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

May 13, 2020

Contents

Chapter 1

Theoretical Introduction

1.1 Introducction

TODO: incluir introducción del problema.

Info: - Handbook incluye una introducción histórica. - Libro verde incluye una introducción excelente.

1.2 Boolean Algebra

We could have started the topic right from the axioms. Nonetheless, given the goal we want to achieve, it seems excesive. We will reffer 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 widely extended in related literature. In particular we follow the approach shown on [6].

Definition 1.2.1. A partial ordered set, also poset, is a pair $\{X, \leq\}$ where X is a set and
le is a partial order of X. A chain Y of a $\{X, \leq\}$ is a subset of X where $\leq$ is a total order.

Proposition 1.2.1 (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 1.2.2. 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 will have two new operations defined, given two elements $x, y \in X$

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

A lattice is complete if every subset has a unique largest element and a unique lowest element. A lattice could be presented generaly as a duple $\{L, \leq\}$, a triple $\{X, \vee, \wedge\}$ and, if posible, would be presented as a quituple $\{X, \vee, \wedge, \top, \bot\}$ where $\top$ is the greatest elemente and $\bot$ the lowest element. a lattice is called distributive i $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

With the contex of lattice just included, we will present the *Knaster and Tarski fixpoint theorem* fixpoint theorem. In order to do that we will introduce some notation. Given $f : \{L, \leq\} \to \{L, \leq\}$ a function 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 1.2.1 (proposition 1.2 [6]). *Let $f : \{L, \leq\} \to \{L, \leq\}$ be a monotone funtion 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 motone.

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

□

Definition 1.2.3. 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. $\neg\neg x = x$

1.3 Propositional Logic

1.4 Definitions and first concepts.

In this section Boolean formulas will be introduced. We first start with the basic building blocks, which collectively form what is called the alphabet. Namely,

- Symbols x, y, z for Boolean variables.
- Symbols p, q, r for Boolean metavariables, that is, a variable that refer to a boolean variable or a negated (see below) boolean variable.
- Values 0 and 1, referring to false and true respectively. The set $\{0, 1\}$ will be named as $\mathbb{B}$.
- Boolean Operators:
  - unary: $\neg$
  - binary: $\wedge, \vee, \to, \oplus, \leftrightarrow$

We will consider $\wedge$ of greater priority than $\vee$. These operator are defined by theirs truth table:

$\neg$	
0	1
1	0

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

$\rightarrow$	0	1
0	1	1
1	0	1

$\oplus$	0	1
0	0	1
1	1	0

$\leftrightarrow$	0	1
0	1	0
1	0	1

Definition 1.4.1. 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 symbol of two formulas is a formula too.

Examples of formulas are $x \vee y$ or $x_1 \wedge x_2 \vee (x_4 \vee \neg x_3 \wedge (x_5 \rightarrow x_6) \vee 0)$.

Definition 1.4.2. Given a set A it has an associated homonym problem that consists on, given an arbitrary element e check if $e \in A$.

Definition 1.4.3. An assignment is a function α from the set of Boolean formulas to the set of Boolean formulas, on which some variables $\{x_1, ..., x_n\}$ are replaced by predefined constants $\{a_1, ..., a_n\}$ respectively.

If none of the variables altered by an assignment α are present on the formula F then $\alpha(F) = F$. We denote as $Var(\alpha)$ the set of those variables that receive a value from α. Analogously, $Var(F)$ will denote the variables present on a formula F.

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. $\alpha = \{x_1 \rightarrow 1, ..., x_n \rightarrow 0\}$. For example given an assignment $\alpha_0 = \{x_1 \rightarrow 1, x_2 \rightarrow 1, x_3 \rightarrow 0\}$ and $F_0 = x_1 \rightarrow (x_2 \wedge x_4)$ then $F_0\alpha_0 = 1 \rightarrow (1 \wedge x_4) = x_4$.

Definition 1.4.4. 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.

Definition 1.4.5. A formula F is called *satisfiable* if $\exists \alpha : F\alpha = 1$. Otherwise it is called *unsatisfiable*. The set of all satisfiable formulas is denoted as SAT. The problem SAT is the associated problem. An assignment α that satisfies F is called a model an is denoted as $\alpha \models F$.

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

Definition 1.4.6. A formula F is said to be in conjunctive normal form if is written as:

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

Where $C_i = (u_{1,i} \vee ... \vee u_{m_i,i})$ and $u_{i,j}$ are literals, that is, variables or negated variables. The set of all formulas in conjunctive normal form is called CNF.

A formula in *CNF* could be seen as a collection of clauses. The associated problem with *CNF* is straightforward on $O(n)$. The problem that we will investigate is whether a arbitrary formula F have a *SAT-equivalent CNF* formula. Equivalently a clause could be seen as a set of literals. The set of all formulas in conjunctive normal form where $|C_i| = N\ i \in 1, ..., n$ is called *NCNF*. The intersection of these set with the *SAT* sets are called *CNF-SAT* y *NCNF-SAT*. If the context is clear enough the problems will be called *CNF* and *NCNF*

We could define an equal relationship on the set of formulas. Let F, G be formulas. Then $F = G$ if it happens that for each α an assignment such that $F\alpha = 1$ then $G\alpha = 1$ and $G\alpha = 1$ then $F\alpha = 1$

Proposition 1.4.1. The given equal relationship is a equivalence relationship.

Proof. All three properties follows from the equivalent properties on the constants. □

We could define a partial order relation between the formulas. Let F, G be formulas. Then $F \leq G$ if it for each α an assignment such that $F\alpha = 1$ then $G\alpha = 1$.

Proposition 1.4.2. The given order relationship is well-defined.

Proof. As we then could see each class of equivalent as the set of assignment that satisfies all of the clauses, this property arises from the order given by the inclusion on sets.

□

TODO: Definir un circuito.

Lemma 1.4.1. *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 1.4.2 (Tseitin [15])**.** *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 computad 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.

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

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.

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

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

In the last item we use the third one.

□

This result is important because, now we could be able to talk only about *3-CNF* formulas. 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-CNF* then we will be able to solve 'full'SAT problems.

Definition 1.4.7. An assignment $\alpha is called autark for a formula F \in CNF$ if for every clause $C \in F$ it happens that if $Var(C) \cap Var(\alpha) \neq \emptyset$ then $C\alpha = 1$, in other words it satisfies all clauses that it 'touches'.

The use of this definition is self-evident, as it would simplify the problem of resolving a *CNF* clause. The strategy would be simple as obvious: try to make every clause positive. This assignment will give simplifications of the problem, and enabling a good method for the search will be useful.

Should it happen that we got an algorithm for autarks clauses, and iterating it, we could find a solution of any given formula. Finding a polynomial algorithm that finds whether there exists any non-empty autark formula and provide it, we could be able of proving that NP = P, as we could solve SAT applying this algorithm iteratively. Anyway, trying to find simple autark assignment, i.e. assignment with not many variables,is a good praxis.

Proposition 1.4.3. We could reduce the SAT-CNF problem 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.

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). □

Proposition 1.4.4. 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

□

Chapter 2

Resolution Algorithms: Solvers

This chapter is fundamental as it 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.

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

2.1 Satisfiability by Combinatorics

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

Nonetheless, given an efficent non-constructive SAT-solver, it is possible to make a constructive solver with a running time of n times the previous by assigning a literal a value, checking whether or not the result is still solvable. EXTENDER LA IDEA.

As general SAT-solver is exponential, 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.

Definition 2.1.1. Let F be a formula. It is in call a k-CNF formula if it is in CNF and $\forall C \in F, |C| = k$.

Proposition 2.1.1. 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 2.1.2. 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 2.1.0.1. *For a formula in CNF, if the sum of the weights of the clauses is less than one then the formula is satisfiable.*

Definition 2.1.2. 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 [12]. For that we will need the well known Hall marriage theorem[3].

Definition 2.1.3. 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 $\Gamma_G(X) = \Uplus_{n \in X} \Gamma_G(n)$

Theorem 2.1.1 (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 2.1.2. *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 2.1.3. 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. □

2.2 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[8].

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.

2.2.1 First definitions

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

Definition 2.2.1. 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 2.2.2. 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 2.2.3. 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*.

2.2.2 Statement of the Lovász Local Lemma

Theorem 2.2.1 (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 2.2.1.1 (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 2.2.1.2 (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 2.2.1.3. *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 2.2.1.2. 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.

□

2.2.3 Nonconstructive proof of 2.2.1

We explain the way Erdös, Lovász and Spencer originally proved the Lemma [2] [13]. The write-up presented here will resemble the one done by [9].

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

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

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

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

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

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

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 $P\left(A \Big| \bigcap_{B\in\mathcal{H}} \overline{B}\right) = 0 \leq P(A)$ 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 $\mathcal{H}_0 = \mathcal{H} \backslash \mathcal{H}_A$.

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

We will bound numerator and denominator. For the numerator:

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

Where the second inequality is given by the definition of lopsidependency graph. On the other hand, for the denominator, we can define $\mathcal{H}_i := \{H_i, ..., H_k\} \cup \mathcal{H}_0$.

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

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 2.2.1.

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

Where in 5. is used 2.2.2 and since $\mu : \mathcal{A} \to (0,1)$ then $P\left(\bigcap_{A\in\mathcal{A}} \overline{A}\right) > 0$.

□

2.2.4 Constructive proof of 2.2.1.2

Moser[9] proves that it exists an algorithm such that it gives an assignment satisfying the SAT formula, should it happen that the formula satisfies 2.2.1.1 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 [12].

Algorithm 1 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 2.2.1.1 it is proved that if will end after running Repair at most $O(\sum_{C\in F} \frac{\mu(C)}{1-\mu(C)})$

2.3 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*[12].

Definition 2.3.1. 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).

2.3.1 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 find autark assignments.

Similar to this concept is the concept of pure literal. A literal u is a pure literal for a formula F if there is no $\neg u$ in F. The assignment that only maps $u \to 1$ is autark for F.

2.3.2 2CNF

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

Proposition 2.3.1. 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.

□

2.3.3 Horn Formulas

In this subsection we will analyze Horn formulas. They named after Alfred Horn[5]. They are of special interest as HORNSAT is P-complete.

Definition 2.3.2. 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 2.3.2. 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 2.3.3. 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 2.3.4. 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 2.3.1. *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.

2.4 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.

2.4.1 Backtracking

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

Algorithm 2 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-CNFSAT. 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 $a^n = (2^k - 1)^{\frac{1}{k}} < 2^n$.

2.4.2 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.

Algorithm 3 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 DPLL(F{x = 0})
```

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.

- 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 roles of heuristics is to reduce the branching steps. Because of this, many heuristics functions have been proposed. For the formulation of some of them we will define:

$$\begin{aligned} f_k(u) &= \text{number of occurrences of literal } u \text{ in clauses of size k} \\ f(u) &= \text{number of occurrences of literal } u \end{aligned} \tag{2.1}$$

  - 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.

2.4.3 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 $MS(F\alpha_i)$). Nonetheless we consider that with a deterministic heuristic (that, for example, choose the first clause between the set of clauses with minimum size) the result is equivalent and this provide a simpler algorithm.

Algorithm 4 DPLL

```
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 u then return MS(F{u = 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 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 $a = \frac{1+\sqrt{5}}{2}$.

2.4.4 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 [14]. The information and definition on UIP is in [16]

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 2.4.1. 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 decision level refer to the process of problem solving in a DPLL algorithm. Each time we could not find any autark to append to our future assignment we have to take a 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 at a particular decision level of the algorithm, 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 implication graph has a conflict if there is two nodes with the same variable and opposite value. The implication graph is made iteratively, adding elements at each decision level. A node is called a decision node if that assignment was not implied (it was made by decision).

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. We would like to choose the nodes that led to the conflict. This could be made choosing nodes such that every path from a decision node to the conflict has to include one of the nodes. This will be named a cut, not confuse with the graph theory concept. A conflict clause can be made from each cut.

A common and efficient approaches are to choose cut are base on the idea of Unique Implication Point(UIP). A UIP is a vertex at the current decision level 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 have smaller clauses and experimental results [14] [16] provide proves in favor of it. It is commonplace on DPLL-based solver.

2.4.5 GRASP

We present now one of the most cited algorithms. GRASP(Generic seaRch Algorithm for the Satisfiability Problem) was introduced by Marques-Silva and Sakallah[7] 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.

Algorithm 5 GRASP

```
1:  procedure SEARCH(d)(F)
2:      //d: decision level
3:      if Decide(d) then return 1
4:      while True do
5:          if Deduce(d) != Conflict then
6:              Search(d+1)
7:          if Deduce(d) == Conflict then
8:              if Diagnose(d) then Erase(); return Conflict
9:          Erase
10: return Search
        =0
```

2.5 Probabilistic Algorithm

In this section we will talk about probabilistic algorithms for CNFSAT and k-CNFSAT. The first one that we will consider is the Paturi-Pudlák-Zane(PPZ) algorithm [10] developed in 1997 and its improvements Paturi-Pudlák-Saks-Zane(PPSZ). It was the first probabilistic algorithm for k-CNFSAT proven to work. It has an associated deterministic version that could well be included in the DPLL section. Then, some improvements have been done to the algorithm in [11] and [4].

2.5.1 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 [11]. 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 [4] 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.

TODO: Explicar lo que es $G_i yu$

Note that in line 5 and 7 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.

TODO: poner los nombres del algoritmo en smallcaps

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.

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 $1 - x \leq exp(-x)$

Algorithm 7 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
```

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 2.5.1. 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 2.5.1. *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:

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

TODO: DECIR que es $\mathrm{E}_p i$
Where 1. is by the convexity of the exponential function.

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

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

Putting it all together we have:

Lemma 2.5.2. *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 2.5.3. *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 2.5.2. 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 2.5.4. *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 lemma2.5.2 we have that $P(v,z,F)$.

Theorem 2.5.5 ([11]). *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).

2.5.2 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 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 $|C_1|, |C_2|, |R(C_1,C_2)| < s$.

Algorithm 8 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
```

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 2.5.6 (theorem 1. [11]). *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[4].

Chapter 3

Reductions

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.

3.1 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 3.1.1. 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 [1], 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] 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).

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

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

[16] 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.