UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Preprocessing techniques for Integer Linear Programming

Franco Sanchez, Víctor

Computer Engineering Graduate Dissertation

Computation specialty

March 2020

Director: Robert Lukas Mario Nieuwenhuis
Co-director: Enric Rodríguez Carbonell

Tutor: Joan Sarda Ferrer

Abstract

P vs NP has been (and still is) one of the most, if not the most important problem in computer science. NP-Complete problems are infamously difficult to solve, and not only that, but they are also very prevalent in almost every field inside computer science research. Because of that, finding efficient methods for solving NP-Complete problems is very important.

In this project, we focus on a couple methods for improving the efficiency of Integer Linear Programing (particularly, the subset of problems in ILP where variables are binary, called *Binary* or *Pseudoboolean Linear Programming*) and compare their performance using the solver "Intsat" developed by Barcelogic, a spinoff of UPC.

The final goal for this project is to implement and study the efficiency of these methods, and to integrate them into Intsat if they appear to make an improvement on the running time and/or the quality of the solutions.

Contents

List of Figures

List of Tables

1. Context

This graduate dissertation is made by Víctor Franco Sanchez, computer engineering student in the Computation specialty at the Polytechnic University of Catalonia (UPC for short, also known as BarcelonaTech). The specialty centers around the development of efficient systems in sectors like computer graphics, AI, compilers, and logical programming.

This project falls into the latter area, being a project focused on improving current satisfiability algorithms, particularly Integer and Binary Linear Programming, by preprocessing the input in certain ways that improve the runtime of the algorithms.

1.1. Terms and concepts

- **Linear programming** is the problem where you try to find the optimum values for a set of variables that can take real values such that they maximise a certain linear expression, given a set of linear constraints of the form $\sum q_i v_i \geq k$.
- **Integer Linear Programming** is a variant on the previous problem where the variables can only take integer values. If the values these variables can take are just 0 and 1, this problem is called *Binary Linear Programming*, or *Pseudo-boolean Linear Programming*. The decisional version of both these problems are both NP-Complete.
- **Satisfiability** is the property of a certain logical function that can be "satisfied". In other words, that exists a certain model that satisfies that formula. Determining whether or not a logical formula is satisfiable or not is known as the Boolean Satisfiability Problem, or SAT for short, and is the first known NP-Complete problem (Cook's theorem)[19].
- **CNF** *(Conjunctive Normal Form)*, also known as Clausal Form, is how most logical formulas are given to modern SAT solving methods. It is a conjunction of clauses ($c_1 \wedge ... \wedge c_n$), each clause being a disjunction of literals ($l_1 \vee ... \vee l_m$) [1, p. 75].
- **Resolution** is a technique for inferring new clauses from a CNF. It works by taking two clauses already in the CNF of the form $p \vee A$ and $\overline{p} \vee B$, and inferring $A \vee B$. A formula in CNF is satisfiable if and only if you cannot find the empty clause in the closure under resolution (that is applying resolution into your formula until you're producing no new clauses).
- **DPLL** *(Davis-Putnam-Logemann-Loveland)* is an algorithm for finding the satisfiability of a certain logical formula via four operations:
  · Decision: Setting a certain literal from *undefined* to *true* in your model.
  · Unit-propagation: Finding which literals can be set to *true* as a direct logical consequence of the literals set to *true* in the current model via resolution. A more in-detail explanation on how unit propagation works can be found in **Annex A.1**.
  · Backtrack: Once a conflict is found, backtrack up to the last decision, and change that literal from *true* to *false*, making that literal no longer a decision. (Alternatively, you can backjump up to the arbitrarily far away decision that caused the conflict, if you keep that information somewhere).
  · Fail: If you find some conflict, but no decision was made, you know the formula is unsatisfiable.
  If you have no more literals to decide on and you have not failed, the formula is satisfiable

and the model you have generated is already a model for your formula.
This algorithm, or slight modifications of it, are the most common algorithms for state-of-the-art SAT solvers, since not only they can be really optimized and parallelized, but also generate as a byproduct a model for the formula in hand if that formula is satisfiable [1, p. 110-114].

Modern implementations of DPLL often also learn additional clauses (lemma learning), and after a while they get rid of added clauses the algorithm deems expendable (clause cleanup).

- **BDDs** *(Binary Decision Diagrams)* are data structures used for encoding boolean functions efficiently in a DAG-like structure, where every node is labeled as a variable, and whether one path is taken or the other depends on whether the variable is set to true or false. The only nodes with no labeled variable are the leaves of the graph, which are labeled either 0 or 1. Given a certain interpretation, the value the BDD 'returns' is the value of the leaf node you get by following the BDD from its root. In the literature, the term BDD almost always refers to ROBDDs, which are a particular subset of the more general BDDs, in which the BDD is the smallest following a prefixed ordering of the variables [3].

- **Probing** is a preprocessing technique in which you obtain some information by setting one or many literals to true, and then obtaining some other formula that is a logical consequence of your initial formula when those literals are set to true, either by unit propagation, DPLL up to n conflicts, or any other method. If, by doing this, you find out that this formula is unsatisfiable, you know those literals are inconsistent, and therefore you can add a clause that encodes that. Also, if by probing the literals $\{l_1, ..., l_n\}$ you get that the literal l_k is true, you can add the clause $(l_1 \wedge ... \wedge l_n) \rightarrow l_k$ to encode that fact.

- **Strengthening** is a preprocessing technique exclusive to Pseudoboolean Linear Programming, in which, after probing a set of literals $\{l_1, ..., l_n\}$, you "strengthen" some constraint $c \leftarrow \sum q_i l_i \geq k$ into $s \sum_{i=1}^{n} \overline{l_i} + \sum q_i l_i \geq k + s$, where s is the "slack" of the constraint, or by "how much" that constraint is satisfied under the current model. As an example, if $c \leftarrow a + 2b + 3c \geq 2$, if by probing $\{d\}$ we obtain the model $\{b, c, d\}$, this formula is now "over-satisfied" by 3, and therefore we can strengthen this clause into $3\overline{d} + a + 2b + 3c \geq 5$ [2, p.76-81].

1.2. Problem formulation

Barcelogic is a UPC spin-off company co-owned by Robert Nieuwenhuis and BarcelonaTech. They are working on a Pseudo-boolean Linear Programming solver intended for cloud computing, and they are interested to see the efficacy of certain preprocessing techniques that have shown promising results in previous papers, theses and dissertations.

My goals in this project are to improve the efficiency and speed of integer and particularly pseudoboolean linear programming solvers by implementing and studying the efficacy of different preprocessing techniques.

It's important to note that CNF-SAT is a particular case of Pseudoboolean Linear Programming (which in turn is a particular case of Integer Linear Programming) because the satisfiability of the clause $l_1 \vee ... \vee l_n$ is equivalent to the linear inequality $l_1 + ... + l_n \geq 1$, and therefore there are two natural ways for developing new techniques for Pseudoboolean Linear Programming:

1. Generalizing techniques used for SAT.
2. Specializing techniques used for ILP.

The first method considered in this work particularly focuses on the former.

We're focusing on papers like Heidi Dixon's dissertation *"Automating pseudo-boolean Inference within a DPLL framework"* [2], which particularly focuses on solving the pigeonhole problem, problem that is explained in **Section 2** of this document, or like the paper by Ignaci Abío Et Al. *"A New Look at BDDs for Pseudo-Boolean Constraints"* [4], which focuses on obtaining pseudoboolean constraints via BDDs.

1.3. Previous work

Of course, many SAT solvers and Integer Linear Programming solvers already exist. Unfortunately, many of these are really secretive about the techniques they use, so we cannot know if they are using or not these techniques. The most popular tools for solving linear programming are Gurobi, CPLEX, GLPK (Open source), and SCIP, amongst others.

Although GLPK is open source, I couldn't find what kind of preprocessing techniques it uses, since its documentation only specifies that it uses *some kind* of preprocessing, and not which techniques they use. I can assume, though, that these techniques are very different to the techniques I've implemented for this dissertation, since they seem to work for real linear programming as well. In this project we're focusing on preprocessing techniques for integer and pseudoboolean linear programming exclusively, since that way we can get stronger constraints.

1.4. Stakeholders

- **Director(s).** The directors are a key figure in the development of this project. They contribute all the needed knowledge and information required for this project, and they also provide the code for their pseudo-boolean solver, as well as useful software and code such as other integer linear solvers, scripts for "fuzzing", etc. They are Robert Nieuwenhuis and Enric Rodríguez, and of course, without them this project would be impossible.
- **Barcelogic.** Barcelogic is the company for which this solver is being developed. Nieuwenhuis and Rodríguez are both members and founders of Barcelogic, and therefore everything I said about them can be said about Barcelogic.
- **End users.** As a Pseudo-boolean Linear Programming solver, this tool has a lot of applications in fields like Operational Research and Scheduling.
  These tools have been used by organizations like FIFA[6], and are also used for public transport planning, automation, floorplanning, and a lot more applications.
- **Research.** This problem is really close to the P vs NP fringe, meaning there's a lot we don't know yet about this, and therefore there are many people researching similar topics. I hope that the results obtained in this dissertation can help to improve our knowledge on this topic.
- **Students.** If the dissertation ends up being accessible enough, it can be used as supporting material for students trying to get into very similar topics, which are often thought of as very counterintuitive.

2. Justification

There are many problems that state-of-the-art satisfiability solvers have trouble solving. One example of these is known as the pigeonhole problem, which states that $n + 1$ pigeons cannot be placed in n holes if each gets its own hole. These problems are believed to be common subproblems in many domains, such as planning and scheduling. This problem should be easy to solve, but traditional satisfiability methods make them way harder.

Traditional satisifiability methods fail to solve this kind of problems efficiently because their proof system is based on resolution, and all proofs by resolution of the pigeonhole problem are exponential in length [2, p.iv-v].

Here is where preprocessing comes into play. The idea is to preprocess the formula in advance, to acquire new information that cannot normally be obtained efficiently with the toolset of modern satisfiability methods, which barely goes beyond just resolution, to try and solve these problems more efficiently.

Also, I am not working from a blank slate, but instead, I am working with the code of a pseudoboolean solver, which already has some minor preprocessing techniques, which produce some byproducts that can, but are not used for further preprocessing. For example, the code I was given implements probing, but not strengthening, which is a technique that builds on top of probing. If I am able exploit these kinds of things that would be even better.

For alternatives to my project, I can think of two things:

1. To implement different preprocessing techniques. The problem with this is that because of how lackluster the information about this topic is, it is very difficult to assess if one technique is better than another just by looking at the theory, and therefore practical experiments must be done to determine the efficacy of these with respect to each other. Of course, if you carry out these experiments, you're already implementing these techniques. Therefore, which techniques we decide to implement is really subjective. The techniques I have implemented are techniques that my directors, which have been working in the field of logic in computer science for years, think that can be effective.

2. To completely change the algorithm for solving integer or pseudoboolean linear programming. That, of course, is not really viable, especially as an end-of-degree dissertation that I've done in two semesters worth of time, since probably no algorithm I could come up with would ever come even close to the efficacy of DPLL for the task at hand.

3. Scope

3.1. Objectives and Requirements

The main objective of this project is to improve the efficiency and speed of integer and pseudoboolean linear programming solvers by implementing different preprocessing techniques, and to evaluate their effectiveness in terms of time gain with or without applying this technique. To achieve this goal, I have to update my knowledge of the state of the art in Operations Research, linear programming, SAT solvers, etc.

I also must verify the correctness of my code. Because this project builds up from other

theoretical papers, the theoretical correctness is assumed to be already proven. But still I must verify that what I programmed is correct, and for that I'll use fuzzing, which is a technique based on generating small problems, and verifying the solution is the same when comparing it with other solvers.

3.2. Obstacles and risks

This section exposes a small set of obstacles and risks that can appear during the development of this project. This same thing is further explained in **Section 8** of this document.

- **Coding mistakes and "bugs".** Every programming project is susceptible to coding mistakes, which, sometimes, can take the better part of a weekend to fix. I can try to use debugging tools and be rigorous on my programming to avoid and solve mistakes. But, generally, these things are unavoidable.
- **Unexpected/underwhelming results.** A lot of academic results are found to be irreplicable, perhaps because of an incomplete methodology on the paper, a lack of information, or just plain luck from the original paper's writer. Because of that, sometimes, we can get some very underwhelming results, which can hold up the whole project.
- **Being overwhelmed by the workload.** I am not used to projects as big as this one. Also, I feel there's a big uncertainty component for this project, and therefore I feel like I could get easily overwhelmed after a couple of things go particularly bad.
- **Miscellaneous other problems.** There's always unexpected problems, such as sickness, accidents, familiar problems, or other unpredictable problems.

4. Implemented Methods

In this section we see the formulation, implementation and results of every implemented technique for this dissertation.

4.1. Strengthening

The first method implemented is a method that profits from the fact that Intsat already did probing, and uses this fact to "strengthen" the clauses as described in Heidi Dixon's PhD dissertation[2].

4.1.1. Formulation

Under a partial assignment of variables P, we say a constraint $c = \sum a_i x_i \geq k$ is *over satisfied* if, by only taking into consideration the literals satisfied under P, c already meets the inequality with some margin. This margin is often called "slack".

The constraint c can also be written as $a \cdot x \geq k$ or $a^T x \geq k$, where a is the vector of terms $(a_1 ... a_n)$, and x is the vector of variables.

We also define s as the slack, or margin of a certain constraint, $s = max(0, k - a^T x)$.

Given a particular Binary Linear Programming problem C, we can strengthen by running the following method:

```
Strengthen(C):
    foreach literal l in C:
        P <- {l}
        P <- UNIT-PROPAGATE(C, P)
        foreach c = a . x >= k with positive slack s:
            c <- a . x + s ~l >= k + s
```

The proof for the correctness of this method can be found in **Annex D**.

To give an example, let C be the following constraint set:

1. $1x_1 + 2\bar{x_2} + 1\bar{x_3} \geq 1$
2. $1\bar{x_4} + 1\bar{x_2} \geq 1$

If we probe the literal x_4, because of *Constraint 2* we can deduce $\bar{x_2}$ must be true for that constraint to be met, meaning that $\textit{UNIT-PROPAGATE}(C, \{x_4\}) = \{\bar{x_2}, x_4\}$. Now, if we plug P into *Constraint 1*, the constraint is over satisfied by one unit, meaning that we can strengthen the constraint:

1'. $1x_1 + 2\bar{x_2} + 1\bar{x_3} + 1\bar{x_4} \geq 2$

More generally, given a constraint set C, a partial assignment $P = \{l_1, ..., l_k\}$, $P' = \textit{UNIT-PROPAGATE}(C, P)$[1] and a constraint $c = \sum a_i x_i \geq r, c \in C$ with positive slack s under the partial assignment of P', the constraint

$$\sum a_i x_i + s \sum_{i=1}^{k} \bar{l_i} \geq r + s$$

is implied by C.

4.1.2. Development

For initial checks of correctness, I implemented the "Strengthen" method mentioned before in Python. Although it worked, it was considerably slow, not being able to finish strengthening for large enough files.

I moved to C++, where I implemented this method using as a basis the source code of Intsat.

This code had to implement the strengthening method previously mentioned, using the same data structures and same programming paradigm as Intsat itself. Optionally, this method could also implement the 2 or more literal strengthening mentioned above. At the end, this should either run Intsat's solve method and output the optimal result, or output the strengthened problem file for another solver to solve.

In terms of implementation, it is important to note that Intsat does not have any methods for modifying constraints, and implementing one would probably be unwise, since I would have to make sure that no invariant inside the code is broken. Refactoring a code that is not yours

[1] A description and explanation for the method of unit propagation for Binary Linear Programming can be found in **Annex C.1**.

is time consuming, hard and prone to instigate new bugs. Instead, together with the help of my directors, we decided on a middle ground: Constraints have a flag for whether or not they are "initial" constraints. If a constraint is not initial, on a cleanup it will get removed. So, instead of modifying the constraint, I add a new constraint with the "initial" flag set to true, and set the "initial" flag of the original constraint to false, so that upon cleanup we get rid of the original constraint.

4.1.3. Implementation

This section is dedicated to important snippets of code on the implementation. Because the implementation was very dependent on the structures put in place by the developers of Intsat, this code will not be immediately useful for anyone not working in Intsat, but it can probably still be useful to understand how to develop this method.

```
1   void Solver::strengthen( Constraint & c, int probingLiteral ){
2       if(not c.isInitial()) return;
```

Line number 2 is to ensure that we only strengthen constraints that have not already been strengthened. As said on **Section 4.1.2**, that strengthened constrains were flagged as "non-initial" so the cleanup would get rid of them. Also, a very important detail is that this function is called right after probing, therefore no probing is necessary.

```
3       long long int slack = -c.getConstant();
4       for(int i = 0; i < c.size(); ++i){
5           if(model.isTrueLit(c.getIthLiteral(i)))
6               slack += c.getIthCoefficient(i);
7       }
8       if(slack <= 0) return;
```

The next thing we do is check whether or not there's slack on that constraint. Remember that strengthening only works when the slack on the constraint is positive.

```
9       vector<int> coeffs(0), lits(0);
10      long long int res = c.getConstant() + slack;
11      bool found = false;
12      for(int i = 0; i < c.size(); ++i){
```

Now we're iterating by every term in the constraint. "res" represents the constant term of the constraint. We must go case by case, checking whether or not the literal is the same as the new literal we're adding, because Intsat has three invariants that I must keep:

1. Constraints do not have repeated variables.
2. The constraints cannot have negative coefficients.
3. The constraints are sorted by increasing order of variable.

```
13          if(c.getIthLiteral(i) == -probingLiteral){
14              found = true;
15              coeffs.push_back(slack+c.getIthCoefficient(i));
16              lits.push_back(-probingLiteral);
17          }
```

The first case we check is whether or not the literal in the clause is equal to the literal we're trying to add. If that is the case, because the slack is always positive, no invariant is broken. We set a flag "found" to true to make sure we don't add the literal again at the end. The next case is a little bit more complicated.

```
18          else if(c.getIthLiteral(i) == probingLiteral){
19              found = true;
20              if(slack - c.getIthCoefficient(i) == 0) {
21                  res -= slack;
22              } else if(slack - c.getIthCoefficient(i) > 0) {
23                  coeffs.push_back(slack - c.getIthCoefficient(i));
24                  lits.push_back(-probingLiteral);
25                  res -= c.getIthCoefficient(i);
26              } else {
27                  coeffs.push_back(- slack + c.getIthCoefficient(i));
28                  lits.push_back(probingLiteral);
29                  res -= slack;
30              }
31          }
```

This is the case where the literal we're adding is found negated in the constraint. That means that we have a situation similar to the following:

$$\sum_{i \neq j} a_i x_i + a_j x_j + s\bar{x_j} \geq k$$

Where $a_j x_j$ is the term in the original constraint with the same literal as the probed one, $s\bar{x_j}$ is the added term because of strengthening, and $\sum_{i \neq j} a_i x_i$ is just the rest of terms in the constraint.

Now, to simplify this constraint we can do one of two things. We either realize that $x_j = 1 - \bar{x_j}$, and therefore this simplifies as follows:

$$\sum_{i \neq j} a_i x_i + a_j x_j + s\bar{x_j} \geq k$$

$$\sum_{i \neq j} a_i x_i + a_j(1 - \bar{x_j}) + s\bar{x_j} \geq k$$

$$\sum_{i \neq j} a_i x_i + a_j - a_j\bar{x_j} + s\bar{x_j} \geq k$$

$$\sum_{i \neq j} a_i x_i + (s - a_j)\bar{x_j} \geq k - a_j$$

Or, otherwise, we realize that $\bar{x_j} = 1 - x_j$, simplifying as:

$$\sum_{i \neq j} a_i x_i + a_j x_j + s\bar{x_j} \geq k$$

$$\sum_{i \neq j} a_i x_i + a_j x_j + s(1 - x_j) \geq k$$

$$\sum_{i \neq j} a_i x_i + a_j x_j + s - s x_j \geq k$$

$$\sum_{i \neq j} a_i x_i + (a_j - s) x_j \geq k - s$$

Which one of these two simplifications we choose depends on which one holds the invariant that coefficients must always be positive (In this case, the coefficient is either $s - a_j$ or $a_j - s$). On line 20 we have the case in which both literals cancel each other out, meaning that the literal disappears, regardless of the simplification chosen, and therefore we arbitrarily chose the second one. On line 22 we have the case in which the first simplification gives out a positive coefficient, and on line 26 is the second one.

```
32          else {
33              if(not found and
34                      abs(c.getIthLiteral(i)) > abs(probingLiteral)){
35                  found = true;
36                  coeffs.push_back(slack);
37                  lits.push_back(-probingLiteral);
38              }
39              coeffs.push_back(c.getIthCoefficient(i));
40              lits.push_back(c.getIthLiteral(i));
41          }
42      } // End of the for loop that began in line 12
```

The if statement in line 33 is important to keep the invariant that the constraints have to be sorted by increasing order of variable. Without that check, if we just add the literal at the end, the order can be wrong.

```
43      if(not found){
44          coeffs.push_back(slack);
45          lits.push_back(-probingLiteral);
46      }
```

If the literal was never inserted anywhere, we add it at the end of the constraint.

```
47      WConstraint newConstraint(coeffs, lits, res);
48      addAndPropagateConstraint(newConstraint, true, 0, 0);
49      c.setInitial(false);
51  } // End of the function
```

Finally, we define the new constraint as an original constraint, and set the "initial" flag of the original constraint to false.

This is not an exact replica of the code, but a slight simplification without some bits of code, because in the original code there were extra lines of code for efficiency, checking whether or not the resulting constraint was equivalent to a CNF clause, because Intsat separates between both, due to the fact that it's able to treat CNF clauses more efficiently than linear constraints. But other than that, the code is the same.

It's important to note that this function is called after probing a certain literal for every constraint, and that every literal is probed.

4.1.4. Results

When talking about results, I think it's important to take into consideration both the correctness and the efficiency of the method.

On the topic of correctness, this implementation had some problems when outputting the strengthened problem into a file and then solving this new problem. This is further explained in **Section 8.7** of this document. When the problem file is not outputted, and instead we solve directly after strengthening, taking advantage of the state of the program we built while strengthening, it worked correctly under the executed test cases. To prove correctness and test efficiency, we tested with auto-generated problems (fuzzing), with pigeon-hole problems, and with problems from the MIPLIB library of problems [15]. In all cases, under the circumstances described, the results were correct. We compared correctness both with unmodified Intsat and with IBM's CPLEX.

Because the study of this method was cut short due to some unexpected problems (See **Section 8.7**), there is no clearly structured data on the efficacy of this method. That being said, the few tests that were conducted showed no serious improvement, or sometimes even a detriment to the efficiency of Intsat. More data can be found in **Annex E.1**.

4.2. Local Search

The second method implemented is the solution-based local search algorithm envisioned by Dimitris Bertsimas et al. in 2012[17]. This method was chosen because it allowed me to detach from Intsat, and instead work from scratch, using only my code, reading the original problem and an initial solution found by Intsat, and trying to find solutions of higher quality from there.

4.2.1. Formulation

Given a certain Boolean Linear Programming problem with n variables and m constraints, with an objective function to maximize $c^T x$, with A being an $m \times n$ matrix of integers, and b being a vector of m elements, defining the constraints $Ax \leq b$ and $x \in \{0,1\}^n$, and a certain feasible solution z_0, we define the following terms:

- Let $x_v = max(Ax - b, 0)$, $x_v \in (\mathbb{Z}^+)^m$ be the amount of constraint violation. That is, by how much each constraint is not met for a certain solution x.
- Let $x_u = max(b - Ax, 0)$, $x_u \in (\mathbb{Z}^+)^m$ be the amount of constraint slack. That is, by how much each constraint is overmet.
- Let $x_w = min(x_u, 1^m)$, $x_w \in \{0,1\}^m$ be a vector of m 0s and 1s, where 0 means there's no slack and 1 means there's some slack.
- Let $trace(x) = [x_v; x_w]$, $trace(x) \in (\mathbb{Z}^+)^m \times \{0,1\}^m$.
- Let h be any hash function.

Also:

- We say two solutions x and y are *adjacent* if the taxicab distance between them is exactly one.
- A certain feasible solution y is *better* than another feasible solution x if $c^T y > c^T x$.

- Let z be the best feasible solution. A solution y is *interesting* if the following properties hold:

  (A1) No constraint is violated by more than one unit by y.

  (A2) The taxicab distance between the traces of y and z is of at most a certain hyperparameter[2] Q.

  (A3) $c^T y > c^T x$ for all x already examined by the algorithm such that $h(trace(x)) = h(trace(y))$

We will also define an object called a *Trace Box TB* that stores the evaluation of the function to optimize for the hash of the trace of every seen solution, and a set of interesting solutions called the *Solution List SL*.

With that in mind, we can finally define the local search procedure as follows:

```
LocalSearch(A, b, c, z0, Q):
    z <- z0
    SL <- {z}
    while SL is not empty:
        x <- SL.pop()
        foreach y adjacent to x:
            if y is feasible and cTy > cTz:
                z <- y
                SL <- {y}
                TB[i] <- -Infinity, for all i
                break out of the for loop
            else if y is interesting:
                TB[h(trace(y))] <- cTy
                SL.add(y)
```

This code adds the initial solution into the SL set, and then the main loop starts. The loop pops an element from the set, and looks through all the adjacent solutions. If an adjacent solution is interesting, this solution is added to the set, and if an adjacent solution is both feasible and better than the current solution, we set this solution to be the new best solution and begin the process all over again, as if this solution was the initial we got. This keeps on going until the set is empty.

4.2.2. Development

Learning from past mistakes, this method was implemented as a stand-alone application in C++. This program had to read and parse:

- An input file using the .lp format for MLP containing a boolean linear programming problem.
- A solution for that same problem in the same format Intsat outputs solutions.
- A value for the hyperparameter Q.

[2]Hyperparameter used in a similar sense of what is often used in Machine Learning: That is an external parameter with no clear optimum value, chosen by the engineer.

And then apply the algorithm described previously.

Because we only care about a trace of a solution when it is either feasible or interesting (as in the definition of "interesting" I gave in **Section 4.2.1**), we can store the trace as a vector of bools (because the second half is already all 0s or 1s, and, if the solution is feasible the first half will be all 0s, and if it's interesting, because of property (A1) mentioned before, will have, at most, some 1s). Vectors of bools in C++ are implemented more efficiently in memory and thus we increase the efficiency of operating with traces, since if they occupy less in memory, the chances of cache misses go down, but often have some time overhead when accessing to a certain position[16]. This overhead was supposed worth it because cache misses are very costly, but no tests were conducted to test whether or not this is the case. To keep track of the lost information, we also add a boolean flag for whether or not some constraint is violated by more than one unit. We also add another flag of whether or not the solution is feasible, for efficiency.

4.2.3. Implementation

The implementation for this method is pretty straightforward, and the final code resembles a lot the pseudocode given in **Section 4.2.1**. That being said, there are some points worth mentioning.

```
1   struct traceStruct {
2       std::vector<bool> violation;
3       std::vector<bool> slack;
4       bool isFeasible;
5       bool isViolatedByMoreThanOneUnit;
6   };
```

This is the definition for the trace. As mentioned before, we're using vectors of bools for greater memory efficiency. The trace is split into its two components "violation" and "slack" for readability.

Bertsimas et al.'s paper grants results for any injective hash function h, so the hash function implemented was a trivial function that only took the trace and casted it into string.

```
7   string h(struct traceStruct & trace){
8       string hash = "";
9       for(int i = 0; i < trace.violation.size(); ++i)
10          hash += std::to_string(trace.violation[i]);
11      for(int i = 0; i < trace.violation.size(); ++i)
12          hash += std::to_string(trace.slack[i]);
13      return hash;
14  }
```

It's important to note that "std::to_string" casts *true* to *"1"*, and *false* to *"0"*. So basically, this just turns the trace into a binary string.

The paper mentioned a different hash function, but due to a lack of time it could not be implemented.

The traces were computed just once per solution, and the traces were passed by reference every time some information the trace contains was needed.

The Solution List SL is implemented with a queue, as recommended by the authors of the original paper, and the Trace Box TB is implemented with a map. If the element is not in the map, the value is considered -Infinity, meaning that to run the line "TB[i] = -Infinity, for all i" all I had to do was to empty the map.

4.2.4. Results

The correctness for this implementation was tested both with fuzzing and with problems from MIPLIB. We also generated "unweighted maximum w-set packing" problems, described by having a constraint matrix $A \in \{0,1\}^{m \times n}$ that has exactly w "1"s in each column, and the function to maximize $c^T x$ having a vector c equal to all ones. These problems were chosen because the original paper gave an upper bound to the expression z^*/zh ($z*$ being the optimal solution and zh being the solution found) equal to $w^2/(2w-1)$. In all cases, the implementation was shown correct under the tested cases, and gave positive results.

As an example, we'll use two problem from MIPLIB called fast0507.lp and seymour.lp. Both are "Set Covering" problems, category of problems for which the paper promises good results on.

First, we'll focus on fast0507.lp[20]. Intsat found 4 solutions after 107, 252, 451 and 11825 seconds, with an objective function of 347, 336, 302 and 291 respectively.

For the following plot (***Figure 1.***), I left the local search method running for at most 10 minutes per execution, with differing values for the hyperparameter Q (1, 2, 5, 10 and 50). Each black point represents an Intsat solution, and the different lines coming out of the points represent the different executions of the local search method for each different value for Q. Every time the local search finds a better solution than the initial one, its corresponding line "dips down", indicating that a new better solution was found. The last solution found by Intsat is omitted, since it's already the optimum and therefore the local search won't be able to improve upon it, and being found so late would mess up with the plot.

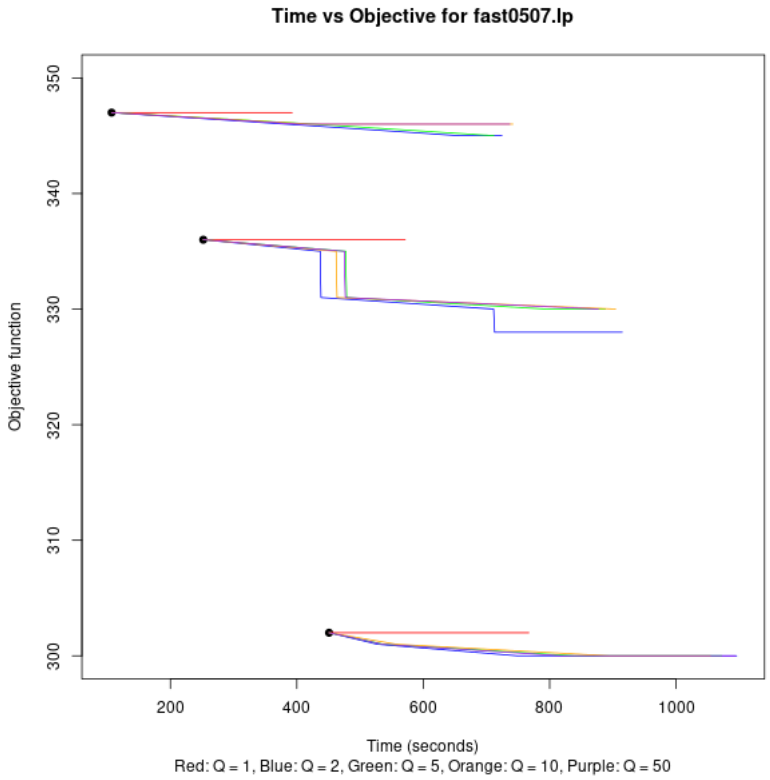

Figure 1: Data extracted from running the local search method on fast0507.lp.

This plot shows some promising results, especially for the last executions. Since the next solution found by Intsat was more than 3 hours after the previous one, this algorithm being able to find better solutions compared to the second to last one found in significantly less time could mean a decent improvement. The next plot, though, is not as promising.

For this next plot (***Figure 2.***), I followed the same steps as in the previous one, this time for seymour.lp[21]. Unlike the other plot, no Q over 2 is plotted. This is not because the tests were not run, but because the results were the same with Q = 5, 10, 50 as with Q = 2. That is, nothing was found.

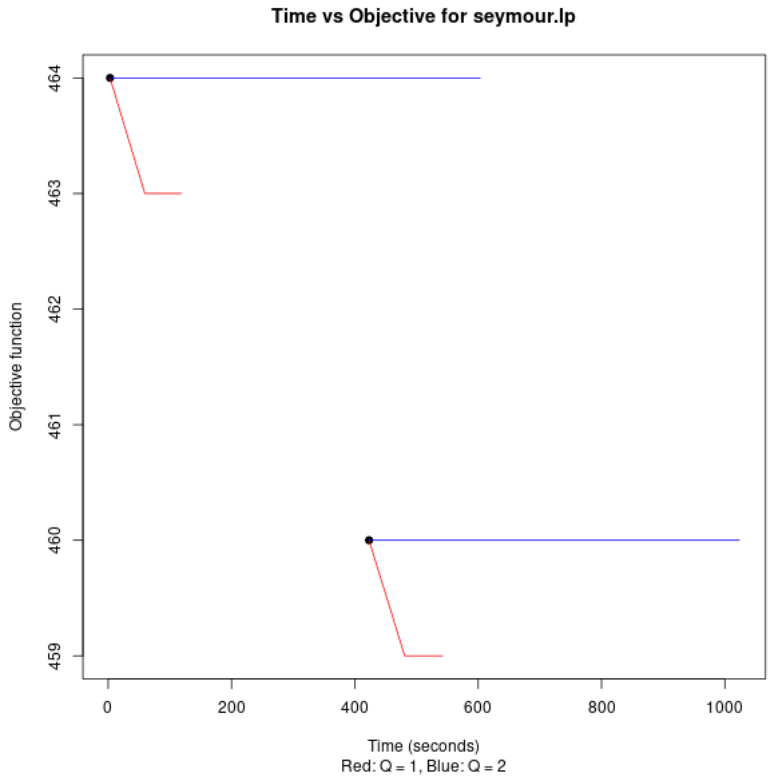

Figure 2: Data extracted from running the local search method on seymour.lp.

The main reason something was found with Q = 1 but not with Q = 2, is because for Q = 2 this algorithm has to look for way more possible solutions, meaning that there's a chance that solutions previously found with Q = 1 are not found because of a lack of time. For the same reason, for "Qs" of 5, 10 and 50 nothing is found.

All of this hints to the idea that this method might be useful under some circumstances, but is not *always* useful. Further data can be found in **Annex E.2**, including the exact data obtained while testing with the unweighted maximum w-set packing problems.

5. Methodology and rigour

5.1. Methodology

The methodology used for this project is an Agile methodology. Agile methodologies are an efficient, iterative and incremental approach to software development, where you make the development cycle short so you always have a functioning version, and you can always give feedback to fix any mistakes made in any of the other development stages.

These cycles are called "sprints". Normally, the time required for these sprints is decided beforehand, but by the nature of my project, it's difficult for me to predict how much an sprint will take me. I'll try to make most sprints one week long, and I'll work to keep the sprints as short as I am capable of to try and span as much as I can.

Every week, I'll meet with my directors, and discuss what my next sprint is going to be (if I am not in the middle of an sprint already). During the week, I'll work to complete that sprint, and by the next week, when I meet with my directors once again, I'll show them the progress I made during this time, and extract conclusions based on the results obtained during the week. If the sprint is finished, we'll discuss my next sprint and so the cycle repeats.

If some obstacle occurs during a sprint that heavily inhibits my capabilities to continue the project, I'll meet with my directors, and we'll try and make some contingency plan for it. This is further explained in **Section 8** of this document.

5.2. Monitoring tools

I'll be using mainly GitLab for version control of the project, and to store the multiple versions of the project memory. This way, I'll have a better view on how this project evolved over time.

Regular meetings will be held with the directors where we'll review, amongst other things, the progress made since the last meeting, the status of the project as a whole, and any doubts and questions that might have arisen. Given the nature of the project, these meetings will sometimes also provide theoretical explanations not seen during the course for the proper development of the project.

If any unforeseen problems arise, I'll get in touch with the directors and, if necessary, a contingency plan or route to follow will be proposed to solve the problem in question.

Due to recent events, not all meetings can be face-to-face. Therefore, while we're on state of alert, we will keep in contact via email or some other online service for communication, such as Skype. In principle, this shouldn't be anything more than a minor inconvenience.

5.3. Update on methodology and monitoring

During the development of the project, as predicted, I couldn't meet with my directors due to the confinement. We've been contacting via email, sometimes once per day, sometimes once per week. That being said, I don't feel as if this was just a *minor* inconvenience as I predicted back when I wrote the previous section. Perhaps it's because of how I learn, but this has been way more difficult for me than what I anticipated, and not being able to talk face-to-face with my directors for sure made it worse.

I feel like I've been way too careless, and I have used GitLab far less than what I expected. Because of that, I have ended up losing part of my progress, especially during the first of the two methods implemented.

I have kept using the Agile methodology during the project. As predicted, the length of each sprint was very inconsistent, and I even had some sprints that were just running tests. At the beginning, sprints were a week long, but especially during the second method, the sprints got on average a lot shorter, mainly because of the lack of time. Every time I found myself in front of a problem that was way above my weight class, I made sure to send an e-mail to my directors as quickly as possible to try and solve this problem as soon as possible.

6. Description of the task and estimates

This project will be developed in the time span of one semester.

Due to the fact that I will not be doing this project full-time, my prediction is a minimum of 14 weekly hours, and an average of 30.

6.1. Research

This being a research-based dissertation, I have to do my research as well. I have to read on the state-of-the-art solutions for preprocessing integer and pseudoboolean problems to be able to develop these techniques.

Effort estimation: It really depends on how difficult the subject at hand is to research. I expect 10 hours for each technique to implement. Maybe more, if the technique is more complex, or less if it's simpler.

Dependencies: None.

Material requirements: Papers.

6.2. Project planning

The GEP deliveries take time to write.

Effort estimation: Since writing is not my forte they might take me a little bit more than the planned time, but as an approximation I'll just take 90 hours estimation from the fact that this subject is 3 credits, and split them accordingly to each deliverable in function of their workload.

Dependencies: This task has no further dependencies besides the fact that, before being able to write this, I have to be somehow familiarized with my project.

6.3. Familiarize with existing code

I am working with pseudo-boolean solver written in C++ given by Barcelogic [7] for me to implement the different preprocessing techniques. I must familiarize myself with the code so I know what's implemented and what's not, the basic structure of the code, and where and how to code everything I have to code.

Effort estimation: Because I've been programming in C++ for a while, and I worked previously on similar (although simpler) code, I don't expect this to take me too long. Between 5 and 10 hours seems like a reasonable estimation.

Dependencies: I depend on having the code.

6.4. Technique #1: Implementing Strengthening

The first technique to implement is strengthening. I talked about strengthening on the previous delivery, but in short, strengthening is a technique to obtain stronger pseudo-boolean

constraints by knowing by how much a certain constraint is over-satisfied by setting some literal to true.

Effort estimation: This technique is not too difficult, so I don't expect it to take too much time to implement. Perhaps 3 weeks worth of time, which would be somewhere around 90 hours according to previous predictions.

Dependencies: I depend on tasks 1.1 and 1.3.

6.5. Studying the improvement of technique #1

After programming technique #1, we need to assess the efficacy of the technique in question.

Effort estimation: Depending on the results, I expect this can take from 5 to 20 hours off my useful time. Of course, these studies take longer to complete, but they can be done automatically, in the background, so this time is just the time it takes me to write the scripts to get the data, and to extract conclusions from this data.

Dependencies: I depend on task 1.4.

6.6. Technique #2: Implementing local search for finding constraints

Knowing what's the strongest constraint that is a logical consequence of a formula is a CO-NP Hard problem (Since the strongest constraint you can get from a contradiction is $0 \geq 1$, and knowing if a formula is a contradiction is a CO-NP Complete problem). Therefore, I'll be looking to get some *strong enough* constraint that is a logical consequence of the formula via local search, instead of just the "strongest" constraint.

Some state-of-the-art methods use many techniques to get constraints that are logical consequences out of logical formulas, such as using BDDs [4], but these still are not the best-possible constraints obtainable, for the reason stated previously. We can study if we can possibly improve these solutions by slightly altering these clauses to make them stronger. I don't have the sufficient knowledge yet to assess how I'd implement this (since that is what the research task is for), but the main idea is that by decreasing the independent term, or by increasing the coefficients, you get a stronger constraint. If that constraint's still a logical consequence, we're good. To implement all of these, I will not only have to implement the local search, but also the BDDs or some other technique [5] to obtain some initial pseudo-boolean constraints.

Effort estimation: This technique is more general and more complex, therefore I suspect it will be more difficult to implement than the previous one. I expect something around two months worth of time, or 240 hours. It might be more, it might be less. It all depends on how difficult it is for me to implement this and how good (or bad) the results I am getting are.

Dependencies: Technically, I just depend on tasks 1.1 and 1.3, but for orderliness sake, I'll begin to work on this technique after I finish task 1.4.
We can split this task into two subtasks: The obtaining of the original constraint, and the local search. The second subtask depends on the first.

6.7. Studying the improvement of technique #2

After programming technique #2, we need to assess the efficacy of the technique in question.

Effort estimation: Same as with technique #1, something between 5 and 20 hours.

Dependencies: I depend on task 1.6.

6.8. Future techniques

Depending on the success of the previous technique, I might pivot to a different technique.

Effort estimation: I'll expend the rest of my time on this project on future techniques, so it all depends on how much it takes me to finish everything else. Maybe I've got time to do another technique, maybe I don't.

Dependencies: I depend on having done task 1.6, and also to have read the theory behind this hypothetical third technique.

6.9. Memory

During the development of this project, I'll be doing the written memory.

Effort estimation: I'll be writing the memory as I go. I've been assuming that the time in implementing and studying the previous techniques already includes the time it'll take me to write the memory.

Dependencies: I depend on having something to write about.

6.10. Updates on tasks

The technique #2 was not implemented. Instead, we implemented a different technique based on the local search of solutions, instead of the local search of constraints. This technique is further explained on **Section 4.2**.

7. Gantt diagram

Because this is planning is based on several assumptions, and because the nature of this thesis, the time and task planning is only a reference, and may be subject to change.

On the previous section I bundled together research, project planning and the memory as just one task, but for the purposes of this Gantt, I've separated them into different tasks.

*Consult **Figure 3** on the next page.*

Figure 3: Gantt diagram for this project. Self elaborated, made with the pgfgantt library on LaTeX.

7.1. Update on planning

This project has been longer than expected, taking me twice as long as predicted. Therefore, the Gantt before is incorrect.

To begin with, each task has taken approximately twice as long, with Strengthening taking around 13 weeks, and Local Search taking 19 weeks, for a total of 32 weeks of work, when before the expected time for the development of both techniques together was 15 weeks.

Also, I've spent more than one week researching each technique, often going back to the research papers during development to better understand the concepts I didn't quite get the first time around.

8. Risk management

Due to the nature and complexity of the task at hand, it's difficult for me to make a contingency plan for what to do when something goes wrong. So, to make this easier for me, I'll go section by section, talking about all plausible problems I can have during the development, and some possible solutions.

8.1. Research

It's possible that I am unable to understand the things I am researching, perhaps due to a lack of knowledge, or because of personal limitations. If that is the case, I'll make sure to contact my directors as soon as this happens, and ask them to help me.

Nevertheless, in the task time setting I have also taken into account 5 hours of overtime that may be used to solve this type of problems.

8.2. Development

Coding mistakes are always unavoidable. I can try to be more careful with what I am coding, add watchdogs and asserts to my code, have code revisions and utilizing tracking tools and debuggers. These problems are accounted for in the time expected to finish the development tasks, though, since these kinds of mistakes are a part of development, and they can take up to half the time of the project to solve.

8.3. Improvement study

The results I get can be seriously underwhelming for reasons I stated on the previous deliverable. If that happens, there's nothing I can do besides pivoting to some new technique. If that happens, the impact it will have on the time planning is difficult to predict, since it can mean restructuring heavily the project.

8.4. Memory

I am not very good at writing, and I can very often get blocked when writing even technical texts. If that happens, I can always ask my associates for help, look previous works for reference, or in case I see it fit, I can ask my directors or my tutor. This could add up to 10 hours to the writing of the memory.

8.5. Miscellaneous

Hardware can fault. If some of my computers faults, because I am using version control, and I have two computers that are functionally interchangeable for the task at hand, the lost data won't be great. In the improbable case that both break, that is more of a problem, but I can still find alternatives.

I can have some other problems that are totally outside my control. If I fall sick, I injure myself, or something bad happens to someone near me, there's little I can do to avoid these things. To get back to schedule, the best thing I can do is explain to my directors what happened, and to try and reschedule tasks and discourse about the planning of the project with my directors.

8.6. Lack of time

If I seriously underestimated the expected time of some of these tasks, or if several of these unforeseen risks happen such that I am unable to finish some of these tasks, I will be obligated to cut the time spent on some of these tasks. When push comes to shove, if we assume we do not reach task 1.8, the next thing to cut would be task 1.6. Particularly the second subtask, since without the second subtask i *still* have something functioning that may be useful, but the second subtask without the first one is useless.

8.7. Halting of the implementation of Technique #1

While implementing this method, I ran into a lot of problems. First, the results were not as good as we were expecting. Also, strengthening was rather slow. Because of that, we decided for the program to run the strengthening method, and then to output the strengthened problem file, but this just lead to way more problems.

There was a test file called *rand27da90.cudf.paranoid* that, when runned with either CPLEX or unmodified Intsat, gave an optimum of 11364, but when the outputted file was runned after the strengthening, it gave an optimum of 11362 instead.

We know this cannot be a problem of correctness by the strengthening method, because the program that ran the strengthening method was a modified version of Intsat, so I could tell the program to optimize the problem before outputting it, and when doing so, it gave the correct answer again. Therefore, the problem had to be with the outputting of the file, and not with the strengthening method itself.

All of this happened amidst the SARS-COVID-19 pandemic, and I spent around 3 weeks trying to solve this problem. At this point, I felt like this project was getting stagnant, and so I contacted my directors for a solution, and they proposed me a different method to implement, one that hopefully wouldn't have the same problems as this one.

9. Budget

Due to the fact that this is a research job, there's no production or commercialization costs. Therefore, I'll have to consider the costs of design, planning, study, implementation, human resources and materials.

To compute the amortization costs, we'll take into consideration the useful life of the item in question when compared with the duration of this project, of at most 6 months in this case.

It's also worth noting that I'll be using the point **(.)** as decimal separator, and I'll be using two significant figures for the computations.

9.1. Hardware

For the development of this project I'll be using, mainly, a desktop computer and a laptop. They have the following specs:

- **Desktop Computer:** AMD A10-6800K APU 4.10 GHz x2, NVIDIA GeForce GTX 960, 16GB RAM, 3TB SSD.
- **Laptop Computer:** Medion Erazer P6679 - Intel Core i5-7200U 2.50 GHz x2, NVIDIA GeForce GTX 950M, 12GB RAM, 256 SSD.

The costs represented in Table 1 are accounting for an average of 6 daily hours of work for 220 work days a year, using the formula following this paragraph. Because both computers are interchangeable for the task in hand, I'll assume I'll be using both equally: Both the laptop and the desktop taking 50% of the 540 approximate hours of the project duration.

$$Cost = Cost\ computer \times \frac{1\ useful\ life}{4\ years} \times \frac{1\ year}{220\ work\ days} \times \frac{1\ day}{6\ hours\ of\ work} \times Dedicated\ hours$$

Device	Cost	Useful life	Hours	Amortization
Desktop	1200€	4 years	270	61.36€
Laptop	712.59€ [8]	4 years	270	36.44€
Total	-	-	540	97.80€

Table 1: Hardware costs. Self elaborated.

9.2. Software

For this project, most of the software used is open source, and the only paid software, that being Windows 10, has already been accounted on **Section 9.1** of this document, with the price of the computers. The following table encompasses everything:

Product	Cost	Amortization
Gitlab	0€	0€
Dev-C++	0€	0€
Intsat (Barcelogic's Solver)	0€	0€
Windows 10 (x2)	278€	0€
Ubuntu OS	0€	0€
LaTeX	0€	0€
g++	0€	0€

Table 2: Software costs. Self elaborated.

9.3. Human resources

The costs for human resources refers to the different tasks that, in this case, the only developer carries out for this project. I've taken the decision to exclude the directors for this project, because I consider it a cost of consultancy and support contributed by the university for free. In this case, I'll be doing, at times, of project manager, developer and tester (Often referred as Test engineer); and the time spent in each of these tasks, together with the associated costs, can be seen in the following two tables. The costs are estimated supposing salaried workers, supposing that the working hours will not exceed 1800 yearly hours, according with the BOE 2016-2856 [9], and consulting the salaries in Spain on *Indeed* [10] for the following calculations. To compute the cost for the social security, we add the 35% of the gross salary.

Position	Gross salary (€/hour)	Hours	Cost	Cost + Social Security
Developer	17.81 €/hour	365 h	6500.65 €	8775.88 €
Tester	18.40 €/hour	20 h	368.00 €	496.80 €
Project Manager	20.375 €/hour	160 h	3260.00 €	4401.00 €
Total	-	545 h	10128.65 €	13673.68 €

Table 3: Human resources costs per role I am taking. Self elaborated.

Activity	Hours	Human Resources	Cost
Project planning	-	-	-
Writing the project planning	90	Project manager	1833.75 €
Research	-	-	-
Familiarizing with the code	5	Developer	89.05 €
Researching techniques #1 and #2	30	Developer	534.3 €
Development	-	-	-
Developing techniques #1 and #2	330	Developer	5877.3 €
Improvement study	-	-	-
Studying techniques #1 and #2	20	Tester	368 €
Memory	-	-	-
Documenting techniques #1 and #2	70	Project manager	1426.25 €

Table 4: Human resources costs per task. Self elaborated.

9.4. Unforeseen events

In the case of unexpected events in the development of this project, some part of the budget has been allocated, for every position corresponding to human resources, to take into consideration extra hours as an additional cost.

Position	Hours	Cost + SS	Probability	Imputation
Developer	20 h	356.2 €	40%	142.48 €
Tester	5 h	92 €	8%	7.36 €
Project Manager	10 h	203.75 €	30%	61.13 €
Total	-	-	-	210.97 €

Table 5: Unforeseen events for human resources. Self elaborated.

The distribution of additional hours is computed by taking into consideration the magnitude of the impact expected from any produced error for each of the positions, and the time spent during this project as each of them. Because I am working under an agile methodology and the hour assignment is quite lax, not many unforeseen events are expected during the testing phase, but there's a high chance that either the developing or the organization parts of the project have to be extended because of some unforeseen error.

Additionally, we must add a provision for material contingencies, hardware failure, or other problems that may circumstantially occur and are alien to the human resources themselves.

Resource	Unexpected event	Cost	Probability	Imputation
Computer	Disk failure	80 €	5%	4 €
Computer	RAM memory failure	50 €	3%	1.5 €
Computer	Repair service	60 €	10%	6 €
Peripherals	Failure	25 €	5%	1.25 €
Material	Substitution	10 €	50%	5 €
Total	-	-	-	17.75 €

Table 6: Unforeseen events for materials. Self elaborated.

9.5. Indirect costs

Because the indirect costs cannot be computed precisely, an approximation has been attempted to be made to the best of my abilities.

For the cost of electricity, I am counting the cost for electricity as 0.142 €/kWh, and the average power has been assumed to be 225W for a desktop computer, and 100W for a laptop computer [11]. Because both are assumed to be used for 270 hours, that makes a consumption of 60.75 kWh for the desktop, and 27 kWh for the laptop, or 8.627€ and 3.834€ of electricity respectively.

We also add the costs for transport to go to the *Campus Nord*, which is where the office of my directors is found. I move via public transport, the place where I have to go is at 2 zones from my house, and the predicted length for this project is 4 months, therefore I'll need both a *T-Jove*, worth 105.20€, and a *T-Usual*, worth 53.85€, making a total of 159.05€ in total for transport.

Finally, we take into consideration the costs for internet connection, according to the monthly cost of 26.98€ for the length of 4 months, which makes a total of 107.92€.

Source	Cost per unit of time	Time	Cost
Desktop computer	0.03195 €/h	270 h	8.63 €
Laptop computer	0.0142 €/h	270 h	3.83 €
Transport	-	-	159.05 €
Internet	26.98 €/month	4 months	107.92 €
Total	-	-	279.43 €

Table 7: Indirect costs. Self elaborated.

9.6. Contingency

We reserve a part of the money to be able to deal with any unforeseen event above mentioned to guarantee the correct realization of the project, at least regarding economic inconveniences. With this goal in mind we reserve a part of the budget depending on the origin of the cost and its objective.

Source	Price	Percentage	Cost
Human resources	13673.68 €	2%	273.47 €
Unforeseen events	228.72 €	15%	34.31 €
Indirect costs	279.43 €	15%	41.91 €
Total	-	-	349.69 €

Table 8: Contingency plan. Self elaborated.

We're assigning only a 2% of contingency for human resources, since the budget of these unforeseen events has already been taken into consideration, and therefore the impact will be lesser. For the rest, a 15% contingency has been assigned to be able to deal with any unexpected problem in this field.

9.7. Final budget

To round all of this up, we take into consideration every budget item we've considered so far, and we add everything together to get a final budget, and to get a general view of the cost of this project and the distribution in budget of it.

Source	Cost
Hardware	97.80€
Software	0 €
Human resources	13673.68 €
Unforeseen events	228.72 €
Indirect costs	279.43 €
Contingency	349.69 €
Total	14629.32 €

Table 9: Final budget. Self elaborated.

The budget for contingency and unforeseen events, in case that it's not consumed shall be reserved for future projects, in such a way that this money will still be used for research.

9.8. Control management

To correctly manage the budget during the development of the project, a set of indicators has been selected to be able to follow, and take into consideration, the evolution of the project and the difference between the real and estimated cost for each task. To detect deviations from the budget the following formulas will be used as indicators:

$$Cost\ deviation = (C_e - C_r) \cdot T_r$$
$$Efficiency\ deviation = (T_e - T_r) \cdot C_e$$

Where C corresponds with the cost in €/hour, and T is the time in hours. The subscripts r and e differentiate between "reality" and "estimation", respectively.
These indicators can help to measure deviations in the budget, especially in the human resources costs per task, since that is what has the biggest impact on the project.

9.9. Economic viability

This is a research project, and therefore is not made with economic viability in mind. That being said, there's a *fairly small* chance that this research ends up giving results that are way better than expected. If that is the case, this project *might* end up being economically viable by giving Barcelogic an edge on the market of satisfiability solvers. Assuming that is not the case, this project economic viability depends solely on grants from interested parties that make this project possible.

Anyway, the economic viability of this project is strictly linked to a tight control of the budget by computing the difference between the expected and real cost, detecting deviations on the budget and correcting them as soon as possible.

9.10. Update on budget

9.10.1. Hardware

During the development of this project, I used almost exclusively the laptop. Because of that, the total costs have gone down from 97.80€ to 72.88€ - or it would've if the project lasted as predicted, but because it lasted twice as long, the cost goes up twofold to 145.76€.

9.10.2. Software

For verifying correctness I ended up using IMB's CPLEX. This software was granted to me by my directors, so for me it was free, but I feel like we should add the license to the cost. Lucky for us, CPLEX offers 1-year free licenses for academic purposes, so it's still free.

9.10.3. Human resources

No considerable changes in this regard, besides being twice as much because of the increased length of this project.

9.10.4. Unforeseen events

For the unexpected events during the development, no considerable changes in that regard, besides the twofold increase.

For the hardware failure part, nothing happened, so that takes 17.75€ out the budget.

9.10.5. Indirect costs

Because I've solely used the laptop for this project, the electricity costs go down by 4.8€. Also, because of confinement, transport costs have been reduced to zero, taking costs down another 159.05€. Both internet and electricity end up taking twice as much, meaning a final cost of 231.16€.

9.10.6. Contingency

No seriously unforeseen event (besides some mild sickness) has occurred during the development of this project, so we can reduce the contingency costs down to zero.

9.10.7. Final Budget

To sum up

Source	Cost
Hardware	145.76€
Software	0 €
Human resources	27347.36 €
Unforeseen events	421.94 €
Indirect costs	231.16 €
Contingency	0 €
Total	28146.22 €

Table 10: Actual final budget. Self elaborated.

9.10.8. Control management

The cost has gone down, but this project was lengthened by 4 additional months, meaning twice the actual time predicted, meaning:
$C_e = 14629.32/545 = 26.84e/hour$
$C_r = 28146.22/1090 = 25.82e/hour$
$T_e = 545\ hours$
$T_r = 1090\ hours$
$Cost\ deviation = (C_e - C_r) \cdot T_r = (26.84 - 25.82) \cdot 1090 = 1111.8e$
$Efficiency\ deviation = (T_e - T_r) \cdot C_e = (545 - 1090) \cdot 26.84 = -14627.8e$
Meaning that the cost has gone down, but so has done the efficiency, and by a lot.

10. Sustainability report

Sustainability is not a topic I think of often, not even close; but it's inherent to any activity, project, concept, product... And especially everything that involves ICT resources, which are found everywhere in this day and age of information.

I find of utmost importance to learn and be cautious about our environment to make this world a better place, to be critical with everything you do, and to try and do things right. It's true that I've never been taught the techniques, concepts or indicators for sustainability, and therefore this analysis will be done with the intent to learn and improve upon myself. By this I mean that this analysis will probably not be close of what an expert on the topic might be able to carry out, but I'll try my best.

10.1. Sustainability dimensions

10.1.1. Economic dimension

The economic cost needed to develop this project has been estimated to be 14629.323 €, as stated in the **Section 9.7** of this document.

Of the budget, human resources (13673.678€) are the biggest contributors to this final result, while hardware, indirect costs and contingency only make a small part. All these costs are, in one way or the other, inevitable, although the human resources ones could end up being less if this project is finished in less time than what's been predicted previously.

On the topic of the useful life of the project, it will be useful for as long as these techniques don't become obsolete by some other techniques. Perhaps, in the future, the state-of-the-art of preprocessing techniques evolves in such a way that these techniques are seen completely useless, or perhaps a paradigmes shift in how we solve these kinds of problems make them obsolete. In any way, it's all tied on how fast the field of satisfiability solvers or integer linear programming solvers evolves.

If the results are positive, and I have a voice on the matter, I would like to publish the results for free, with details of the implementation, to further improve the general knowledge on this topic. This would give state-of-the-art solvers a new set of tools that would allow them to be faster and more efficient, therefore reducing time costs to the users.

10.1.2. Environmental dimension

Both computers, together, consume a total of 87.75 kWh, as we've concluded in **Section 9.5** of this document. Converting that to CO_2 emissions by multiplying the result by 0.23 according to the MITECO's report on 2018's Factor Mix for EDP [12], resulting in a total of 20.18 kg of CO_2.

Also, I am using public transport to go from Sabadell to Barcelona, producing approximately 0.78 kg of CO_2 for each travel according to *EcoPassenger* [13]. I have one meeting per week, each one requiring two travels, meaning a total of 1.56 kg of CO_2 per week, or approximately 24.96 kg of CO_2 for the whole project.

All of this would make a total CO_2 footprint of 45.14 kg of CO_2 for the whole project.

I could reduce the carbon footprint by using more the laptop, and less the desktop computer, allowing me to reduce the footprint from 45.15 kg to a minimum of 37.38 kg. The transport costs are considered to be unavoidable, unless I am able to schedule virtual meetings. That wouldn't help that much, though, since I still need to take public transport to attend class, so even if I could detach these costs from the project, I would still be emitting the same amount of CO_2.

For the useful life of my project, it might help reduce future carbon footprint by allowing other solvers to be faster, therefore taking less time, and therefore less energy. It's very difficult to quantify how good this improvement might be, though, since it depends on how good the techniques studied in here are with respect to the already established techniques, for how long it's useful, by how many solvers implement these techniques, and by how many people end up using those solvers.

10.1.3. Social dimension

Thanks to this project I am learning - and will learn - things about satisfiability, logic in informatics, Operations Research, optimization, and many other topics that I might have never learned about if it weren't for this project. This project dances in the fine line between efficient and non-efficient algorithms that is the NP-Complete class, and therefore there's still a lot to learn.

In the social field, this project might not only be good for improving the knowledge on preprocessing techniques for these algorithms, but also it would help to point out the flaws and limitations of the current methods for solving these algorithms, which are why they can benefit of these preprocessing techniques in the first place.

Regarding the negative social aspects of this project, I can't really see any big, glaring problem. The only problem I am able to see is that, if these techniques seem *too* good, they might allow us to break some problems that, before, we though unbreakable. Like, for example, assume that these techniques are *so good* they allow us to make programs that factorize a number into its prime factors efficiently. Then, the RSA algorithm for cryptography would be compromised, since its main assumption is that that problem cannot be solved efficiently [14]. Of course, I just say this for completion, because I find it *highly unlikely* (if not impossible) that these techniques can be this good.

10.2. Sustainability matrix

To summarize, the following matrix is a synthesis of everything said so far in the topic of sustainability.

	PPP	Useful life	Risks
Economics	14629.32 €	No profits	Not being viable
Environment	45.15 kg CO_2	+ Efficiency	Could not improve
Social	8/10	6/10	Realistically, none

Table 11: Sustainability matrix. Self elaborated.

10.3. Update on sustainability

Now, at the end of this project, let's go back on the topic of sustainability to study the differences between prediction and reality.

10.3.1. Economic dimension

As shown in **Section 9.10.7** of this document, the economic cost has gone up from 13673.68€ to 28146.22€.

10.3.2. Environmental dimension

Because of confinement, I didn't use public transport, therefore reducing the carbon footprint by 24.96 kg of CO_2.

Because I have only used my laptop for this project, the carbon footprint from electricity would have reduced from 20.18 to 12.41 kg of CO_2 if the project lasted as expected, but because it was twice as long the total amount of CO_2 is multiplied by 2, to a total of 24.82 kg. That is, still, almost half of the original prediction of 45.15 kg.

10.3.3. Social dimension

This dimension is one I find somewhat iffy. On one side, I *for sure* didn't break RSA with this project, as expected. If anything, the data raises questions on the efficacy of these techniques, while leaving lots of things unanswered. These methods seem to, sometimes, improve the efficiency of the algorithm, while sometimes doing nothing, or even worse than nothing. Further research is required to study the efficacy of these methods in finer detail, but they surely don't seem game-breaking in any way.

10.3.4. Sustainability matrix

To summarize, the following matrix is a synthesis of everything said so far in the topic of sustainability.

	PPP	Useful life	Risks
Economics	28146.22 €	No profits	Not being viable
Environment	24.96 kg CO_2	+ Efficiency	Could not improve
Social	8/10	6/10	None

Table 12: Final sustainability matrix. Self elaborated.

11. Conclusions

11.1. Technical Skills

- **CCO1.1 Assess the computational complexity of a problem, know algorithmic strategies that could lead to the solution, and recommend, develop and implement the one that grants the best performance according to the established requirements.**

  This project revolves around the P vs NP fringe, trying to more efficiently solve NP Complete and CO-NP complete problems.

- **CCO1.2: Display knowledge of the theoretical fundamentals of programming languages and the techniques for lexical, syntactic and semantic processing associated with such, and knowing how to apply them in the creation, design and processing of languages.**

  Although it is not a primary objective for this project, I had to develop a parser for the .lp format for Mixed Integer Programming while working on the Local Search method.

- **CCO1.3: Defining, evaluating and selecting hardware and software development and production platforms for the development of applications and computer services of diverse complexity.**

For this project, I used g++ as the compiler because it's free, and it's better suited to compile in ARM than other compilers like LLVM, which is important taking into consideration that this project is thought of for cloud computing, and the servers that offer this kind of service are often ARMs.

- **CCO2.1: Showing knowledge of the fundamentals, the paradigms and the techniques of smart systems, and analyzing, design and make system, services and computer applications that use these techniques in any area of application.**

  Smart systems often need to solve NP-complete problems internally, for which they can benefit from using ILP and SAT solvers.[18]

- **CCO2.2: The ability to acquire, obtain, formalize and represent the human knowledge in a computable way for the resolution of problems through a computer system in any field of application, particularly in those in which there are related aspects of computation, perception and action in smart environments.**

  SAT, Linear Programming, and every related problem, all focus on the automated resolution of problems through the formalization of data from the real world.

- **CCO2.4: Displaying knowledge and developing techniques of computational learning; Design and implement applications that use them, including those dedicated to the automatic extraction of information and knowledge in large volumes of data.**

  The first method of Strengthening infers additional knowledge (the new constraints) out of the original constraints.

- **CCO3.1: Implementing critical code following running time, efficiency and safety criteria.**

  The methods chosen were all chosen because of their reported running time costs, and studies were conducted to study the efficiency and efficacy of such methods.

- **CCO3.2: Programming taking into consideration the hardware architecture, both in assembly and in high level.**

  Decisions were made for better placement of data into the cache, to reduce overhead in that regard.

11.2. Future Work

This project is yet to be completely settled. There are many things still to be done for this project to be done. These things include, but are not limited to:

- Fixing the Strengthening method.
- Recollecting more data to better study the improvement of the methods.
- Implementing the particular hash function described in the paper for the local search.
- Integrating both methods with Intsat using Redis.
- Implementing parallelism.

11.3. Special thanks

I'd like to thank everyone that is been with me during the long development of this project.

First and foremost, I'd like to thank my directors Robert Nieuwenhuis and Enric Carbonell for their help, support, and for wanting to do this project with me in the first place.

I'd also like to thank my family that has helped me all throughout this project in every way they could, and gave me the emotional support I needed.

Finally, I'd like to thank my friends for having to deal with me being a bore about this project. Special thanks to Pol Martin, Antoni Casas and Adrian Tormos for being kind enough to help me proofread this document.

12. References

[1] Biere, A., Heule, M., Van Maaren, H., Walsh, T. *Handbook of Satisfiability*, 2009. ISBN-10: 1586039296

[2] H. Dixon *Automating pseudo-boolean Inference within a DPLL framework*, Ph.D. Dissertation, University of Oregon, December 2004

[3] Randal E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, August 1986

[4] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, Valentin Mayer-Eichberger: *A New Look at BDDs for Pseudo-Boolean Constraints*. J. Artif. Intell. Res. 45: 443-480 (2012)

[5] Armin Biere, Daniel Le Berre, Emmanuel Lonca, Norbert Manthey: *Detecting Cardinality Constraints in CNF. SAT* 2014: 285-301

[6] El Periódico, *Un software creado por la UPC verificará el sorteo del Mundial de fútbol*, 2017

[7] Barcelogic - Efficiency, simplified. https://barcelogic.com/en/

[8] Medion Erazer P6679 Intel Core i5-7200U/12GB/256GB SSD/GTX950M/15.6" in PcComponentes

[9] Ministerio de Empleo y Seguridad Social, Gobierno de España, *Disposición 2856 del BOE núm. 70 de 2016*, article 22

[10] Indeed.es, *Comparación de sueldos, buscar sueldos*

[11] Energuide, *How much power does a computer use? And how much CO2 does that represent?*

[12] MITECO, *Información completa para la toma de decisiones sobre la sección de proyectos de absorción de dióxido de carbono y la sección de compensación del registro*, April 2019 [Pag. 30]

[13] HaCon Ingenieurgesellschaft mbH., *EcoPassenger*

[14] RSA Laboratories, *The RSA factoring challenge* (archived)

[15] MIPLIB 2017 – The Mixed Integer Programming Library

[16] vector bool - C++ Reference

[17] D. Bertsimas et al., *A New Local Search Algorithm for Binary Optimization*, INFORMS Journal on Computing, 11 April 2012

[18] Kłosowski, Grzegorz & Kozłowski, Edward Gola, Arkadiusz. *Integer Linear Programming in Optimization of Waste After Cutting in the Furniture Manufacturing*. 260270. 10.1007978-3319644653_26. International Conference on Intelligent Systems in Production Engineering and Maintenance (2018)

[19] Stephen A. Cook, *The complexity of theorem-proving procedures*. STOC '71: Proceedings of the third annual ACM symposium on Theory of computing, May 1971

[20] fast0507 details - miplib.zib.de

[21] seymour details - miplib.zib.de

A. Boolean Satisfiability Problem

The boolean satisfiability problem (SAT) is a famous problem in computer science. The statement is the following:

Given a boolean function $f : \{0,1\}^n \to \{0,1\}$ expressed in the language of propositional logic, is there some input $a \in \{0,1\}^n$ such that $f(a) = 1$?

This is a well-known NP-Complete problem, and it has been studied and optimized a lot, because of its versatility and because of the shear amount of problems that can be encoded with it. Often, instead of a general formula in propositional logic, the input problem is a formula in Conjunctive Normal Form.

In this document, in **Section 1.1**, I've already explained both what CNF is and a common algorithm for solving the CNF-SAT problem (DPLL).

A.1. Unit propagation

A problem in CNF is expressed as a conjunction of clauses $(c_1 \wedge ... \wedge c_m)$, each clause c_i is a disjunction of literals $(l_{i,1} \vee ... \vee l_{i,k})$, and each literal is either a variable or the negation of a variable (x or $\bar{x}$).

If a clause c_i only contains a single literal $l_{i,1}$, we say this clause is an unitary clause, or unit for short. If the formula is satisfiable, a unit gives the mandatory value for the variable the literal represents. The unit x means x is equal to one, and the unit $\bar{x}$ means it's equal to zero.

If a different clause c_j contains the literal $l_{i,1}$, that clause is automatically true because of the rules of propositional logic, and therefore can be disregarded and erased without affecting the truth table of the formula.

If, instead, c_j contains the literal $\bar{l_{i,1}}$, we can simplify the clause down via resolution, turning $c_j = \bar{l_{i,1}} \vee C$ into $c_j = C$. If C contained only a single literal $l_{j,1}$, c_j turns into a new unit.

The process of repeating this procedure of disregarding clauses that contain the literals that are also present in units, and simplifying down clauses via resolution using these same units is called unit propagation.

Unit propagation is a very powerful tool because of its simplicity and efficiency. This is important because to implement DPLL in Binary Linear Programming, we will need an equivalent to this method for it.

For an example on unit propagation, let a CNF be made of the following clauses:

1. x_1
2. $\bar{x_1} \vee x_2$
3. $x_1 \vee \bar{x_2}$
4. $\bar{x_1} \vee \bar{x_3}$
5. $\bar{x_2} \vee x_3 \vee x_4$

Clause 1 is a unitary clause, meaning we can do unit propagation with x_1. In *Clause 2* $\bar{x_1}$ is present, meaning we can simplify down the clause by resolution to x_2. In *Clause 3* x_1 is present, meaning we can disregard this clause. In *Clause 4* $\bar{x_1}$ is present again, so we can also simplify. The variable x_1 is nowhere to be found in *Clause 5*, meaning we can do nothing. Now the clauses look like this:

1. x_1
2. x_2
3. $\bar{x_3}$
4. $\bar{x_2} \vee x_3 \vee x_4$

Now *Clause 2* is also a unitary clause. *Clause 4* is the only clause besides *Clause 2* that has the variable x_2, meaning we can simplify:

1. x_1
2. x_2
3. $\bar{x_3}$
4. $x_3 \vee x_4$

Clause 3 is also a unitary clause, meaning we can simplify down *Clause 4* even further.

1. x_1
2. x_2
3. $\bar{x_3}$
4. x_4

Clause 4 is also a unit, but there's nothing else to do, therefore the procedure of unit propagation has ended.

B. Linear Programming

Linear Programming is an optimization problem defined the following way:

Given a linear function $f : \mathbb{R}^n \to \mathbb{R}$ and a set of linear constraints $\{c_1, ..., c_m\}$, such that $c_j = \sum a_{i,j} x_i \leq k_j$, is there some solution $x \in \mathbb{R}^n$ such that it meets every constraint c_j? And, if there is, what is the solution x with the maximum value for $f(x)$ that meets all constraints?

Just like SAT, this problem is also very well studied because a lot of problems in real life can be reduced to a linear programming problem. The most common algorithms for solving LP are Simplex and Karmarkar's Inner Point Method, the latter being a polynomial time algorithm. How these two algorithms work, although fascinating, is not really important for this dissertation.

C. Integer/Pseudoboolean linear programming

Integer linear programming is the same as Linear programming, with the additional restriction that the solution x must be in $\mathbb{Z}^n$. This additional constraint makes the problem way more difficult, to the point where no polynomial time algorithm is known to solve this problem (and is most often thought that it's impossible). The most common method for solving this kind of problems is to ignore the integer constraints, solve the problem as a normal LP problem, and then with this additional information adding additional constraints, called "cutting planes" to try and find the best integer solution.

Pseudoboolean linear programming (Or Binary linear programming) is a subset of problems in Integer linear programming, where you limit the solution to be in $\{0,1\}^n$.

You can reduce CNF SAT to Pseudoboolean linear programming easily, by turning each clause $a_1 \vee ... \vee a_k$ into the constraint $a_1 + ... + a_k \geq 1$, and then optimizing by any linear function, meaning that Pseudoboolean linear programing and, therefore, also Integer linear programming are NP-Hard.

It's important to note that, whether or not a set of linear constraints in an integer/pseudo-boolean linear programming are feasible is an NP Complete problem, but saying whether or not a particular feasible solution is the optimum is a CO-NP Complete problem.

C.1. Unit propagation for Pseudoboolean Linear Programming

In this dissertation is often mentioned that you can use DPLL to solve BLP, but that would require BLP to have some method for unit propagation, but it's not trivially obvious how you'd implement such a method when the formula is defined with linear constraints instead of with clauses, making the definition for unit propagation used in **Annex A.1** not ideal.

First, let's define what an "unit" is in this context, since we're working with constraints, not with clauses. In DPLL, variables take one of three possible states: true, false, and undefined. At the beginning of the execution every variable is set to undefined, and via decisions, unit propagation and backtracking we deduce the state of every variable until no variable has an undefined state, or until the method fails because a contradiction in the formula was found. In every step of the execution, we can say that the "units" are defined by the states of the non-undefined variables. That is, if a certain variable x is true, we say we have the unit x, or if x is false instead, we would have the unit $\bar{x}$.

Now, for a given constraint $c_i = \sum a_i x_i \geq k$, we can say this constraint is impossible to meet if the sum of all positive coefficients a_i adds up to less than k. What we can do is plug the values for all the units we have at the moment inside the formula, to be left with a new constraint c_i' with less coefficients. From this constraint, we say that we can "unit-propagate" a literal if, by supposing this literal is false, we make this constraint impossible to be met, therefore concluding that this literal must be true.

Again, unit propagation is not the action of doing this once, but instead, is the action of repeatedly applying this method from an initial given set of units until nothing can be done anymore.

D. Proof of correctness for Strengthening

Given the literal l, the constraint set C and the constraint $c = \sum a_i x_i \geq k$, $c \in C$:

Let $P = UNIT - PROPAGATE(C, \{l\})$, $current(c, P)$ ="c's slack with the literals satisfied under P", and $c' = \sum a_i x_i + current(c, P)\bar{l} \geq k + current(c, P)$, assuming $current(c, P) > 0$

1. Proof that $C \wedge l \vDash c'$:
Let ρ be any full assignment that satisfies $C \wedge l$. Because l is true, $P \subseteq \rho$ by unit propagation in C. And because c is oversatisfied by an amount of $current(c, P)$ when the partial assignment P is set, it will be oversatisfied by an amount of at least that much after the full assignment[3]. Meanwhile, c' simplifies down to $\sum a_i x_i \geq k + current(c, P)$, which is the same as c with an additional $current(c, P)$ in the constant term, which is the same by which c is over satisfied, meaning that c' is also satisfied.

2. Proof that $C \wedge \bar{l} \vDash c'$:
This is trivially true, since if $\bar{l}$ is true, the constraint c' simplifies down to $\sum a_i x_i \geq k$, which is the constraint c, and because $c \in C$, and constraint logic is constructive, $C \vDash c$, and therefore $C \vDash c'$.

3. Corollary from 1 and 2: Because both $C \wedge l$ and $C \wedge \bar{l}$ imply the constraint c', $C \vDash c'$.

4. Proof that $C \setminus \{c\} \cup \{c'\} \wedge l \vDash c$:
If l is true, c' simplifies down to $\sum a_i x_i \geq k + current(c, P)$.
For every full assignment that satisfies this constraint, is trivially true that it will also satisfy c, therefore, $c' \wedge l \vDash c$ and, therefore, $C \setminus \{c\} \cup \{c'\} \wedge l \vDash c$

5. Proof that $C \setminus \{c\} \cup \{c'\} \wedge \bar{l} \vDash c$:
Again, c' simplifies down to c when $\bar{l}$ is true, and therefore this is trivially true.

6. Corollary from 4 and 5: $C \setminus \{c\} \cup \{c'\} \vDash c$

Because of corollaries 3 and 6, we come to the conclusion that, because c' can be deduced out of C, and c can be deduced out of C, without c but with c', c and c' are logically interchangeable in this context, and therefore you can change one by the other without changing the logical equivalence of the problem.

□

Note that this proof only applies to one-literal strengthening. For multiple-literal strengthening, 3 is still true, but 6 is not, and therefore you cannot replace the constrain when strengthening with 2 or more variables.

E. Experimental data

In this section of the document we find some additional experimental data on the methods described in this dissertation. The decision for leaving this data at the end of the document was

[3] Dixon's Strengthening assumes - and needs for the method to work - that no negative coefficients are found in the constraints. This can be done because any literal l is equal to $1 - \bar{l}$, so if some coefficient is negative in some constraint, you can always turn the coefficient into a positive one by using this substitution. This is why no matter the new literals in the assignment, it won't make the evaluation of the expression go down.

so that it wouldn't hinder the reading of this document.

E.1. Strengthening

The study of the strengthening method was cut short due to problems previously described. Still, many tests were carried out, and some data was collected. On a lot of cases, no constraint was strengthened. The following table is an extract of cases in which the algorithm actually was able to strengthen something.

problemName	Opt. w/o str.	Time w/o str	Opt. with str	Time with str	str. ctrns	Time spent str.
$j1201_7 - sat.lp$	0	5.19s	0	23.84s	12	17.897s
$j12021_1 - unsat.lp$	None	13.17s	None	30.55s	22	13.01s
$j12042_1 - unsat.lp$	None	35.82s	None	65.36s	9	18.264s
$j3010_7.std.lp$	49	458.87s	49	658.24s	1021	55.259s
$j3013_4.std.lp$	72	415.58s	72	467.41s	32	44.905s
$j3015_1.std.lp$	46	525.98s	46	773.38s	1016	50.179s
$j3016_2.std.lp$	48	513.35s	48	761.91s	1008	41.512s
$j3022_2.std.lp$	45	371.66s	45	979.61s	5627	72.395s
$j3025_4 - sat.lp$	0	0.89s	0	1.82s	24	0.966s
$j3025_4 - unsat.lp$	None	1.04s	None	2.20s	24	0.929s
$j3026_7.std.lp$	56	381.33s	56	553.99s	1028	46.422s
$j3026_9 - sat.lp$	0	0.09s	0	0.07s	7	0.007s
$j3029_9 - unsat.lp$	None	3.33s	None	5.32s	6	1.859s
$j3032_1.std.lp$	61	420.69s	61	628.60s	975	37.336s
$j3032_9.std.lp$	65	423.33s	65	616.04s	933	36.754s
$j3034_7.std.lp$	58	321.06s	58	361.53s	9	37.829s
$j3034_9.std.lp$	60	304.01s	60	407.20s	991	50.927s
$j3035_6.std.lp$	58	323.39s	58	1019.43s	11120	102.662s
$j3038_9.std.lp$	63	360.29s	63	1477.28s	10656	112.26s
$j3041_7.std.lp$	92	313.05s	92	861.10s	4825	74.025s
$j3044_5.std.lp$	55	458.85s	55	665.94s	994	51.719s
$j3045_1.std.lp$	82	363.71s	82	431.22s	43	48.428s
$j3045_6.std.lp$	129	196.23s	129	318.16s	1044	57.189s
$j3045_7.std.lp$	101	249.58s	101	294.32s	19	41.783s

Table 13: Multiple runs of Intsat solving a selection of problems with and without strengthening. Self elaborated.

Unfortunately, the data collected was not very promising. Almost every time strengthening was applied, Intsat actually took *longer* to solve the problem in question, even if we disregard the preprocessing time it took the algorithm to apply strengthening. This was the reason we decided to output the strengthened problem into an external file, to further research this problem, but then the issues described in **Section 8.7** happened, and the implementation of the method was halted.

E.2. Local Search

The paper for local search granted a certain level of efficacy on problems of the category "Unweighted Maximum w-Set Packing". Unweighted Maximum w-Set Packing is a kind of problem in Boolean Linear Programming in which the constraint matrix $A \in \{0,1\}^{m \times n}$ has exactly w "1"s in each column, and the function to optimize $c^T x$ is a maximization function, and has a c equal to all ones. The paper gives an expression $(w^2/(2w-1))$ for the maximum value z^*/zh

can take when the hyperparameter Q takes a value of w^2. Consecutive runs of the code worked better than described, often finding the optimal right away.

problemName	Q	opt.	tCPLEX	iniCost	LSCost	percent	tLS	z*/zh	expr	ok?
n-100-m-32-w-2-1.lp	4	16	0.01	0	16	100	73.753	1	1.333	yes
n-100-m-32-w-2-2.lp	4	16	0.02	0	16	100	98.106	1	1.333	yes
n-100-m-32-w-2-3.lp	4	16	0.01	0	16	100	98.669	1	1.333	yes
n-100-m-32-w-2-4.lp	4	16	0	0	16	100	99.223	1	1.333	yes
n-100-m-32-w-2-5.lp	4	16	0.02	0	16	100	98.837	1	1.333	yes
n-100-m-32-w-2-6.lp	4	16	0.02	0	16	100	97.493	1	1.333	yes
n-100-m-32-w-2-7.lp	4	16	0.01	0	16	100	97.52	1	1.333	yes
n-100-m-32-w-2-8.lp	4	16	0.02	0	16	100	80.014	1	1.333	yes
n-100-m-32-w-2-9.lp	4	16	0.02	0	16	100	73.149	1	1.333	yes
n-100-m-32-w-2-10.lp	4	16	0.02	0	16	100	72.797	1	1.333	yes
n-100-m-32-w-2-11.lp	4	16	0.01	0	16	100	73.104	1	1.333	yes
n-100-m-32-w-2-12.lp	4	16	0.01	0	16	100	72.721	1	1.333	yes
n-100-m-32-w-2-13.lp	4	16	0.05	0	16	100	99.463	1	1.333	yes
n-100-m-32-w-2-14.lp	4	16	0.03	0	16	100	99.836	1	1.333	yes
n-100-m-32-w-2-15.lp	4	16	0.09	0	16	100	98.876	1	1.333	yes
n-100-m-32-w-2-16.lp	4	16	0.05	0	16	100	100.225	1	1.333	yes
n-100-m-32-w-2-17.lp	4	16	0.03	0	16	100	98.633	1	1.333	yes
n-100-m-32-w-2-18.lp	4	16	0.02	0	16	100	99.535	1	1.333	yes
n-100-m-32-w-2-19.lp	4	16	0.02	0	16	100	98.635	1	1.333	yes
n-100-m-32-w-2-20.lp	4	16	0.02	0	16	100	96.849	1	1.333	yes
n-100-m-32-w-2-21.lp	4	16	0.01	0	16	100	97.431	1	1.333	yes
n-100-m-32-w-2-22.lp	4	16	0.03	0	16	100	98.23	1	1.333	yes
n-100-m-32-w-2-23.lp	4	16	0.02	0	16	100	97.618	1	1.333	yes
n-100-m-32-w-2-24.lp	4	16	0.03	0	16	100	115.641	1	1.333	yes
n-100-m-32-w-2-25.lp	4	16	0.03	0	16	100	119.275	1	1.333	yes
n-100-m-32-w-2-26.lp	4	16	0.05	0	16	100	127.685	1	1.333	yes
n-100-m-32-w-2-27.lp	4	16	0.03	0	16	100	116.576	1	1.333	yes
n-100-m-32-w-2-28.lp	4	16	0.05	0	16	100	116.28	1	1.333	yes
n-100-m-32-w-2-29.lp	4	16	0.03	0	16	100	114.292	1	1.333	yes
n-100-m-32-w-2-30.lp	4	16	0.03	0	16	100	112.573	1	1.333	yes
n-100-m-32-w-2-31.lp	4	16	0.03	0	16	100	117.442	1	1.333	yes
n-100-m-32-w-2-32.lp	4	16	0.03	0	16	100	112.163	1	1.333	yes
n-100-m-32-w-2-33.lp	4	16	0.05	0	16	100	112.447	1	1.333	yes
n-100-m-32-w-2-34.lp	4	16	0.03	0	16	100	113.849	1	1.333	yes
n-100-m-32-w-2-35.lp	4	16	0.05	0	16	100	112.351	1	1.333	yes
n-100-m-32-w-2-36.lp	4	16	0	0	16	100	92.49	1	1.333	yes
n-100-m-32-w-2-37.lp	4	16	0.03	0	16	100	92.063	1	1.333	yes
n-100-m-32-w-2-38.lp	4	16	0.02	0	16	100	91.594	1	1.333	yes
n-100-m-32-w-2-39.lp	4	16	0.02	0	16	100	100.936	1	1.333	yes
n-100-m-32-w-2-40.lp	4	16	0.02	0	16	100	92.694	1	1.333	yes
n-100-m-32-w-2-41.lp	4	16	0.02	0	16	100	92.207	1	1.333	yes
n-100-m-32-w-2-42.lp	4	16	0.02	0	16	100	99.471	1	1.333	yes
n-100-m-32-w-2-43.lp	4	16	0.02	0	16	100	100.05	1	1.333	yes
n-100-m-32-w-2-44.lp	4	16	0.02	0	16	100	99.893	1	1.333	yes
n-100-m-32-w-2-45.lp	4	16	0	0	16	100	99.511	1	1.333	yes

Table 14: Multiple runs of local search solving autogenerated Unweighted Maximum 2-Set Packing with the optimal recomended Q. Self elaborated.

More runs were done, all confirming the claim. Also, additional runs were done with suboptimal values for Q, to verify whether or not it made a difference. Evidently, a lower Q gave worse

results, but took a fraction of the time to find these suboptimal solutions.

problemName	Q	opt.	timeCPLEX	iniCost	LSCost	percent	timeLS
n-100-m-32-w-2-1.lp	1	16	0.02	0	14	87.5	0.006
n-100-m-32-w-2-1.lp	2	16	0.02	0	14	87.5	0.034
n-100-m-32-w-2-1.lp	3	16	0.02	0	14	87.5	0.018
n-100-m-32-w-2-2.lp	1	16	0.05	0	14	87.5	0.005
n-100-m-32-w-2-2.lp	2	16	0.02	0	14	87.5	0.018
n-100-m-32-w-2-2.lp	3	16	0.02	0	14	87.5	0.046
n-100-m-32-w-2-3.lp	1	16	0.02	0	14	87.5	0.011
n-100-m-32-w-2-3.lp	2	16	0.02	0	14	87.5	0.021
n-100-m-32-w-2-3.lp	3	16	0.01	0	14	87.5	0.024
n-100-m-32-w-2-4.lp	1	16	0.02	0	14	87.5	0.01
n-100-m-32-w-2-4.lp	2	16	0.02	0	14	87.5	0.035
n-100-m-32-w-2-4.lp	3	16	0.01	0	14	87.5	0.019
n-100-m-32-w-2-5.lp	1	16	0.01	0	14	87.5	0.006
n-100-m-32-w-2-5.lp	2	16	0.03	0	14	87.5	0.033
n-100-m-32-w-2-5.lp	3	16	0.02	0	14	87.5	0.026
n-100-m-32-w-2-6.lp	1	16	0.00	0	14	87.5	0.01
n-100-m-32-w-2-6.lp	2	16	0.02	0	14	87.5	0.022
n-100-m-32-w-2-6.lp	3	16	0.02	0	14	87.5	0.018
n-100-m-32-w-2-7.lp	1	16	0.01	0	14	87.5	0.005
n-100-m-32-w-2-7.lp	2	16	0.02	0	14	87.5	0.021
n-100-m-32-w-2-7.lp	3	16	0.01	0	14	87.5	0.022
n-100-m-32-w-2-8.lp	1	16	0.03	0	14	87.5	0.007
n-100-m-32-w-2-8.lp	2	16	0.03	0	14	87.5	0.015
n-100-m-32-w-2-8.lp	3	16	0.03	0	14	87.5	0.02
n-100-m-32-w-2-9.lp	1	16	0.08	0	14	87.5	0.005
n-100-m-32-w-2-9.lp	2	16	0.01	0	14	87.5	0.026
n-100-m-32-w-2-9.lp	3	16	0.02	0	14	87.5	0.025
n-100-m-32-w-2-10.lp	1	16	0.02	0	14	87.5	0.012
n-100-m-32-w-2-10.lp	2	16	0.02	0	14	87.5	0.02
n-100-m-32-w-2-10.lp	3	16	0.03	0	14	87.5	0.016
n-100-m-32-w-2-11.lp	1	16	0.03	0	14	87.5	0.005
n-100-m-32-w-2-11.lp	2	16	0.02	0	14	87.5	0.016
n-100-m-32-w-2-11.lp	3	16	0.03	0	14	87.5	0.016
n-100-m-32-w-2-12.lp	1	16	0.01	0	14	87.5	0.006
n-100-m-32-w-2-12.lp	2	16	0.01	0	14	87.5	0.018
n-100-m-32-w-2-12.lp	3	16	0.00	0	14	87.5	0.015
n-100-m-32-w-2-13.lp	1	16	0.03	0	15	93.75	0.006
n-100-m-32-w-2-13.lp	2	16	0.06	0	15	93.75	0.021
n-100-m-32-w-2-13.lp	3	16	0.06	0	15	93.75	0.016

Table 15: Multiple runs of local search solving autogenerated Unweighted Maximum 2-Set Packing with suboptimal Q when compared to the recommendation. Self elaborated.

This is probably a quirk of the generated test cases, but it happens that every run of the algorithm for this batch of tests gave results such that z^*/zh was below the theoretical maximum of $w^2/(2w-1)$ for an optimal Q, even though the Q was suboptimal. This is generally not the case.

For not-so well behaved cases, the algorithm gave very lackluster results, often finding no better solution than the originally given one.

problemName	Q	opt.	timeCPLEX	iniCost	LSCost	percent	timeLS
air04.lp	1	0	9.38	57894	57894	0	3.464
air04.lp	2	0	9.38	57894	57894	0	2.978
air04.lp	5	0	9.38	57894	57894	0	411.153
air04.lp	10	0	9.38	57894	57894	0	600.48
air04.lp	50	0	9.38	57894	57894	0	603.253
mitre.lp	1	115155	0.77	115965	115965	0	1.361
mitre.lp	2	115155	0.77	115965	115965	0	1.35
mitre.lp	5	115155	0.77	115965	115965	0	1.337
mitre.lp	10	115155	0.77	115965	115965	0	1.835
mitre.lp	50	115155	0.77	115965	115965	0	1.394
p0201.lp	1	7615	0.33	10365	10365	0	0.001
p0201.lp	2	7615	0.33	10365	10365	0	0.001
p0201.lp	5	7615	0.33	10365	10365	0	0.001
p0201.lp	10	7615	0.33	10365	10365	0	0.02
p0201.lp	50	7615	0.33	10365	9475	32.3636	600.005
p0201.lp	1	7615	0.33	10315	10315	0	0.001
p0201.lp	2	7615	0.33	10315	10315	0	0.001
p0201.lp	5	7615	0.33	10315	10315	0	0.002
p0201.lp	10	7615	0.33	10315	10315	0	0.019
p0201.lp	50	7615	0.33	10315	9475	31.1111	600.001
p0201.lp	1	7615	0.33	8490	8490	0	0.001
p0201.lp	2	7615	0.33	8490	8490	0	0.001
p0201.lp	5	7615	0.33	8490	8490	0	0.001
p0201.lp	10	7615	0.33	8490	8490	0	0.022
p0201.lp	50	7615	0.33	8490	8390	11.4285	600.001
p2756.lp	1	3124	0.52	16684	16684	0	0.065
p2756.lp	2	3124	0.52	16684	16684	0	0.085
p2756.lp	5	3124	0.52	16684	16684	0	0.07
p2756.lp	10	3124	0.52	16684	16684	0	0.314
p2756.lp	50	3124	0.52	16684	16684	0	600.003
p2756.lp	1	3124	0.52	3872	3872	0	0.11
p2756.lp	2	3124	0.52	3872	3872	0	0.123
p2756.lp	5	3124	0.52	3872	3872	0	0.092
p2756.lp	10	3124	0.52	3872	3872	0	0.335
p2756.lp	50	3124	0.52	3872	3872	0	600.006

Table 16: Multiple runs of local search solving a small selection of problems in MIPLIB. If the same problem is shown twice with the same values for Q means that we're testing execution with a different initial solution. Self elaborated.

For these not-so well behaved cases, whether the algorithm finds a better solution than the original is pretty arbitrary, often requiring large values for Q and long periods of time, and sometimes even that is not enough.

There were other relatively well-behaved examples, like the aforementioned *fast0507.lp* and *seymour.lp* showcased in **Section 4.2.4** of this document.