DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2017

Study of efficient techniques for implementing a Pseudo-Boolean solver based on cutting planes

ALEIX SACREST GASCON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Undersökning av effektiva tekniker för implementering av en pseudo-boolsk lösare med hjälp av skärande plan

ALEIX SACREST GASCON

Degree Project in Computer Science, DD142X
Supervisor: Dilian Gurov
Examiner: Örjan Ekeberg

School of Computer Science and Communication
KTH Royal Institute of Technology

June 2017

Abstract

Most modern SAT solvers are based on resolution and CNF representation. The performance of these has improved a great deal in the past decades. But still they have some drawbacks such as the slow efficiency in solving some compact formulas e.g. *Pigeonhole Principle* [1] or the large number of clauses required for representing some SAT instances.

Linear Pseudo-Boolean inequalities using cutting planes as resolution step is another popular configuration for SAT solvers. These solvers have a more compact representation of a SAT formula, which makes them also able to solve some instances such as the Pigeonhole Principle easily. However, they are outperformed by clausal solvers in most cases.

This thesis does a research in the CDCL scheme and how can be applied to cutting planes based PB solvers in order to understand its performance. Then some aspects of PB solving that could be improved are reviewed and an implementation for one of them (division) is proposed. Finally, some experiments are run with this new implementation. Several instances are used as benchmarks encoding problems about graph theory (dominating set, even colouring and vertex cover).

In conclusion the performance of division varies among the different problems. For dominating set the performance is worse than the original, for even colouring no clear conclusions are shown and for vertex cover, the implementation of division outperforms the original version.

List of abbreviations

The following table shows the meaning of some of the most important acronyms and abbreviations used in this thesis.

Abbreviation	Meaning
BCP	Boolean Constraint Propagation
CNF	Conjunctive Normal Form
DNF	Disjunctive Normal Form
DPLL	Davis–Putnam–Logemann–Loveland
CDCL	Conflict Driven Clause Learning
LPB	Linear Pseudo-Boolean
PB	Pseudo-Boolean
SAT	Satisfiability Problem, could be used for Satisfiable
UNSAT	Unsatisfiable

Contents

Chapter 1

Introduction

A wide range of combinatorial problems can be codified in terms of propositional logic. This means that such problems can be expressed as propositional satisfiability (SAT) problems [2]. The key point of this process is that such combinatorial problems expressed as satisfiability problems, can often represent an easier approach. This is because the satisfiability problem is a very studied topic and various well-known techniques and algorithms are provided.

Because its nature, these combinatorial problems can be easily expressed using propositional logic's language, so that solving the adequate propositional logic statement gives a solution to the actual problem. It is important to acknowledge that in this process there are two separated parts: on the one hand we have the codification of the problem according to the logic we are using; on the other hand we have the actual solving of the reformulated problem. Consequently, this second part raises the need of algorithms which solve such satisfiability problem instances. Such algorithms are called SAT solvers.

As a result, SAT solving has become a procedure used in finding a solution for many of these combinatorial problems; e.g. Model Checking (hardware / software verification), cryptography, schedule planning, resource planning, combinatorial design and many others.

Nevertheless, very often the codification of complex combinatorial problems into SAT leads to a very large number of propositional logic equations. For this reason, efficiency in SAT solvers is a very important issue.

SAT was one of the first problems which was proven to be NP-complete, hence finding polynomial-time algorithm that solves any

SAT instance would involve proving $P = NP$. Despite the fact that there is no such algorithm that can be considered to have polynomial time, in practice, modern SAT solvers, which contain really advanced heuristics, are capable of solving problems with formulas formed by millions of symbols among tens of thousands of different variables.

There exist many different possible representations of the knowledge in terms of logic. As a result of its simplicity and easy reasoning, Conjunctive Normal Form (CNF), based on propositional logic, is the most used among SAT solvers. This format is basically a conjunction of disjunctions, namely a conjunction of clauses, having as clause a disjunction of literals.

The satisfiability problem can also be expressed as set of Linear Pseudo-Boolean (LPB) inequalities, where in each of them we have Boolean variables instead of regular mathematical variables. This is also a popular representation for SAT solvers. Whereas CNF solvers use as solving technique *resolution*, LPB solvers use an analog operation called *cutting planes*. This is why these solvers can be often referred as PB solvers based on cutting planes.

Although state-of-the-art CNF SAT solvers are able to solve really complex and long formulas, they spend a great amount of time or they do not finish at all with some particular compact problems e.g. *Pigeonhole Principle* [1]. Another drawback of modern SAT solvers comes from being (most of them) based on CNF representation. The power of expression of Clausal Normal Form is very low compared to other different representations for SAT instances, such as Linear Pseudo-Boolean (LPB) inequalities. LPB has a much more higher level of expression compared to CNF. This means a prohibitively larger number of clausal constraints is needed for expressing what in LPB domain could be regarded as a short problem.

It seems reasonable that keeping a representation of information more compact could lead to a more compact reasoning process when solving. Moreover, due to the compact expression of LPB inequalities in addition to specific Pseudo-Boolean (PB) techniques, some problems which may be intractable for a clausal solver, may actually be easy for a LPB solver based on cutting planes. An example of this is the *Pigeonhole* problem, which will be further detailed in this thesis.

The SAT solver field is in constant race for efficiency, which in terms of time complexity means being able to afford problems that used to be intractable in the past.

1.1 Problem statement

The main purpose of this thesis is to carry out a study about Pseudo-Boolean solvers based on cutting planes in order to increase the efficiency of a Pseudo-Boolean solver. The approach will be studying some not yet implemented specific techniques of PB and implement them. This could be sumed up in the following research question:

Are there any Pseudo-Boolean techniques that could be applied to a SAT solver based on cutting planes which could improve its efficiency?

1.2 Motivation

The vast majority of modern SAT solvers use the *Conflict Driven Clause Learning* (CDCL) scheme with clausal (CNF) representation. This configuration for the implementation of a solver is apparently getting the best results considering time consumption while execution.

However, as it was introduced in the previous subsections Linear Pseudo-Boolean inequalities have a higher expression capacity than CNF clauses. Moreover, PB resolution step (called cutting planes) is believed to be stronger than resolution for CNF clauses.

There are many operations that can be applied to PB constraints but for most of them, there has not been found an efficient implementation of them. This could be one of the main reasons why clausal solvers outperform PB solvers. The purpose of this project is to develop an efficient implementation of a PB solver that is competitive with state-of-the-art solvers.

1.3 Outline

This thesis is structured into five chapters. In the first chapter the topic is introduced, as well as, it is defined, in an introductory way, the purpose and the problem statement. The second chapter explains the background, giving basic knowledge about SAT, the CDCL scheme, CNF and LPB. Whereas in the third chapter there is a formal definition of the problem, as well as, the method used. Finally the results are shown and discussed in the fourth chapter and the conclusion is developed in the fifth.

Chapter 2

Background

In this chapter we introduce some background on SAT solvers and the satisfiability problem. The concepts of this chapter will be used and referenced in the following chapters. There is a review of the main features and characteristics of modern SAT solvers.

2.1 The Satisfiability Problem

The Boolean Satisfiability problem (SAT) [2] is defined as determining if there exists an interpretation (model) that satisfies a given set of constraints expressed as a Boolean formula. In other words, its aim is to find if there exists an assignment, for each of the variables in the formula, which satisfies all the constraints. We say a formula is *unsatisfiable* when there is no such combination of assignments for the variables that evaluates the formula to true, fulfilling all constraints; otherwise we say it is *satisfiable*.

There exists many different logics (e.g. *propositional logic* or *first-order logic*) in terms of which it is possible to express the SAT problem and each different logic may have different possible representations (for propositional logic for instance *CNF* or *DNF*). In this thesis the focus will be on propositional logic. According to this, the SAT problem will be further defined below in terms of propositional logic.

Let us consider $x_1, \cdots, x_n$ are Boolean variables. We say a Boolean formula is formed by clauses $C_1, \cdots, C_m$. Each clause $C_j = (l_1 \vee l_2 \vee \cdots \vee l_k)$, where $l_z = x_i$ or $l_z = (\overline{x_i})$ is a literal. Then the formula F has the following form:

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

This is expressed in Conjunctive Normal Form (CNF), which is a conjunction of clauses. Consider the formula (2.1), this formula has two clauses $(\overline{x} \vee \overline{y})$ and $(x \vee y)$. And it is satisfiable because the assignment of values $x = 1$ and $y = 0$ satisfies the formula and hence it is a model.

$$(\overline{x} \vee \overline{y}) \wedge (x \vee y) \tag{2.1}$$

But for the formula (2.2) there is no combination of Boolean assignments to the variables which satisfy the formula. We say it is unsatisfiable.

$$x \wedge \overline{x} \tag{2.2}$$

The SAT problem is proven to be NP-Complete. Problems in NP class are those for which there is not an efficient (i.e. polynomial time complexity) algorithm found to solve them, and it is believed that such algorithm does not exist. However, state-of-the-art SAT solvers can solve input formulas containing a high number of different variables and a huge number of symbols.

2.1.1 Resolution

Resolution is the reasoning method applied to clauses in order to prove that a given formula is unsatisfiable. Resolution implies a new clause from two clause that have a complementary literal. Let us consider the clauses $x_1 \vee \cdots \vee x_n \vee \overline{c}$ and $y_1 \vee \cdots \vee y_m \vee c$, resolution is applied as follows:

$$\frac{x_1 \vee \cdots \vee x_n \vee \overline{c} \quad y_1 \vee \cdots \vee y_m \vee c}{x_1 \vee \cdots \vee x_n \vee y_1 \vee \cdots \vee y_m}$$

However, to reach a conclusion this operation has to be applied correctly among the clauses, varying the order in which it is applied may lead to proofs exponentially larger than others. The complexity in resolution process is the reason why efficient algorithms with advanced heuristics are needed to get an approach for solving a formula.

2.2 Conflict Driven Clause Learning

There can be found a great deal of practical applications where SAT-solvers are applied e.g. cryptography, bio-informatics, schedule planning and many others [2]. This could be said it is mainly because of the

good performance of Conflict Driven Clause Learning (CDCL) solvers. CDCL is the name given to the structure for the solving algorithm that most modern SAT solvers are using.

CDCL structure was inspired by DPLL (Davis–Putnam–Logemann–Loveland) [3, 4], a backtracking search algorithm from 1960s. Although CDCL has many new features introduced, it still maintains the search structure in DPLL. For this reason in the following subsection (2.2.1) there is explained some background about DPLL in order to get a better understanding of the modern algorithm.

2.2.1 DPLL

The main idea of DPLL is assigning values to the literals appearing in the formula, keeping track of these assignments. Then, when a conflict is found, act accordingly. To understand the DPLL algorithm it is easier to talk about its stages separately, these are also repeated in CDCL. There are 4 stages in the algorithm [5]:

- **Unit Propagation**: the algorithm searches for all clauses in which there is only one literal without value assigned and all other literals in the clause, if any, falsified by the assignment. In order to get a valid model, these literals have to be set to true, otherwise we would have falsified clauses. Therefore, the corresponding values are assigned to each variable in order to satisfy the clauses.

- **Conflict**: this happens when the actual assignation of Boolean values to variables gives a contradiction in the formula and hence its evaluation with respect to the current assignment is false.

- **Decision**: this stage comes when there is no more assignments to do in unit propagation and no conflict has appeared. Then an unassigned variable is picked and a value is assigned to it. The method how to pick the next variable and which value to assign (True or False) depends on the heuristics used in the algorithm. We will call this variable *decision variable*.

- **Backtrack**: it is the stage performed when a conflict is found. Here it is important to notice that it is different a value assigned to a variable in unit propagation (if we assigned the opposite value to the variable it would give conflict, cannot be flipped) than a value assigned in a decision. The backtracking consists

in removing the value assigned to the variables in reverse order until a decision variable is found. Then the value of the decision variable is flipped.

The Davis–Putnam–Logemann–Loveland algorithm starts with Unit Propagation, propagating literals that are alone in a clause (as no values will be yet assigned). After each propagation the assignation database will be updated and each of them may produce more propagations as consequence. When there are no more propagations to perform and no conflicts are found, DPLL makes a decision. A variable with no value assigned previously is chosen and a value is given to it, this is marked as decision variable. After this, unit propagation takes place again as there may be some literals to propagate, and so on.

Whenever a false clause is detected in Unit Propagation this process stops and starts the backtrack. As it was already explained before, the algorithm backtracks until the last decision variable. The variable is unmarked as decision variable and it is assigned the opposite Boolean value to the one it had. After that, the execution follows with unit propagation.

Finally, the algorithm can stop in two cases. First case is in the decision step there are no unassigned variables left, then all variables are assigned which means a model to the formula is found. This is the case when DPLL returns SATISFIABLE. The opposite case is when the backtracking undoes all assignments and no decision variable is found, then UNSATISFIABLE is returned [6].

Example of DPLL execution

We will represent the trail of assignments as a string of literals; if in this string the literal $\overline{var}$ appears means that false Boolean value is assigned to var, being var any variable. Otherwise if what appears in the string is just var, the value assigned would be true. The decision variables will be shown with the upper-index $\overline{var}^d$. In this case, we will say that var was a decision variable, and its value is false. Note that literals not having this upper-index attached will be propagations.

Let us consider a formula (2.3) formed by the variables u, v, x:

$$(u \vee \overline{v}) \wedge (v \vee \overline{x}) \wedge (x \vee \overline{u}) \wedge (\overline{u} \vee \overline{x}) \tag{2.3}$$

The process depends totally on the order in which the variables are picked, but let us fix that to u, v, x. Then the execution would be the

following:

$$u^d \rightarrow decision \tag{2.4}$$
$$u^d x \rightarrow propagation\ (x \vee \overline{u}) \tag{2.5}$$
$$u^d x \rightarrow conflict\ (\overline{u} \vee \overline{x}) \tag{2.6}$$
$$\overline{u} \rightarrow backtrack \tag{2.7}$$
$$\overline{u}\ \overline{v} \rightarrow propagation\ (u \vee \overline{v}) \tag{2.8}$$
$$\overline{u}\ \overline{v}\ \overline{x} \rightarrow propagation\ (v \vee \overline{x}) \tag{2.9}$$
$$\overline{u}\ \overline{v}\ \overline{x} \rightarrow conflict\ (x \vee \overline{u}) \tag{2.10}$$

In the last conflict (2.10) there is no branching decision to backtrack so that the algorithm finishes its execution returning UNSATISFIABLE, hence the formula has no model that satisfies it.

2.2.2 Organization of CDCL Solvers

The CDCL scheme was introduced in the mid-90s and with it many new features were introduced and the combination of them give the good performance these SAT solvers. In general terms, the most important techniques found in CDCL SAT solvers a part from the DPLL structure are the following [2]:

- Unit Propagation optimizations for speeding this process.
- Conflict analysis able of generating new clauses describing conflicts, the aim of this is avoiding to explore areas in the search that lead to a conflict that was already seen.
- Backjump, the difference with backtrack in DPLL is that this backjump can be to any previous decision level, not necessarily the one before the current when the conflict arises.
- Use of lazy data structures for the representation of formulas [2, 6, 7, 8].
- Better heuristics for choosing next decision variable.
- Restarting the search often. It is possible that eventually the search goes very deep in the search tree, if the path leads to conflicts it may take a long time in backjumpings until the solver

gets back into good tracks. To avoid this behavior in the solver, restarts in the search are placed often [7, 9, 10]. With each restart the trail of assignments is all erased, but learnt clauses stay in the clause database.

Additional techniques can be found in CDCL solvers depending on the implementation, this may include the different implementations of the lazy data structures, also erasing unused learnt clauses periodically or the organization of unit propagations. For the purposes of this project we will only focus on Conflict Analysis + Backjumping and Unit Propagation as main characteristics of CDCL.

As it was mentioned before the structure of CDCL is based on the DPLL with the integration of these features. There can also be seen the stages of **decision**, **unit propagation**, **conflict** and **backjumb** (which was called backtrack in DPLL). The pseudo-code is shown in Algorithm 2.1 [11]. There are some functions which will be further explained below:

Algorithm 2.1 CDCL Algorithm [11]

```
1: procedure SEARCH
2:     while true do
3:         while propagate_gives_conflict() do
4:             if decision_level == 0 then return UNSAT
5:             else analyze_conflict()
6:         restart_if_applicable()
7:         remove_lemmas_if_applicable()
8:         if !decide() then return SAT
```

- $propagate\_gives\_conflict()$: This function performs the unit propagation and if a conflict is found during the process, it stops and returns true, false is returned otherwise.

- $decision\_level$: This represents the count of decisions taken. At each decision level only one decision is performed, this may trigger some propagations and these are also associated with that level. If its value is zero no decisions are taken yet. If we find a conflict in the initial decision level there is no model that satisfies the current formula because there is no possible backtrack point.

- $analyze\_conflict()$: This function analyses the conflict. This function generates a new clause that explains the conflict and avoids to explore it again, this clause is added to the clause database. More information in section 2.2.3.

- $restart\_if\_applicable()$: According to some predefined parameters a frequency of restart is fixed. Periodically a restart of the search will be applied. This function evaluates the parameters and if it is time for a restart it is applied. This avoids too long dead-ends for solver.

- $remove\_lemmas\_if\_applicable()$: As mentioned before one of the possible features in CDCL is learnt clauses erasure. There are also some parameters that define how often to do it, and how many of them will be erased. In this function these parameters are checked and if it is time, the predefined amount of clauses is erased. Notice that erasures are always from learnt, never from original clauses. It is also important to notice that clauses that are currently reasons for propagated literals in the trail are locked and cannot be erased.

- $decide()$: Applies the heuristics to find an unassigned variable, decide its Boolean value and this is added to the trail.

Although Algorithm 2.1 is the main scheme of CDCL, in each different implementation of the algorithm the functions defined above may differ, e.g. using different heuristics or data structures. These heuristics and implementations of the data structures may make a big difference between versions of the solver.

2.2.3 Clause Learning

CDCL solvers have several new techniques and rules that make the difference with DPLL solvers, but the most important, which gives the name to the Conflict Driven Clause Learning method is learning clauses from conflicts. CDCL solvers are capable of extracting a clause that explains the conflict in order avoid exploring the same conflict again in future search. Once a conflict is found resolution is applied to obtain the clause to learn. The clause learnt needs to contain only one literal from the current decision level so that when the backjump is performed it triggers unit propagation and we assure that same conflict

does not happen again. These clauses that result form conflict analysis that only contain one literal from the current conflicting decision level are called *Unique Implication Point* (UIP).

Note that there can be more than one UIP found in the resolution process from the conflict analysis. In this case, they will be sorted related in the order in which they are found in the resolution process, and the first on the sequence will be the clause to learn. This is called First UIP or *1UIP*, the authors of [12] note that gives the best results in CNF-based solvers.

The 1UIP, which is the clause that will be learnt, also determines the level to which backjump. Among the decision levels of all literals in the 1UIP clause, the backjumping level is the biggest that is not the conflicting level. Once the clause is learned, all literals in the trail asserted later than the backjumping level are erased, so that they become unassigned.

Example of Clause Learning

Let us consider the formula 2.11:

$$\begin{aligned}&(\overline{u} \vee \overline{v} \vee y) \wedge \\ &(\overline{u} \vee \overline{v} \vee x) \wedge \\ &(\overline{u} \vee v \vee \overline{x}) \wedge \\ &(u \vee \overline{v} \vee \overline{x}) \wedge \\ &(\overline{u} \vee \overline{v} \vee \overline{x}) \wedge \\ &(a \vee y)\end{aligned} \tag{2.11}$$

Let us consider that the order in which the variables are picked for decision is u, a, v, y, x and that the decision will set first the variables to true. The decision level will be labelled as *DLx* being x the level. The execution of CDCL goes as follows:

$$\text{DL1} \quad u^d \rightarrow \mathit{decision} \tag{2.12}$$
$$\text{DL2} \quad u^d a^d \rightarrow \mathit{decision} \tag{2.13}$$
$$\text{DL1} \quad u^d a^d v^d \rightarrow \mathit{decision} \tag{2.14}$$
$$\text{DL3} \quad u^d a^d v^d y \rightarrow \mathit{propagation}\ (\overline{u} \vee \overline{v} \vee y) \tag{2.15}$$
$$\text{DL3} \quad u^d a^d v^d y x \rightarrow \mathit{propagation}\ (\overline{u} \vee \overline{v} \vee x) \tag{2.16}$$
$$\text{DL3} \quad u^d a^d v^d y x \rightarrow \mathit{conflict}\ (\overline{u} \vee \overline{v} \vee \overline{x}) \tag{2.17}$$

Now a conflict has been found, so conflict analysis is going to be applied to get the clause to learn and also the level to which backjump:

$$\frac{\overline{u} \vee \overline{v} \vee \overline{x} \quad \overline{u} \vee \overline{v} \vee x}{\overline{u} \vee \overline{v}} \tag{2.18}$$

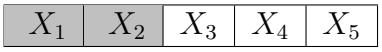
Resolution 2.18 is applied between the conflict clause 2.17 and the one that is the reason for the previous propagation 2.16. In this case a UIP is immediately found and since it is the first resolution step it is a 1UIP. The learnt clause will be $\overline{u} \vee \overline{v}$. It is in fact a UIP because it only contains one variable decided in the current decision level, which is v. As mentioned the backjump will be until the biggest decision level of the variables in the clause which is not the conflicting level. That is decision level 1.

It is possible that in the first resolution step the result clause is not a UIP, then resolution will be applied again between the clause obtained and the previous propagation reason, in this case would be 2.15.

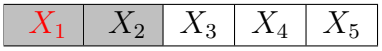
2.2.4 Unit Propagation: the two watched literal scheme

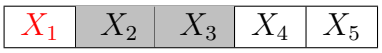
Statistically what solvers spend most time doing is unit propagation, approximately #propagations/#decisions = 323 in state-of-the-art CDCL solvers. Hence, it is a matter of fact that a good implementation of unit propagation is an important factor for the efficiency in SAT solvers. In this section the watching literal scheme for unit propagation will be introduced. This scheme is widely used in modern SAT solving.

When a value is assigned to a variable all clauses in which it is present could become unit so the solver should be aware of that. Visiting all clauses is not an efficient implementation. The watched literal scheme keeps track only of two pointers per clause. In this method at the beginning the first two positions of each clause are watched, namely the pointers.

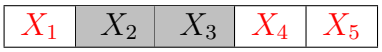


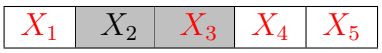
As long as, this two watched literals are not falsified there is no need to visit this clause. When one of them is falsified, another not-falsified literal in the clause is searched and this becomes the new watch (keeping the old not-falsified and the new).

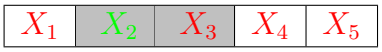




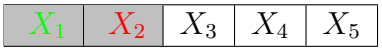
When one of the literals is falsified and there is no other not-falsified literal in the clause to pick as watch, propagate the other one.



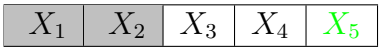


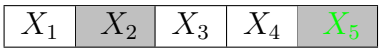


When a watched literal is satisfied then the other literal does not matter anymore because the clause is satisfied.



If another a satisfied literal is found, that becomes watched.





With this scheme the solver only needs to keep track of two literals per clause which represent unit propagation or not in that clause and over which literal. The most part of the computational time of CDCL solvers they are performing unit propagation, efficiency in this stage is very important.

2.3 The Pseudo-Boolean approach

As it was introduced before, there exists several representations for expressing a Boolean formula. In this section it will be introduced the Pseudo-Boolean (PB) interpretation and how the CDCL scheme can be applied to it.

An ordinary SAT instance is defined as a conjunction of clauses, which are formed by the disjunction of literals, as it was introduced in the section 2.1. Let x_i be a Boolean variable and $l = x_i$ or $l = \overline{x_i}$ be a literal. Then a clause is of the form $C = (l_1 \vee l_2 \vee \cdots \vee l_k)$ and finally a the SAT instance expressed in *clausal* form can be defined as $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$.

For the Pseudo-Boolean interperetation, the representation of Boolean formula in clauses is redefined. In this case the SAT represented is codified as inequalities of sums of weighted Boolean variables [13], which may also be referred as Linear Pseudo-Boolean (LPB) constraints.

Let us consider $x_1, \cdots, x_n$ are Boolean variables and $c_1 \cdots c_n$ are integer positive coefficients. The SAT instance is a set of m inequalities $C_1, \cdots, C_m$. The right-hand side of the inequalities is an integer and it is often referred as degree. Where each inequality is of the form

$$C_j = \sum_i c_i \cdot l_i \geq w, \;\; a, w \in \mathbb{Z}, \;\; l_i \in \{x_i, \overline{x_i}\}, \;\; \overline{x_i} = (1 - x_i)$$

Typically, the constraints may have coefficients with real values, for the scope of this thesis, all coefficients are considered integer-valued as it is assumed in [14].

For the formulas expressed in this thesis we will also make the convention to have all coefficients with positive value, as well as, we will also use "$\geq$" as only inequality symbol. For instance given the inequality $-5 \cdot x + 3 \cdot y \leq 1$, we can transform the inequality in the following way so that we get the desired format (note that for LPB constraints $x = 1 - \overline{x}$):

$$-5 \cdot x + 3 \cdot y \leq 1 \Leftrightarrow +5 \cdot x - 3 \cdot y \geq -1 \Leftrightarrow +5 \cdot x - 3 \cdot (1 - \overline{y}) \geq -1 \Leftrightarrow$$

$$+5 \cdot x + 4 \cdot \overline{y} \geq -1 + 3 \Leftrightarrow +5 \cdot x + 4\overline{y} \geq 2$$

The SAT problem in this case is the same, finding an assignment that satisfies all inequalities or otherwise proving it is UNSAT. The

SAT instance expressed as LPB constraints is much more powerful in terms of representation, in fact, the number of CNF clauses required for expressing the LPB constraints is prohibitively large [14]. This allows us to compactly describe problems, note that given a LPB formula it may take an exponential number of CNF clauses to express the same problem. LPB problems can be solved by generic integer linear programming (ILP) solvers. But this is a more mathematical approach rather than Boolean, getting a wider search space due to not using specialized cutting planes methods.

As it is stated in [13] there are three keys to the modern SAT solvers performance: 1) fast Boolean constraing propagation (BCP) based on effective filtering of irrelevant parts of the problem structure; 2) learning of compact facts representing the large infeasible parts of the solution space; 3) fast selection of decision variables.

In terms of CDCL clausal solvers, the previous list can be mapped as follows: 1) corresponds to Unit Propagation; 2) Clause Learning; 3) decision heuristic for picking the next literal to be decided. In this section there will be a review of how can 1 and 2 (sections 2.3.3 and 2.3.4) be performed changing the definition of the solver representation to LPB. For the scope of this thesis 3 will be considered independent from the representation.

The main operation on CNF clauses that leads to the proof throughout the SAT solver execution is resolution (2.1.1) and the analog resolution step for LPB inequalities is called cutting planes and it is explained in the following sub-section. Note that this is a very characteristic operation of solvers based on LPB inequalities, that is why these solvers may be often referred as *PB solvers based on cutting planes*.

2.3.1 Cutting Planes

Cutting planes is the corresponding LPB operation to the CNF resolution. It consists in an addition of two constraints possibly multiplied by a coefficient, namely a non-negative linear combination of them. It can also be referred as *clashing addition*.

Consider the constraints $\sum c_i \cdot l_i \geq w$ and $\sum c'_i \cdot l_i \geq w'$ and the integer coefficients λ_1 and λ_2, the cutting planes operation is shown as

follows:

$$\frac{\begin{array}{c}\lambda_1 \cdot (\sum c_i \cdot l_i \geq w) \\ \lambda_2 \cdot (\sum c'_i \cdot l_i \geq w')\end{array}}{\lambda_1 \cdot (\sum c_i \cdot l_i) + \lambda_2 \cdot (\sum c'_i \cdot l_i) \geq \lambda_1 \cdot w + \lambda_2 \cdot w'}$$

In LPB there exists many specific operations among the constraints, a brief introduction to some of these operations can be found in the next sub-section.

2.3.2 Operations on LPB constraints

Division

This can be applied when all coefficients have a *gcd* greater than 1. Then they are all divided, consider the gcd is now a:

$$\frac{\sum (a \cdot c_i) \cdot l_i \geq w}{\sum c_i \cdot l_i \geq \lceil w/a \rceil}$$

Coefficient Rounding

Because of the Boolean nature of the variables the coefficients may be rounded up, note that $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$.

$$\frac{\sum c_i \cdot l_i \geq w}{\sum \lceil c_i \rceil \cdot l_i \geq \lceil w \rceil}$$

Saturation

Saturation changes a coefficient of a constraint to w if the coefficient's value is greater than it, note that this operation is correct due to the Boolean nature of the constraints. It can be expressed as follows:

$$\frac{\sum c_i \cdot l_i \geq w}{\sum min(c_i, w) \cdot l_i \geq w}$$

Weakening

This operation *weakens* the coefficients on the left-hand side of the inequality and reduces accordingly the the degree.

$$\frac{\sum_{i \neq j} c_i \cdot l_i + c_j \cdot l_j \geq w}{\sum_{i \neq j} c_i \cdot l_i \geq w - c_j}$$

2.3.3 Boolean Constraint Propagation

This part is analog to the unit propagation in CNF clauses, here it is presented an adaptation for LPB constraints. For CNF it suffices to keep track of just two literals of each clause, as it is shown in 2.2.4. This method is based on the rule that whenever a clause has all its literals falsified but one, this has to be set to true in order to satisfy the clause.

For the Pseudo-Boolean approach the idea is focused on the fact that for LPB constraints whenever the falsification of one literal would falsify the whole clause, this needs to be set to true. This happens when the coefficient of such literal is greater than *the maximum possible amount by which a constraint can be over-satisfied*. This is called *slack* and it is computed with the coefficients whose literal has no value assigned it is or set to true and the degree of the constraint, 2.19 shows how the slack is computed. Let us consider S as the set of assignments to variables at the current moment of computing the slack.

$$slack = \sum_{i:\overline{l_i} \notin S} c_i - w \tag{2.19}$$

An unassigned literal needs to be propagated when its coefficient is greater than the slack. The slack represents how much *over-satisfied* can be a constraint respect its degree. This takes into account all coefficients whose literal is still unassigned or is assigned to true. This represents the maximum value that can get the left side of the inequality. If one literal l_k has a coefficient such that $slack - c_k < 0$ then it has to be implied to true. The proof of this is given in 2.20.

$$\begin{aligned}
&slack - c_k < 0 \Leftrightarrow \\
&\sum_{i:\overline{l_i} \notin S} c_i - w - c_k < 0 \Leftrightarrow \\
&\sum_{i:\overline{l_i} \notin S \wedge i \neq k} c_i - w < 0 \Leftrightarrow \\
&\sum_{i \neq k} c_i \cdot l_i - w < 0 \Leftrightarrow \\
&\sum_{i \neq k} c_i \cdot l_i < w \Leftrightarrow \\
&constraint\ falsified
\end{aligned} \tag{2.20}$$

It is important to note that the step

$$\sum_{i:\overline{l_i} \notin S \wedge i \neq k} c_i - w < 0 \Leftrightarrow \sum_{i \neq k} c_i \cdot l_i - w < 0$$

is given because the coefficients not taking part in the slack are the ones assigned to false, which are also the ones not taking place in the sum, namely not adding value.

In conclusion, the literals which need to be watched are the ones whose coefficient is greater than the slack.

2.3.4 Pseudo-Boolean Learning

This subsection shows how clause learning from conflict analysis can be accomplished for a Pseudo-Boolean solver based on cutting planes.

For a clausal CDCL solver this process is based on applying resolution among the conflicting clause and the reason clause for the last propagated literal. If the result of this operation is not a UIP resolution this step is performed once again with the resulting clause and the reason clause of the previous propagated literal and so on, until a UIP is found. Once a UIP is found, by definition it only contains exactly one literal propagated in the conflicting decision level, which guarantees that will trigger a propagation in a previous decision level. Following the same pattern we need to ensure the following two premises for the PB clause learnt:

- The learnt PB clause must guarantee that there exists a decision level to which backjump, in which one or more propagations will be implied, according to the variable assignment.

- The learnt PB clause has to remain in conflict with the variable assignment, ensuring a backjump form the conflicting decision level.

A clause fulfilling the first property will be referred as assertive. The second property it is not mentioned with regular clauses since the opposite never occurs, hence it does not need to be checked for them. But for PB constraints it is possible that after applying the analog of resolution step with PB, the result is no longer in conflict with respect to the trail of variable assignments.

Let us consider the trail of assignments $\{\overline{x}, y\}$, the conflict clause 2.21 and the reason clause for the last assignment in the trail 2.22. Note

that to see if a constraint is falsified under the current trail it is only necessary compute its slack and see that it is negative. For instance the slack of the constraint 2.21 is -2, therefore it is conflicting with the current assignment.

$$1 \cdot x + 3 \cdot \overline{y} + 3 \cdot z \geq 5 \tag{2.21}$$

$$3 \cdot y + 1 \cdot z \geq 2 \tag{2.22}$$

According to conflict analysis resolution (clashing addition for PB) is applied to these two clauses:

$$\frac{\begin{array}{r} 1 \cdot x + 3 \cdot \overline{y} + 3 \cdot z \geq 5 \\ 3 \cdot y + 1 \cdot z \geq 2 \end{array}}{1 \cdot x + 3 \cdot y - 3 \cdot y + 3 \cdot z + 1 \cdot z \geq 5 + 2 - 3 \Leftrightarrow 1 \cdot x + 4 \cdot z \geq 4}$$

Note that as it was introduced at the beginning of this section for PB constraints $\overline{y} = 1 - y$. The resulting constraint of the resolution step is $1 \cdot x + 4 \cdot z \geq 4$ which is no longer in conflict with the assignation trail, since its slack has value 0. The resulting constraint does not show the second property.

The slack of a constraint shows if it is falsified or not, namely a negative slack determines that a constraint is in conflict with the trail. In order to maintain the second property when performing the conflict analysis it is necessary to keep the learned clause with negative slack. In the previous example we added the conflicting clause, which had slack -2, with the reason clause, which has slack 2; the resulting clause had slack 0. When resolving, the addition with a clause with positive slack will increase the slack of the conflicting clause, eventually making it positive or zero and therefore losing the conflict information.

Nevertheless, this can be avoided by weakening the reason clause until its slack is lower than the absolute value of the conflict clause's slack. The weakening operation can be applied to slack contributing literals until the clashing addition can be applied, this is shown in Algorithm 2.2.

Algorithm 2.2 Resolve for PB constraints [15]

1: **procedure** RESOLVE($C_{confl}, l_0, C_{reason}, S$)
2: **while** $true$ **do**
3: $C \leftarrow ClashingAddition(C_{confl}, l_0, C_{reason})$;
4: **if** $slack(C, S) < 0$ **then return** $saturation$(C);
5: $l^* \leftarrow$ *any literal occurring in* $C_{reason} \setminus \{l_0\}$ *such that* $\neg l^* \notin S$;
6: $C_{reason} \leftarrow saturation(weaken(C_{reason}, l^*))$

In the previous example the variable z would be picked as l^* since it is the only one not falsified (and it is not the variable which we want to resolve). After applying weakening over z the result is:

$$3 \cdot y \geq 1$$

Then saturation is applied:

$$1 \cdot y \geq 1 \tag{2.23}$$

Then considering the clause 2.23 the resolve step is as follows:

$$\frac{\begin{array}{l} 1 \cdot x + 3 \cdot \overline{y} + 3 \cdot z \geq 5 \\ 3 \cdot (1 \cdot y \geq 1) \end{array}}{1 \cdot x + 3 \cdot z \geq 5}$$

Note that for the clashing addition accomplishes its purpose (resolve over y) both constraints need to have the same coefficient for opposite literals. This is ensured by λ_1, λ_2 in the clashing addition. In this example $\lambda_1 = 1$ and $\lambda_2 = 3$.

Chapter 3

Methodology

In this chapter the problem statement is described in depth, explaining further details according to the concepts introduced in the Background chapter 2. In addition, it is described the methodology used in order to tackle with the problem stated.

3.1 The Problem

The great performance of state-of-the-art SAT solvers is mostly due to the CDCL scheme. Its capacity of learning from errors, the techniques for fast propagation of literals, in addition to some heuristics, make the solvers being able to deal with formulas which were intractable with previous implementations. Nowadays SAT solving has become a very used tool for problem solving and optimization, with many practical applications e.g. Model Checking (hardware / software verification), cryptography, schedule planning, resource planning, combinatorial design and many others.

However, although state-of-the-art SAT solvers are able to solve highly complex and long formulas, they spend a great amount of time or they do not finish at all with some particular compact problems. One example of this is the *Pigeonhole Principle* [1], which will be further described in this subsection 3.1.1. Another drawback of modern SAT solvers comes from being (most of them) based on CNF representation. The capacity of expression of Conjunctive Normal Form is very low compared to other different representations for SAT instances, such as Linear Pseudo-Boolean (LPB) inequalities. LPB is much more expressive than CNF, in fact, the number of CNF clauses required for

expressing a LPB instance is exponential.

In order clearly represent the main topics, the problem formulation is structured in the following subsections. The first two, 3.1.1 The Pigeonhole Principle and 3.1.2 The AtMost-k encoding show two examples of the main drawbacks that can be found in clausal SAT solvers, respectively, a simple problem that becomes intractable for many solvers and an encoding of formulas that require extremely large number of clauses. Then the subsection 3.1.3 The Focus explains where the main study of this thesis is settled and finally 3.1.4 The Approach describes how the problem is going to be undertaken.

3.1.1 The Pigeonhole Principle

The Pigeonhole Principle states that given a number n of pigeons and a number m of holes, having $n > m$, it is impossible to place each pigeon in one hole and not have more than one pigeon per hole. In SAT terminology, this means that placing each pigeon in one hole and having only one pigeon per hole is UNSATISFIABLE.

This problem can easily be translated into both CNF and LPB. Let us consider the variable $x_{i,h}$, which expresses if pigeon i is placed in hole h, with the truth value assigned to it.

CNF representation:

$$x_{i,1} \vee \cdots \vee x_{i,n}, \forall i \tag{3.1}$$

$$\overline{x_{i,h}} \vee \overline{x_{j,h}}, \forall h, \forall i \neq j \tag{3.2}$$

The clauses 3.1 represent that every pigeon i has to be in at least one hole h. And the clauses 3.2 represent that two pigeons cannot be placed into the same hole. We could add some clauses restricting physics laws, such as, that one pigeon cannot be placed in two holes at a time, but since we want to keep it simple and it will be restricting a part of the search space that is already unsatisfiable we leave them apart.

LPB representation:

$$\sum_{h=1,\cdots,m} x_{i,h} \geq 1, \forall i \tag{3.3}$$

$$\begin{aligned} &\sum_{i=1,\cdots,n} x_{i,h} \leq 1, \forall h \Leftrightarrow \sum_{i=1,\cdots,n} -x_{i,h} \geq -1, \forall h \Leftrightarrow \\ &\sum_{i=1,\cdots,n} -(1-\overline{x_{i,h}}) \geq -1, \forall h \Leftrightarrow \sum_{i=1,\cdots,n} \overline{x_{i,h}} \geq n-1, \forall h \end{aligned} \tag{3.4}$$

The representation for the LPB constraints it is analog to the CNF the encoding, 3.3 represents that every pigeon has to be in at least one hole. Whereas the constraint 3.4 represents that in each hole can be placed at most 1 pigeon. The first inequality of the derivation in 3.4 is the most intuitive. However, the derivation to get to the last part is done because we will use the convention of having all coefficients with positive value and only the "$\geq$" as inequality symbol. Note that for PB $x = 1 - \overline{x}$.

This problem has clearly a compact number of formulas as input. But for the clausal CNF approach, solved by resolution, having $m = n-1$ it was proven by [1] that it takes an exponential length in terms of resolution steps to solve it. Concretely, $exp(\Omega(m))$, being m the number of holes. While for the LPB approach it becomes a much more shorter process, by its construction of the constraints.

This kind of reasonably short formulas that take an exponential time to solve is one of the main drawbacks in SAT solving. Its main problem comes from a bad encoding of cardinality constraints. It will be further explained in 3.2.4.

3.1.2 The AtMost-k encoding

Besides to this low efficiency in solving bad cardinality constraints encodings, as introduced above. LPB is also more compact in terms of knowledge representation, requiring a prohibitively large number of CNF clauses for representing LPB constraints [14]. An example of this can be shown with the representation of the well-known encoding *AtMost-k*. This states that at most k of certain variables can be true. The encoding in LPB can be achieved with one constraint:

$$x_1+\cdots+x_n \leq k \Leftrightarrow -(1-\overline{x_1})+\cdots+-(1-\overline{x_n}) \geq -k \Leftrightarrow \overline{x_1}+\cdots+\overline{x_n} \geq n-k$$

Note that also in this encoding we follow the convention of having only positive coefficients and "$\geq$" as the only inequality symbol.

However, the encoding for CNF is gets larger in terms of clauses, taking $\binom{n}{k+1}$ clauses to encode it, with n being the number of variables. It needs to be created a clause for each possible combination of $k+1$ negated elements of n (without repetitions). For instance, if $k = 2, n = 5$:

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$$

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}$$

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_5}$$

$$\overline{x_1} \vee \overline{x_3} \vee \overline{x_4}$$

$$\overline{x_1} \vee \overline{x_3} \vee \overline{x_5}$$

$$\overline{x_1} \vee \overline{x_4} \vee \overline{x_5}$$

$$\overline{x_2} \vee \overline{x_3} \vee \overline{x_4}$$

$$\overline{x_2} \vee \overline{x_3} \vee \overline{x_5}$$

$$\overline{x_2} \vee \overline{x_4} \vee \overline{x_5}$$

$$\overline{x_3} \vee \overline{x_4} \vee \overline{x_5}$$

And the number of clauses required for the encoding raises at high speed, for $n = 50, k = 2$ we need $19,600$ clauses and for $n = 50, k = 19$ $\#clauses = 47,129,212,243,960$.

3.1.3 The Focus

Taking into account these facts in addition to the different operations that can be applied to PB constraints (2.3.2) together with the CDCL scheme applied to PB solvers (2.3), it seems reasonable to bet for PB solvers in the race for efficiency. However, CNF clausal CDCL solvers still outperform PB solvers in terms of execution time. One possible reason is that the advantages of these solvers do not overtake the overhead related with saving and managing the coefficients of the inequalities.

Nevertheless, PB solving and cutting planes resolution are very complex and there is much research to be done related to LPB. Further research in the field of PB solvers based on cutting planes could possibly lead to techniques that have not been implemented yet.

3.1.4 The Approach

The intention of this thesis is to do research about the CDCL scheme and the adaptation of it to PB solvers (2). Then study different topics regarding PB solving and cutting planes that could be improved and eventually develop some techniques that could speed up the execution of a PB solver. And finally get involved with some actual PB solver and implement those techniques with the aim to come to a better solution in terms of solving time.

In the following sections of this chapter there is the introduction and explanation of the different topics under study and description of which techniques could be applied, a description of the solver used as target for the implementation and finally the implementation of the studied techniques.

3.2 Pseudo-Boolean topics under study

A Pseudo-Boolean solver implemented on top of cutting planes, requires to be based on an adaptation of the CDCL scheme to be competitive with the modern SAT solvers. But Pseudo-Boolean solving is a complex field and there are many questions to be answered and aspects in which more research is needed. Fully understanding some of this questions could lead to a more efficient implementation of a cutting planes CDCL solver. In this section some of the topics under study of PB cutting planes SAT solving are going to be reviewed. Finally, at the end of this chapter an implementation involving some of the following questions will be proposed.

3.2.1 Constraint Propagation

SAT solvers spend most of the computational time performing Boolean Constraint Propagation, hence the efficiency in this process plays an important role in the SAT solver performance. For solvers based on resolution, there exists a really efficient implementation in terms of both memory usage and access time, the two-watched literal scheme (2.2.4). It is based on the idea that, only two literals per clause need to be watched to know if a constraint is propagating or not.

However, for solvers based on cutting planes it is not that easy. The PB approach needs to keep track of the literals whose coefficient is

greater than the slack as explained in 2.3.3. The slack represents how much a constraint can be *over-satisfied*. In other words if all remaining unset literals where set to true, the sum on the left side of the inequality would be greater than the weight by the value of the slack. Therefore, the slack represents also how much weight can still be negated to keep the constraint satisfied. Note that if a literal whose coefficient is greater than the slack is falsified, the whole constraint is falsified.

The watching literals scheme for PB solvers is not nearly as efficient as the one for clausal solvers. In fact, some experimental results, like the ones in [13], determine that the performance is only good when the value of the weight is low in comparison to the coefficients in the left part of the inequality. Otherwise it is easy to end up watching a lot of literals, maybe even all of them.

Consequently, we can say PB solvers do not have a very efficient implementation of BCP in comparison with clausal solvers and this is one of the keys for the good results of CDCL solvers. Finding an efficient implementation would be crucial for boosting the efficiency of PB cutting planes SAT solvers.

Nevertheless, this is a very studied topic. Since research does not seem to come to a conclusion for the best implementation of BCP in PB solvers, for the scope of this project the implementation used will be based on the idea showed in 2.3.3, as it is how the target solver (3.3) is implemented.

3.2.2 Weakening criteria

In the section 2.3.4 was introduced how the clause learning could work according to the CDCL scheme for a SAT solver based on cutting planes. When applying the cutting planes step it can happen with PB constraints that the resulting constraint has positive slack, not being in conflict with the trail anymore. This can be avoided by following the algorithm 2.2. The main idea is to systematically apply weakening and saturation to the reason until the new clause's slack is negative. This will lower the value of the slack for the constraint and eventually get a negative slack.

In order to make it work the chosen literals to weaken in the clause have to be slack contributing, namely not being assigned to false. So that its removal from the constraint can reduce the value of the slack.

Although it has been proved to work, there is no clue of which is

the best way to implement it. There is no knowledge about which literal is better to be chosen when weakening, a part that it has to be slack contributing.

3.2.3 Division

Division is a very powerful operation on LPB constraints. It is capable of reducing the value of all coefficients, without loss of information, when they have a *GCD* greater than 1.

An inefficient implementation of this operation during runtime could take a lot of time. But one could imagine some efficient implementations to apply during conflict analysis so that the clauses get the value of coefficients reduced, if possible, without losing information expressed. Having lighter constraints (namely lower values of coefficients) could yield shorter resolutions of formulas.

3.2.4 Cardinality constraints detection

Constraints can sometimes be expressed in various different ways, some of them may be more efficient than others when it comes to solving. An example of this can be found with the *AtMost-k* encoding that was introduced in the subsection 3.1.2. The encoding showed for LPB constraints is the easiest for both writing it and for the SAT solver to solve it. But it can often happen that this is not the encoding we get in the input formula.

Consider an input formula in CNF format encoding *AtMost-k* that we want to translate to LPB constraints. It is easy to literally translate the clauses so that we get a PB encoding that it is as inefficient as the CNF encoding.

Let us consider the same example used above. Let us encode *AtMost-k* for $k = 2, n = 5$ as follows:

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$$

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}$$

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_5}$$

$$\overline{x_1} \vee \overline{x_3} \vee \overline{x_4}$$

$$\overline{x_1} \vee \overline{x_3} \vee \overline{x_5}$$

$$\overline{x_1} \vee \overline{x_4} \vee \overline{x_5}$$

$$\overline{x_2} \vee \overline{x_3} \vee \overline{x_4}$$
$$\overline{x_2} \vee \overline{x_3} \vee \overline{x_5}$$
$$\overline{x_2} \vee \overline{x_4} \vee \overline{x_5}$$
$$\overline{x_3} \vee \overline{x_4} \vee \overline{x_5}$$

Whereas the LPB encoding the formula as showed in 3.1 for this example is the following:

$$\overline{x_1} + \overline{x_2} + \overline{x_3} + \overline{x_4} + \overline{x_5} \geq 3$$

However one could translate the CNF clauses into LPB constraints and get the following encoding:

$$\overline{x_1} + \overline{x_2} + \overline{x_3} \geq 1$$
$$\overline{x_1} + \overline{x_2} + \overline{x_4} \geq 1$$
$$\overline{x_1} + \overline{x_2} + \overline{x_5} \geq 1$$
$$\overline{x_1} + \overline{x_3} + \overline{x_4} \geq 1$$
$$\overline{x_1} + \overline{x_3} + \overline{x_5} \geq 1$$
$$\overline{x_1} + \overline{x_4} + \overline{x_5} \geq 1$$
$$\overline{x_2} + \overline{x_3} + \overline{x_4} \geq 1$$
$$\overline{x_2} + \overline{x_3} + \overline{x_5} \geq 1$$
$$\overline{x_2} + \overline{x_4} + \overline{x_5} \geq 1$$
$$\overline{x_3} + \overline{x_4} + \overline{x_5} \geq 1$$

This encoding for LPB constraints is as inefficient as the one showed for CNF. The detection of this kind of bad encodings is called *cardinality constraints detection* and there are several methods for preprocessing of the formula, before the execution of the solver. However, an efficient implementation of this during runtime would make solvers able to detect this bad encodings from the input formula as well as from the constraints learned from conflict analysis.

3.3 The Solver

In this section it is presented the solver that is used as base for the implementation.

3.3.1 CDCL-CuttingPlanes

The cdcl-cuttingplanes solver was developed by Jan Elffers [15] a PhD student in the Theoretical Computer Science group (TCS) in KTH. The solver was the best in the DEC-SMALLINT-LIN track of the Pseudo-Boolean Evaluation 2016. It is a CDCL solver built on top of cutting planes.

3.4 Implementing division

In the section 3.2 there is a review of some topics of PB SAT solving that could be improved, some of them because they are not implemented yet in these SAT solvers while others because its implementation could be improved. In this section we propose an implementation of division for the solver cdcl-cuttingplanes (3.3.1).

The idea is to implement this operation on the solver so that when it is possible it is applied on a constraint in order to make it lighter for the solver. The aim of this implementation is trying to redirect the solver somehow to shorter solutions in terms of resolution steps (i.e. cutting planes steps).

For applying division on a constraint we need to compute the GCD of all the coefficients and then divide them by the value. For computing the the GCD we will start with the first two coefficients and compute the GCD of them, then we will compute the GCD of the first result with the third coefficient and so on. Eventually, we will get a 1 as result and that ends the computation process. Otherwise the computation will end up finding a value by which all coefficients can be *integrally* divided. The algorithm is structured as follows, where $coef$ and w are parameters passed by reference to the function which respectively represent the array of coefficients and the weight of the constraint:

Algorithm 3.1 Apply division to a constraint

```
1:  procedure DIVISION(coef, w)
2:      nCoefs ← coef.size()
3:      if nCoefs ≤ 1 then return false
4:      GCD ← coef[0]
5:      for all i ∈ {1, ··· , nCoefs − 1} do
6:          GCD ← gcd(GCD, coef[i])
7:          if GCD == 1 then return false
8:      for all i ∈ {0, ··· , nCoefs − 1} do
9:          coefs[i] = coefs[i]/GCD
10:     w = ⌈w/GCD⌉
11:     return true
```

There are many possible places throughout the CDCL scheme to apply division. We are going to consider the following emplacements for applying division:

- **Learned clause**: Apply the division operation at the end of the conflict analysis procedure, to the clause that will be learnt.

- **During conflict analysis**: Apply the division operation during conflict analysis, to each new clause appearing from cutting planes resolution (Clashing Addition 2.3.1).

Both configurations will be tested and compared with the results of executions without the division operation implemented.

In the cdcl-cuttingplanes solver there are various options for configuring the execution. One of them involves rounding of the reason when performing the cutting planes step. This option rounds the reason in a way that it is divided by the coefficient to be resolved and then rounded. This could reduce the effectivity of division. For this reason, the experiments will be tested both with this rounding turned on and off. By default this setting is enabled.

Consequently, it is decided to test and compare 4 different configurations of division for the solver cdcl-cuttingplanes. All configurations are described bellow and the name of each subsection will be the one used to refer to them from now on. Note that the rounding of the reason is enabled by default, so if nothing is said it means it is turned on.

3.4.1 Original

This configuration corresponds to the solver as it was before the implementation of division on it.

3.4.2 Div1

Here division is only applied to the learnt clause, namely not applied during conflict analysis process.

3.4.3 Div2

Here the solver is configured to apply division to the learnt clause as well as during the conflict analysis.

3.4.4 Div3

Finally, for this configuration we have same settings as in *Div2* but turning off the rounding of the reason setting.

3.5 Benchmarks

Several benchmarks have been used in order to systematically test the performance of the different configurations of the implementation of division in the solver. These benchmarks are grouped in three different types of instances that codify three different problems about graph theory. These problems are: finding a *dominating set* of a given size, *even colouring* and finding a *vertex cover* of a given size. All of them are detailed in the following subsections.

3.5.1 Dominating Set

A dominating set of a graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that all vertices of the graph that are not in V' are adjacent to at least one of its vertices. In figure 3.1 some dominating sets of the graphs are highlighted in red.

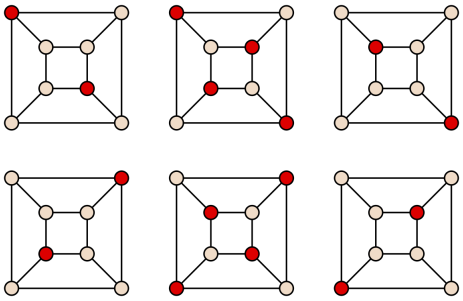


Figure 3.1: Dominating sets highlighted in red.

In particular the benchmarks used where instances codifying the dominating set problem for hexagonal grid graphs, an example of this is shown in picture 3.2. But particularly where the picture finishes the nodes are connected with the ones form opposite part in the picture, having in fact a 3-dimensional graph like the one in figure 3.3.

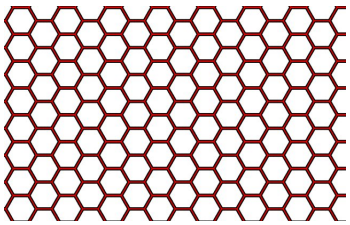


Figure 3.2: Hexagonal grid graph.

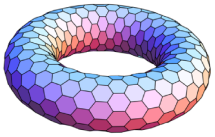


Figure 3.3: 3-dimensional hexagonal grid.

This graphs are represented as shown in figure 3.4, so that there is only needed two measures to define them, these are the height and width in terms of vertices, this will be respectively represented with m and n.

The size of the dominating set for the problems codified in the instances is expressed in terms of these m and n measures, $|DS| = m{\cdot}n/4$. It is important to notice that whenever this division has an integer as result it is possible that the instance is satisfiable (only sometimes).

Note that dominating set will be the only type of benchmarks used that has satisfiable instances, all others only contain unsatisfiable instances.

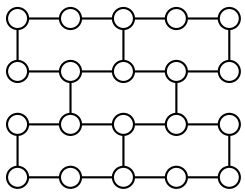


Figure 3.4: Representation of the hexagonal grids.

3.5.2 Even Colouring

The even colouring problem is a particular case of the edge colouring problem. The aim is to determine if given a graph $G = (V, E)$ there exists a $0/1$ coloration of edges $e \in E$, such that all vertices $v \in V$ have the same amount of adjacent edges of each colour.

The benchmarks codify the even colouring problem for random graphs.

Random Graphs

These graphs are randomly generated and they have two attributes that define each of them. The total number of vertices which is named n and the degree of each vertex, named deg.

There are two different values of degree among the instances: 4 and 6. For making instances with degree 4 unsatisfiable they all have an even number of vertices and there is one of the edges which is split into two inserting a vertex in the middle. For the instances with degree 6, just having an odd number of vertices it suffices to make them unsatisfiable.

3.5.3 Vertex Cover

A vertex cover of a graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that, for all edges $(u, v) \in E$ either $u \in V'$ or $v \in V'$ or both. In figure 3.5 vertex covers of the graphs are highlighted in red.

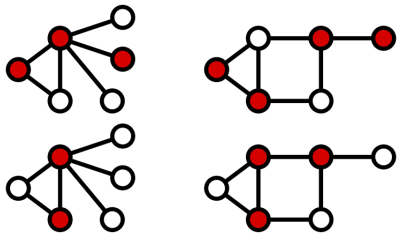


Figure 3.5: Vertex covers highlighted in red.

In particular the benchmarks used are instances codifying the vertex cover problem for regular grids. These are $m \times n$ size regular grids. For these graphs the minimum possible size for a vertex cover is $m/2 \cdot (n-1) + m$. Taking this into account the vertex cover size searched in the benchmarks is smaller so that all instances are unsatisfiable. Three different sizes of vertex cover are found among the instances. These are shown in table 3.1. For all families of instances ($v1$, $v2$ and $v3$) they have all an odd n (width of the gird).

Table 3.1: Vertex cover sizes.

name	Vertex Cover size
v1	$m \cdot \lfloor n/2 \rfloor$
v2	$m \cdot \lceil n/2 \rceil - 1$
v3	$m \cdot \lfloor n/2 \rfloor - 1$

Chapter 4

Results

In this chapter the results of the performance of the implementation of division on cdcl-cuttingplanes are shown. As explained in in section 3.4, 3 different configurations of the solver varying where division operation is applied are tested. The experiments will be carried out considering these 3 configurations plus the original solver implementation. Consequently, we have in total 4 different configurations of the solver so each benchmark will be used 4 times. In this chapter the names given to each configuration will be the same as in 3.4.

In the following sections the results for the different benchmarks are presented. Further details of the benchmarks and its characteristics are described in the section 3.5.

The results are divided in different families of instances, considering a family one of the three problems (i.e. dominating set, even colouring or vertex cover) with a specific configuration. Each family has a fixed value of m for dominating set and vertex cover and a fixed value of deg in case of the even colouring. And for each family the instances have an increasing n. The main idea is to observe the exponential growth respective to the value of n for each of the configurations of the solver.

For each family the running times of the executions and the number of conflicts for each instance are plotted in order to visually display the exponential growth. Then there is also a table showing the number of divisions performed every 1000 conflicts for each of the configurations of the solver in order to determine how often division is applied. For further details about the runtimes in seconds and the number of conflicts, in appendix A there are the tables showing the numbers for each execution, note that the shorter running times are highlighted in green.

4.1 Dominating Set m = 6

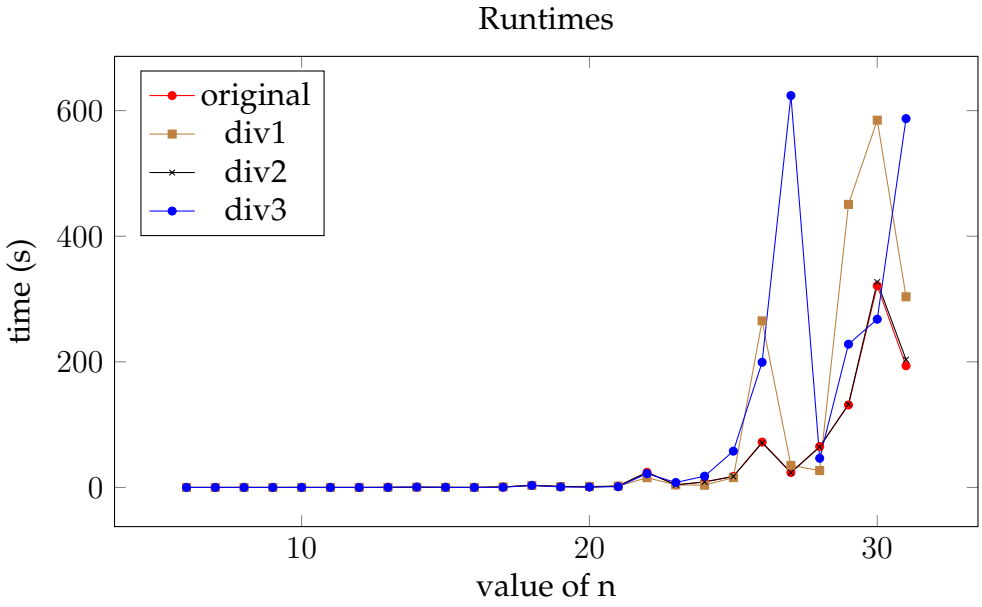


Figure 4.1: Comparison of runtimes for dominating set with m = 6.

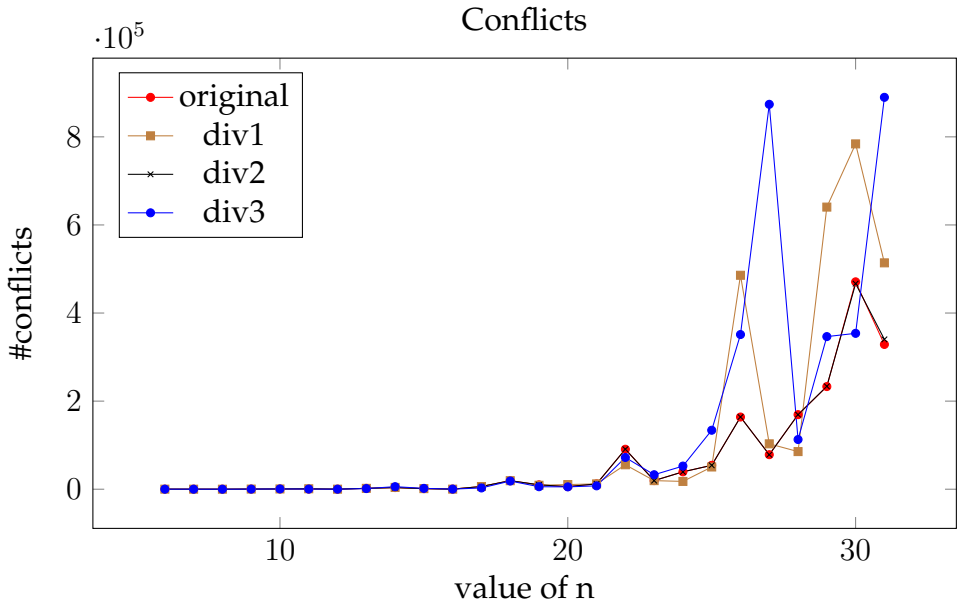


Figure 4.2: Comparison of conflicts for dominating set with m = 6.

Table 4.1: Number of divisions for each 1000 conflicts for dominating set with m = 6.

n	#Divisions / 1000 Conflicts		
	div1	div2	div3
6	0.00	0.00	0.00
7	0.00	0.00	0.00
8	0.00	0.00	0.00
9	0.00	0.00	0.00
10	0.00	0.00	0.00
11	702.59	702.59	0.00
12	0.00	0.00	0.00
13	1.03	3.51	0.00
14	0.74	0.83	0.52
15	0.78	0.78	1.06
16	0.00	0.00	0.00
17	0.16	0.16	0.00
18	0.10	0.15	0.38
19	0.45	0.30	0.35
20	0.38	0.17	0.00
21	0.00	0.00	0.26
22	0.05	0.25	0.06
23	0.26	0.05	0.15
24	0.17	0.20	0.10
25	0.04	0.07	0.02
26	5.12	0.15	0.01
27	0.11	0.11	0.01
28	9.05	0.21	0.15
29	2.37	0.09	0.07
30	3.13	0.07	0.03
31	6.74	0.14	0.05

4.2 Dominating Set m = 8

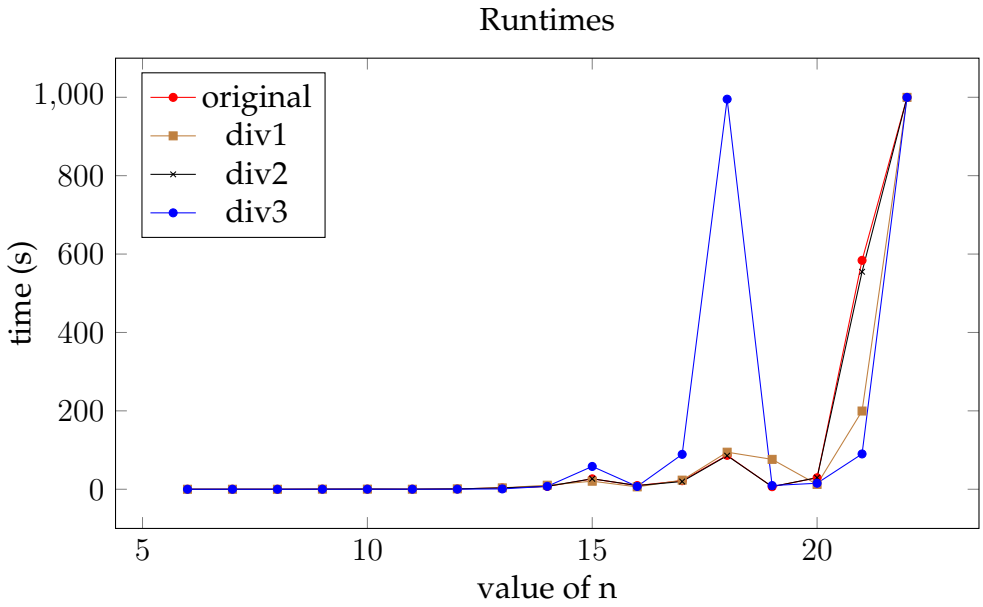


Figure 4.3: Comparison of runtimes for dominating set with m = 8.

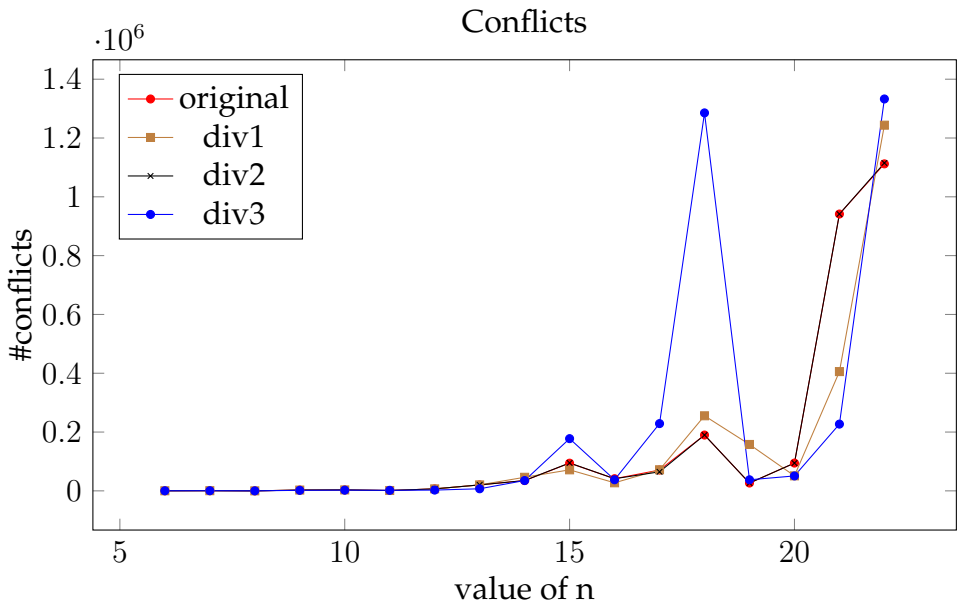


Figure 4.4: Comparison of conflicts for dominating set with m = 8.

Table 4.2: Number of divisions for each 1000 conflicts for dominating set with m = 8.

n	#Divisions / 1000 Conflicts		
	div1	div2	div3
6	0.00	0.00	0.00
7	0.00	0.00	0.00
8	0.00	0.00	0.00
9	0.73	1.09	0.00
10	0.00	0.00	0.00
11	0.00	0.00	0.00
12	0.00	0.00	0.38
13	0.10	0.10	0.00
14	0.13	0.20	0.00
15	9.15	0.17	0.03
16	7.40	0.07	0.21
17	0.03	0.06	0.04
18	0.79	0.03	0.00
19	0.08	0.11	0.00
20	0.02	0.22	0.04
21	2.18	0.05	0.09
22	1.18	0.05	0.02

4.3 Even Colouring random deg = 4

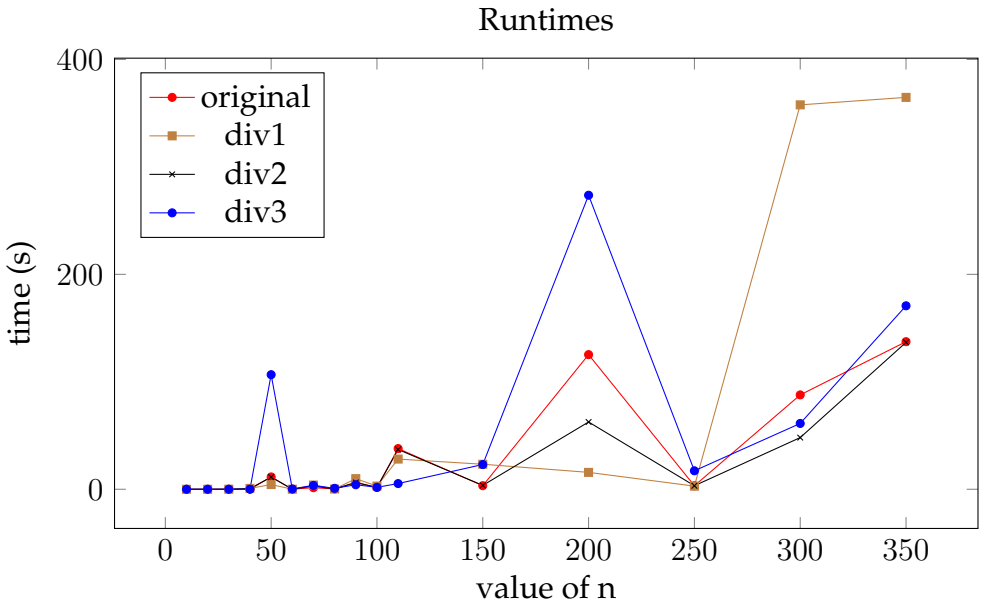


Figure 4.5: Comparison of runtimes for even colouring with deg = 4.

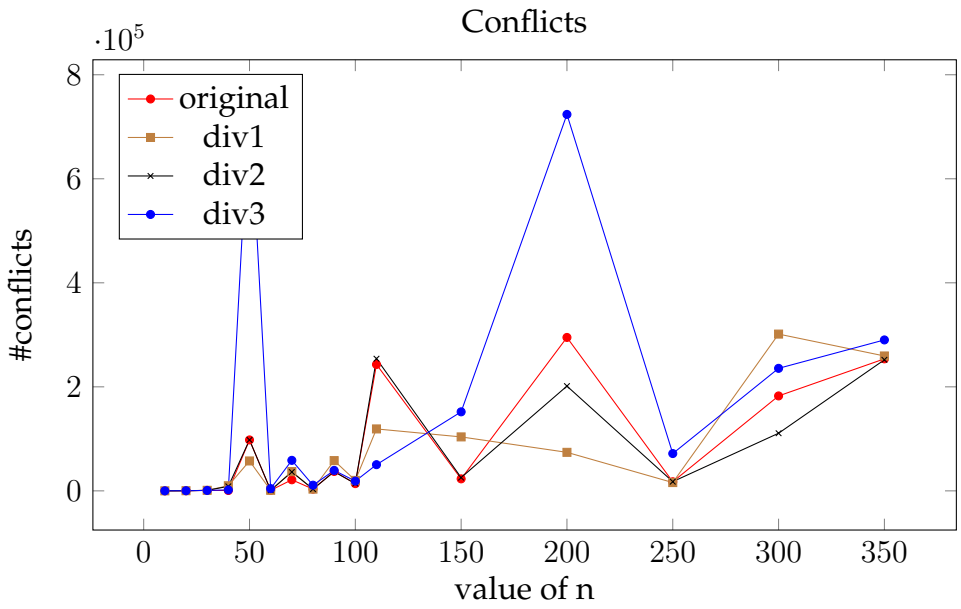


Figure 4.6: Comparison of conflicts for even colouring with deg = 4.

Table 4.3: Number of divisions for each 1000 conflicts for even colouring with deg = 4.

n	#Divisions / 1000 Conflicts		
	div1	div2	div3
10	44.44	68.18	68.18
20	20.76	46.81	51.72
30	11.97	18.79	44.58
40	18.00	57.57	7.11
50	3.27	4.82	3.04
60	8.29	8.26	10.26
70	81.29	65.90	1.57
80	2.07	2.07	14.65
90	71.45	133.04	2.77
100	50.05	15.32	7.29
110	122.30	24.83	3.42
150	76.66	25.63	1.59
200	1.61	4.78	0.54
250	1.13	12.66	0.25
300	39.00	1.89	0.11
350	3.84	4.07	0.32

4.4 Even Colouring random deg = 6

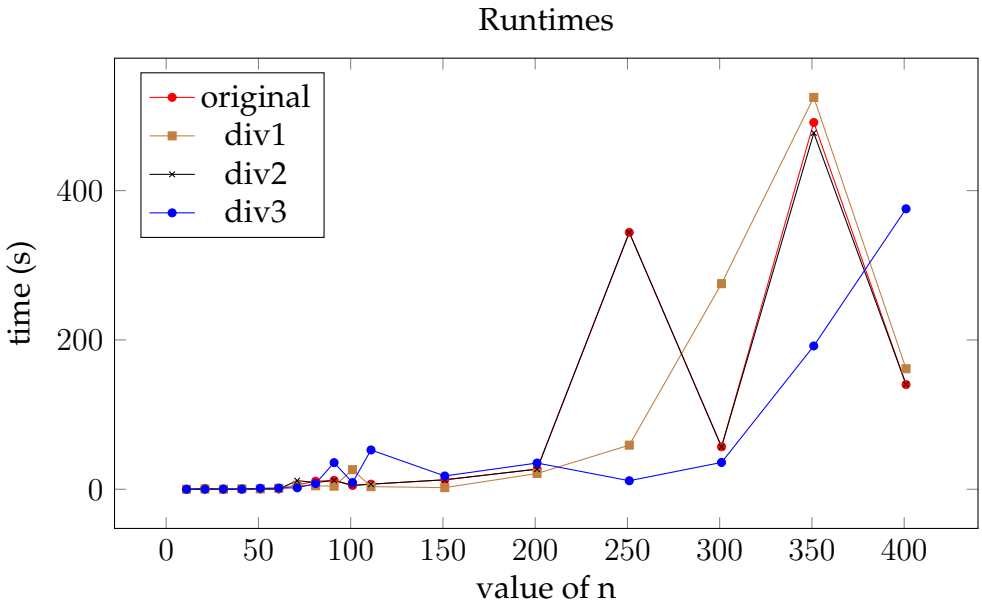


Figure 4.7: Comparison of runtimes for even colouring with deg = 6.

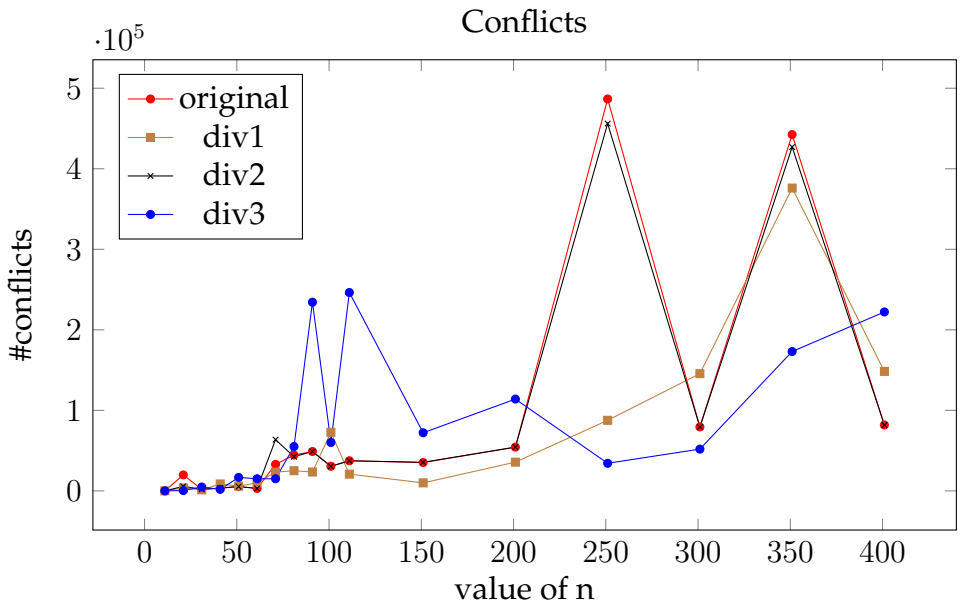


Figure 4.8: Comparison of conflicts for even colouring with deg = 6.

Table 4.4: Number of divisions for each 1000 conflicts for even colouring with deg = 6.

n	#Divisions / 1000 Conflicts		
11	25.00	51.28	51.28
21	9.06	62.54	24.72
31	30.53	22.02	4.36
41	34.79	58.40	8.45
51	79.28	56.47	3.94
61	203.88	12.39	1.26
71	251.96	243.82	6.38
81	42.40	26.25	2.23
91	6.06	9.51	1.75
101	52.47	8.60	0.12
111	96.68	30.27	1.56
151	7.63	25.55	1.84
201	5.52	7.90	0.54
251	35.64	0.65	0.55
301	3.17	9.18	0.62
351	7.73	3.32	0.80
401	6.61	4.96	0.11

4.5 Vertex Cover v1 m = 10

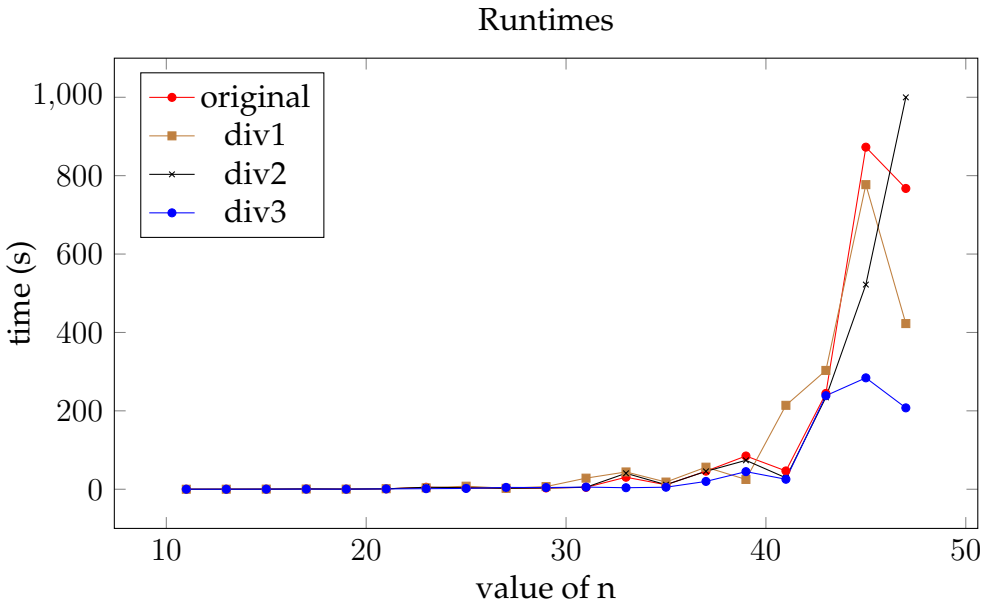


Figure 4.9: Comparison of runtimes for vertex cover v1 with m = 10.

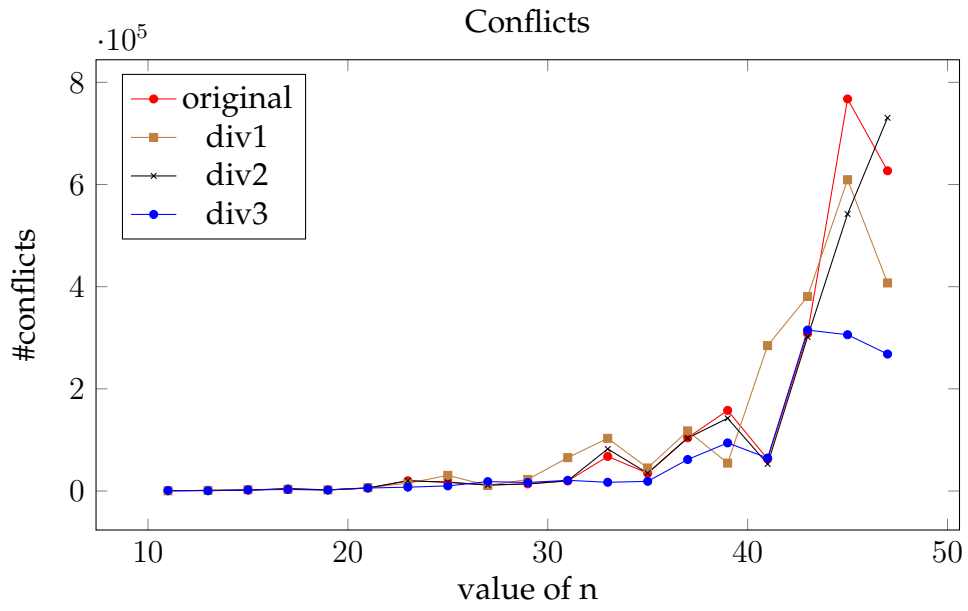


Figure 4.10: Comparison of conflicts for vertex cover v1 with m = 10.

Table 4.5: Number of divisions for each 1000 conflicts for vertex cover v1 with m = 10.

n	#Divisions / 1000 Conflicts		
11	0.00	0.00	4.46
13	0.00	0.00	0.00
15	0.37	1.18	1.33
17	3.62	3.74	1.93
19	1.82	3.52	0.00
21	2.68	1.69	2.04
23	1.19	0.40	1.58
25	0.46	0.69	1.09
27	1.29	0.77	2.06
29	1.15	0.29	0.90
31	0.60	0.74	1.00
33	1.33	0.89	1.28
35	0.80	1.06	1.58
37	0.48	1.39	0.83
39	0.62	0.90	0.32
41	1.07	1.45	1.38
43	0.54	0.52	0.48
45	0.42	0.45	0.93
47	0.44	0.28	0.73

4.6 Vertex Cover v2 m = 8

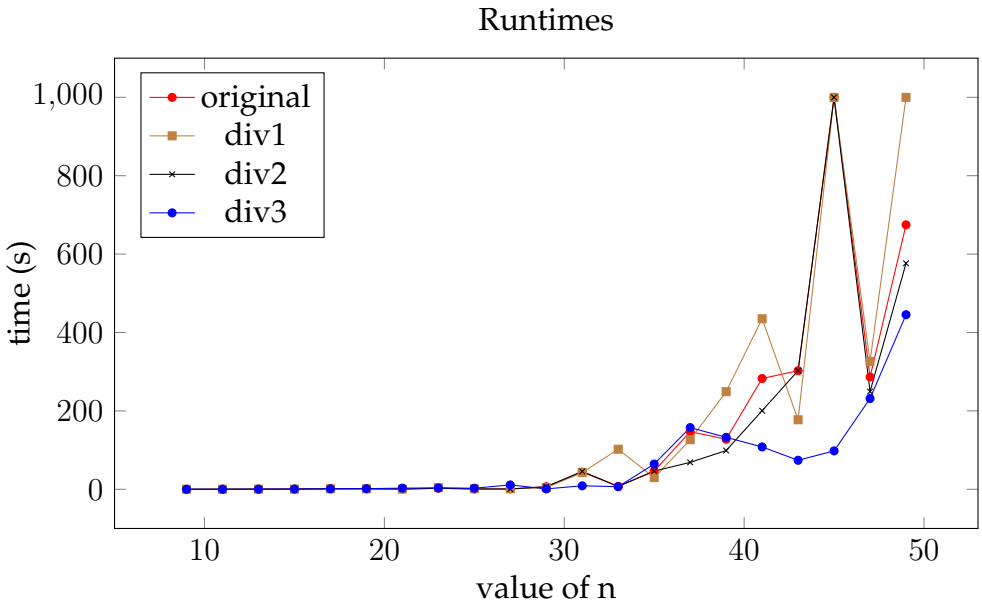


Figure 4.11: Comparison of runtimes for vertex cover v2 with m = 8.

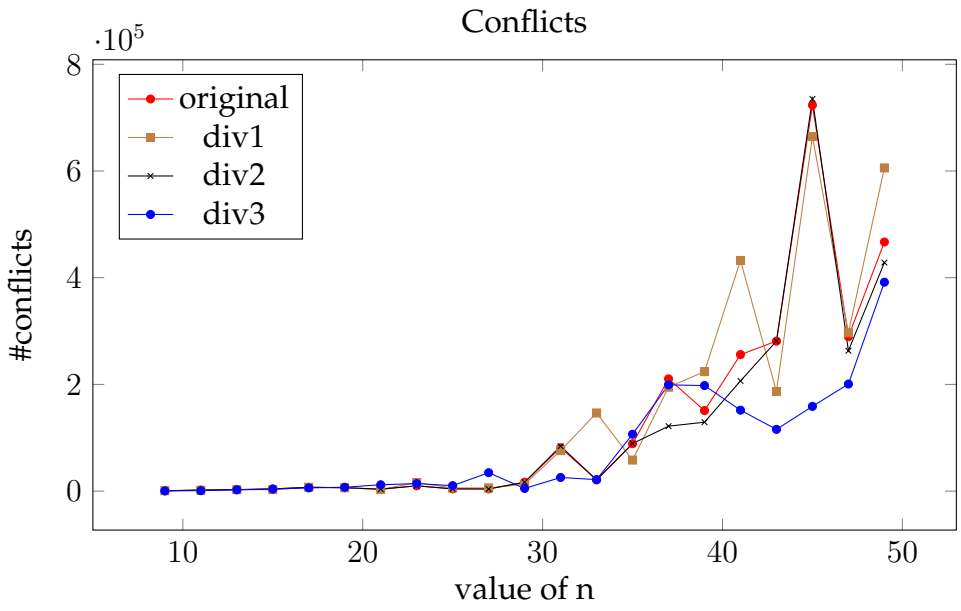


Figure 4.12: Comparison of conflicts for vertex cover v2 with m = 8.

Table 4.6: Number of divisions for each 1000 conflicts for vertex cover v2 with m = 8.

n	#Divisions / 1000 Conflicts		
9	18.13	15.20	21.38
11	3.92	10.54	7.54
13	4.67	9.20	7.46
15	13.46	7.80	7.35
17	4.14	5.89	5.98
19	4.37	5.97	4.56
21	4.62	13.02	3.53
23	2.81	3.84	4.21
25	8.15	6.02	3.03
27	10.64	11.22	2.09
29	4.49	4.20	3.21
31	1.75	2.11	2.62
33	1.46	2.38	2.83
35	1.65	1.86	1.08
37	0.84	1.64	0.83
39	1.11	1.12	0.50
41	0.79	0.99	0.93
43	1.00	1.04	0.62
45	0.68	1.08	1.21
47	1.39	0.83	1.98
49	0.57	0.77	0.70

4.7 Vertex Cover v3 m = 10

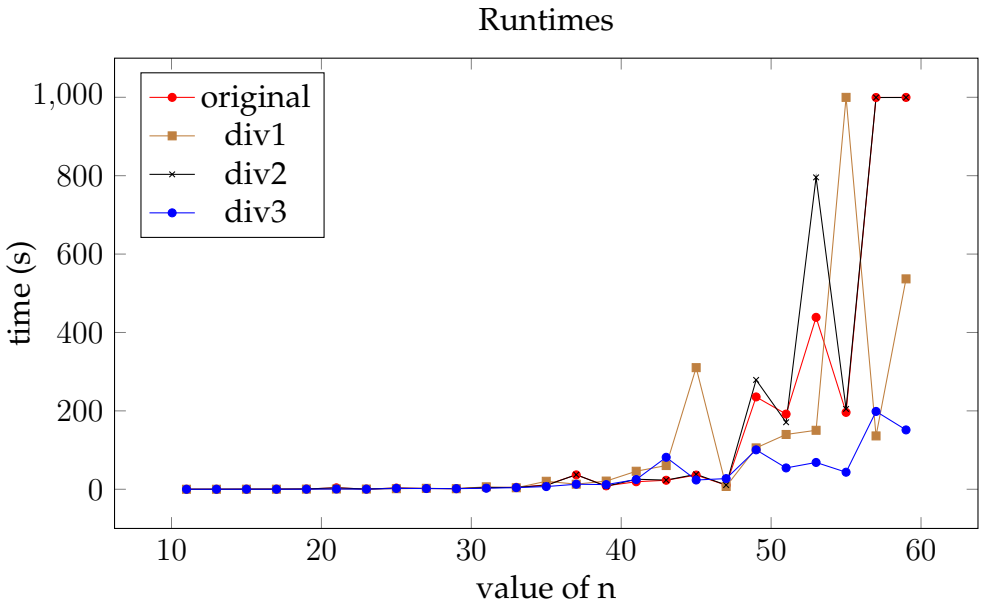


Figure 4.13: Comparison of runtimes for vertex cover v3 with m = 10.

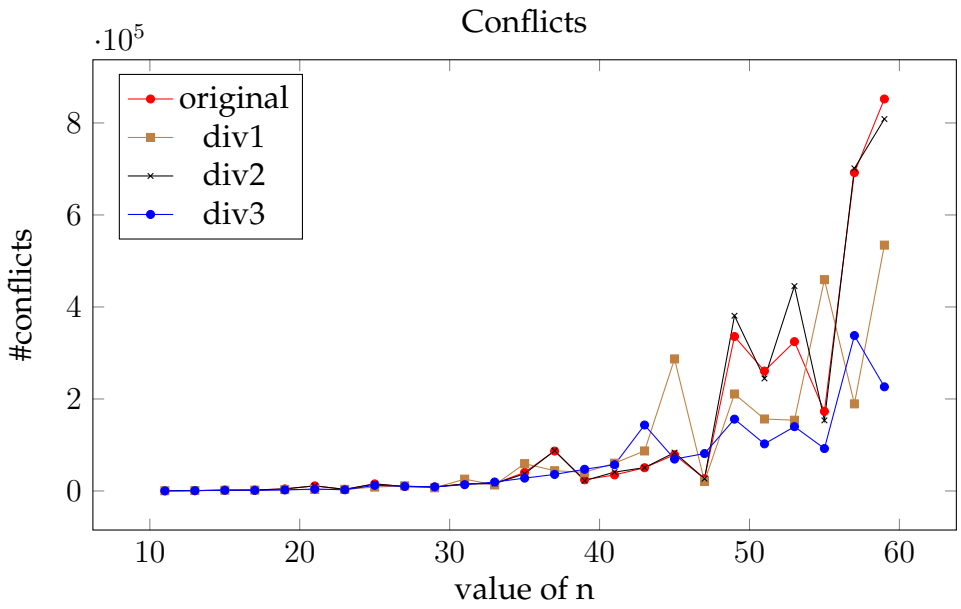


Figure 4.14: Comparison of conflicts for vertex cover v3 with m = 10.

Table 4.7: Number of divisions for each 1000 conflicts for vertex cover v3 with m = 10.

n	#Divisions / 1000 Conflicts		
11	0.00	0.00	0.00
13	0.00	0.00	0.00
15	2.46	4.54	0.64
17	3.82	0.95	0.75
19	1.35	1.10	0.46
21	0.64	0.90	0.24
23	1.14	2.56	0.39
25	0.95	0.46	0.41
27	0.99	0.94	1.95
29	0.72	0.93	1.12
31	0.58	0.97	0.80
33	0.92	1.43	1.34
35	0.62	0.48	0.75
37	0.14	0.22	0.50
39	1.01	0.58	0.57
41	0.48	0.59	0.82
43	0.76	0.61	0.49
45	0.22	0.60	1.36
47	1.07	0.52	0.41
49	0.19	0.12	0.28
51	0.62	0.49	0.28
53	0.49	0.41	0.17
55	0.35	1.83	0.35
57	0.22	0.23	0.22
59	0.09	0.17	0.30

Chapter 5

Discussion

In this chapter, it is presented the correspondent discussion to the results shown in the previous chapter. This chapter is structured in three different sections, one for each type of problem codified in the benchmarks (i.e. dominating set, even colouring and vertex cover). Since the results obtained are in general consistent throughout each problem to solve, they will be presented accordingly. Grouping the discussion of results for of each problem.

5.1 Dominating Set

5.1.1 Runtime and number of conflicts

In general, for the dominating set division does not seem to make an improvement. This is clearer for the family of instances having $m = 6$ where it can be seen that the configurations of the solver with division almost always perform at most as good as the original. Particularly in plots 4.1 and 4.2 it can be observed that both running times and number of conflicts of the div1 and div3 are normally above the performance of original, getting worse results. Whereas for the configuration div2 it gets similar results as the original, which is interesting. This can be also observed in the table A.3 of the appendix, where it is shown that div2 and original have really similar executions both in terms of conflicts and runtimes. But in case of the number of conflicts they both get the exact same number in several times.

In case of the family of benchmarks with $m = 8$ the results shown are less clear. Due to a high pick of div3 and the small amount of

instances is difficult to understand the behaviour of the solver. For this family the instances where harder for the solver, this is the reason why less instances are presented. Looking in detail at figure 4.4, at the last part of the plot could seem that the exponential growth of div3 is slower. However this seems to be a visual effect due to the point next to the last, because for the last execution both div3 and div1 grow over the original in terms of number of conflicts. Hence it is difficult to extract clear patterns for the dominating set $m = 8$. Nevertheless, what is also observed in this families of instances is that the results for div2 are also very similar to the ones from the original.

As a result of the performance of the different configurations seems that in case division is applied during the conflict analysis it is better to not disable the rounding options. As seen in the plots, div2 was performing nearly as the original having better performance than div3 in general. Also it is clear that performing division during conflict analysis is better than just at the end of it, as div2 performs also better than div1.

As it was explained in subsection 3.5.1 among the instances of dominating set some of them are satisfiable and others unsatisfiables. Taking this into account an interesting thing to observe was whether there is a correlation between being SAT or UNSAT with the efficiency of divisions. This was not the case. As explained previously, in general the performance of division can be seen as worse as the original version for dominating set and it does not have anything to do with being SAT or UNSAT.

5.1.2 Number of divisions

The results in terms of number of divisions are presented in the tables 4.1 and 4.2 for $m = 6$ and $m = 8$ respectively. This numbers show two interesting aspects. First, the number of divisions applied is relatively low. Compared to the number of conflicts for each of the executions.

Another interesting aspect about the number of divisions is that several times the highest value is the one of div1. This fact is quite interesting, since one would expect to be div2 and div3 larger than div1. As explained in 3.4, for div1 division is only applied at the end of the conflict analysis whereas for div2 and div3 if possible this is applied also within the analysis of the conflict.

5.2 Even Colouring

5.2.1 Runtime and number of conflicts

For the benchmarks encoding even colouring, there are two families of instances encoding the problem for random graphs $deg = 4$ and $deg = 6$. For neither of the families nor the runtimes nor number of conflicts are highly conclusive.

For $deg = 4$ it can be noticed that in general when an instance is hard it is specially hard for div3, since when a peak is found in general terms div3 has a higher peak. Analyzing the last part of the plot we can observe what appears to be an exponential growth. In the runtimes plot figure 4.5, we can say that div1 has an exponential growth faster than the rest. However this is not seen in the conflicts figure 4.6.

Taking into consideration the family of instances with $deg = 6$ the results are also mixed. However, if we were to draw the exponential curve of each of the configurations of the solver, div3 seems to have the slower growth. This can be better seen in the runtimes plot 4.7 but the pattern it is also repeated (although slightly) with number of conflicts, as observed in figure 4.8. It can also be said that div1 is the configuration that appears to have the faster exponential growth. And both div2 and original have very similar executions for this family of instances.

5.2.2 Number of divisions

In contrast to the results shown for dominating set 5.1.2, for even colouring the number of divisions seem to be larger in general. For both families of instances as shown in tables 4.3 and 4.4, the number of divisions is greater than for dominating set compared to the numbers of conflicts. Note that the number of conflicts is shown in the tables A.6 and A.7 of the appendix.

For these families of instances the pattern of the number of divisions of div1 being often the greatest is repeated. Having several instances in which div1 is the one performing more divisions compared to the number of conflicts.

5.3 Vertex Cover

5.3.1 Runtime and number of conflicts

The results for the vertex cover problem show that apparently division is improving the efficiency of the solver when tested with these benchmarks. Both in terms of runtimes and number of conflicts the exponential growth can be appreciated and div3 appears to have the best results. This is very clear for v2 with $m = 8$ in the plots in figures 4.11 and 4.12. Among the rest of configurations (div1, div2 and original) there is no clear conclusion for any of the families of instances tested, except that they seem worse in terms of execution than div3. These results are very clear for v2 but they can also be observed for v1 and v3.

Consequently, it can be said that in case division is applied during the conflict analysis it is better to disable the rounding options.

For vertex cover benchmarks it is also interesting to see that the performance of div2 is also very similar to the original's performance. This is replicated both in terms of runtime and conflicts.

5.3.2 Number of divisions

The number of divisions observed in tables 4.5, 4.6 and 4.7 show that here the number of divisions is larger in general than the ones from dominating set but smaller than the ones from even colouring, compared to the number of conflicts (shown in tables A.1, A.2 and A.5).

In this case, the pattern shown in dominating set and even colouring, in which the number of divisions for div1 was often larger than the rest, is not replicated. Here also there are instances in which this occurs but not as many as for the other problems.

Finally as a comment the number of divisions for v2 with $m = 8$ is larger in general than the ones for v3 with $m = 10$ and for v1 with $m = 10$ (tables 4.6, 4.7 and 4.5). Note that although the value of m is larger for v3 and v1 than for v2, in general v2 instances are harder to solve.

Chapter 6

Conclusion

Essentially the conclusions of this thesis are the following:

- The performance of the different configurations of division (i.e. div1, div2 and div3) depends on the instances that are solved.
- For dominating set the original version of the solver performs better than when division is applied (except with div2 that is almost the same).
- For dominating set if division is applied during conflict analysis it is better to enable the rounding options.
- For the even colouring benchmarks although div3 seems to have the lowest exponential growth (for $deg = 6$), it is not so clear. So for even colouring no clear conclusions can be extracted.
- For vertex cover division seems to improve the efficiency. Concretely with div3 configuration.
- For vertex cover if division is applied during conflict analysis it is better to disable the rounding options.
- As seen with the dominating set in 5.1 there appears to be no correlation between being SAT or UNSAT and the performance of division.
- When division is applied during conflict analysis but with the rounding options enabled (i.e. div2) the results are very similar in general (both in terms of runtime and number of conflicts) to the execution of the original.

- The number of divisions is in general low compared to the conficts, but this is even more clear for dominating set.

6.1 Future work

Pseudo-Boolean SAT solving is a wide field of study which still needs a lot of research to be fully understood. The race of SAT for efficiency keeps constantly upgrading so any possible improvements in the efficiency of a solver are always welcome. This thesis is an introduction to some of the research that can be done for improving the efficiency of CDCL solvers based on cutting planes, but still much work can be done. In section 3.2 there was a list of topics to be studied in this field (among others), from which this thesis only focuses in one (division). Future work extending this thesis could start doing more research on the other topics and implementing the results.

Bibliography

[1] Armin Haken. "The intractability of resolution". In: *Theoretical Computer Science* 39.C (1985), pp. 297–308. ISSN: 03043975. DOI: 10.1016/0304-3975(85)90144-6.

[2] Armin Biere et al., eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. {IOS} Press, 2009. ISBN: 978-1-58603-929-5.

[3] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM* 7.3 (1960), pp. 201–215. ISSN: 00045411. DOI: 10.1145/321033.321034.

[4] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (1962), pp. 394–397. ISSN: 00010782. DOI: 10.1145/368273.368557. URL: http://portal.acm.org/citation.cfm?id=368557.

[5] Albert Oliveras and Enric Rodr. *The DPLL algorithm Overview of the session Problem Solving w ./ Prop . Logic DPLL : A Bit of History*. 2009. URL: https://www.cs.upc.edu/%7B~%7Doliveras/LAI/dpll.pdf.

[6] Jakob Nordström. *Understanding Conflict-Driven SAT Solving Through the Lens of Proof Complexity*. 2016. URL: http://www.csc.kth.se/%7B~%7Djakobn/research/TalkProofComplexityLensCDCL.pdf.

[7] Daniel Le Berre. *Introduction to SAT*. 2014. URL: http://satsmt2014.forsyte.at/files/2014/07/SAT-introduction.pdf.

[8] Jakob Nordström. "On the Interplay Between Proof Complexity and {SAT} Solving". In: *ACM SIGLOG News* 2.3 (2015), 19\nobreakdash–44. URL: http://www.csc.kth.se/%7B~%7Djakobn/research/TalkInterplaySummerSchool2016.pdf.

[9] Laurent Simon. *Implementation of CDCL SAT Solvers*. 2016. URL: http://ssa-school-2016.it.uu.se/wp-content/uploads/2016/06/LaurentSimon.pdf.

[10] Joao Marques-Silva. *Introduction to SAT*. 2014. URL: http://ssa-school-2016.it.uu.se/wp-content/uploads/2016/06/jpms-satsmtar16-slides.pdf.

[11] Albert Oliveras. *From DPLL to CDCL SAT solvers*. 2009. URL: https://www.cs.upc.edu/%7B~%7Doliveras/LAI/cdcl.pdf.

[12] Lintao Zhang et al. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In: *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*. 2001, pp. 279–285. ISBN: 0-7803-7249-2. DOI: 10.1109/ICCAD.2001.968634. URL: http://dl.acm.org/citation.cfm?id=603095.603153.

[13] Donald Chai and Andreas Kuehlmann. "A fast pseudo-Boolean constraint solver". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.3 (2005), pp. 305–317. ISSN: 02780070. DOI: 10.1109/TCAD.2004.842808.

[14] F.A. A Aloul et al. "Generic ILP versus specialized 0-1 ILP: an update". In: *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.* (2002), pp. 450–457. ISSN: 1092-3152. DOI: 10.1109/ICCAD.2002.1167571. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1167571.

[15] Jan Elffers and K T H Royal. *Pseudo-boolean CDCL SAT solvers*. 2015.

Appendix A

Tables of execution times and conflicts

A.1 Vertex Cover v1 m = 10

Table A.1: Runtimes and #conflicts for vertex cover v1 with m = 10.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
11	0.024	0.024	0.024	0.092	427	427	427	1121
13	0.12	0.12	0.12	0.044	1302	1302	1302	546
15	0.22	0.364	0.216	0.26	1695	2710	1697	2255
17	0.7	0.528	0.776	0.38	4308	3311	4809	3116
19	0.324	0.32	0.328	0.26	2271	2193	2271	2046
21	1.012	1.328	1.012	0.86	5904	6723	5904	5872
23	5.092	3.588	5.124	1.448	20109	15119	20167	7600
25	3.724	7.804	3.72	2.008	17268	30685	17268	10058
27	2.476	2.388	2.484	4.664	11690	10822	11690	18461
29	3.208	6.784	3.22	4.284	13946	22628	13946	16698
31	4.78	28.3	5.028	5.46	19634	65356	20144	20964
33	30.64	44.128	40.696	3.872	67772	103171	82370	17137
35	11.428	18.336	11.536	5.136	34767	45194	34974	19000
37	46.112	56.292	45.628	19.948	104294	117691	103724	61740
39	84.816	24.984	73.736	44.912	157873	54752	142404	94222
41	46.716	214.116	29.112	25.484	62668	284702	52950	64639
43	244.384	303.144	234.384	238.724	308637	380850	301801	315084
45	872.696	777.056	522.128	284.2	767486	609287	542319	305972
47	767.396	422.624	999.756	207.592	626882	407325	730569	268084

A.2 Vertex Cover v2 m = 8

Table A.2: Runtimes and #conflicts for vertex cover v2 with m = 8.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
9	0.048	0.06	0.048	0.024	657	717	658	421
11	0.224	0.156	0.224	0.064	1991	1530	1993	796
13	0.34	0.316	0.34	0.308	2715	2567	2716	2546
15	0.576	0.36	0.58	0.556	4112	2748	4102	3807
17	1.368	1.34	1.42	1.14	7134	7255	7133	6022
19	1.392	1.076	1.276	1.576	6363	6174	6365	7234
21	0.6	0.432	0.604	2.92	3609	3029	3610	11901
23	2.5	4.188	2.54	3.368	9891	16026	9892	14012
25	0.796	1.196	0.796	2.708	4154	5887	4153	10237
27	0.776	1.328	0.696	11.128	4142	6014	3921	34502
29	6.548	4.092	6.592	0.968	16668	12925	16669	4986
31	43.732	42.448	45.608	8.832	81180	75804	84197	25531
33	7.324	102.224	7.34	6.676	20984	146467	20986	21560
35	46.1	29.724	46.392	64.368	88865	58193	89053	106642
37	147.1	126.412	68.632	157.38	210411	194845	121700	199168
39	127.7	248.988	98.792	132.5	150974	224031	129148	197816
41	282.592	434.944	200.412	108.048	255888	432285	206519	151666
43	302.468	177.268	302.644	73.98	281077	186779	281236	115701
45	999.728	999.756	999.664	97.924	723521	664532	734969	158793
47	286.376	326.096	249.328	231.6	289412	296761	263115	200676
49	674.668	999.844	576.508	445.344	466902	605769	428376	391567

A.3 Dominating Set m = 6

Table A.3: Runtimes and #conflicts for dominating set with m = 6.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
6	0	0.004	0.004	0.004	114	114	114	111
7	0	0.004	0.004	0	86	86	86	68
8	0	0	0	0	23	23	23	23
9	0.012	0.012	0.012	0.008	312	312	312	233
10	0.028	0.028	0.024	0.028	550	550	550	580
11	0.012	0.016	0.016	0.012	287	965	965	317
12	0.004	0.004	0.004	0.004	98	98	98	112
13	0.16	0.148	0.152	0.1	1988	1941	1993	1441
14	0.368	0.428	0.356	0.584	3716	4072	3600	5764
15	0.1	0.104	0.1	0.176	1283	1284	1284	1893
16	0.012	0.004	0.004	0.012	157	157	157	255
17	0.832	0.832	0.808	0.304	6126	6127	6126	2846
18	3.212	3.196	3.228	3.232	19438	19321	19438	18576
19	1.624	1.424	1.632	0.74	9922	8975	9922	5641
20	0.692	1.652	0.688	0.704	5881	10551	5881	5325
21	2.16	2.16	2.18	1.192	12064	12064	12064	7634
22	23.98	15.452	24.06	21.98	90760	55498	90751	72056
23	4.072	4.048	4.08	7.868	19562	19474	19562	32876
24	8.744	3.564	8.756	17.948	39433	17684	39433	52541
25	17.472	15.648	17.556	57.744	54265	50451	54265	133898
26	72.192	265.212	70.956	199.292	163893	485766	163895	351184
27	23.84	35.136	23.62	623.996	78400	103020	78400	873846
28	65.12	26.84	64.276	46.412	169309	85411	169309	112829
29	131.3	450.684	132.36	228.06	233021	640541	233614	346492
30	321.112	584.82	327.088	267.852	470841	784023	466739	353892
31	193.68	303.5	203.596	587.24	328691	513930	340167	889747

A.4 Dominating Set m = 8

Table A.4: Runtimes and #conflicts for dominating set with m = 8.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
6	0.008	0.008	0.008	0.008	253	253	253	254
7	0.028	0.024	0.024	0.02	549	549	549	509
8	0.004	0	0.004	0	31	31	31	31
9	0.216	0.24	0.24	0.14	2742	2747	2747	1823
10	0.308	0.312	0.348	0.228	3232	3232	3232	2418
11	0.104	0.1	0.1	0.164	1244	1244	1244	1838
12	0.924	0.96	0.896	0.244	6777	6777	6777	2658
13	3.496	3.5	3.508	0.84	20275	20277	20275	7106
14	7.22	10.336	7.312	7.444	34971	46197	35048	35165
15	26.536	20.516	26.776	58.208	94157	71495	94915	177593
16	9.44	6.044	9.448	7.068	41308	27429	41308	37797
17	21.796	23.244	19.704	89.044	70819	71731	64739	228763
18	86.516	94.548	85.676	995.14	189821	255296	189821	1285395
19	6.54	76.264	6.596	9.492	26616	157858	26616	37644
20	29.264	12.4	29.564	15.304	94473	51052	94473	50778
21	583.944	199.564	554.856	90.076	941477	405607	941477	227004
22	999.64	999.78	999.708	999.708	1112072	1243647	1114968	1332768

A.5 Vertex Cover v3 m = 10

Table A.5: Runtimes and #conflicts for vertex cover v3 with m = 10.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
11	0.032	0.032	0.032	0.012	436	436	436	265
13	0.068	0.072	0.072	0.068	816	816	816	674
15	0.228	0.208	0.228	0.168	1761	1624	1761	1573
17	0.288	0.24	0.312	0.144	2103	1834	2103	1335
19	0.688	0.396	0.736	0.252	4538	2959	4555	2170
21	3.688	0.584	3.676	0.632	11105	3140	11105	4137
23	0.528	0.5	0.552	0.292	3515	3500	3516	2563
25	3.08	1.38	3.004	2.148	15412	8452	15356	12184
27	1.936	2.428	1.98	1.848	9543	11109	9543	10234
29	1.62	1.352	1.672	1.536	8664	6957	8644	8939
31	4.244	6.276	4.312	2.604	15386	25761	15386	13668
33	4.92	3.736	4.8	4.212	16052	13069	16052	19472
35	11.112	20.392	9.892	6.844	40432	59368	37288	27909
37	36.692	13.016	37.268	12.848	86704	44003	87783	35942
39	8.732	20.92	7.564	11.9	24333	40569	22462	47108
41	19.216	45.976	25.596	24.8	34911	60438	40970	57007
43	22.972	60.512	23.36	81.536	50474	86990	50783	143404
45	36.252	310.244	38.112	23.644	78758	287068	83441	69293
47	10.396	7.132	10.752	27.204	26711	20594	26712	81350
49	235.572	105.58	278.984	100.548	335786	210820	380911	156128
51	191.788	139.784	170.924	54.416	260692	156473	244592	102371
53	438.596	150.484	795.704	68.428	324624	153790	445414	139817
55	196.14	999.812	204.972	43.512	172914	459342	153389	92255
57	999.46	136.08	999.476	198.588	692059	189316	701316	337863
59	999.692	536.752	999.46	151.412	852119	534459	808548	226415

A.6 Even Colouring random deg = 4

Table A.6: Runtimes and #conflicts for even colouring with deg = 4.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
10	0	0	0	0	43	45	44	44
20	0.01	0.01	0.01	0.01	228	289	235	290
30	0.04	0.04	0.04	0.03	956	1086	1011	830
40	0.03	0.82	0.61	0.1	687	9833	8667	1829
50	11.46	4.39	11.34	106.61	97931	57510	97921	753459
60	0.06	0.05	0.06	0.28	1087	1085	1089	4384
70	1.47	3.92	2.97	3.56	21309	37026	36311	58670
80	0.24	0.23	0.23	0.87	3378	3381	3380	10919
90	6.32	9.83	6.32	4.19	37265	58150	36891	39369
100	1.66	2.95	1.68	1.81	13902	20138	13904	18781
110	37.87	28.09	37	5.29	243023	119079	254122	50361
150	3.31	23.28	3.59	23.03	23087	103752	25512	151958
200	125.29	15.72	62.6	273.61	294974	74083	201443	723759
250	3.26	2.9	3.27	17.19	17484	15973	17698	71747
300	87.81	357.69	48.07	61.24	182530	301430	110790	235571
350	137.34	364.68	136.69	170.71	253805	259356	252014	290199

A.7 Even Colouring random deg = 6

Table A.7: Runtimes and #conflicts for even colouring with deg = 6.

n	Runtime (s)				#Conflicts			
	original	div1	div2	div3	original	div1	div2	div3
021	1.19	0.26	0.36	0.01	19855	3641	5660	445
031	0.13	0.07	0.13	0.22	1953	1310	1953	4821
041	0.21	0.63	0.21	0.12	3441	8565	3442	1893
051	0.38	0.43	0.38	1.27	5366	5966	5366	16731
061	0.24	1.21	0.25	1.60	3066	9947	3066	15078
071	5.78	5.42	11.59	1.90	32926	23270	63630	15058
081	10.83	4.58	8.92	7.69	44521	25026	42403	55106
091	11.79	4.15	11.82	35.64	48885	23448	48885	234293
101	5.10	26.17	5.13	8.88	30572	72766	30572	60093
111	6.80	3.52	6.81	52.61	37399	20821	37399	246365
151	12.63	2.15	12.69	17.77	35187	9957	35504	72171
201	26.97	21.06	26.94	35.14	54337	35713	54337	113989
251	344.01	59.08	343.95	11.41	486672	87645	456105	34278
301	56.86	275.38	56.79	35.91	79439	145550	79303	51901
351	491.31	524.95	477.27	191.97	442464	376029	427015	173053
401	140.25	161.39	140.50	375.66	81813	148375	81816	222190