UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Preprocessing algorithms for SAT and Pseudo-Boolean solvers

Final project

Ramon Cano Aparicio
Director: Robert Nieuwenhuis, Computer Science Department

June 2022

Contents

1 Introduction and contextualization

1.1 Introduction

Many real-life problems can be expressed as a set of restrictions. If those restrictions can be satisfied, then there's a solution to the problem, otherwise, the problem can be classified as not solvable.

The set of those problems that can be codified using boolean formulas, that is to say, where the variables of the problems can take the values 0 and 1, are wildly known as SATISFIABILITY (SAT) problems. Therefore, if a program is capable of assigning values to the variables such that all the restrictions of a given problem are satisfied, then the solution to the problem is found.

Although many problems can be easily represented this way, deciding whether there's a solution to the problem or giving a solution if it exists is not easy to compute.

SAT is the first family of problems proven to be NP-Complete by Stephen Cook and Leonid Levin. Therefore, there's no known algorithm that runs in polynomial time capable of solving this kind of problem.

This project tries to transform the SAT problem into an LP problem, as well as applying preprocessing techniques to express the information in a more optimized way for the LP solver.

The code of the project is available in `https://github.com/rcanoaparicio/tfg-sat-preprocessing`.

1.2 Context

This TFG project takes place in the context of the Logics and Programming research group at the UPC and its spin-off Barcelogic. Barcelogic is a company that uses Logic Programming to solve various types of problems: sports league scheduling, mobility or planning.

This project aims to further improve the solver technology developed in this group. Concrete solvers include systems for deciding/optimizing combinatorial problems expressed as logical formulas or as 0-1 Integer Linear Programs (ILPs). We will focus on formula preprocessing techniques: how to express input knowledge in the most compact and powerful form, and also on inprocessing, that is, doing the same for new formulas being learned by the solver.

This work has a theoretical side, studying the correctness and power of different methods and devising new algorithms for them, but it also has a practical experimental side, implementing (mostly in C++) and benchmarking.

1.3 Concepts

In this section, some key concepts are described in order to better follow the explanations.

1.3.1 SAT Problem

SAT Problem is a family of problems consisting of if there exists an interpretation of a boolean formula that satisfies it.

1.3.2 NP and NP-complete

These are terms used in computational complexity theory.

NP (Nondeterministic Polynomial time) is a class of problems "for which there exists an efficient certifier"[4].

NP-Complete is a class of problems "computationally hard for all practical purposes, though we can't prove it"[4].

1.3.3 Clause

A clause is a set of literals (can take the value of true or false) and a logical connection between them.

1.3.4 CNF

CNF (Conjunctive Normal Form) is a conjunction of clauses, where each of the clauses is a disjunction of literals.

1.3.5 Constraint logic programming

Constraint Logic Programming is "a combination of logic programming and constraint solving [...] Programmers are given more control by having constraints for variables in the body of a program. The body of a constraint logic program is evaluated similarly to normal logic programming, however, the constraints which are in the body must be satisfied."[5]

1.3.6 Cardinality constraint

A cardinality constraint is a constraint of the form $l_1 + ... + l_n >= k$, where the l_i are the literals and k is a natural number.

1.3.7 Subsunction

A is said to subsume B if only if A is an equivalent or stronger representation of B. Let $A = L \geq k$ and $B = L' \geq k'$

$L \geq k \models L' \geq k'$ iff $|L/L'| \leq k - k'$

For example:

$L_1 + L_2 \geq 10 \models L_1 + L_3 \geq 6$ iff $|L2| \leq 4$

1.3.8 Probing

Probing is the procedure used for detecting *hidden binary clauses*, that is, clauses that are not explicit in the problem.

2 Justification

2.1 Previous studies

Due to the importance of the subject, there already exist various previous studies addressing the preprocessing of this kind of problem. Most of them are focused on solving the problem using the well known DPLL algorithm and trying to find ways to improve it.[8] There are some cases, though, in which the modern solves still struggle, for example with the *pigeon wholes* problem.

Other studies focused on this kind of problem [9], using techniques adapted to this kind of problem in order to solve them faster.

2.2 Justification

This project, instead of being centred on improving the existing algorithm, will focus on the previous step, that is, the preprocessing of the input, transforming it into an Integer Linear Programming problem.

3 Scope

3.1 Objectives and sub-objectives

The general objective of this project is to find a more powerful form of expressing the original input of the problem such that the final program computes the result at a faster speed.

For doing so the input will first obtain information from the input that is not explicit. Then will express that information in a compact way.

3.1.1 Theoretical objectives

- Find existing algorithms of preprocessing
- Make modifications of the existing algorithms in order to make them fast enough
- Identify the possible parallelization of the algorithms
- Compute the temporal and spatial cost of the different solutions

3.1.2 Practical objectives

- Implement the different algorithms in C++
- Benchmark the results and make comparisons
- Obtain a faster way to calculate the solution for the problems
- Make this project act as a base for future development from other students who are interested in the field

3.2 Requirements

Some requirements are needed to ensure the quality of the project:

- Write clean code that can be understood by other programmers
- Write tests in order to ensure the correctness of the result
- Make use of the most appropriate structures for each problem
- Make an implementation being conscious of the architecture of the machines, accesses to memory and cache
- Machines with certain computational power to perform the execution and benchmarks

3.3 Potential obstacles and risks

Through the investigation, implementation and documentation of the process, some risks and possible obstacles can occur and, thus, need to be taken into account:

- **Not being able to find a path in an early stage of the project**. Since it's an unsolved and investigation project, it's possible not to be able to find a path to advance.
- **Inexperience in the field**. Not having prior experience in investigation projects in general, and more specifically in Logic Programming projects.
- **Having no access to proper equipment**. Having no access to the necessary equipment in order to run the experiments and obtain reliable results.

4 Methodology and rigour

From the beginning of the project, an agile methodology is defined.

4.1 Methodology

Since it's an investigation project, continuous communication, as well as fast prototyping of ideas, are key factors for the success of the dissertation. Therefore, there will be a meeting every week. Every meeting will be divided into two main parts:

- **Analysis of the results**. Explain the results obtained through the week and detect existing or potential problems.
- **Next step**. Although a general path is defined from the beginning, it's important to define smaller tasks. This way it will be easier to react to possible problems, explore new ways and reject the ones not useful.

In the beginning, when there are no results yet, more time will be spent ensuring the correct understanding of the problem and the approaches for solving it.

4.2 Monitoring tools

Github will be used as version control tool. The existence of branches makes the creation and maintenance of different versions of the code easier. The code of the project will be public and accessible to everyone through the development of the project.

5 Description of tasks

The duration of the project will be of 9 months, with an expected total duration of 540 hours approximately. The project start date is on September 28th of 2021 and its delivery date is between the 27th of June and 1st of July 2022.

Each week 15 hours distributed in 3 days will be dedicated to the development of this project. This estimated time includes a variety of tasks such as planning, investigation and implementation.

Following each task will be defined, providing a definition and an estimation of the time needed for each one in hours. Important factors to take into account are: identifying dependencies between the tasks, as well as taking into account which tasks are more critical and the potential risks and obstacles.

The tasks defined below are not listed in sequential order, since some of them may be done in parallel, but they're ordered according to the natural sequence of events in the project.

5.1 Weekly meetings

Through the project, there will be a weekly meeting of 1 hour with the tutor of the thesis. What will be done in each meeting will vary according to the phase of the project.

- **Instruction.** During the first phase of the project the main purpose of the meetings will be to understand the project itself, this is, define the goals, learn the theoretic base needed for the development of the project and solve mostly theoretical doubts that may emerge.
- **Investigation.** The next step will be to investigate and experiment. In this phase, the meetings will be used to define the smaller short-time objectives, solve more practical doubts, extract conclusions from the results obtained and, again, define the new objectives. This phase will follow a very cyclic pattern in form of sprints. Each sprint will have a duration of around a month, depending on the estimated difficulty of the task planned.
- **Conclusions and final results.** Once the investigation phase is done, the results obtained so far will be benchmarked and conclusions will be extracted.

5.2 Project planning

This will be the first task of the project. When developing a project, especially if it's a complex and long term one like this one, it's important to define the scope of the project, the resources needed to achieve the goals and the resources available to succeed.

In each of the items listed below, the time dedicated in the previous study of *project management* is also taken into account.

- **Contextualization and project scope.** In this step is defined the main goal of the project, as well as secondary goals that can vary according to the resources available and accomplishments. Therefore, it's important in this step to well define which goals are the most important ones to prioritize them if needed. All this will be done with the tutor and director of the thesis.
- **Time planning.** Once the tasks are defined, the next step will be to define the estimated duration of each task.
- **Economic management and sustainability.** Definition of the cost for each of the tasks and resources needed as well as the sustainability of the project.
- **Final document**. Once all the subtasks are finished, a final document containing all the information will be redacted, making sure everything is coherent and making the needed corrections.

5.3 Previous study

Before starting the implementation, a previous study must be done. Being mostly an investigation thesis, only having a good theoretical basis we can ensure the success of this project.

- **Logic Programming basis.** All the basic theoretic concepts needed to proceed and understand the project.
- **Problem definition.** Once the theoretic basis is established, the next step will be to understand the problem.
- **Current solutions.** Find already existing algorithms for solving the problem.

5.4 Practical implementation

This phase will be very cyclic, mainly consisting of implementing an idea, obtaining results and, according to those results, identifying the problems that can be solved and defining the next steps of the investigation.

- **Implementation of the base algorithm.** Naive implementation of the theoretical algorithm using C++. The implementation will be then tested using a small test input to identify the problems of the solution.
- **Problem identification.** The implementation will be tested using a small test input to identify the problems of the solution.
- **Investigate a solution.** For each of the problems, a solution will be investigated and a new algorithm will be defined.
- **Implementation of the new algorithm.** Implementation of the new theoretical algorithm using C++. The implementation will be then tested using a small test input to identify the problems of the solution.

5.5 Final experimentation, analysis and conclusions

The algorithms will be tested using a final set of inputs and then conclusions will be taken.

- **Testing the algorithms.** The different versions of the algorithms will be tested using a set of inputs. For each one, the results will be benchmarked.
- **Results analysis and conclusions.** Analyze the results obtained and make the conclusions.

5.6 Documentation

Through the project, all the tasks and results will be documented.

- **Project planning**. Making the pertinent corrections and adding the project planning information to the final document.
- **Implementation**. Explanation of the algorithms used their costs and problems.
- **Results and conclusions**. Final results and conclusions obtained.
- **Exposition**

5.7 Resources

To develop the thesis there exist a few needed resources, both human and material.

5.7.1 Human Resources

The human resources needed are

- **Project manager**. In charge of planning the project.
- **Researcher**. In charge of analysing results and thinking of new ways to solve the problems.
- **Developer**. In charge of implementing the algorithms.
- **Tester**. In charge of testing the implementations and gathering the results.
- **Mentoring**. Tutor of the thesis.

5.7.2 Material resources

The material resources needed for this project are

- **Computers**. 16GB of RAM and AMD Ryzen 5 3600 6-Core 3.60GHz CPU
- **IDE**. Atom and Visual Studio
- **Compiler**. g++
- **Atenea**.
- **Version Control software**. Github is used for this project.

6 Time estimations

6.1 Task time estimations and dependencies

Task	Description	Time(h)	Dependencies
T1	**Weekly meetings**	**38**	
T2	- Instruction	6	-
T3	- Investigation	30	T2
T4	- Conclusions and final results	2	T3
T5	**Project planning**	**65**	
T6	- Contextualization and project scope	25	-
T7	- Time planning	15	T6
T8	- Economic management and sustainability	15	T7
T9	- Final document	10	T6, T7, T8
T10	**Previous study**	**60**	
T11	- Logic Programming basis	40	-
T12	- Problem definition	10	-
T13	- Current solutions	10	-
T14	**Practical implementation**	**300**	
T15	- Implementation of the base algorithm	20	T10
T16	- Problem identification	100	T15
T17	- Investigate a solution	100	T15
T18	- Implementation of the new algorithm	80	T15
T19	**Final experimentation, analysis and conclusions**	**15**	
T20	- Testing the algorithms	10	T16, T17, T18
T21	- Results analysis and conclusions	5	T20
T22	**Documentation**	**55**	
T23	- Project plannig	5	-
T24	- Implementation	30	-
T25	- Results and conclusions	10	-
T26	- Exposition	10	T23, T24, T25

Table 1: Time estimation and dependencies for each task.

6.2 GANTT diagram

Estimated schedule using a GANTT diagram.

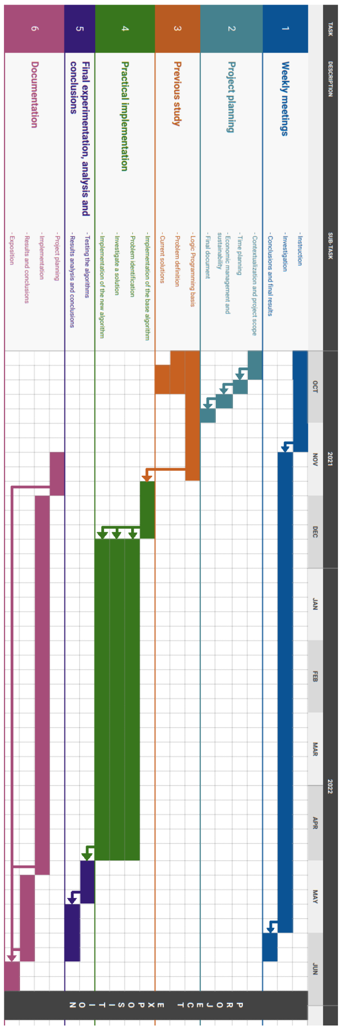


Figure 1: GANTT diagram of the tasks

7 Risk management

There exist some potential risks in the course of developing this project. Here are described the ways of preventing them or acting once they appear to eliminate them or try to reduce their effects as much as possible.

7.1 Not being able to find a path in an early stage of the project

Since it's an unsolved and investigation project, it's possible not to be able to find a path to advance towards to.

- **Impact:**. High
- **Probability:** Low. Before starting the project, the thesis tutor has already some ideas.
- **Proposed solution:**. Try to work with the tutor, look for already existing solutions to the problem or try to exploit those parts where some results were achieved.

7.2 Inexperience in the field

Not having prior experience in investigation projects in general, and more specifically in Logic Programming projects.

- **Impact:**. Low
- **Probability:** High
- **Proposed solution:**. Asking for help from the tutor, spend more time looking for resources.

7.3 Having no access to proper equipment.

Having no access to the necessary equipment in order to run the experiments and obtain reliable results.

- **Impact:**. Having no access to proper equipment
- **Probability:**. Very low.
- **Proposed solution:**. Nowadays finding new equipment is relatively easy. In case it wasn't possible, any other machine could be used, although it may alter the rigour of the results slightly.

8 Budget

In this section, the economic cost of the project will be discussed. The cost of the project will be estimated following the tasks listed before, which will include material and personnel costs.

8.1 Personnel Costs

First of all, the different roles needed will be defined for each task. The salary for each role will be estimated using the average salary[1] for that position and the hours calculated for each of the tasks.

As for the positions, many are considered Junior roles, since a lot of time dedicated to formation due to inexperience is taken into account when estimating the time of the project.

- **Weekly meetings**. In all the weekly one-hour meetings the *Junior Project Manager* will be present, as well as the *Junior Researcher* and the *Project Tutor*.
- **Project planning**. This task will be executed by the *Junior Project Manager*.
- **Previous investigation**. This task will be executed by the *Junior Researcher*, in charge of thinking of new solutions.
- **Practical Implementation**. This task will be executed both by the *Junior Researcher* and the *Junior Developer*.
- **Final experimentation, analysis and conclusions**. The tests will the driven by the *Tester*, and then the results will be analysed by the *Junior Researcher* and *Project Tutor*.
- **Documentation**. The final documentation of the project will be redacted by the *Junior Researcher*.

Once the roles for each task are defined, the cost of each one using the following salaries. For the sake of simplifying the calculations, the salary will be expressed as €/h. Also, since we're computing the cost, the 35% of the average salary will be added as Social Security contribution:

Therefore, we conclude that the personal economic cost of the project will be around 14.626€.

[1]The numbers will be obtained from www.glassdoor.es

Role	**Cost(€/h)**
Project mentor	27 [10]
Junior Project Manager	33 [11]
Junior Researcher	20 [12]
Junior Developer	20 [13]
Tester	26 [14]

Table 2: Cost estimated for each role.

Task	Project Mentor(h)	Junior Project Manager(h)	Junior Researcher(h)	Junior Developer(h)	Tester(h)
Weekly meetings	38	38	38	0	0
Project planning	0	65	0	0	0
Previous investigation	0	0	60	0	0
Practical implementation	0	0	100	200	0
Final experimentation, analysis and conclusions	0	2	5	0	10
Documentation	0	55	0	0	0

Table 3: Cost in hours estimated for each task.

Task	Cost(€)
Weekly meetings	3040
Project planning	2145
Previous investigation	1200
Practical implementation	6000
Final experimentation, analysis and conclusions	426
Documentation	1815

Table 4: Economic cost estimated for each task.

8.2 Material costs

All the software used in the development of the project is free.

8.2.1 Amortization of the resources

The hardware used is a computer with an estimated cost of 1.000€. The computer will be used in the *Project planning*, *Previous investigation*, *Practical implementation*, part of the *Final experimentation, analysis and conclusions* and *Documentation*, a total of 490 hours.

$Amortization = 1.000€ * \frac{1}{5\ years} * \frac{1}{12\ months} * \frac{1}{160\ hours\ of\ work} * 490\ hours\ worked = 51,05€$

8.2.2 Indirect costs

The indirect costs of the project also need to be taken into account:

- **Electricity**. 0.50€/kWh [15]. The total cost will be $0.50€/kWh * 490\ hours = 245€$
- **Travel cost**. The cost of the public transport for 9 months using a T-Jove [16] is $105,20€ * 3 = 315,60€$
- **Internet cost**. Assuming a cost of 40€/month, the total cost will be of $40€/month * 9 months = 360€$

8.2.3 Generic cost of the project

Taking into account all the costs listed above, the cost will be 441,25€.

Item	**Cost(€)**
Amortization	51,05
Electricity	245,00
Travel	105,20
Internet	40,00
Total	**441,25**

Table 5: Estimation of the indirect costs

Item	**Cost(€)**
Personnel	16088,60
Material - Electricity	214,86
Electricity	343
Total	**16647,48**

Table 6: Estimated total budget.

8.3 Deviations of the budget

Since some unexpected events can appear through the development of the project, the budget will be incremented by 10% to be able to deal with those circumstances. As for the light price, instead of a 10% extra, 40% will be applied, since recently its price it's been quite unstable.

8.4 Final budget

The total budget is 16647,48€.

9 Sustainability report

The sustainability report will be divided into three sections: environmental, economic and social sustainability.

Each section will contain three parts: **project put into production (PPP)**, **exploitation** of the project and the **risks** inherent to the project.

9.1 Environmental sustainability

9.1.1 PPP

Have you estimated the environmental impact of undertaking the project? Have you considered how to minimise the impact, for example by reusing resources?

Since a great part of the project will need the use of a computer, there's a minimal environmental impact inherent in the development of the project.

Especially the testing will require executing the code a larger amount of time. To reduce this, during the development, the code will be tested in a smaller set of problems than the final test.

Although the hardware used also consumes more resources than other products in the market, the cost of producing a new one instead of using the hardware already available would be more expensive both economically and environmental in a project of such short duration.

Since the development of this project doesn't require any special equipment, the amount of energy consumed can be seen in the previous section.

9.1.2 Exploitation

How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution environmentally improve existing solutions?

The success of this project would lead to a great reduction in energy consumption. Since the main goal of the study is to reduce the computational time and resources needed to solve SAT Problems, the environmental beneficial impacts are evident.

9.2 Economic

9.2.1 PPP

I Have you estimated the cost of undertaking the project (human and material resources)?

The cost of undertaking the project has been estimated and can be seen in section Budget.

9.2.2 Exploitation

How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution economically improve existing solutions?

The success of this project would lead to a great reduction in energy consumption, as well as being able to solve more problems. Since the main goal of the study is to reduce the computational time and resources needed to solve SAT Problems, the economic beneficial impacts are evident.

9.3 Social

9.3.1 PPP

What do you think undertaking the project has contributed to you personally?

First, this project will bring me the opportunity to develop a research project, something I've no prior experience with.

Also, I'll be able to learn more about the treatment of SAT Problems, an area that caught my attention but I'm very ignorant about.

How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution socially improve (quality of life) existing? Is there a real need for the project?

Currently, there's no clear solution to this problem. SAT is a widely known NP-Complete problem, and the preprocessing of this type of problem it's still under research.

Last but not least, a positive result in this thesis would lead to solving SAT Problems faster. Since many real-life problems can be represented as SAT Problems, it would have an impact on society.

10 SAT Problems

SAT problems are the set of problems consisting on deciding the satisfiability of formula expressed in Conjunctive Normal Form.

A formula is said to be in CNF when it's a conjunction of one or multiple clauses. Conjunctions are represented using the symbol $\vee$.

All the clauses in a CNF formula must be a disjunction of literals. Disjunctions are expressed using the symbol $\wedge$.

Literals are atomic formulas represented using lowercase letters and can be positive or negated. For example, the variable a can be the literal a or its negation $\neg a$.

All the following formulas belong to this group:

1. p. A formula consisting of a single clause p
2. $p \vee q$. A formula consisting of a single clauses: $p \vee q$
3. $p \wedge (q \vee \neg q) \wedge r \wedge s$. A formula consisting of two clauses: $p \wedge q$ and $\neg q \wedge r \wedge s$

All the following formulas are not CNF formulas:

1. $(p \vee q) \wedge r$.
2. $p \wedge (q \vee \neg q \wedge r) \wedge s$.

Deciding SAT means determining if, for a given CNF formula, there is an interpretation that satisfies it, that is, if is it possible to assign values to the variables such that the formula is true.

To this question there are two possible answers:

- **Satisfiable**: an interpretation such that the formula is satisfied does exist
- **Unsatisfiable**: there's no possible interpretation that satisfies the formula

This problem belongs to NP-Complete.

10.1 Proving our problem is NP

According to the computational complexity theory, Nondeterministic Polynomial-time (NP) problems are the set of decision problems in which the positive answer can be verified in polynomial time by a deterministic Turing machine.

In the case of the SAT problem, a *certificate function* running in polynomial time can be provided, that is, a function that is able to verify a solution for a given SAT problem.

As described before, the input for any SAT problem is a formula in Conjunctive Normal Form. Therefore, the *certificate function* provided has to check that at least one of the literals in each of the DNF clauses is evaluated to be true in the given solution. A straightforward algorithm can solve this problem in polynomial time as shown in *Algorithm 1* below.

Algorithm 1 Boolean satisfiability problem certificate

```
$F$ contains the SAT formula
$I$ contains the interpretation of $F$ to verify
for $c \in F$ do                          ▷ Iterate through each clause in F
    $Found \leftarrow False$
    for $l \in c$ do                      ▷ Iterate through each literal in c
        $Found \leftarrow Found \lor I[l]$
    end for
    if $Found == False$ then
        return $False$                    ▷ Any of the literals was interpreted as true
    end if
end for
return $True$        ▷ For the given interpretation, in all the clauses there's at
least one literal that evaluates true
```

The provided algorithm iterates once for every clause. For each of the clauses iterates once for every literal. The interpretation of the literal can be evaluated in constant time if the value is stored as an array. Therefore, the answer can be computed in linear time $O(N)$, where N is the size of the input, SAT problem is proved to belong to NP.

10.2 Proving our problem is NP-Complete

The next step will be to prove SAT problem belongs to the NP-Complete class.

In fact, the SAT problem is a very well-known problem in the *computational complexity theory* field, since it was the first known NP-Complete problem. It was proven to belong to this class by the computer scientist Stephen Cook in 1971 and Leonid Levin in 1973.

The theorem is known as the *Cook-Levin theorem*. In order to prove our problem is NP-Complete, it will be necessary to prove the problem belongs to both NP and NP-hard. It was already proven in the previous section *Proving our problem is NP* and that the problem belongs to NP. Therefore, in this section, the focus will be on proving that the problem belongs to NP-hard too.

10.2.1 NP-hard definition

An NP-hard problem can be defined as a problem *at least as hard as any NP-problem*[2], that is, if $\forall p \in NP, p \leq x$, then x is an NP-hard problem.

[2]https://mathworld.wolfram.com/NP-HardProblem.html

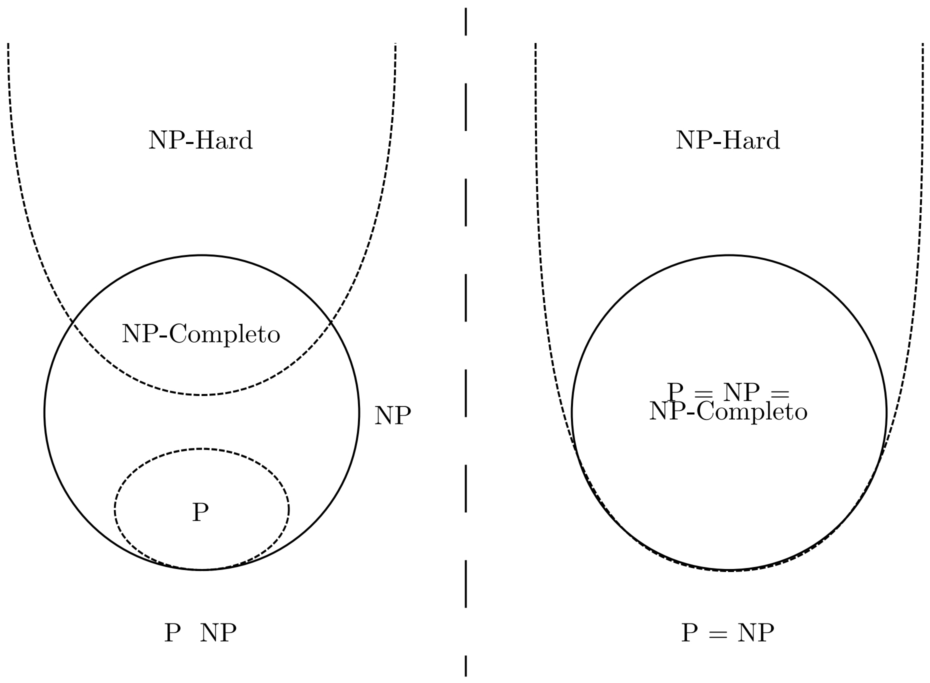


Figure 2: Euler diagram of P, NP, NP-complete and NP-hard families. Source
.

In this case, the goal will be to prove that $\forall p \in NP, p \leq SAT$. That is, reduce every problem to the SAT problem.

The idea behind the proof is, that for each problem in NP we can construct a non-deterministic touring machine that solves the problem in polynomial time. In order to visualize the idea better, all the different configurations of the previously mentioned touring machine can be encoded into a table. Each of the rows of the table will represent a configuration of the machine, while the columns will represent the different symbols of the tape. Each of the cells will describe the transition.

From the definition given above, it's known that the table will have a polynomial number of rows since it has a polynomial number of configurations and a polynomial number of columns since it's known that runs in polynomial time.

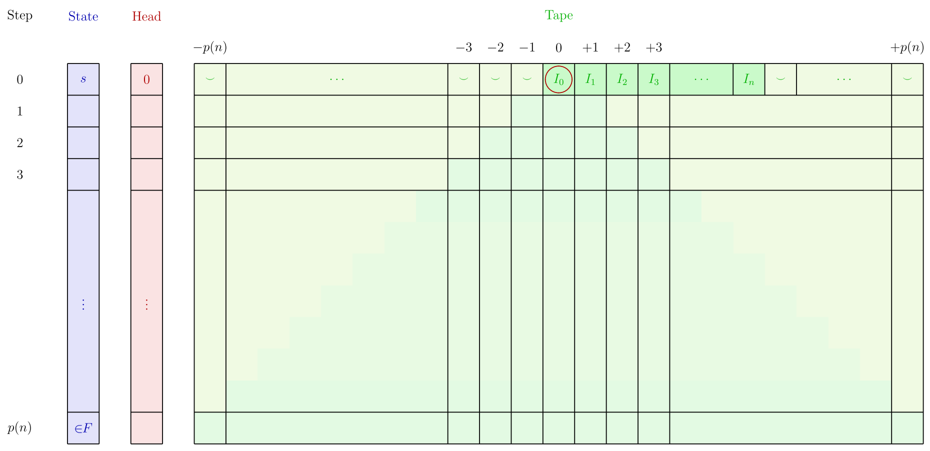


Figure 3: Table encoding the configurations of a touring machine. Source
.

In the table, each cell will correspond to a variable identified as $X_{i,j,s}$, where i will correspond to the row, j to the column and s to the value.

The goal will be to create a formula that satisfies the following restrictions:

1. **Each cell has exactly one value**.
2. **The first row is the start configuration**.
3. **At least one row is the end configuration**.
4. **Each row yields next**.

The final formula will be the conjunction of all four formulas.

10.2.2 Each cell has exactly one value

When working with SAT problems, in order to express extitexactly one restriction we need to divide the work in two simpler parts: *at least one* and *at most one*.

At least one: $\forall i \forall j \bigvee_{s \in Z} (X_{i,j,s})$

At most one: $\forall i \forall j \bigwedge_{s,t \in Z, s \neq t} (\neg X_{i,j,s} \vee \neg X_{i,j,t})$

$\varphi_1 = \forall i \forall j \bigvee_{s \in Z} (X_{i,j,s}) \wedge \bigwedge_{s,t \in Z, s \neq t} (\neg X_{i,j,s} \vee \neg X_{i,j,t})$

10.2.3 The first row is the start configuration

Being I the input, the initial configuration must contain all the input elements at the beginning of the tape.

$$\varphi_2 = \forall i \bigwedge_{j}^{|I|} X_{i,j,I_j}$$

10.2.4 At least one row is the end configuration

At least one of the states must be an accepting state.

$$\varphi_3 = \forall i \forall j \bigvee_{s \in Z} (X_{i,j,q_{accept}})$$

10.2.5 Each row yields next

First of all, what is a valid transition must be defined. In each step, the touring machine can move either to the right or to the left.

Move to the left: $X_{i,j-1,s} \wedge X_{i,j,t} \wedge X_{i,j+1,u} \wedge X_{i+1,j-1,t} \wedge X_{i+1,j,s}$

Move to the right: $X_{i,j-1,s} \wedge X_{i,j,t} \wedge X_{i,j+1,u} \wedge X_{i+1,j-1,s} \wedge X_{i+1,j+1,t}$

At least one movement: $X_{i,j-1,s} \wedge X_{i,j,t} \wedge X_{i,j+1,u} \wedge X_{i+1,j-1,t} \wedge X_{i+1,j,s} \vee X_{i,j-1,s} \wedge X_{i,j,t} \wedge X_{i,j+1,u} \wedge X_{i+1,j-1,s} \wedge X_{i+1,j+1,t}$

At most one movement: $\neg X_{i,j-1,s} \vee \neg X_{i,j,t} \vee \neg X_{i,j+1,u} \vee \neg X_{i+1,j-1,t} \vee \neg X_{i+1,j,s} \vee \neg X_{i,j-1,s} \vee \neg X_{i,j,t} \vee \neg X_{i,j+1,u} \vee \neg X_{i+1,j-1,s} \vee \neg X_{i+1,j+1,t}$

$ValidMovement_{i,j} = (X_{i,j-1,s} \wedge X_{i,j,t} \wedge X_{i,j+1,u} \wedge X_{i+1,j-1,t} \wedge X_{i+1,j,s} \vee X_{i,j-1,s} \wedge X_{i,j,t} \wedge X_{i,j+1,u} \wedge X_{i+1,j-1,s} \wedge X_{i+1,j+1,t}) \wedge (\neg X_{i,j-1,s} \vee \neg X_{i,j,t} \vee \neg X_{i,j+1,u} \vee \neg X_{i+1,j-1,t} \vee \neg X_{i+1,j,s} \vee \neg X_{i,j-1,s} \vee \neg X_{i,j,t} \vee \neg X_{i,j+1,u} \vee \neg X_{i+1,j-1,s} \vee \neg X_{i+1,j+1,t})$

Once defined what a valid transition is, there can only be one valid transition in any row.

At least one transition: $\forall i \bigvee_{j \in Z} ValidMovement_{i,j}$

At most one transition: $\forall i \bigwedge_{j \in Z, k \in Z, j \neq k} (\neg ValidMovement_{i,j} \vee \neg ValidMovement_{i,k})$

$$\varphi_4 = \forall i (\bigvee_{j \in Z} ValidMovement_{i,j}) \wedge (\bigwedge_{j \in Z, k \in Z, j \neq k} (\neg ValidMovement_{i,j} \vee \neg ValidMovement_{i,k}))$$

The final formula is $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$. Since it is possible to transform the input of any NP problem into a SAT problem input, it's proven that the

SAT problem belongs to NP-Hard.

As stated before, if the problem belongs both to NP and NP-Hard, it means the problem belongs to the NP-complete family of problems.

10.3 SAT Solvers

If the problem dealing with belongs to NP-hard, how can a problem deal with it?

Different SAT solvers follow different strategies in order to solve this kind of problems. In this section, there will be provided with a brief introduction about how SAT Solvers work, first describing a simple and straightforward approach, and later enumerating and explaining different techniques that can help the program reduce the computational cost of finding the solution.

10.3.1 Naive algorithm

In this section, the naive recursive algorithm to solve SAT problems will be explained. Although it may seem that it's too inefficient, most SAT solvers use this algorithm as a base and, then, implement some other techniques over this base.

This approach consists in testing all the possible assignments of true or false to the different variables. If one of them is a valid solution (can be done in polynomial time - in fact, linear time - as shown before) then the algorithm can stop. Otherwise, the algorithm will keep trying combinations. Once all the combinations are exhausted and there was no valid assignment found, the algorithm can conclude that the given input is unsatisfiable, that is, there's no possible assignment of variables that satisfies the given formula.

In the algorithm will be used the function described in the section *Proving our problem is NP* to check if a solution was found.

Algorithm 2 Naive solution for the SAT problem

```
$P$ contains the SAT problem in CNF
$V$ contains the variables, initially contains all the variables
if $V = \emptyset$ then
    return $isValidSolution(P)$
end if
$v = V.pop()$                                    ▷ extract a variable from V
$setTrue(v)$
if $naiveSolveSAT(P, V)$ then
    return $True$
end if
$setFalse(v)$
return $naiveSolveSAT(P, V)$
```

If the initial calculation of the variables is taken into account, it can be computed in time $O(N)$, being N the number of literals in the input formula and storing them in a hash set. This is computed in the beginning and then it's passed as an argument to the function.

Except for the recursive calls, all the operations in the functions have a constant computational cost. As for the recursive calls, each time the function does at most 2 calls. In fact, in the worst case, that is, when all of the calls return false, all the executions of the function will make 2 calls. Therefore the cost is $O(2^N)$.

Since the first part, before calling the function, has a cost of $O(N)$ and the function itself has a cost $O(2^N)$, the total cost is $O(2^N)$.

10.3.2 Improvements: Propagate conflict

The cost of this solution is too high. Therefore, some strategies are often applied in SAT solvers in order to minimize the cost. In this section, we'll cover some of the most basic ones, since they'll serve as a base to understand better the problem and will serve in future sections of this project.

Propagate conflicts is one of the most basic, yet most effective techniques used in this kind of SAT solver.

It consists in, as the name says, propagating the conflicts after deciding the value of a variable. In order to understand it better, the concept of conflict propagation is shown in the next example:

$(\neg p \vee q) \wedge (\neg q \vee \neg r) \wedge (r \vee s)$

In the formula shown above, if the variable p is set to $True$, then q must be set to $True$, otherwise, the clause can not be satisfied.

Following the same logic, if q is set to $True$, then $\neg q \vee \neg r$ can only be satisfied if r is set to $False$.

Finally, if r is set to $False$, s must be $True$ in order to satisfy $r \vee s$.

Therefore, a satisfiable assignment would be $p = 1, q = 1, r = 0, s = 1$. It's worth noticing that we arrived at this assignment just propagating the conflict from the first assignment. When working with real-life problems, this situation occurs very often. Although it's not common to be able to satisfy all the CNF formulas just through one decision as in the short example shown, we can avoid making decisions that will provoke the final assignment to be unsatisfiable.

Adding the modification to the previous algorithm would result in the following algorithm:

Algorithm 3 Propagate conflict solution for the SAT problem

```
$P$ contains the SAT problem in CNF
$V$ contains the variables, initially contains all the variables
if $V = \emptyset$ then
    return $isValidSolution(P)$
end if
$v = V.pop()$                                    ▷ extract a variable from V
$setTrue(v)$
$propagateConflict(P, v)$
if $naiveSolveSAT(P, V)$ then
    return $True$
end if
$setFalse(v)$
$propagateConflict(P, \neg v)$
return $naiveSolveSAT(P, V)$
```

The cost of the algorithm was previously explained in the previous section, but this time there's a new function $propagateConflict$.

Algorithm 4 Propagate conflict

```
$P$ contains the SAT problem in CNF
$v$ contains the literal to propagate
$S$ stack
$S.push(v)$
while $notS.empty()$ do
    $l = S.pop()$
    for $c$ in $P$ do
        if $not\ exists\ l \in C, l == True$ then
            if $all\ l \in C, l == False$ then
                return $Conflict$
            end if
            if $countUndef(c) == 1$ then
                $setTrue(undefined\_literal)$
                $S.push(undefined\_literal)$
            end if
        end if
    end for
end while
```

The algorithm above iterates for every variable in the stack. At most, there will be M variables. For each one, iterates through all the N literals in the formula. Therefore the cost of the function is $O(N * M)$.

If the costs of the two functions are combined, the cost of the total function is the same as before, that is, $O(2^N)$.

10.3.3 Improvements: Occurrence lists

This is not the only technique that can be applied to improve performance. In this section, another technique will be explained: the usage of Occurrence lists.

Following the same strategy as before, conflict propagation, there's a better way to look for where the conflicts may occur. Using the same example as before, we pay attention to which clauses the conflict appears: the clauses in which the exists the negation of the literal.

The first part of the modification can be done when reading the input. It just consists in creating a map, where the key will be the literal (which can be positive or its negation) and the value a list of occurrences.

Once all the occurrences are stored, the rest of the algorithm can continue as before. The only modification will be in the *Propagate conflict* function: instead of iterating through every clause in the formula, it will just iterate through the clauses where the negation of the literal treating appears. That is, if we're treating the literal l, the algorithm will loop over the occurrences of $\neg l$. If the literal is negated, $\not l$, the algorithm just needs to loop through the occurrences of l.

Algorithm 5 Propagate conflict using Occurrences List

```
$P$ contains the SAT problem in CNF
$v$ contains the literal to propagate
$S$ stack
$S.push(v)$
while $notS.empty()$ do
    $l = S.pop()$
    for $c$ in $occurrences(\neg l)$ do
        if $not\ exists\ l \in C, l == True$ then
            if $all\ l \in C, l == False$ then
                return $Conflict$
            end if
            if $countUndef(c) == 1$ then
                $setTrue(undefined\_literal)$
                $S.push(undefined\_literal)$
            end if
        end if
    end for
end while
```

This simple modification, although doesn't change the complexity of the algorithm, in practice will make it faster, since it's avoiding many unnecessary operations.

These concepts explained above are enough to understand the future techniques that will be explained in future sections.

11 Linear Programming

In this section, the concept of Linear Programming will be explained. Although the explanations about it will not be exhaustive, it's worth having a base, since this project takes advantage of this kind of programming.

Linear Programming is a programming technique used to *optimize a linear objective function, subject to linear inequality and linear inequality constraints*[3].

Usually, the input of a Linear Programming problem has the following parts:

1. A linear function to be maximized or minimized
2. Problem constraints
3. Variables

[3]https://en.wikipedia.org/wiki/Linear$_programming$

12 Integer Programming

12.1 From SAT to IP

The goal of this project is to transform a SAT input into an IP problem after applying some preprocessing techniques. Although the preprocessing techniques will be explained later, this section will introduce the concept of the transformation from SAT to IP.

Let's start with a simple example in order to understand it better. The clause $p \vee q$ expresses that at least one of the two literals p and q must be positive. That is:

$p \vee q$ is equivalent to $p + q >= 1$.

Now let's see what happens with a longer clause. For example, the clause $p \vee q \vee r \vee s$. This clause implies that at least one of the literals must be true, therefore as before:

$p \vee q \vee r \vee s$ is equivalent to $p + q + r + s >= 1$.

It's easy to see that, when all the literals in a clause are positive, the transformation is quite straightforward. When dealing with clauses with false literal the transformation requires a bit more effort.

Let's take, for example, the clause $p \vee \neg q$. We can rewrite as before:

$p \vee \neg q$ is equivalent to $p + \neg q >= 1$.

Although $p + \neg q >= 1$ it's understandable, it's not in a format that can be inputted to a IP solver. Therefore, the negation has to be transformed into a positive literal, while maintaining the meaning. It can be done by changing all the negated literals for $1-$ the literal.

$p + \neg q >= 1$ is equivalent to $p + (1 - q) >= 1$.

If we isolate the literals on the left side of the inequation:
$p + (1 - q) >= 1$ is equivalent to $p - q >= 0$.

Finally, the whole formula, that is, the whole set of clauses can be expressed as a set of restrictions following the steps shown above. To illustrate it with an example, let's take the CNF formula $p \wedge (\neg p \vee q) \wedge (\neg p \vee \neg r)$.

$p \wedge (\neg p \vee q) \wedge (\neg p \vee \neg r)$ can be expressed as the set of restrictions:

$p >= 1$

$-p + q >= 0$
$-p - r >= -1$

12.2 Taking advantage of IP

All the explained above is just a plain translation from one kind of problem to another. In that case, only if the IP is able to solve the problem faster, then the mentioned transformation will be worthwhile. SAT solvers, especially the most modern ones, are already very good at solving problems, using techniques more advanced and complex than the ones explained. Therefore, in order to obtain better results, the transformation has to express the information in a more powerful way for the LI or LP solver.

There are cases where various SAT clauses can be expressed in a smaller number of restrictions in LP. By doing so, the LP solver may have a chance to perform faster than the SAT solver. This will be the main goal of the following sections of the project.

The reason will be detailed in the next section, but to get an idea of what it looks like, below there will be an example:

The formula $(p \vee q) \wedge (p \vee r) \wedge (q \vee r)$ can be rewriten as three constraints, like in previous sections, but it can also be expressed as one single constraint $p + q + r >= 2$.

This way, the LP solver will just have to deal with a single constraint and will be able to perform faster than when having multiple.

13 First approach

The first approach will be, as explained in the previous section, to try to transform the clauses of the SAT problem into LP clauses. After explaining the concept of subsumption and how the algorithm will use it, the algorithm designed by Robert Nieuwenhuis will be presented.

The algorithm will mainly work around cardinality constraints of the form $l_1 + l_2 + ... + l_n \geq k$, where l_i are the literals and k is a natural number.

Only the constraints of form ... $\geq$ ... will be considered, since we have that $l_1 + ... + l_n \leq k \equiv$-l$_1$ $+ ... + -l_n \geq n - k$.

The focus of the problem of this project will be on answering the following question: *Given a set of constraints, how to detect all the cardinality constraints that are logical consequences of it?*

13.1 Definition: q = n - k

In this section there are three variables that will be used quite often, those are q, n and k. The value of the three is related.

It's easy to understand the values of n and k if they're presented in a LP constraint like the following:

$a_0 + a_1 + ... + a_n \geq k$

The value of n stands for the number of variables on the left side of the inequation, while the k represents the constant on the right side of the inequation.

The remaining variable q is the result of $n-k$, $q = n-k$. q can be understood as the *degrees of freedom* in the constraint, that is, the number of variables in the constraint that can be false with the constraint still remaining satisfiable.

The smaller the value of q, the more restrictive the constraint is.

13.2 Definition: subsumption

Once well defined the concepts presented above, it's time to introduce what will be a very key concept in this project: the concept of subsumption.

$L \geq k \models L' \geq k'$ iff $|\frac{L}{L'}| <= k - k'$

In order to understand better the statement, the following example is provided:

$x + y + z + u + v \geq 3 \models x + y + z \geq 1$

In the example above, the $|\frac{L}{L'}| = |\frac{x+y+z+u+v}{x+y+z}| = 2$. Then, it's easy that $2 <= k - k'$, since $k - k' = 3 - 1 = 2$. We have that $2 <= 2$, which is true, therefore the second constraint is subsumed by the first one.

This concept may not be intuitive at first, but it's worth taking some time to understand it since it will be used to determine the relationship between constraints in future algorithms.

13.2.1 Prove

In order to use the statement above, we first need to prove that such a statement is true.

Lemma:
$L \geq k \models L' \geq k'$ iff $|\frac{L}{L'}| <= k - k'$.

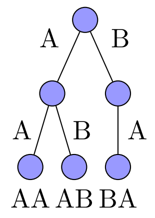
Prove:
$L_1 + L_2 \geq k \models$
$L_1 \geq k - |L2| \models$
$L_1 + L_3 \geq k - |L2|$

13.3 Definition: trie

The concept presented above requires a good data structure that allows the algorithm to store the required data, as well as perform the necessary comparisons as fast as possible. For that reason, the *trie* data structure will be used.

The *trie* is simply a prefix tree, that is, all the children under a node have a common prefix. For that reason, this data structure is often used when working with strings.

For example, a trie encoding the words $\{AA, AB, BA\}$ would be like the following:



When working with a trie, we can perform some operations at a very low cost, for example, those related to the search. The cost of checking if a word is in the *Trie* is, in the worst case, the height of the *Trie*.

The insert operation is not expensive either. Since we need to have a node for each character in a word, it's doable in $O(N)$ time, where N is the length of the word.

In the case of this project, instead of strings, the constraints will be stored in it.

How can we store the constraints in the trie?

In order for the algorithm to be consistent, the data structure used must be well defined. First, all the values in the constraint will be ordered in ascended order. This way it can be assured that constraints having the same *prefix* will be under the same node.

If we use an example similar to the one before using words, but using constraints instead, we obtain a trie like the one below. In this case, the constraints to store in the trie will be:

$a_1 + a_2 - a_3 >= 1$

$-a_1 + a_2 >= 1$

$a_3 + a_4 >= 1$

$-a_1 + a_3 + a_4 >= 2$

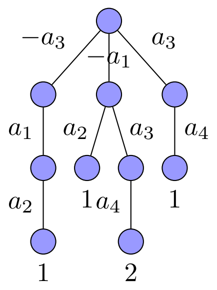
First of all, the constraints must be all in order as described. Since any variable is more important than another it doesn't matter which goes first as long as it's consistent. In the case of this project, in order to determine the *value* (only important for the order of the variable in the constraint) of a variable, it will k for a_k for positive variables and $-k$ for a_k when the variable has a negative sign:

$-a_3 + a_1 + a_2 >= 1$

$-a_1 + a_2 >= 1$

$a_3 + a_4 >= 1$

$-a_1 + a_3 + a_4 >= 2$



In case multiple constraints *collide*, that is, they're the same but the value of k, then for the sake of the goal of the algorithms that will be explained in future sections, the bigger value of k will be stored in the $Trie$, since it's more restrictive.

13.4 Algorithm

In this section, first, the goal of the algorithm will be presented and then the details of the algorithm.

13.4.1 Introduction

Given a cardinality constraint $C : l_1 + ... + l_n \geq k$, we have, from stronger to weaker:

- if $k > n$, then C is unsatisfiable
- if $k = n$, then **all** l_i must be true (C is equivalent to $l_1 \wedge ... \wedge l_n$)
- if $k = n-1$, then at most one l_i can be false (C is equivalent to $\bigwedge_{1\leq i<j\leq n} (l_i \vee l_j)$)
- if $k = n-2$, then at most one l_i can be false (C is equivalent to $\bigwedge_{1\leq i<j<r\leq n} (l_i \vee l_j \vee l_r)$)
- ...

From the succession presented above, we make the following observation.

Lemma: *we have* $l_1 + ... + l_n \geq k$ iff $S \geq k - 1$ for all subsets S of $l_1, ..., l_n$ with $|S| = n - 1$.

In the presented lema, when it's said *we have* means that exists the constraint literally or something stronger, that is, a constraint that subsumes the constraint (defined in the section *Definition: subsumption*).

By induction, this goes down to all clauses $S \geq 1$, where $|S| = n - k + 1$.

It's worth noticing that the shorter the clauses we start with, the stronger the cardinality constraints obtained will be.

Starting with clauses of size 2 will produce cardinality constraints $l_1 + ... + l_n \geq k$, where $k = n - 1$.

Starting with clauses of size 3 will produce cardinality constraints $l_1 + ... + l_n \geq k$, where $k = n - 2$.

Starting with clauses of size 3 will produce cardinality constraints $l_1 + ... + l_n \geq k$, where $k = n - 3$.

13.4.2 Pseudocode

The base algorithm is the following:

Algorithm 6 Detect constraints by Robert Nieuwenhuis

```
$Clauses$ contains the clauses
$Constraints = \emptyset$
for $q = 1$ to $maxQ$ do
    for $clause$ in $Clauses$ where $1 = n - q$ do
        $constraint = toConstraint(clause, 1)$    ▷ transform clause $l_1 \vee ... \vee l_n$
into $l_1 + ... + l_n \geq 1$
        $Constraints.add(constraint)$
    end for
    for $k = 1$ to $maxK$ do
        for $constraint$ in $Constraints$ where $k = n - q$ do
            for $l'$ such that "we have" $S + l' \geq k$ for all subsets $S$ of $\{l_1, ..., l_n\}$
with $|S| = n - 1$ do
                $Constraints.add(l_1 + ... + l_n + l' >= k + 1)$
            end for
        end for
    end for
end for
```

The algorithm looks quite simple and straightforward, following the rules stated in the previous sections. That's because there's one part of the algorithm missing, that is the *we have* check condition.

In order to create that function, the *Trie* structure explained above will be used to store the constraints.

For this algorithm is assumed that the *Trie* structure is received as a parameter, as well as the constraint and the literal trying to extend with.

Algorithm 7 We have

```
$trie$
$constraint$ constraint trying to extend
$l$ literal trying to extend with
$newConstraint = \emptyset$
for $v$ in $constraint$ do
    $newConstraint = substitute(constraint, v, l)$
    if $have(trie, newConstraint) == False$ then
        return $False$
    end if
end for
return $True$
```

The recursive function *have* is the following:

Algorithm 8 Trie - we have

```
$trieNode$ initially the root of the $Trie$
$constraint$ new constraint
$e$ margin of "error", in the first call $e =$
if $trieNode.hasConstraint$ and $isConsequence(constraint, trieNode.constraint)$
then
    return $True$
end if
if $trieNode.next == null$ then
    return $False$
end if
if $trieNode.maxK < e$ then
    return $False$
end if
$f = False$
for $c$ in $trieNode.next$ do
    if $constraint.contains(c)$ then
        $f = have(trie.next[c], constraint, e)$
    end if
    if $f == False$ then
        $f = have(trie.next[c], constraint, e + 1)$
    end if
end for
return $f$
```

If the *Trie* structure, the algorithm would have to iterate through all the constraints in order to make the checks, also there will be no option to discard branches when a constraint for the necessary k doesn't exist.

This function time's execution is too large. When trying to execute the test files, the execution didn't finish in a reasonable time. Therefore we must rethink the strategy to follow.

14 Reducing the scope

The first change will be to reduce the scope. In the previous version of the algorithm, all the values of q were considered.

If we think about what each value of q means (see Introduction), it's easy to realize that the bigger the value of q, the smaller the probability of it existing is since more clauses are required to satisfy the condition.

For example, in order to have: $l_1 + ... + l_n \geq n - 3$, that is, $q = 3$, all the combinations in $\{l_1, ..., l_n\}$ of groups of 4 elements.

Therefore in this approach, the algorithm will just focus on the $q = 1$ and $q = 2$ cases. Since we're reducing the problem, maybe there's a way to adapt the algorithm to work with a more specific problem. In addition, the nature of the problem now is much smaller, since for each q the algorithm was iterating through every variable and every constraint.

14.1 Q1 and Q2 clauses

The algorithm now will get rid of q in its complexity, since only the cases of $q = 1$ and $q = 2$ will be taken into account.

The algorithm will remain quite similar to the used before:

Algorithm 9 Detect constraints q=1 and q=2

```
$Clauses$ contains the clauses
$Constraints = \emptyset$
for $q = 1$ to 2 do
    for $clause$ in $Clauses$ where $1 = n - q$ do
        $constraint = toConstraint(clause, 1)$    ▷ transform clause $l_1 \vee ... \vee l_n$ into $l_1 + ... + l_n \geq 1$
        $Constraints.add(constraint)$
    end for
    for $k = 1$ to $maxK$ do
        for $constraint$ in $Constraints$ where $k = n - q$ do
            for $l'$ such that "we have" $S + l' \geq k$ for all subsets $S$ of $\{l_1, ..., l_n\}$ with $|S| = n - 1$ do
                $Constraints.add(l_1 + ... + l_n + l' >= k + 1)$
            end for
        end for
    end for
end for
```

As said before, with this version of the algorithm the q is not a problem anymore, but it still being not fast enough. The program is not fast enough to process the input tests in a reasonable time.

Therefore, for the next version of the algorithm, the strategy to follow will be to further reduce the scope of the problem, trying first to work only with the $q = 1$ case, and only once achieving success in the most simple case, try to expand it to $q = 2$.

14.2 Focus on Q1

Since this time the algorithm just needs to focus on the $q = 1$ case, this time the strategy will be quite different.

According to the previous definitions (see Introduction), in order to have $q = 1$, that is, a constraint $l_1 + ... + l_n \geq k$, where $k = n - 1$, we need to have all the binary combinations of the set $\{l_1, ..., l_n\}$ as clauses.

The difference now is that *we have* this time is literal. Without taking the unary clauses into account, it's for sure that there will not be stronger constraints in the set of constraints created from reading the input clauses than the binary clauses.

Therefore, when checking if a constraint exists, the operation will be more straightforward, since it's known beforehand the exact constraint we're looking for.

Also, the algorithm will try to be more selective with the literals trying to extend the constraints with. It will do so using the *Trie* structure once again.

This time the main algorithm will remain simple, and the complexity will rely on the auxiliary function *getCandidateVariables*.

Algorithm 10 Detect constraints q=1

```
$Constraints$ contains the initial binary clauses as constraints
$constraints = Constraints$
$k = 1$
$result = \emptyset$
while $constraints.empty() == False$ do
    $trie = build(constraints)$
    $nextConstraints = \emptyset$
    for $constraint$ in $constraints$ do
        $candidateVariables = getCandidateVariables(trie, vars(constraint), 0, k)$
        for $variable$ in $candidateVariables$ do
            $newConstraint = constraint$
            $newConstraint.add(variable)$
            $newConstraint.k+ = 1$
            $nextConstraints.push(newCOnstraint)$
        end for
    end for
    $result.append(constraints)$
    $constraints = nextConstraints$
    k += 1
end while
```

The next algorithm to describe will be *getCandidateVariables*.

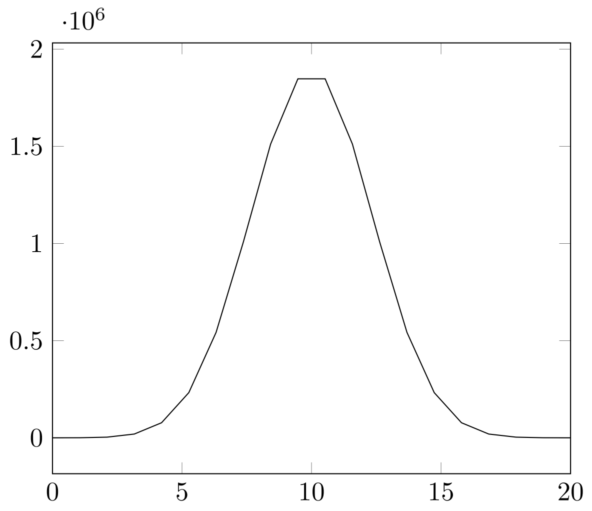
Algorithm 11 Get candidate variables

```
$trie$ contains the contrains in a $Trie$ structure
$variables$ contains the variables of the $constraint$
$idx$ index of the variable checking
$left$ variables left to check
$result$
if $left == 0$ then
    for $node$ in $trie.nodes$ do
        $result.push(node.variable)$
    end for
    if $trie.nodes.contains(vars[idx]) == False$ then
        return $result$
    end if
    $nextCandidates = getCandidateVariables(trie.nodes[idx], vars, idx + 1, left - 1)$
    if $idx + left \geq vars.size()$ then
        return $nextCandidates$
    end if$siblingCandidates = getCandidateVariables(trie, vars, idx + 1, left)$
    return $intersection(nextCandidates, siblingCandidates)$
end if
```

This algorithm is much simpler than the previous versions, but it's still not able to preprocess the input in a reasonable time. After further analyzing and understanding what the algorithm is doing, there's a clear reason for the execution being too slow.

Let's imagine the result is a constraint of length n. Because of the algorithm we're using, we know that the constraint is made up of constraints of length $n-1$. Those constraints are all made up of constraints of length $n-2$, and so forth until arrived at the constraints of length 2.

What that means is the algorithm is generating all the combinations of the variables $l_1, ..., l_n$ of sizes from 2 to n.



Number of combinations of size x of 20 elements

As the plot shows, the number grows very fast. Therefore, we need another strategy, an algorithm that is able to find the solution without needing to generate all the middle steps.

15 Find all maximal cliques

When studying different strategies to solve the problem, representing the problem as a graph was one of them.

Each variable is represented as two vertexes in the graph: one for the positive value and another one for the negative. In the graph, there will be an edge between a vertex l_i and l_j if the clause $l_i \vee l_j$ exists.

This way, the goal of the algorithm will be to find the groups of vertex where all the vertex are connected to the rest since that will mean that for every $edge(l_i, l_j)$ in the graph there is a clause $l_i \vee l_j$. We want each group of vertexes to be as big as possible. This is, in fact, solving the problem of Finding all maximal cliques, a well-known NP-Complete problem.

15.1 Proving our problem is NP

To prove the problem of the Maximal Cliques is NP, as before, we prove that there's an algorithm that can verify the result in a polynomial time.

Algorithm 12 Max Clique certificate

```
$G$ is the graph
$clique$ is the set of vertex
for $v$ in $G$ do
    if $v \notin clique$ then
        $found = False$
        for $u$ in $clique$ do
            if $edge(u, v) == \emptyset$ then
                $found = True$
            end if
        end for
        if $found == False$ then
            return $False$
        end if
    end if
end for
return $True$
```

The algorithm iterates for each vertex v in the graph G, and then checks for each vertex if is connected to the clique. The check can be done in constant time. The cost is $O(N^2)$, where N is the number of vertex in the graph G. Therefore there exists a certificate that runs in polynomial time.

15.2 Proving our problem is NP-Complete

In order to prove the problem is NP-Complete, it needs also to be proven that the problem belongs to NP-Hard. It can be done by reducing a known NP-hard problem to the one trying to prove since if the problem we're trying to prove can be solved in polynomial time, it means that the NP-hard problem can be solved in polynomial time too.

In the case of the problem, it's already described above how the reduction works in the previous section, Find all maximal cliques. The algorithm is the following:

Algorithm 13 Max Clique certificate

```
$G$ is the graph
$binaryClauses$
**for** $clause$ in $binaryClauses$ **do**
    $createEdge(clause[0], clause[1])$
**end for**
```

The algorithm iterates through each clause in the input and then creates an edge. Assuming the edge can be created in constant time, the algorithm will have a linear cost of $O(N)$.

15.3 Using the find-all-maximal-cliques algorithm to solve our problem

The first approach will be to use a recursive algorithm that solves the problem without the need of creating all the combinations as before.

Algorithm 14 Enumerate Max Cliques by Robert Nieuwenhuis

```
$V$ contains a list of vertices. $|V| \geq 1$
if $V = \{x\}$ then
    return $\{\{x\}\}$
end if
$x = V.pop()$                                    ▷ Extract the element from $V$
$L = \{\}$
$A = adj(x)$
for $k$ in $enumerateMaxCliques(V)$ do
    if $A \supseteq k$ then
        $add$ $\{x\} \cup k$ to $L$
    end if
    if $A \supseteq k == False$ then
        $add$ $k$ to $L$
        if $\{x\} \cup (A \cap k)$ not subsumed then
            $add$ $\{x\} \cup (A \cap k)$ to $L$
        end if
    end if
end for
return $L$
```

For the *not subsumed* check, the *Trie* structure can be used to make the search faster, as shown in previous algorithms.

The algorithm described belongs to a class of algorithms called *Output-sensitive* algorithms. That is, the running time of the algorithm is determined by the size of the output.

In the case of our algorithm, it means the more cliques and the bigger they are, the more it takes for the algorithm to finish its execution.

After testing with this algorithm and other implementations[4] [5], the algorithm used will be the *Tomita* algorithm. The implementation by Darren Strash can be found here.

After finding an implementation of the algorithm fast enough, another strategy will be used in order to profit from it.

[4]https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch$_{a}lgorithmWithout_{p}ivoting$
[5]https://github.com/darrenstrash/quick-cliques

16 Definition: Probing

When working with SAT problems, sometimes the relationship between two variables is explicitly expressed as a clause, $l_i \vee l_j$, but other times it may not be explicit.

For example, if there's a formula $(p \vee q) \wedge (p \vee r) \wedge (q \vee r) \wedge (\neg r \vee s) \wedge (\neg r \vee t) \wedge (s \vee t)$. In this case, there's also a not-explicit relation between p and s, p and t, q and s, and q and t.

Detecting those relationships will allow the algorithm to detect longer constraints if they exist. *Probing* is a technique used to find those *hidden clauses*.

The algorithm used is very similar to the *conflict propagation* explained in a previous section. In fact, the same $propagateConflict$ function can be used. In the main function, each time we decide on a variable and propagate the conflicts, the algorithm will create a clause for each variable propagated and the decided variable.

Algorithm 15 Propagate conflict solution for the SAT problem

```
$P$ contains the SAT problem in CNF
$V$ contains the variables
for $v$ in $V$ do
    $setTrue(v)$
    $propagateConflict(P, v)$
    for $pv$ in $propagatedVariables$ do
        $createClause(v \vee pv)$
    end for
    $clearVariables()$
    $setFalse(v)$
    $propagateConflict(P, v)$
    for $pv$ in $propagatedVariables$ do
        $createClause(v \vee pv)$
    end for
    $clearVariables()$
end for
```

Since the algorithm is just iterating through all the variables, setting a value and then propagating the conflicts, the algorithm should be fast enough.

After running the algorithm, we've explicitly set the clauses, but there's a new problem: the size of the input is much bigger now, so the execution time will be longer. On the other hand, the results obtained will be more powerful, since the output will need fewer constraints to express the same information.

17 Definition: Pigeon Hole problems

In this section, the concept of *Pigeon Hole problems* will be described, since they're a quite common kind of problem when working with SAT problems, and it's extremely hard for the SAT-solvers to deal with them, even for the most advanced ones.

The problem is the following: there's a total of n pigeons and m holes, having $n > m$. Each pigeon must be in a hole, and in each hole can be at most one pigeon.

With the example, it's easy to see where's the problem. But when working with real problems, where there are thousands of clauses and literals, it may not be that intuitive. As shown in the previous section, the relationship between the variables may not be explicit in the problem, which makes it even harder to detect. This kind of problem appears especially when treating scheduling problems.

As stated before, it's very hard for the solvers to solve this kind of problem. The graph below shows the execution time of the SAT Solver *PicoSAT*[6] needed for *Pigeon Hole Problems* of different sizes. The number of seconds starts to grow fast for problems of relatively small size.

[6] http://fmv.jku.at/picosat/

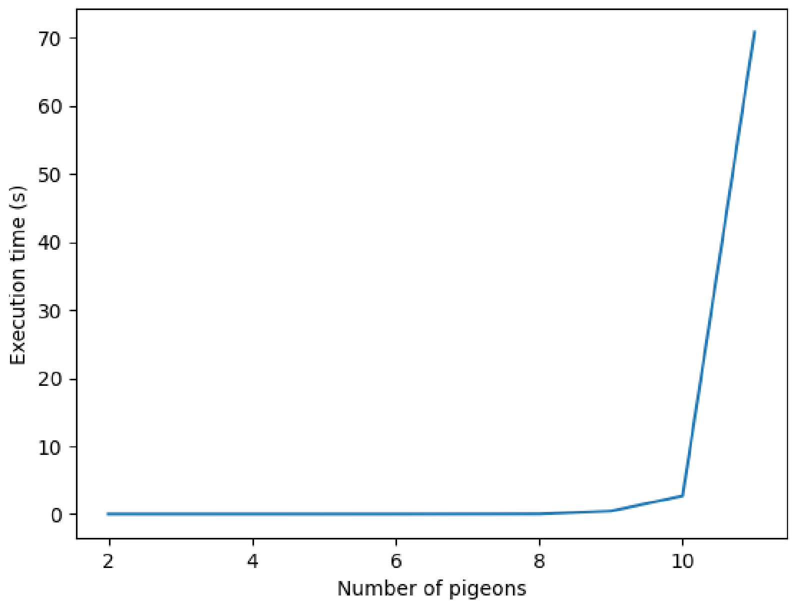


Figure 4: Execution time for PicoSAT to solve Pigeon Hole problems of different sizes

Since SAT Solvers can not deal easily with this kind of problem, maybe it's easier for an LP program to solve it using the preprocessing explained.

18 Tests

At this point, the preprocessing will be tested using different problems as input, to see if there's some improvement or not. First, the input will receive a *Probing* preprocessing, in order to extract the not-explicit relationships between the literals. Then the clauses will be converted into a graph for the *Max Cliques solver*. The result of the solver will be then transformed into *constraints* in order to feed the *LP solver* with them.

Therefore the total preprocessing time will consist of:

$T_{preprocessing} = T_{probing} + T_{max\_cliques} + T_{constraints}$

The result will be successful for the cases in which:

$T_{preprocessing} + T_{LP\_solver} > T_{PicoSAT}$

In concrete, the LP Solver used will be *pbsat*.

18.1 Pigeon Hole problems

Testing with the pigeon problems the following results are obtained:

Number of Pigeons	Execution time PicoSAT (s)	Execution time Our Method (s)
2	0.020259	0.004850
3	0.016199	0.006847
4	0.018553	0.004940
5	0.021218	0.005306
6	0.017857	0.005435
7	0.020573	0.005331
8	0.049005	0.005671
9	0.460942	0.006199
10	2.673638	0.007319
20	-	0.007568
30	-	0.007867
40	-	0.007354
50	-	0.007863
100	-	0.008821

Table 7: Execution time for Pigeon Hole Problems

As the table shows, when working using the preprocessing and the LP solver Pigeon Hole problems of bigger sizes can be treated too without much cost.

This is a good sign, but those problems are just *pure* Pigeon Hole problems, therefore, the number of variables and clauses is considerably smaller than the ones in real problems.

18.2 Real problems

After testing with some experimental inputs, we tested the algorithm with some real-life problems.

Problem	Execution time PicoSAT (s)	Execution time Our Method (s)	**Satisfiability**
aes_24_4_keyfind_2-sc2013.cnf	-	-	Satisfiable
Bebel-toughsat_24bits_1.cnf	-	-	Satisfiable
Biere-eqbpwtrc10bpdtlf10.cnf	-	-	Satisfiable
hid-uns-enc-6-1-0-0-0-0-3251.cnf	-	-	Satisfiable
Savicky-size_4_4_4_i0566_r8.cnf	-	-	Satisfiable
horaris1.cnf	-	8.024	Unsatisfiable
horaris2.cnf	0.055	24.511	Satisfiable
horaris3.cnf	-	176.669	Unsatisfiable

Table 8: Execution time for real SAT Problems

None of the algorithms was able to solve most of the problems in the given time (at most 5 minutes). This is expected from the PicoSAT solver since it's not a commercial solver and the techniques it uses are already outdated.

As for our algorithm, it was only able to solve three of the problems, two of them which were unsatisfiable. That may be due to those problems containing Pigeon Hole problems in them.

19 Conclusions

Observing the results obtained, the algorithm performs better than the PicoSAT for some problems. Since they're *unsatisfiable* scheduling problems, it's very likeable they contain pigeonhole problems in them.

Also, the algorithm wasn't able to decide on the problems of big size, just the smallest ones. In order to decide whether it's better than PicoSAT or not, the experiments would need to last longer. As for the commercial SAT Solvers, the preprocessing implemented can't compete with them.

The preprocessing algorithm is fast enough to preprocess most of the problems, even the bigger ones, in a considerable time. It may be possible for other LP Solvers to handle the preprocessed input in a shorter time.

Since the program was able to decide a subset of the test problems faster than the PicoSAT solver, the result of this project is successful.

When trying to treat SAT problems, many different strategies can be executed at the same time and stop once one of them finishes. Given the solvers and problems of this project, our solution will be faster in some of the cases, making the execution time much shorter. This will lead to reducing the resources needed to compute the solution for this kind of problem.

19.1 Mistakes

Through the course of developing this project, many mistakes were made. Detecting what the most important mistakes are and well identifying them will help future studies.

Delay the tests and documentation

Since the initial results were not satisfactory, the focus of the1 project was moved to finding new ways to solve the problem and implementing them.

This resulted in leaving the documentation of the project and the necessary tests for the future. Although finally, the results obtained are satisfactory, the documentation of the process may not reflect it well enough.

Scope too broad

The initial goal of the project was too ambitious and expensive. Pretending to solve the problem totally from the beginning was a big mistake. Unfortunately, the scope of the project wasn't reduced until months into the project. It would have been smarter to tackle first the smallest problem, as we did in the end, and then try to extend the solution into the other cases.

One of the reasons was underestimating the problem. Even after reducing the problem to the clauses of size 2, it was an NP-hard problem. It should have been more evident that the initial goal of the project was too much.

The code

The code of the project is hard to follow. The versions of the code are distributed in different files and the files contain unused code.

Since one of the goals of the project is to serve as a base for future investigations, they could be clear and easy to read.

19.2 Obstacles

Through the development of the project some obstacles appeared, some of them expected and some unexpected. Down are enumerated the most important ones.

Not having access to commercial solvers

Since commercial solvers were not available for the tests, the time it took to solve the test problems was too much. Therefore, the tests were not able to finish.

To make a better comparison, it would have been better to execute the tests with commercial state-of-the-art, in order to determine if the developed program is or not an improvement in practice.

Lack of time

In the beginning, the number of hours for the project was estimated. According to this, the hours were assigned through the weeks. Since the length of this project has been from September 2021 to June 2022, it was difficult to expect some changes from the start.

In concrete, the biggest unexpected change was starting a full-time job. The available daily time was largely reduced.

19.3 Future work

In this section, different directions for future studies will be enumerated. The intention of this project is, not only to make conclusions but also to serve as a base for future explorations of the problem.

19.3.1 Improving the tests

As stated before, the quality of the tests was not good enough to determine if this project's solution was good enough or to measure how good it is. There are two main points where this can be improved.

The first of them is to use state-of-the-art solvers. Since the intention of the project is to determine if applying the preprocessing will result in a reduction of the execution time needed to decide the problems, testing it with outdated solvers will not give a good idea if this goal was achieved or not.

The second point is to provide more time for the algorithm to solve the problems. Since usually, the time designated for solving a problem is larger, it would be more realistic if the same time was given to test the algorithm too.

19.3.2 Extend the solution to q=2

Is it possible to apply a similar solution to solve the $q = 2$ case? When working with the $q = 1$ case, the input was first expressed as a graph, and then the algorithm for enumerating the maximal cliques was applied.

Is it possible to extend this idea to more dimensions using a *hypergraph*? This could be an idea worth studying.

References

[1] Uwe Schöning. *Logic for Computer Scientists.* Germany: Universit it Ulm, 1989. ISBN 139780817647629

[2] Stephen A. Cook. *The complexity of theorem-proving procedures.* University of Toronto, 1971

[3] Levin, Leonid (1973). *Problems of Information Transmission* .

[4] leinberg, Jon; Tardos, Éva (2006). *Algorithm Design* .

[5] Imperial College London - Logic Programming. [Accessed: 19 March 2022]. Available at `http://www.doc.ic.ac.uk/~cclw05/topics1/constraint.html`

[6] Franco Sanchez, Víctor. *Preprocessing techiques for Integer Linear Programming*

[7] Niklas Eén and Armin Biere. *Effective Preprocessing in SAT through Variable and Clause Elimination*

[8] Raihan H. Kibria. *Soft Computing Approaches to DPLL SAT Solver Optimization*

[9] Heidi Dixon. *AUTOMATING PSEUDO-BOOLEAN INFERENCE WITHIN A DPLL FRAMEWORK*

[10] Glassdoor. [Accessed: 19 March 2022]. Available at: `https://www.glassdoor.es/Sueldos/barcelona-researcher-sueldo-SRCH_IL.0,9_IM1015_KO10,20.htm?clickSource=searchBtn`

[11] Glassdoor. [Accessed: 19 March 2022]. Available at: `https://www.glassdoor.es/Sueldos/barcelona-junior-project-manager-sueldo-SRCH_IL.0,9_IM1015_KO10,32.htm?clickSource=searchBtn`

[12] Glassdoor. [Accessed: 19 March 2022]. Available at: `https://www.glassdoor.es/Sueldos/barcelona-junior-researcher-sueldo-SRCH_IL.0,9_IM1015_KO10,27.htm?clickSource=searchBtn`

[13] Glassdoor. [Accessed: 19 March 2022]. Available at: `https://www.glassdoor.es/Sueldos/barcelona-junior-developer-sueldo-SRCH_IL.0,9_IM1015_KO10,26.htm?clickSource=searchBtn`

[14] Glassdoor. [Accessed: 19 March 2022]. Available at: `https://www.glassdoor.es/Sueldos/barcelona-tester-sueldo-SRCH_IL.0,9_IM1015_KO10,16.htm?clickSource=searchBtn`

[15] Tarifa Luz Hora. Available at: `https://tarifaluzhora.es/?tarifa=pcb&fecha=10%2F03%2F2022`

[16] TMB. [Accessed: 19 March 2022]. Available at: https://www.tmb.cat/es/tarifas-metro-bus-barcelona/sencillos-e-integrados/t-jove

[17] Boolean satisfiability problem . Wikipedia. [Accessed: 06 May 2022]. Available at: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[18] Stephen Cook. Wikipedia. [Accessed: 06 May 2022]. Available at: https://en.wikipedia.org/wiki/Stephen_Cook

[19] Leonid Levin. Wikipedia. [Accessed: 06 May 2022]. Available at: https://en.wikipedia.org/wiki/Leonid_Levin

[20] Cook-Levin theorem. Wikipedia. [Accessed: 06 May 2022]. Available at: https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem

[21] NP (complexity). Wikipedia. [Accessed: 06 May 2022]. Available at: https://en.wikipedia.org/wiki/NP_(complexity)

[22] NP-completeness. Wikipedia. [Accessed: 06 May 2022]. Available at: https://en.wikipedia.org/wiki/NP-completeness

[23] The Cook-Levin Theorem FULL PROOF (Boolean Satisfiablility is NP-complete). Youtube. [Accessed: 11 May 2022]. Available at: https://www.youtube.com/watch?v=LW_37i96htQ

[24] NP-hardness. Wikipedia. [Accessed: 11 May 2022]. Available at: https://en.wikipedia.org/wiki/NP-hardness

[25] Linear Programming. Wikipedia. [Accessed: 14 May 2022]. Available at: https://en.wikipedia.org/wiki/Linear_programming

[26] Elements of a Linear Programming Problem (LPP). Towards Data Science. [Accessed: 16 May 2022]. Available at: https://towardsdatascience.com/elements-of-a-linear-programming-problem-lpp-325075688c18

[27] Trie. Wikipedia. [Accessed: 16 May 2022]. Available at: https://en.wikipedia.org/wiki/Trie

[28] Clique problem. Wikipedia [Accessed: 20 May 2022]. Available at: https://en.wikipedia.org/wiki/Clique_problem

[29] Output-sensitive algorithm. Wikipedia. [Accessed: 20 May 2022]. Available at: https://en.wikipedia.org/wiki/Output-sensitive_algorithm

[30] Etsuji Tomita, Akira Tanaka, Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. https://snap.stanford.edu/class/cs224w-readings/tomita06cliques.pdf

[31] Pigeohone Principle. Wikipedia. [Accessed: 21 May 2022]. Available at: `https://en.wikipedia.org/wiki/Pigeonhole_principle`

[32] Hypergraph. Wikipedia. [Accessed: 22 May 2022]. Available at: `https://en.wikipedia.org/wiki/Hypergraph`

List of Algorithms