we have defined a quotient space on the formulas in terms of satisfiability. In order to solve GSAT, we are not in need of solving all formulas. Instead we can learn how to solve a language of formulas F such that for every class $[\phi_1] \in Form/\sim$ there is a formula $f \in F$ such that $f \in [\phi_1]$. Also, we will need a method that allow us to find such $f \in F$ given any element of $[\phi_1]$.We want to prove that SAT is a language that satisfy this restrictions. The naive approach to the problem is straightforward:

Proposition 3.3.1. There is a CNF formula in each equivalence class. Moreover, given a function $f \in Form$ we are able to find an equivalent CNF formula.

Proof. Given $\phi_1 \in Form$ we make the truth table of ϕ_1. Two formulas are in the same equivalent classes if, and only if, they share the same truth table.

We can generate a CNF formula that has the same table this way: for every row $[x_1 \to a_1, ..., x_n \to a_n]$ (x_i variables, $a_i \in \{0,1\}$) that falsify ϕ_1 we add a clause $(z_1 \vee ... \vee z_n)$ with $z_i = x_i$ if $a_i = 0$ and $z_i = \neg x_i$ if $a_i = 1$. □

This method is interesting as it show the truth table, as the collection of all two-valued assignmentsα such that $Var(\alpha) = \phi_1$. Nonetheless is not a method that should be considered useful, as is has exponential time. Tseitin theorem provide us with a solution to this problem that run in polynomial time. We will need a lemma first:

Lemma 3.3.1. *For every* SAT *formula there is an associated circuit.*

Proof. Every operator can be seen as a gate and every variable as an input.

□

Theorem 3.3.2 (Tseitin [22])**.** *There is a 3-CNF formula on each equivalent class. Moreover, given an element F there is a equivalent formula G in 3-CNF which could be computed in polynomial time.*

Proof. We will show that for every circuit with n inputs and m binary gates there is a formula in *3-CNF* that could be constructed in polynomial time in n and m. Then, given a formula we will work with it considering its associated circuit.
< We will construct the formula considering variables $x_1, ..., x_n$ that will represent the inputs and $y_1, ..., y_m$ that will represents the output of each gate.

$$G = (y_1) \wedge \bigwedge_{i=1}^{m} (y_i \iff f_i(z_{i,1}, z_{i,2}))$$

Where f_i represents the formula associated to the i-gate, $z_{i,1}, z_{i,2}$ each of the two inputs of the i-gate, whether they are $x_\_$ or $y_\_$ variables. This formula is not *3-CNF* yet, but for each configuration being f_i a Boolean operator there would be a *3-CNF* equivalent.

- $z \iff (x \vee y) = \neg(z \vee x \vee y) \vee (z \wedge (x \vee y)) = \neg(z \vee x \vee y) \vee (z \wedge x) \vee (z \wedge y) =$
  $= (\neg z \wedge \neg x \wedge \neg y) \vee (z \wedge x) \vee (z \wedge y) = (\neg z \vee (z \wedge x) \vee (z \wedge y)) \wedge (\neg x \vee (z \wedge x) \vee (z \wedge y)) \wedge (\neg y \vee (z \wedge x) \vee (z \wedge y)) = (\neg z \vee x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$

- $z \iff (x \wedge y) = \neg(z \vee (x \wedge y)) \vee (z \wedge (x \wedge y)) = (z \wedge x \wedge y) \vee (\neg z \wedge \neg x \wedge \neg y) =$
  $((z \vee (\neg z \wedge \neg x \wedge \neg y)) \wedge (x \vee (\neg z \wedge \neg x \wedge \neg y)) \wedge (y \vee (\neg z \wedge \neg x \wedge \neg y))) = (\neg x \vee z) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (\neg y \vee x) \wedge (\neg z \vee y) \wedge (\neg x \vee y)$
- $z \iff (x \iff y) = \neg(z \vee (x \iff y)) \vee (z \wedge (x \iff y) = \neg(z \vee (\neg x \wedge \neg y) \vee (x \wedge y)) \vee (z \wedge (\neg x \wedge \neg y) \vee (x \wedge y))) = (\neg z \wedge \neg(\neg x \wedge \neg y) \wedge \neg(x \wedge y)) \vee (z \wedge (\neg x \wedge \neg y) \vee (x \wedge y))) = (\neg z \wedge (x \vee y) \wedge (\neg x \vee \neg y)) \vee (z \wedge (\neg x \wedge \neg y) \vee (x \wedge y))) = z \vee (\neg x \wedge \neg y) = (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg z \vee y) \wedge (y \vee z \vee x) \wedge (y \vee \neg y \vee x) \wedge (\neg z \vee z \vee x) \wedge (\neg z \vee \neg y \vee x)$
- $z \iff (x \oplus y) = z \iff (\neg x \iff y)$

In the last item we use the third one.

□

The fact that they are reachable on polynomial time is important because it means it could be done efficiently. Should this be impossible it will not be of much relevance in practice, as we yearn to solve this problem as efficient as possible (in fact, as polynomial as possible). This result implies that if we know how to solve 3-SAT we know how to solve GSAT.

3.3.2 Tautologies

Definition 3.3.1. A formula F such that for every assignment α happens that $F\alpha = 1$ is a tautology. Given two formulas G, F it is said that G follows from F if $F \to G$ is a tautology.

Proposition 3.3.2. Given a tautology $F \to G$, there exists a formula I such that $Var(I) = Var(F) \cap Var(G)$ and both $F \to I$ and $I \to G$ are tautologies. A polynomial algorithm to solve this problem is not known.

Proof. Let $\{x_1, ..., x_k\} = Var(F) \cup Var(G)$ then we will build I by defining its truth table in the following way: Given an assignment α:

$$I\alpha = \begin{cases} 1 & \text{if } \alpha \text{ could be extended to an assignment that } \textit{satisfies } F, \\ 0 & \text{if } \alpha \text{ could be extended to an assignment that } \textit{nullifies } G, \\ * & \text{otherwise.} \end{cases}$$

Where * mean that it could be either 0 or 1. This is well defined because if for an arbitrary α it happens that $G\alpha = 0$ then $F\alpha = 0$.

For every assignment β such that $Var(\beta) = Var(F) \cup Var(G)$ then if $\beta(F) = 1$ then $\beta(I) = 1$ so $F \to I$ is a tautology. Similarly it can not happen that $I\beta = 1$ and $G\beta = 0$, because the second will imply that $I\beta = 0$.

For the last part we will refer to the paper on the topic by: TODO □

3.3.3 From non-constructive to constructive

This subsection we explain how a constructive SAT-solver can be made from a non-constructive SAT-solver without changing is asymptotic time complexity, assuming true the exponential time hypothesis[3.2.1]

Proposition 3.3.3. Let ϕ be an oracle that decides SAT in $O(\varphi(n+m))$ where n is the number of variables and m the number of clauses. Then we can make an algorithm that computes FSAT on $O((n(\varphi(n+m))+m)$

Proof. We will iteratively expand a partial assignment α. α initially maps all variables to v. The procedure take as input a CNF formula F. The algorithm that solve FSAT is described in [1]. It is based in the notion that, if F is satisfiable, either $F\{x=1\}$ or $F\{x=0\}$ is satisfiable. We are able to explore the variable lineally being sure that we are always assigning the correct value to each variable.

Algorithm 1 FSAT routine

```
1:  procedure SOLVER(F)
2:      F_0 ← F
3:      α ← empty partial assignment.
4:
5:      for x ∈ Var(F) do
6:          if φ(F_0{x = 1}) then
7:              α+ = {x = 1}
8:              F_0 ← F_0{x = 1}
9:          else
10:             if φ(F_0{x = 0}) then
11:                 α+ = {x = 0}
12:                 F_0 ← F_0{x = 0}
13:             else
14:                 Unsatisfiable
15:     return α
```

Let's analyze the complexity of this algorithm. We make at most n repetitions of the for loop, and on each repetition we call ϕ and assign a variable in a formula.Therefore the procedure run in $O(n\varphi(n+m)+m)$ □

Assuming ETH we assume that ϕ is exponential in time, an therefore asymptotic complexity $O(\varphi(n+m))$ is the same as $O(n(\varphi(n+m)+m))$, so, until ETH is proved wrong we can consider SAT and FSAT as being equal in complexity. On this text we will only deal with non-constructive solver on the combinatorics section[**??**].

Part II

Solvers

Chapter 4

Special Cases

On this part we attack the main problem of SAT: explain the different techniques that can be applied.Onward we will see how it could be solved, and develop applied techniques. There are a lot of approaches to this problem and they differ on their way to attack it. We have to realise that three things are important to judge a algorithm.

- The simplicity: following Occam's razor, between two solutions that do not appear to be better or worse, one should choose the easiest one. This solution are far more comprehensible and tends to be more variable and adaptable for our problem. We should not despise an easy solution to a complex problem only because a far more difficult approach give slightly better results.
- The complexity: and by that I mean its algorithmic ('Big O') complexity. It is important to get good running times in all cases and have a analysis of the worst cases scenario that the algorithm could have.
- The efficiency: Some algorithms will have the same complexity as the most simple ones, but will use some plans to be able to solve most part of the cases fast (even in polynomial time). There are some cases that would make this algorithms be pretty slow, but more often than not a trade-off is convenient.

On this chapter we are going to talk about solvability in special situations where we work with a restricted subset of formulas (more restricted than CNF formulas). We want to exploit special characteristics of these subsets in our favor in order to get a resolution without involving a complex exponential time algorithms (as the ones needed to solve SAT).

The first section of this chapter will talk about combinatorics. We proceed to analyze solvability in special cases, i.e., algorithms that works really well in formulas given that they satisfy some restriction.

4.1 Satisfiability by Combinatorics

To get an intuition about how unsolvable clauses are, we gonna state some simple result about combinatorics and resolution. These techinques present some cases where we can solve the decision problem efficently, although more often that not we would not provide a satisfying assignment, i.e., we do not solve the function problem.

As there is no complete SAT-solver known to work in polymial time complexity, a polynomial increase does not affect overall the assymptotical complexity.

Firstly, it is easy to break a big clause on some smaller ones, adding one another on this way: Suppose we got two positive integers n, m such that $m < n$ a clause $x_1 \vee$

$x_2 \vee ... \vee x_n$ we could split it into two parts $x_1 \vee x_2 \vee ... \vee x_{m-1} \vee y, \neg y \vee x_m \vee ... \vee x_n$. Also given the same clause with a given length n we could enlarge it one variable adding $x_1 \vee ... \vee x_n \vee y$ and $x_1 \vee ... \vee x_n \vee \neg y$ where y is a new variable. Note that to enlarge a clause from a length m to a length $n > m$ we would generate 2^{n-m} clauses.

Proposition 4.1.1. Let F be a k-CNF formula, if $|F| < 2^k$ then F is satisfiable.

Proof. Let $n = Var(F)$, it happens that $n > k$. For each clause $C \in F$ there are 2^{n-k} assignments that falsify F, so in total there could be strictly less than $2^k \cdot 2^{n-k} = 2^n$. Therefore it exists an assignment that assigns all variables and not falsifies the formula F. □

Proposition 4.1.2. Let $F = \{C_1, ..., C_n\}$ be a CNF formula. If $\sum_{j=1}^{m} 2^{-|C_j|} < 1$, then F is satisfiable.

Proof. Enlarging clauses the way it is explained to the maximum length k and applying the previous result. □

Following this idea we could define the weight of a clause $C \in F$ as

$$\omega(C) = 2^{-|C|}$$

being this the probability that a uniform-random assignment violates this clause.

Corollary 4.1.0.1. *For a formula in CNF, if the sum of the weights of the clauses is less than one then the formula is satisfiable.*

Definition 4.1.1. Let F be a CNF formula. It is said to be minimally unsatisfiable if:

- F is unsatisfiable.
- $F \backslash \{C\}$ is satisfiable $\forall C \in F$.

Then the following prove will be shown as in [19]. For that we will need the well known Hall marriage theorem[6]. A similar result is proved in chapter 7 of [12], using the König theorem. Both of the version take the idea of associate a graph to each set of clauses.

Definition 4.1.2. Let $G = (N, E)$ be a graph where N is the set of nodes and E the set of edges, represented as pair of nodes. Given $n \in X$, the neighborhood of n, denoted as $\Gamma_G(n)$, is defined as:

$$\Gamma_G(n) = \{n' \in X : n' \neq n, \exists e \in E \text{ such that } n, n' \in e\}$$

Analogously, the inclusive neighborhood is defined as $\Gamma_G^+(n) = \Gamma_G(n) \cup \{n\}$. The neighborhood of a subset $W \subset X$ is defined as $\Gamma_G(X) = \Cup_{n \in X} \Gamma_G(n)$

Theorem 4.1.1 (Hall marriage graph version)**.** *Let G be a finite bipartite graph with finite sets of vertex X, Y. There is a matching edge cover(a cover such that every vertex only participate in one edge) of X if and only if $|W| \leq |\Gamma_G(W)|$ for every $W \subset X$.*

Lemma 4.1.2. *Let F be a CNF formula. If for every subset G of F it holds that $|G| \leq |Var(G)|$, then F is satisfiable.*

Proof. We will associate a bipartite graph with F: U, V be the two set of vertexes where U consists on the set of clauses and V on the set of variables. There is an edge (u, v) if v takes part on u.

By the marriage theorem every clause can be associated to a variable. Therefore we could make an assignment that take every variable associated to a clause to the value that the clause requires. □

This idea or neighbourhood in clause is important and curious. It defines a relation between clauses and give clauses resolution some nice graph-tools to work with.

Proposition 4.1.3. If F is minimally unsatisfiable, then $|F| > Var(F)$.

Proof. Since F is unsatisfiable, there must be a subset G such that is maximal and satisfy $|G| > Var(G)$. If $G = F$ them the theorem is proved.

Otherwise, let $H \subset F \backslash G$ be an arbitrary subset. If $|H| > |Var(H)(G)|$ then $|G \cup H| > |Var(G \cup H)|$ and G would not be maximal. Therefore F satisfies the condition of the lemma and is satisfiable using an assignment that does not use any variable $x \in Var(G)$. As G is minimally unsatisfiable G is satisfiable by and assignment β. We could them define an assignment:

$$\gamma(x) = \begin{cases} \beta(x) & \text{if } x \in Var(G) \\ \alpha(x) & \text{otherwise.} \end{cases}$$

this assignment would satisfy F against the hypothesis. We proved $G = F$ by contradiction and therefore we proved the lemma. □

4.2 Lovász Local Lemma

We continue to prove an interesting lemma on the theoretical analysis of satisfiability problem: the Lovász Local Lemma (LLL). This lemma was first proven on 1972 by Erdös and Lovász while they were studying 3-coloration of hypergraphs. Then it was Moser which understood the relationship between this result and constraint satisfaction problem. SAT could be regarded as the simplest of these problems.

This section is going to be based on the works of Moser, Tardos, Lovász and Erdös. As it will be shown, LLL is applicable to set a sufficient condition for satisfiability. We will explain the lemma for theoretical purposes and prove the most general version, and give a constructive algorithm to solve a less general statement of the problem. The principal source of bibliography for the whole section would be Moser PhD. Thesis[14].

The main contribution of Moser's work to this problem is finding an efficient constructive algorithm to find what assignment satisfies the formula, given that F satisfies the hypothesis of the lemma. Previously only probabilistic approaches had been successful.

The probabilistic method is a useful method to prove the existence of objects with an specific property. The philosophy beneath this type of proofs is the following: in order to prove the existence of an object we do not need to give the object, instead, we could just consider a random object in the space we are exploring an prove that

the probability is strictly positive. Then we can deduce that an object with that property exists. It is not necessary to provide the exact value, bounding it by a constant greater that zero would be enough.

This technique was pioneered by Paul Erdös. LLL is an useful tool to prove lower bounds for probabilities that is commonly used to prove that a probaility is strictly positive.

This section will follow this order:

- Present the notation and general expression for the LLL.
- Use the result to prove an interesting property on satisfiability on CNF.
- Prove the general result with the probabilistic result.
- Provide the more concise CNF-result with a constructive algorithm.

4.2.1 First definitions

We will work here with a very specific type of formulas.

Definition 4.2.1. Let C be a clause in F, the neighborhood of C, denoted as $\Gamma_F(C)$ as

$$\Gamma_F(C) = \{D \in F : D \neq C, Var(C) \cap Var(D) \neq \emptyset\}$$

Analogously, the inclusive neighborhood $\Gamma_F^+(C) = \Gamma(C) \cup \{C\}$.

Further on Γ and Γ^+ will respectively denote inclusive or exclusive neighborhood on CNF formulas or graphs

Definition 4.2.2. Two clauses are *conflicting* if there is a variable that is required to be true in one of then and to be false in the other. G_F^* is the graph such that there is an edge between C and D iff they *conflict* in some variable.

Definition 4.2.3. Let Ω be a probability space and let $\mathcal{A} = \{A_1, ..., A_m\}$ be arbitrary events in this space. We say that a graph G on the vertex set $\mathcal{A}$ is a *lopsidependency graph* for $\mathcal{A}$ if no event is more likely in the conditional space defined by intersecting the complement of any subset of its non-neighbors. In others words:

$$P\left(A \,\middle|\, \bigcap_{B \in S} \overline{B}\right) \leq P(A) \qquad \forall A \in \mathcal{A},\ \forall S \subset \mathcal{A} \backslash \Gamma_G^+(A)$$

If, instead of requiring the event to be more likely, we require it to be independent (i.e. to be equal in probability) the graph is called *dependency graph*.

4.2.2 Statement of the Lovász Local Lemma

Theorem 4.2.1 (Lovász Local Lema)**.** *Let Ω be a probability space and let $\mathcal{A} = \{A_1, ..., A_m\}$ be arbitrary events in this space. Let G be a lopsidependency graph for $\mathcal{A}$. If there exists a mapping $\mu : \mathcal{A} \to (0, 1)$ such that*

$$\forall A \in \mathcal{A} : P(A) \leq \mu(A) \prod_{B \in \Gamma_G(A)} (1 - \mu(B))$$

then $P\left(\bigcap_{A \in \mathcal{A}} \overline{A}\right) > 0$.

By considering the random experiment of drawing an assignment uniformly, with the event corresponding to violating the different clauses we could reformulate this result. The weight of each clause is the probability of violating each clause. Therefore, we can state a SAT-focused result.

Corollary 4.2.1.1 (Lovász Local Lema for SAT)**.** *Let F be a CNF formula. If there exists a mapping $\mu : F \to (0,1)$ that associates a number with each clause in the formula such that*

$$\forall A \in \mathcal{A} : \omega(A) \leq \mu(A) \prod_{B \in \Gamma^*_G(A)} (1 - \mu(B))$$

then F is satisfiable.

Proof. To prove the result it would only be necessary to show that Γ^* is the lopside-pendency graph for this experiment. Given $C \in F$ and $\mathcal{D} \subset F \backslash \Gamma_{G^*_F}(D)$ (i.e. no $D \in \mathcal{D}$ conflict with C). We want to check the probability of a random assignment falsifying C given that it satisfies all of the clauses in $\mathcal{D}$, and prove that it is at most $2^{-|C|}$.

Let α be an assignment such that it satisfies $\mathcal{D}$ and violates C. We could generate a new assignment from α changing any value on $Var(C)$, and they still will satisfy $\mathcal{D}$ (as there are no conflict) so the probability is still at most 2^{-k}.

□

The result that we will prove in a constructive way will be slightly more strict, imposing the condition not only in Γ^* but in Γ^+

Corollary 4.2.1.2 (Constructive Lovász Local Lema for SAT)**.** *Let F be a CNF formula. If there exists a mapping $\mu : F \to (0,1)$ that associates a number with each clause in the formula such that*

$$\forall A \in \mathcal{A} : \omega(A) \leq \mu(A) \prod_{B \in \Gamma_G(A)} (1 - \mu(B))$$

then F is satisfiable.

In order to get a result easier to check we will present a new criteria. If $k \leq 2$ the k-SAT problem is polynomially solvable so we will not be interested on such formulas.

Corollary 4.2.1.3. *Let F be a k-CNF with $k > 2$ formula such that $\forall C \in F$ and $|\Gamma_F(C)| \leq 2^k/e - 1$ then F is satisfiable.*

Proof. We will try to use [4.2.1.2]. We will define such $\mu : F \to (0,1)$, $\mu(C) = e \cdot 2^{-k}$. Let $C_0 \in F$ be an arbitrary clause.

$$2^{-k} = \omega(C) \leq \mu(C) \prod_{B \in \Gamma_F(C)} (1 - \mu(B)) = e2^{-k}(1 - e2^{-k})^{|\Gamma_F(C)|}$$

With the hypothesis

$$2^{-k} \leq e2^{-k}(1 - e2^{-k})^{2^k/e-1}$$

$$1 \leq e(1 - e2^{-k})^{2^k/e-1}$$

Being famous that the convergence of the sequence $\{(1 - e2^{-k})^{2^k/e-1}\}_k$ to $1/e$ is monotonically decreasing.

□

4.2.3 Nonconstructive proof of [4.2.1]

We explain the way Erdös, Lovász and Spencer originally proved the Lemma [4] [20]. The write-up presented here will resemble the one done by [15].

Thorough the proof we will use repeatedly the definition of conditional probability, i.e. for any events $\{E_i\}_{i\in 1,...,r}$,

$$P\left(\bigcap_{i=1}^{r} E_1\right) = \prod_{i=1}^{r} P\left(E_i \middle| \bigcap_{j=1}^{i-1} E_j\right)$$

Further on this subsection we will consider Ω to be a probability space and $\mathcal{A} = \{A_1, ..., A_m\}$ to be arbitrary events in this space, G to be a lopsidependency graph, and $\mu : \mathcal{A} \to (0, 1)$ such that the conditions of the theorem are satisfied. We first prove an auxiliary lemma.

Lemma 4.2.2. *Let $A_0 \in \mathcal{A}$ and $\mathcal{H} \subset \mathcal{A}$. then*

$$P\left(A \middle| \bigcap_{B\in\mathcal{H}} \overline{B}\right) \leq \mu(A)$$

Proof. The proof is by induction on the size of $|\mathcal{H}|$. The case $H = \emptyset$ follows from the hypothesis easily:

$$P\left(A \middle| \bigcap_{B\in\mathcal{H}} \overline{B}\right) = P(A) \leq^{1.} \mu(A) \prod_{B\in\Gamma_G^*(A)} (1 - \mu(B)) \leq^{2.} \mu(A)$$

Where 1. uses the hypothesis and 2. uses that $0 < \mu(B) < 1$. Now we suppose that $|\mathcal{H}| = n$ and that the claim is true for all $\mathcal{H}'$ such that $|\mathcal{H}'| < n$. We distinguish two cases. The induction hypothesis will not be necessary for the first of them

- When $\mathcal{H} \cap \Gamma_G^*(A) = \emptyset$ then $P\left(A \middle| \bigcap_{B\in\mathcal{H}} \overline{B}\right) = 0 \leq P(A)$ by definition of Γ_G^* and $P(A) \leq \mu(A)$ by definition of μ.
- Otherwise we have $A \notin \mathcal{H}$ and $\mathcal{H} \cap \Gamma_G^*(A) \neq \emptyset$. Then we can define to sets $\mathcal{H}_A = \mathcal{H} \cap \Gamma_G^*(A) = \{H_1, ..., H_k\}$ and $\mathcal{H}_0 = \mathcal{H} \backslash \mathcal{H}_A$.

$$P\left(A \middle| \bigcap_{B\in\mathcal{H}} \overline{B}\right) = \frac{P\left(A \cap \left(\bigcap_{B\in\mathcal{H}_A} \overline{B}\right) \middle| \bigcap_{B\in\mathcal{H}_0} \overline{B}\right)}{P\left(\bigcap_{B\in\mathcal{H}_A} \overline{B} \middle| \bigcap_{B\in\mathcal{H}_0} \overline{B}\right)}$$

We will bound numerator and denominator. For the numerator:

$$P\left(A \cap \left(\bigcap_{B\in\mathcal{H}_A} \overline{B}\right) \middle| \bigcap_{B\in\mathcal{H}_0} \overline{B}\right) \leq P\left(A \middle| \bigcap_{B\in\mathcal{H}_0} \overline{B}\right) \leq P(A)$$

Where the second inequality is given by the definition of lopsidependency graph. On the other hand, for the denominator, we can define $\mathcal{H}_i := \{H_i, ..., H_k\} \cup$

$\mathcal{H}_0$.

$$P\left(\bigcap_{B\in\mathcal{H}_A}\overline{B}\,\middle|\,\bigcap_{B\in\mathcal{H}_0}\overline{B}\right)=\prod_{i=1}^{k}P\left(\overline{B_i}\,\middle|\,\bigcap_{B\in\mathcal{H}_i}\overline{B}\right)$$
$$\geq^{3.}\prod_{i=1}^{k}(1-\mu(H_i))\geq^{4.}\prod_{B\in\Gamma_G^*(A)}(1-\mu(B))$$

Where in 3. the induction hypothesis is used, and in 4. is considering that $H_i\in\Gamma_G^*(A)$ Considering now both parts:

$$P\left(A\,\middle|\,\bigcap_{B\in\mathcal{H}}\overline{B}\right)\leq\frac{P(A)}{\prod_{B\in\Gamma_G^*(A)}(1-\mu(B))}\leq\mu(A)$$

Where the last inequality uses the hypothesis on μ.

□

proof of the theorem 4.2.1.

$$P\left(\bigcap_{A\in\mathcal{A}}\overline{A}\right)=\prod_{i=1}^{m}P\left(\overline{A_i}\,\middle|\,\bigcap_{j=1}^{i-1}\overline{A_j}\right)\geq^{5.}\prod_{i=1}^{m}(1-\mu(A_i))$$

Where in 5. is used 4.2.2 and since $\mu:\mathcal{A}\to(0,1)$ then $P\left(\bigcap_{A\in\mathcal{A}}\overline{A}\right)>0$.

□

4.2.4 Constructive proof of [4.2.1.2]

Moser[15] proves that it exists an algorithm such that it gives an assignment satisfying the SAT formula, should it happen that the formula satisfies 4.2.1.1 conditions. This is no a big deal, as a backtrack would be also capable of providing the solution, given that we know its existence. Not so trivial is that it would run in $O(|F|)$. We will show the version of the algorithm shown in [19].

Algorithm 2 Moser's Algorithm

```
1: C_1, ..., C_m ← Clauses in F to satisfy, globally accessible
2: α ← assignment on Var(F)
3:
4: procedure REPAIR(α, C)
5:     for v ∈ Var(C) do
6:         α(v) = random ∈ {0,1}
7:     for j := 1 to m do
8:         if (Var(C_j) ∩ Var(C) ≠ ∅) ∧ (C_j α = 0) then
9:             Repair(C_j)
10:
11: Randomly choose an initial assignment α
12: for j := 1 to m do
13:     if α(C_j) = 0 then
14:         Repair(C_j)
```

At first sight it is not clear if it terminates. If F verifies 4.2.1.1 it is proved that if will end after running Repair at most $O(\sum_{C \in F} \frac{\mu(C)}{1-\mu(C)})$

4.3 Special Cases Solvable in Polynomial Time

In this section we will discuss some cases of the SAT problem solvable in P. These cases are of interest because polynomial is no achievable in all cases. Nonetheless, they only work with a subset of all possible formulas. They should be use whenever possible as no general polynomial time is believed to exist, nor it is proved its non-existence. In general thorough the section we will follow *The Satisfiability Problem: Algorithms and Analyses*[19].

Definition 4.3.1. Let F be a formula. A subset $V \subset Var(F)$ is called a backdoor if $F\alpha \in$ P for every assignment α that maps all V.

Let us explain this concept. Given a formula F a backdoor is a expecial subset of the variables such that if all of it is assigned then we can solve the remaning formula in polynomial time, i.e., once we have assigned this variables the problem is easy. The trivial backdoor is the set of all variables. For a backdoor the smaller, the better.

A goal for a SAT-solver could be to find a backdoor of minimum size. DPLL would try to search for a backdoor, using heuristics in order not to explore all subsets (only achievable if such backdoor exists).

4.3.1 Unit Propagation

Unit propagation is a simple concept that is worth standing out because it is commonplace. Given a CNF formula F if there is a clause with only one element then the value of the variable should be assigned accordingly to the clause, otherwise F is unsatisfiable. This lead to the unit propagation concept. Whenever we have a unitary clause $\{p\}$ we should *resolve* it and start working with $F[p = 1]$ being $[p = 1]$ the assignment that maps the value of the metavariable p to 1, which could possibly imply mapping a variable to 0.

Also, the unit propagation might result on a recursive problem, as other unit clauses could appear. Unit propagation is a usefull way to simplify .

4.3.2 2CNF

It is already know that 3CNF is equivalent to SAT. This is not known for 2CNF and is believed to be false.

Proposition 4.3.1. 2CNF is in P

Proof. To prove that 2CNF is in P, a polynomial algorithm on the number of clauses will be given. Let $F \in$ 2CNF. Without loss of generality, we will consider that there are no clauses in F $\{u, u\}$ or $\{u, \neg u\}$ as the first one should be handle with unit propagation and the second one is a tautology. Therefore each clause is $(u \vee v)$ with $var(u) \neq var(v)$, which could be seen as $(\neg u \to v)(vu)$.

We would consider a step to be as follow: we choose a variable $x \in Var(F)$ and set it to 0. Then a chain of implications would arise, which might end on conflict. If no conflict arises, then is an autark assignment, so repeat the process. Otherwise set it to 1 and proceed. If conflict arise, then F is unsatisfiable. If no conflict arise, then is an autark assignment, so repeat the process.

Each step is of polynomial time over the number of clauses. Also there would be at most as many steps as variables, therefore we have a polynomial algorithm.

□

4.3.3 Horn Formulas

In this subsection we will analyze Horn formulas. They named after Alfred Horn[9]. They are of special interest as HORNSAT is P-complete.

Definition 4.3.2. Let F be a formula in CNF. F is said to be a Horn formula if for every $C \in F$ there is at most one non-negated literal. HORN will be the set of all horn formulas.

HORNSAT will be the intersection of HORN and SAT problems. Nonetheless, given the easiness of checking whether a formula is in HORN, it would usually consider as the problem that check the satisfiability of a horn formula.

Proposition 4.3.2. HORNSAT is in P.

Proof. Given a formula F it could have a clause with only one non-negated literal or not. If it does not have a clause like this, set all the variables in to 0 and is solved. Otherwise, unit-propagate the unary clause and repeat the process recursively. If a contradiction is raised, them the F is not satisfiable. □

Now we will discuss a simple generalization of Horn formulas: the renamable Horn Formulas. These formulas allow us to give some use to the otherwise not really useful Horn definition. They also add a condition that can be checked efficiently.

Definition 4.3.3. Let F be a CNF formula. F is called renamable Horn if there is a subset U of the variables $Var(F)$, so that $F[x = \neg x | x \in U]$ is a Horn formula. That set is called a renaming.

Definition 4.3.4. Let F be a CNF formula. Then a 2CNF formula F^* is defined as:

$$F^* = \{(u \vee v) | u, v \text{ are literals in the same clause } K \in F\}$$

Theorem 4.3.1. *The CNF formula F is renamable Horn if and only if the associated F^* formula is satisfiable. Moreover, if satisfying assignment α for F^* exists then it encodes a renaming U in the sense that $x \in U \iff \alpha(x) = 1$.*

Proof. Let F be renamable Horn and U be a renaming. We consider the assignment

$$\alpha(x) = \begin{cases} 1 & x \in U, \\ 0 & \text{otherwise.} \end{cases}$$

Let $\{u \vee v\} \in F^*$ after the renaming. There should be at least one negative variable so if every variable is set to 0, F^* is satisfiable.

The other direction is analogous: let α be an assignment that satisfy F^*. Then there is no to literals in the same clause set to 0. Defining $U = \{x \in Var(F) : \alpha(x) = 1\}$ there is no two positives variables in a clause. □

If a renaming exists, it can be obtained efficiently, and then solve efficiently with the HORNSAT algorithm.

Chapter 5

Complete Algorithms

5.1 Backtracking and DPLL Algorithms

In this section we will talk about algorithms that explore the space of possible assignments in order to find one that satisfies a given formula, or otherwise prove its non-existence. Onward whenever a formula is given, it would be a CNF formula.

5.1.1 Backtracking

We will start with the approach based on the simple and well-known backtracking algorithm.

Algorithm 3 Backtrack

```
1: procedure BACKTRACKING(F)
2:     if 0 ∈ F then return 0
3:     if F = 1 then return 1
4:     Choose x ∈ Var(F)
5:     if backtracking(F{x = 0}) then return 1
6:     return backtracking(F{x = 1})
```

This algorithm describe a recursion with $0(2^n)$ complexity with n being the number of variables. It also lends itself to describe a plethora of approaches varying how we choose the variable x in line 4. This algorithm will be an upper bound in complexity and a lower bound in simplicity for the rest of algorithms in this section.

An easy modification can be done to improve a little its efficiency in the context of k-SAT. Choosing a clause of at most k variable we could choose between $2^k - 1$ satisfying assignments. The recursion equation of this algorithm will be $T(n) = (2^k - 1) * (T(n - k))$, so it would have asymptotic upper bound $O(a^n)$ with $a^n = (2^k - 1)^{\frac{1}{k}} < 2^n$.

5.1.2 Davis-Putman-Logemann-Loveland (DPLL) algorithm

This algorithm is an improvement of the backtracking algorithm, still really simple and prone to multiple modifications and improvements.

We could see to main differences:

- The algorithm try to look for backdoors and simplifications in lines 5 and 6. Although only some of these techniques are present, and even some implementations skip the pure literal search, is an improvement. Search for autarks assignments or renames could also be a good idea.

Algorithm 4 DPLL

```
1:  procedure DPLL(F)
2:      if 0 ∈ F then return 0
3:      if F = 1 then return 1
4:
5:      if F contains a unit clause {p} then return DPLL(F{p = 1})
6:      if F contains a pure literal u then return DPLL(F{u = 1})
7:
8:      Choose x ∈ Var(F) with an strategy.
9:      if DPLL(F{x = 0}) then return 1
10:     return DPLL1(F{x = 1})
```

- It uses heuristics to select variables. It does not imply that they always are better chosen (and there would be cases that run worse), but tend to be better. In practice, hard heuristics approaches give excellent results. citation needed . The idea behind heuristics is trying to reduce as much as possible the number of branching steps. Many heuristics functions have been proposed. For the formulation of some of them we will define:

$$\begin{aligned} f_k(u) &= \text{number of occurrences of literal } u \text{ in clauses of size k} \\ f(u) &= \text{number of occurrences of literal } u \end{aligned} \tag{5.1}$$

  - DLIS (dynamic largest individual sum): choose u that maximizes f. Try first $u = 1$.
  - DLCS (dynamic largest clause sum): choose u that maximizes $f(u) + f(\neg u)$. Try first whichever has largest individual sum.
  - Jeroslaw-Wang: For the one sided version choose u such that maximizes the sum of the weights of the clauses that include the literal. For the two sided version choose a variable instead of a literal.
  - Shortest Clause: choose the first literal from the shortest clause, as this clause is one of the clauses with the biggest weight in F.
  - VSIDS: This heuristics function is a variation of DLIS. The difference is that once a conflict is obtained and the algorithm need to back track, the weight of that literals are increased by 1.

5.1.3 Clause Learning

Despite not being an algorithm, clause learning is a rather useful technique in order to improve any search based algorithm (as DPLL variations). The technique works adding clauses to ensure that once reached a contradiction it would not be reached again, that is, providing new clauses to the CNF formula that, without being satisfied, the formula could not be satisfied. When we add those clauses we avoid the repetitions that led to the contradiction, bounding some branches in a problem specific manner. The content of this subsection is in [21]. The information and definition on UIP is in [23]

In the context of Clause Learning we have to think about an algorithm that works by iteratively expanding a partial assignment as is done in DPLL.

In order to add clarity to the explanation we will introduce some definitions: Conflict clause, decision level, and implication graph. A conflict clause would represent part of an assignment that will never be part of a solution.

Definition 5.1.1. A clause C is a conflict clause of the formula F if:

- $Var(C) \subset Var(F)$
- Each variable in $Var(C)$ appear only once is the clause C.
- $C \notin F$ and for every assignment α such that $C\alpha = 0$ it happens $F\alpha = 0$.

It is clear that the third condition of the definition is the one that add meaning to it. Nonetheless the first two are important to bound the clauses that can be interesting. By adding conflict clauses more constraints are added to the formula, avoiding searching on assignments that will not satisfy the formula. The purpose of clause learning is to find conflict clauses. In order to do that we will make a implication graph and examine it when a conflict happens.

A decision is made every time a variable is assigned and its not part of a unit clause or is a pure literal, i.e., each time we make a non-forced decision. These decisions anidate, and the decision level refer to the number of anidations done when the literal u was assigned to the value a.

The implication graph is the directed graph that has as nodes a pair with a variable an a value assigned to that variable, and there is an edge from (x, a_x) to (y, a_y) if at some point, assign x to a_x make mandatory that y is assigned to a_y. The idea behind this graph is to has a log of the decisions taken to expand the current partial assignment. Formally:

Definition 5.1.2. Let F be a CNF formula,$\{x_i : 1, ..., n\} = Var(F)$, and $A = \{a_i \in 0, 1\} : i \in 1, ..., n\}$. The associated implication graph $\mathcal{G}_{F,A}$ is defined inductively:

- $\mathcal{G}_0$ is the empty graph, and $F_0 = F$.
- From G_{i-1}, F_{i-1} we define G_i, F_i:
  - Let x_j be the first variable in $Var(F_i)$. We add the node named (x_j, a_j).
  - We define $F_i' = F_{i-1}\{x_j \to a_j\}$ and $G_i = G_{i-1}$.
  - Every unit clause $\{l\}$ in F_i, we add to G_i a node (x, a) such that $(l)\{x \to a\} = 1$. Note that (x, a) is unique as l is a literal.
  - Every unit clause $\{l\}$ has an associated clause $C = \{l_{i_1}, .., l_{i_k}\} \in F$ and a associated node (x, a). Necessarily, all other literals in C has been assigned already, so they has an associated node in the graph (y, b). We add to G_i an edge from every associated node $(y, b) \to (x, a)$.
  - We define:

$$F_i = F_i'\{x \to a : (x, a) \text{ is a node associated to a unit clause in } F_i'\}$$

- We repeat the process until either all variables are assigned or a conflict arise. The implication graph has a conflict if there is two nodes with the same variable and opposite value. The resulting graph will be $\mathcal{G}_{F,A}$

We say that x was assigned at *decision level* i if $x \in Var(F_{i-1})$ and $x \notin F_i$.

As we can see a decision graph is dependent of the order on which the variables are assigned (should a decision be made) and the value chosen for each variable when this decision happens.

We will show a little example in order to clarify this definition.

Example 5.1.1. Suppose that we have $F = \{(x_1 \vee x_2 \vee x_3), (x_1 \vee x_2), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (x_4 \vee x_2 \vee \neg x_3)\}$. We make a list $A = \{1, 1, 1, 0\}$ of values associated to each variable. We start by adding the node $(x_1, 1)$ to $\mathcal{G}_1$.

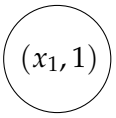

FIGURE 5.1: $\mathcal{G}_1$ before unit propagation

Then we define $F_1' = F_0\{x_1 \to 1\} = F_0\{x_1 \to 1\} = F\{x_1 \to 1\} = \{(\neg x_2), (\neg x_3 \vee \neg x_4), (x_4 \vee x_2 \vee \neg x_3)\}$. We have an unit clause, therefore, we add a node$(x_2, 0)$, as this clause where associated in F with the clause $(\neg x_1 \vee \neg x_2)$ we add an edge $(x_1, 1) \to (x_2, 0)$. As there were only a unit clause, we can define $F_1 = \{(\neg x_3 \vee \neg x_4), (x_4 \vee \neg x_3)\}$.

FIGURE 5.2: $\mathcal{G}_1$ after unit propagation

Once we have F_1 and $\mathcal{G}_i$ we continue the iteration. Therefore we choose the variable x_3 (as x_1 and x_2 where already assigned) and assign it to 1 as we initially decided. Therefore we define $\mathcal{G}_2 = \mathcal{G}_1$, and immediately after we add the node $(x_3, 1)$. We define $F_2' = \{(\neg x_4), (x_4)\}$. We have two unit clauses. Solving them as done above we have:

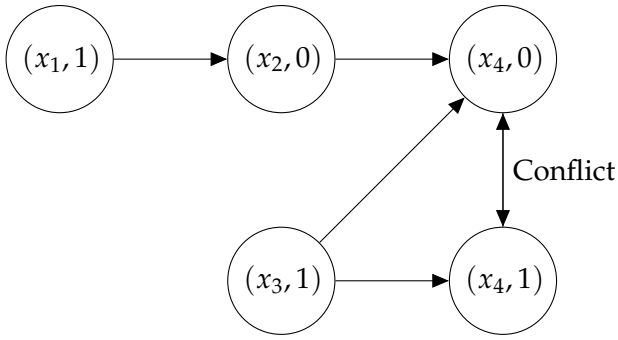

FIGURE 5.3: $\mathcal{G}_2$ after unit propagation

We can see that a conflict has arise. Therefore we can not continue iterating throw the process and $\mathcal{G}_{F,A} = \mathcal{G}_2$. Note that have we wanted to assign the values in other order, only a renaming would have been necessary.

As we already stated, the purpose of the implication graph is to show the root of the conflict. That is, we want to know what assignments led to the conflict. This

could be made by making a set of nodes $N = \{(x_j, a_j : j \in 1, ..., k)\}$ such that every path from a decision node to the conflict has to include one node of the set. This will be named a cut, not confuse with the graph theory concept.

A conflict clause can be made from each cut N: once we have the set N, we can add a new clause $C = (l_1, ..., l_k)$to F such that $l_k = x_k$ if $a_k = 0$ and $l_k = \neg x_k$ otherwise.

Let's summarize what we know until know:

- We know how to make implication graph.
- We know how to add conflict clauses from a cut of a implication graph with conflict.

So we only has to learn strategies in order to detect cuts on implication graphs. Although every cut is enough to add conflict clauses, the most two common approaches are to choose cut are base on the idea of Unique Implication Point(UIP). A UIP is a vertex that dominates both vertices corresponding to the conflicting variable.

- Last UIP - choosing every decision node that has a path to the conflict.
- First UIP - choosing the first unique point encountered. That is, following backward the implication graph from the conflict, choosing the first UIP.

The first UIP tend to produce smaller clauses and experimental results [21] [23] provide proves in favor of it. It is commonplace on DPLL-based solver. The GRASP[5.2.3] algorithm was one of the first conflict driven solver, that is, a sat solver that implement a DPLL procedure based mainly on Clause Learning.

5.2 Other complete algorithms

5.2.1 Monien-Speckenmeyer (MS) Algorithm

This algorithm is a variation of the DPLL-Shortest Clause algorithm, specifying that once you choose the shortest clause, all variables you choose should be from that clause until you satisfy it, as it will continue to be the shortest given that there is no clause with repeated literals as well as no clause that is a tautology. This algorithm (DPLLSC) on k-SAT generates a recursion such that $T(n) = \sum_{i=1}^{k} T(n-i)$. Under the hypothesis that MS does not has a under-exponential worst case complexity, then $T(n) = a^n$ for some $a \in (1, \infty)$. Then

$$a^k = \sum_{i=1}^{k} T(i) = \frac{1 - a^k}{1 - a}$$

that solved in the equation $a^{k+1} + 1 = 2a^k$. The difference between MS and DPLLSC is that MS includes an autark assignment search in addition to the unit clause search and generalizing the pure literal search (that would be a search of autarks of size 1). When we select a clause (the shortest) we first try to generate an autark with its variables and otherwise continue the algorithm.

Other version of the algorithm repeats the last for-loop in the successive calls of F (calling $\texttt{MS}(F\alpha_i)$). Nonetheless we consider that with a deterministic heuristic (that,

Algorithm 5 Monien-Speckenmeyer

```
1: procedure MS(F)
2:     if 0 ∈ F then return 0
3:     if F = 1 then return 1
4:
5:     if F contains a unit clause {p} then return MS(F{p → 1})
6:     if F contains a pure literal l then return MS(F{l → 1})
7:     Choose the shortest clause C = {u_1, ..., u_m}
8:     for i ∈ {1, ..., m} do
9:         α_1 := {u_1 → 0, ..., u_{i-1} → 0, u_i → 1}
10:        if α_i is autark then return MS(Fα_i)
11:    if MS(F{u_1 = 1}) then return 1
12:    return MS(F{u_1 = 0})
```

for example, choose the first clause between the set of clauses with minimum size) the result is equivalent and this provide a simpler algorithm.

For the k-SAT complexity analysis we have to consider whether or not an autark was found. If so, $T(n) \leq T(n-1)$. Otherwise we are applying a non autark assignment that necessarily collide with a clause which size is at most $k-1$. Let us denote by $B(n)$ the number of recursive calls with n variables and under the hypothesis that there is a clause with at most $k-1$ variables. In this case $T(n) \leq \sum_{i=1}^{k} B(n-i)$ and $B(n) \leq \sum_{i=1}^{k-1} B(n-i)$. Both of these cases are worse than $T(n-1)$ so in order to study a worst case complexity we have to study the case when no autark is found. Under the hypothesis that $B(n) = a^n$ we get $a^k + 1 = 2^{k-1}$. For $k = 3$ we obtain $a = \frac{1+\sqrt{5}}{2}$.

5.2.2 Deterministic Local Search

The local search procedure on sat context is the same as in other branches of computer science. The idea is that we start with an initial assignment α and search in the *neighborhood* of α for a satisfying assignment, that is, those assignments that are close to α according to a distance d.

Definition 5.2.1. Let α and β be assignments, we define the Hamming distance $d_{\mathcal{H}}$ as:

$$d_{\mathcal{H}}(\alpha, \beta) = |\{x \in Var(\alpha) \cup (\beta)\} : \alpha(x) \neq \beta(x)\}$$

Note that in case that for every $y \in Var(\alpha) \backslash Var(\beta)$, we can consider that $\alpha(y) = v$, and respectively with β.

For every α we define its neighborhood as $D(\alpha, \delta) = \{\beta : d_{\mathcal{H}}(\alpha, \beta) \leq \delta\}$. In order for this algorithm to work is necessary that $\delta > 0$ and is preferable that $\delta << |Var(\alpha) \cup Var(\beta)|$, in order to avoid doing a backtrack. The procedure determine whether or not there is a satisfying assignment for F in $D(\alpha, \delta)$. The procedure take as input a CNF formula F, an assignment α and a positive integer δ.

For 3-SAT the running time is $O(m3^{\delta})$ where m is the number of clauses. This technique is useful on formulas with a great density of satisfying assignments. Nonetheless, until now this is an incomplete algorithm. The strategy to prove incompleteness is the following. Let F be a CNF formula, such that $Var(F) = \{x_1, ..., x_n\}$

Algorithm 6 Local Search[19]

```
1: procedure LS(F, α, δ)
2:     if Fα = 1 then return Satisfiable
3:     if δ = 0 then return Unsatisfiable
4:     Choose C = l_1, ..., l_n such that Cα = 0
5:     for i ∈ {1, ..., n} do
6:         return LS(F, α ∘ {l_1 → 1}, δ -1)
```

and consider $\delta = n//2 + n\%2$ where $//$ is the integer division and $\%$ is the modulo, and $\alpha_a = \{x_1 \to a, ..., x_n \to a\}$. Then by running $\texttt{LS}(F, \alpha_0, \delta)$ and $\texttt{LS}(F, \alpha_1, \delta)$ we have a complete algorithm. The asymptotic complexity for this algorithm is $O(3^{n/2}) \approx O(2^{0.793n})$. Note that in this context we only work with two-valued assignment, as we are not dealing with partial assignments.

Algorithm 7 Complete Local Search

```
1: procedure CLS(F)
2:     n ← |Var(F)|
3:     α_0 ← {x_i → 0 : 1 ≤ i ≤ n}
4:     α_1 ← {x_i → 1 : 1 ≤ i ≤ n}
5:
6:     if LS(F,α_0, n//2 + n%2) then return Satisfiable
7:     return LS(F,α_1, n//2 + n%2)
```

There is a natural way to generalize the idea of exploring all the space of assignments by coordinates local searches, and that is the covering codes.

Definition 5.2.2. Let X be a set of variables and let $A = \{\alpha_i : 1 \leq i \leq n\}$ a set of two valued assignments over X, and δ be a positive integer. The pair (A, δ) is a *covering code with Hamming radius* δ if for any two-valued assignment α over X, there exists $\alpha' \in A$ such that $d_{\mathcal{H}}(\alpha, \alpha') < \delta$.

We can note that the only important thing about X on a covering code is the number of variables that it has. Therefore a covering code (A, δ) for $X = \{x_1, ..., x_n\}$ is also, after a renaming, a covering code for $Y = \{y_1, ..., y_n\}$, therefore we can consider that A is a *covering code of length* n. When no details about the set of variables is given other than its length n we assume is a covering code over the set $X = \{x_1, ..., x_n\}$

Lemma 5.2.1 (lemma 5.3[19])**.** *For every $\epsilon > 0$ and $\delta \in (0, \frac{1}{2})$ there is a length n_0 so that there is a covering code $C_0 = (A = \{\alpha_i : 1 \leq i \leq t\}, \delta n_0)$ of length n_0, with $t \leq 2^{1-h(\delta)+\epsilon)n_0}$, where $h(\delta)$ is a polynomial function on δ.*

Proof. As done with the LLL, we will prove this result with probabilistic existence. We fix a set of variables $X = \{x_i : 1 \leq i \leq n\}$ choose t random assignments over x following a uniform distribution.

We are going to prove that

$$P(\exists \alpha_0 \forall i \in 1, ..., t : d_{\mathcal{H}}(\alpha_0, \alpha_1) > \delta n) < 1.$$

We can see that

$$
\begin{aligned}
P(\exists \alpha_0 \forall i \in 1,...,t : d_{\mathcal{H}}(\alpha_0,\alpha_1) < \delta n) &= \sum_{\alpha_0 \in A_X} \prod_{\alpha \in A} P(d_{\mathcal{H}}(\alpha_0,\alpha) > \delta n) \\
&= \sum_{\alpha_0 \in A_X} \prod_{\alpha \in A} (1 - P(d_{\mathcal{H}}(\alpha_0,\alpha) \leq \delta n)) \\
&=^{1.} \sum_{\alpha_0 \in A_X} \prod_{\alpha \in A} \left(1 - \frac{\sum_{j=0}^{\delta n} \binom{n}{j}}{2^n}\right) \\
&=^{2.} 2^n \left(1 - \frac{\sum_{j=0}^{\delta n} \binom{n}{j}}{2^n}\right)^t \\
&\leq^{3.} 2^n e^{-t\frac{\sum_{j=0}^{\delta n} \binom{n}{j}}{2^n}} \\
&=^{4.} \left(\frac{2}{e}\right)^n \to_{n\to\infty} 0.
\end{aligned}
\tag{5.2}
$$

Where A_X is the set of all two-valued assignments over X, 1. is because of results on binomials distribution, 2. is because the expresion is independent of eithet α_0 and α, 3. is because properties derived on the fact that $\lim(1 - \frac{1}{n})^n \to e$ and 4. is because we have yet to define t and we choose to define it as:

$$t = \frac{n2^n}{\sum_{j=0}^{\delta n} \binom{n}{j}}.$$

As $(\frac{2}{e})^n \to_{n\to\infty} 0$ for some n_0 big enough we have that $P(\exists \alpha_0 \forall i \in 1,...,t : d_{\mathcal{H}}(\alpha_0,\alpha_1) < \delta n) < 1$ and therefore there exists a covering on which such α does not exists. To end the proof, we can also consider that

$$t = \frac{n_0 2_0^n}{\sum_{j=0}^{\delta n_0} \binom{n_0}{j}} \leq 2^{1-h(\delta)+\epsilon)n_0},$$

for some polynomial function h over δ.

□

Remark 5.2.1. Every covering code $C = (\{\alpha_i : 1 \leq i ł 1k\}, \delta)$ of length n can be *truncated* to a covering code $C' = (\{\alpha'_i : 1 \leq i \leq k\}, \delta)$ of length $m < n$ with α'_i defined as:

$$\alpha'_i(x) = \begin{cases} \alpha_i(x) & x \in \{x_i : 1 \leq i \leq m\} \\ v & \text{otherwise.} \end{cases}$$

Remark 5.2.2. Every covering code $C = (\{\alpha_i : 1 \leq i ł 1k\}, \delta n_0)$ of length n can be *extended* to a covering code $C' = (\{\alpha'_{i_1,...,i_{m//n+1}} : 1 \leq i_j \leq k\}, \delta n)$ of length $m > n$ with α'_i defined as:

$$\alpha'_{i_1,...,i_{m//n+1}}(x) = \begin{cases} \alpha_{i_j}(x) & x \in \{x_i : n(j-1)+1 \leq i \leq nj\} \\ v & \text{otherwise.} \end{cases}$$

Then, for every $\epsilon > 0$, setting $\delta = 0.5$ we have a covering code C_0. We can suppose that for implementing our algorithm we have such covering without any need of processing for our algorithm, as we can brute-force look for it once, and then the algorithm can run as many time as required without any need of repeating those computations.

Algorithm 8 Covering Code Local Search

1: $C_0 \leftarrow$ the covering code provided by the lemma for $\epsilon > 0 \wedge \delta = 0.5$
2: **procedure** COVERING-CODES-LS(F)
3: $n \leftarrow |Var(F)|$
4: **if** $n \leq n_0$ **then return** CLS(F)
5: $C = (A, \delta n_0) \leftarrow$ the extended covering code of C_0 to n variables
6: **for** $\alpha \in A$ **do**
7: **if** LS(F, α, δn) = Satisfiable **then return** Satisfiable
8: **return** Unsatisfiable

This algorithm for 3-SAT run on $O((1.5+\epsilon)^n)$**??**.

In fact what made this algorithm relevant is that its performs different from DPLL algorithm. No only on complexity but on what formulas it is able to solve efficiently.When we reduce a problem to SAT we have to take into account how a SAT will behave with our formulas, and from that we have to consider how to code it (if several alternative codings are available). LS allows us to not only design formulas that will work well in a DPLL search. For more information on how to take advantage of these differences see**??**.

5.2.3 GRASP - Todavía sin rehacer

We present now one of the most cited algorithms. GRASP(Generic seaRch Algorithm for the Satisfiability Problem) was introduced by Marques-Silva and Sakallah[13] that works on CNF formulas. It is based on clause learning techniques, and unit propagation. It divides the search process in four parts:

1. `Decide`: Chooses a decision assignment at each stage of the search process. Based of experimental results it uses the heuristic DLIS.
2. `Deduce`: Which implement a recursive unit propagation as done before.
3. `Diagnose`: Which implement a clause learning procedure.
4. `Erase`: Which delete assignments implied by the last decision.

The method `Erase` is needed as the assignment is considered a global variable. The way that the algorithm work is that each time, either a new conflict clause is added to the formula, and therefore we `Erase` our last assignment to explore other options, or we find an assignment that satisfy the formula.

Chapter 6

Probabilistic Algorithms

In this chapter we will talk about probabilistic algorithms for SAT and k-SAT.When we talk about probabilistic algorithms, we are trying to define an incomplete SAT-solver, with a bounded probability error. This might seems like a big loss in power. Nonetheless, given the complexity of the problem, neither are complete solvers capable of solving all formulas in a feasible time. Therefore, dropping completeness could be a fair exchange in order to get better time complexity.

6.1 Paturi-Pudlák-Zane

The first one that we will consider is the Paturi-Pudlák-Zane(PPZ) algorithm [16] developed in 1997 and its improvements Paturi-Pudlák-Saks-Zane(PPSZ). It was the first probabilistic algorithm for k-SAT proven to work. It has an associated deterministic version that could well be included in the DPLL chapter. Then, some improvements have been done to the algorithm in [17] and [7].

6.1.1 Paturi-Pudlák-Zane

In this subsection we will present the PPZ algorithm and in the next subsection its improved version PPSZ. The information presented here follows the discussion in [17]. The difference between PPZ and PPSZ is some added preprocessing. At the time of release, PPSZ was the asymptotically fastest algorithm for random k-SAT with $k \geq 4$ only improved in 3-SAT by the Schönning random walk algorithm and its improved version the Hofmeister algorithm, because PPSZ were not able to extend the results they found but it was suggested that it should be extendable. At the end, it was proved 9 years later by Hertli [7] that the bounds hold on general.

To define the algorithms, we first define some subroutines. The first of them take a CNF formula F, an assignment α and a permutation π and returns other assignment u.Note that in line 5 and 7 on the procedure `modify`[9]is only checking whether or not we can unit propagate the variable $x_{\pi(i)}$. The algorithm `Search` is obtained by running `Modify` on many pairs (α, π) where α is a random assignment and π a random permutation.

This procedure is the named PPZ algorithm. As we can see is a pretty simple algorithm, but more often than not the work on random algorithms is not to program but to prove them correct. Therefore we will proceed to prove why this algorithm is, in fact, a correct probabilistic algorithm.

Algorithm 9 `Modify subroutine`

```
1: procedure MODIFY(α, π, F)
2:     F_0 ← F
3:     u ← empty partial assignment.
4:
5:     for i ∈ {0, ..., m − 1} do
6:         if {x_π(i)} ∈ F_i then
7:             u+ = {x_π(i) = 1}
8:         else
9:             if {¬x_π(i)} ∈ F_i then
10:                u+ = {x_π(i) = 0}
11:            else
12:                u+ = {x_π(i) = α(x_π(i))}
13:        F_{i+1} = F_i u
14:    return u
=0
```

Algorithm 10 `Search subroutine`

```
1: procedure SEARCH(F, I)
2:     for i ∈ {0, ..., I} do
3:         α ← random assignment on Var(F)
4:         π ← random permutation on 1, ..., |Var(F)|
5:         u ← Modify(α, π, F)
6:         if u(F) = 1 then
7:             return Satisfiable
8:     return Unsatisfiable
```

Search always answers Unsatisfiable if F is unsatisfiable. The only problem is to upper bound the error probability in the case that F is unsatisfiable. In fact, we only have to to find $\tau(F)$: the probability that modify(F, π, α) find a satisfying assignment. The error probability of search would be therefore $(1-\tau(F))^I$. As $1 - x \leq exp(-x)$ with $x \in [0,1]$ them $(1-\tau(F))^I \leq exp(-I\tau(F))$, which is at most $exp(-n)$ where $n = |Var(F)|$ provided $I > n/\tau(F)$. it suffices to give good upper bounds on $\tau(F)$. In order to do that we will prove first two lemmas.

To prove the first lemma we introduce some notation:

Definition 6.1.1. A variable x is forced for an assignment α, a formula F and a permutation π if x is unit propagated in the procedure Modify(α, π, F). $Forced(\alpha, \pi, F)$ is the set of all variables that are forced for (α, π, F)

Lemma 6.1.1. *Let z be a satisfying assignment of a CNF formula G, and let π be a permutation of $\{1, ..., n\}$ and y be any assignment to the variables. Then, Modify(G, π, y)=z if and only if $y(x) = z(x) \ \ \forall x \in Var(G) \backslash Forced(z, \pi, G)$.*

Proof. If $y(x) = z(x) \ \ \forall x \in Var(G) \backslash \text{Forced}(z, \pi, G)$ we prove that $u = z$ where u is the assignment provided by Modify(i, π, F). by induction on i. $x_{\{}\pi(0)\}$ is forced only if F has a unit clause on x, therefore either it is forced for all assignments or it is not forced for any of them. Otherwise $u(x_{\pi(0)}) = z(x_{\pi(0)}) = y(x_{\pi(0)})$ Therefore $u(x_{\pi(0)}) = z(x_{\pi(0)})$. Let suppose that $u(x_{\pi(j)}) = z(x_{\pi(j)})$ for $j < i$. If the variable $x_{\pi(i)}$ is forced on z it should be forced on u to (and to the same value). Otherwise

$u(x_{\pi(j)}) = z(x_{\pi(j)}) = y(x_{\pi(j)})$.

Let i be the first index such that $y(x_{\pi(i)}) \neq z(x_{\pi(i)})$ with $x_{\pi(i)} \notin \text{Forced}(z, \pi, G)$ therefore $u(x_{\pi(i)}) = y(x_{\pi(i)}) \neq z(x_{\pi(i)})$. □

Now, let $\tau(F, z)$ the probability that $Modify(\alpha, \pi, F)$ would return z with random π and α. From the previous lemma:

$$\tau(F, z) = 2^{-n}\mathbb{E}_\pi[2^{|\text{Forced}(z,\pi,F)|}] \geq^{1.} 2^{-n+E_\pi[|\text{Forced}(z,\pi,F)|]}$$

Where 1. is by the convexity of the exponential function and $\mathbb{E}_\pi$ is the expected value with π as variable.

Let v be a variable in $Var(f)$ and z a satisfying assignment of F. let C be a clause in F, then we say that C is critical for (v, z, F) if the only true literal in C is the one corresponding to v. Suppose that π is a permutation such that v appears after all other variables in C. It is easy to follow that $v \in \text{Forced}(z, \pi, F)$ if C is critical for (v, z, F). Conversely, if z is forced it must be critical and appears last on the permutation. Let $Last(v, G, z)$ be the set of permutation of the variables such that for at least one critical clause for (v, G, z), v appears last on the permutation. That is, the set of permutations where v is forced. Let $P(v, z, F)$ the probability that a random permutation is in $Last(v, G, z)$. It follows that

$$\mathbb{E}_\pi[|\text{Forced}(z, \pi, F)|] = \sum_{v \in Var(F)} \mathbb{E}_\pi[v \in \text{Forced}(z, \pi, F)] = \sum_{v \in Var(F)} P(v, z, F)$$

Putting it all together we have:

Lemma 6.1.2. *For any satisfying assignment z of a CNF formula F*

$$\tau(F, z) \geq 2^{-n+\sum_{v \in Var(F)} P(v,z,F)}$$

In particular, if $P(v, z, F) \geq p$ for all variables v then $\tau(G, z) \geq 2^{-(1-p)n}$.

Theorem 6.1.3. *Let F be a k-CNF formula. If F is satisfiable by an isolated assignment, $\tau(F) \geq 2^{-(1-\frac{1}{k})n}$, where n is the number of variables.*

Proof. Let z be a satisfying assignment of F. Then $\tau(F) \geq \tau(F, z)$. If z is an isolated assignment, them for each variable v there is a critical clause C_v and the probability that for a random permutation v appear last is $1/k$. Therefore by the previous lemma

$$\tau(F) \geq \tau(F, z) \geq 2^{-(1-\frac{1}{k})n}$$

□

Then we can think that it is unusual that it is easier to guess a satisfying assignment with such a simple method when there is less satisfiable assignments. We are now going to formalize that intuition, growing on the previous lemmas, and giving similar arguments. For that we will introduce a new concept.

Definition 6.1.2. Let α be an assignment of a proper subset $A \subset Var(F)$. Then the subcube defined by α is the set of the assignments that extends α, i.e. all β that assign all elements in $Var(F)$ and $\beta(x) = \alpha(x), \forall x \in A$.

Lemma 6.1.4. *Let V be a set of variables and let $A \neq \emptyset$ be a set of assignments that map all variables in V. The set of all assignments that map all V can be partitioned into a family $(B_z : z \in A)$ of distinct disjoint subcubes so that $z \in B_z \ \forall z \in A$.*

Proof. If $|A| = 1$ choose B_z as the set of all possible assignments. Otherwise there is two assignments that differ on one variable X. We will partition two subcubes: the one from the assignment that map x to 0 and the assignment that map x to 1. Then we proceed recursively on both subcubes. □

Given a formula F we will apply this lemma to the set $sat(F)$ of assignments that satisfy F, and obtain a family of $\{B_z : z \in sat(F)\}$. We will analyze the probability $\tau(F, z|B_z)$, that is, the probability of $Modify(y, \pi, F)$ returns z given that $y \in B_z$. It is easy to follow that:

$$\begin{aligned}\tau(G) \geq \sum_{z \in sat(F)} \tau(G, z|B_z) Prob(y \in B_z) &\geq \sum_{z \in sat(F)} \min_{\chi \in sat(F)} \{\tau(G, \chi|B_\chi)\} Prob(y \in B_z) \\ &= \min_{\chi \in sat(F)} \{\tau(G, \chi|B_\chi)\}\end{aligned}$$

Further on let z be a satisfying assignment and $B = B_z$. Let N be the set of unassigned variables in B_z (the set of variables that are not assigned equal for all α in B). Writing $Forced_z(y, \pi, F) = N \cap Forced(y, \pi, F)$ we have

$$\tau(F, z|B) \geq 2^{-N + E[|Forced_z(z, \pi, G|]}$$

Therefore $P(v, z, F) \geq 1/k$ for $v \in N$. This is true because z is the unique satisfying assignment in B, hence changing the value in v produce a nonsatisfying assignment. Therefore v is critical on some permutation and analogously as the lemma6.1.2 we have that $P(v, z, F)$.

Theorem 6.1.5 ([17]). *Let F be a k-CNF formula, z a satisfying assignment and let B be a subcube on $Var(F)$ that contains z and no other satisfying assignment. Then:*

$$\tau(G, z|B) \geq 2^{-(1 - \frac{1}{k})|N|}$$

With that we could analyze the complexity of this algorithm. Modify run on $O(nC)$ where n is the number of variables and C is the number of clauses (assign CNF-formula has a worst case of C). Search run on $O(I \cdot O(\text{Modify}))$ supposing that we can get a random number in $O(1)$ and therefore a random assignment and a random permutation on $O(n)$. As we will set $I > n/\tau(G, z|B) > n/\tau(F)$ we get a running time of $O(n \cdot C \cdot 2^{1 - \frac{1}{k}n})$, with a one-sided error probability of e^{-n} (0.049 for 3-SAT).

6.1.2 Paturi-Pudlák-Saks-Zane

This algorithm includes a preprocessing of the formula prior to the searching algorithm. This preprocessing will try to find isolated assignments improving its running time (or at least its complexity analysis). The preprocessing takea as input a CNF formula F and a positive integer I. It uses the concept of resolution: should it happen that we have to clauses $C_1 = \{x_1, ..., x_n\}$, $C_2 = \{y_1, ..., y_{n'}\}$ such that $C_1, C_2 \in G$ and the literal $x_i = \neg y_j$; $i \in \{1, ..., n\}$, $j \in \{1, ..., n'\}$ them we could generate a clause $C = R(C_1, C_2) = \{x_k : k \in \{1, ..., n\} \backslash i\} \cup \{y_k : k \in \{1, ..., n'\} \backslash j\}$ and

Algorithm 11 Resolve subroutine

```
1: procedure RESOLVE(F, s)
2:     F_s = F
3:     while F_s has a s-bounded pair (C_1, C_2) with R(C_1, C_2) ∉ F_s do
4:         F_s = F_s ∧ R(C_1, C_2)
5:     return F_s
6:
7: procedure RESOLVESAT(F, s, I)
8:     F_s = Resolve(F, s)
9:     return Search(F, s)
   =0
```

the formula $F' = F \wedge C$ has the same satisfying assignment that F. A pair of clauses (C_1, C_2) are said to be s-bounded if they are resolvable and $|C_1|, |C_2|, |R(C_1, C_2)| < s$.

With this prepossessing added to the algorithm a better upper bound is proved. Defining

$$\mu_k = \sum_{j=1}^{\infty} \frac{1}{j\left(j + \frac{1}{k-1}\right)}$$

Theorem 6.1.6 (theorem 1. [17]). *Let $k \geq 3$[1], let $s(n)$ a function going to infinity. Then, for any satisfiable k-CNF formula F on n variables,*

$$\tau(F_s) \geq 2^{-(1-\frac{\mu_k}{k-1})n - o(n)}$$

Hence ResolveSat(F,s,I) with $I = 2^{+(1-\frac{\mu_k}{k-1})n + o(n)}$ has an error probability $o(1)$ and running time $2^{-(1-\frac{\mu_k}{k-1})n - o(n)}$ on any satisfiable k-CNF formula, provided that $s(n)$ goes to infinity sufficiently slowly.

By slowly the theorem means that $s(n)$ diverge in $o(log(n))$. Also the term $o(n)$ can be reduced as wished.

[1] Here we are also using the Hertli Result[7].

Part III

Reductions

Chapter 7

Path based Problems

In order to demonstrate the utility a series of reductions will be developed. This will imply a formal approach to the resolution of the problems, as well as deploying a little theoretical background to some problems when needed. Also we would like to show that this techniches provide sometimes really simple approximations to the problems.

7.1 Hamiltonian Cycle

The problem of, given a graph, find a Hamiltonian Cycle is well know to be NP-Complete. Then by Cook Theorem it is known that a reduction from the problem of the Hamiltonian Cycle to SAT exists. This theorem is constructive, so it effectively does give a reduction. Nonetheless, this reduction is unmanageable and in order to use SAT-solvers to improve Hamiltonian cycle resolution it would be necessary to improve it. On this subsection an alternative reduction will be proven.

Definition 7.1.1. A Hamiltonian cycle is a cycle that visit every node in a graph. The associated problem is to check, given a graph, whether whether cycle exists.

We will consider the problem of the Hamiltonian cycle of undirected graphs. Therefore an edge would have two sources instead of a source and a target as it is regarded on directed graphs.

This problem is a very good example to represent what mean to use a SAT-solver to solve a hard problem. The presented reduction is done as shown in [2], with a minor error solved. It move the complexity of the problem from how to solve it to how to implement a SAT-solver. Therefore it only left a worry about what do I need to satisfy in order to solve this problem In order to make the reduction we will represent with Boolean clauses the conditions:

Let $G = (V = \{v_1, ..., v_n\}, E = \{e_1, ..., e_m\})$ be a graph. To reduce it to a CNF-SAT problem, we will first define the variables $\{x_{i,j} : i \in 1, ..., n; j \in 1, ..., n+1\}$. If the variable $x_{i,j}$ is assign to true it would mean that the vertex v_i is in position j in the path. We would like to find a assignment of these variables that satisfy the following clauses:

1. Each vertex must appear at least once in the path, for every vertex v_i: $x_{i,1} \vee ... \vee x_{i,m+1},\ i \in 1, ..., n$.
2. Each vertex must not appear twice in the path, unless it is the first and last node: $\neg x_{i,j} \vee \neg x_{i,k},\ i \in 1, ..., n,\ j \in 2, ..., m+1, k \in 1, ..., m$.
3. Every position in the path must be occupied: $x_{1,i} \vee ... \vee_{n,i},\ i \in 1, ..., m+1$.

4. Two consecutive vertex have to be adjacent: $\neg x_{i,j} \vee \neg x_{i+1,k} \; \forall (k,j) \notin E$.

Let now solve that this is a correct reduction, i.e., that an assignment that can satisfy this clauses exists if, and only if, the graph G has a Hamiltonian graph. If such an assignment exists we can make a Hamiltonian cycle with the variables sasigned to 1. On the other hand if such cycle exists an assignment that assign to 1 the variable $x_{i,j}$ given that the vertex v_i is in position j in the path would satisfy all the clauses.

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[2] Dor Cohen. *Find hamilton cycle in a directed graph reduced to sat problem*. Computer Science Stack Exchange. URL:https://cs.stackexchange.com/q/49593 (version: 2016-07-29).

[3] Evgeny Dantsin et al. "Deterministic algorithms for k-SAT based on covering codes and local search". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2000, pp. 236–247.

[4] Paul Erdős and László Lovász. "Problems and results on 3-chromatic hypergraphs and some related questions". In: *Colloquia Mathematica Societatis Janos Bolyai 10. Infinite And Finite Sets, Keszthely (Hungray)*. Citeseer. 1973.

[5] Carla P Gomes et al. *Exploiting Runtime Variation in Complete Solvers.* 2009.

[6] Philip Hall. "On representatives of subsets". In: *Classic Papers in Combinatorics*. Springer, 2009, pp. 58–62.

[7] Timon Hertli. "3-SAT Faster and Simpler—Unique-SAT Bounds for PPSZ Hold in General". In: *SIAM Journal on Computing* 43.2 (2014), pp. 718–729.

[8] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. "Introduction to automata theory, languages, and computation". In: (2007).

[9] Alfred Horn. "On sentences which are true of direct unions of algebras". In: *The Journal of Symbolic Logic* 16.1 (1951), pp. 14–21.

[10] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. "PySAT: A Python Toolkit for Prototyping with SAT Oracles". In: *SAT*. 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26. URL: https://doi.org/10.1007/978-3-319-94144-8_26.

[11] Russell Impagliazzo and Ramamohan Paturi. "On the complexity of k-SAT". In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375.

[12] Victor W Marek. *Introduction to mathematics of satisfiability*. CRC Press, 2009.

[13] João P Marques-Silva and Karem A Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[14] Robin Moser. "A constructive proof of the Lovász local lemma". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 343–350.

[15] Robin Moser. *Exact Algorithms for Constraint Satisfaction Problems*. Logos Verlag Berlin GmbH, 2013, pp. 17–45.

[16] Ramamohan Paturi, Pavel Pudlák, and Francis Zane. "Satisfiability coding lemma". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE. 1997, pp. 566–574.

[17] Ramamohan Paturi et al. "An improved exponential-time algorithm for k-SAT". In: *Journal of the ACM (JACM)* 52.3 (2005), pp. 337–364.

[18] Karem A Sakallah. "Symmetry and Satisfiability." In: *Handbook of Satisfiability* 185 (2009), pp. 289–338.

[19] Uwe Schöning and Jacobo Torán. *The Satisfiability Problem: Algorithms and Analyses*. Vol. 3. Lehmanns media, 2013.

[20] Joel Spencer. "Asymptotic lower bounds for Ramsey functions". In: *Discrete Mathematics* 20 (1977), pp. 69–76.

[21] Richard Tichy and Thomas Glase. "Clause learning in sat". In: *University of Potsdam* (2006).

[22] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[23] Lintao Zhang et al. "Efficient conflict driven learning in a boolean satisfiability solver". In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE. 2001, pp. 279–285.