MINIZSAT

A semi SAT-based pseudo-Boolean solver

Master thesis by
Rogier Poldner

MINIZSAT

A semi SAT-based pseudo-Boolean solver

Master thesis by
Rogier Poldner

committee:
Dr. H. van Maaren
Dr. M.J.H. Heule
Prof. Dr. C. Witteveen
Dr. P.G. Kluit

Acknowledgment

The development of MiniZSat did not happen overnight. Before starting with the actual development, research had to be done on both the Boolean satisfiability problem (SAT) and integer linear programming. Especially the complete understanding of SAT techniques took some time. Later on the pace picked up and MiniZSat was born.

We initially started with the facility location problem and the idea to improve performance on this specific problem by using a SAT solver as a basis. During the process we realized that this was very difficult and we tried to create a general purpose semi pseudo-Boolean solver. This resulted in the solver MiniZSat. The facility location problem still remains an integral part of the project. It became a case study of this thesis and it is extensively used to compare performance of various solvers.

MiniZSat is a product of Hans van Maaren, Marijn Heule and myself. I want to thank them for their great assistance and suggestions while working on my project. They guided and supported me to create the MiniZSat solver and deliver this thesis. I also would like to thank my family and friends for their support through out this period.

Abstract

This thesis explores the possibility to increase the performance of a SAT-based pseudo-Boolean solver. There are generally three groups of pseudo-Boolean solvers; SAT-based, ILP-based or a hybrid between both. A MiniSat-based solver, named MiniZSat, is created in an attempt to improve the performance of a SAT-based pseudo-Boolean solver. MiniZSat embeds the objective function in the solver itself. Various ILP techniques have been included in this solver, such as bounding. The usage of bounding allowed MiniZSat to prune large parts of the solution space and thereby increase the overall performance.

MiniZSat combined with its bounding techniques is able to outperform all solvers on random 3SAT instances. It is also capable of delivering a large improvement on the performance of SAT based pseudo-Boolean solvers on facility location problem instances. ILP (based) solvers however still remain the best performer in this category. Due to the improvements MiniZSat was able to solve a large number of divers problem instances from the pseudo-Boolean evaluation.

Contents

1 Introduction **1**
- 1.1 Problem classes . . . 1
  - 1.1.1 Integer linear programming problem . . . 1
  - 1.1.2 Boolean satisfiability problem . . . 2
  - 1.1.3 Pseudo-Boolean problems . . . 2
  - 1.1.4 Expressiveness . . . 3
- 1.2 Increasing performance of SAT based pseudo-Boolean solvers . . . 3
  - 1.2.1 Embedding the objective function . . . 4
  - 1.2.2 Bounding technique . . . 5
  - 1.2.3 Bound conflict analysis . . . 5
  - 1.2.4 MiniZSat . . . 6
- 1.3 Outline . . . 6

2 Integer linear programming (ILP) **7**
- 2.1 Linear programming . . . 7
  - 2.1.1 Problem class . . . 9
  - 2.1.2 Solving methods . . . 10
- 2.2 Description . . . 11
- 2.3 Complexity . . . 11
- 2.4 Solving techniques . . . 12
  - 2.4.1 Branching . . . 12
  - 2.4.2 Bounding . . . 12
  - 2.4.3 Cutting planes . . . 13
- 2.5 Solving algorithms . . . 13
  - 2.5.1 Branch and bound . . . 13
  - 2.5.2 Branch and cut . . . 13
  - 2.5.3 Branch and price . . . 16
- 2.6 Search options . . . 16
  - 2.6.1 Node selection . . . 17
  - 2.6.2 Branching . . . 17
  - 2.6.3 Cutting planes . . . 17
  - 2.6.4 Preprocessing . . . 17
  - 2.6.5 Primal heuristics . . . 17
- 2.7 Solvers . . . 18
  - 2.7.1 Commercial solvers . . . 18
  - 2.7.2 Noncommercial solvers . . . 19

3 SAT **21**
3.1 Description . . . 21
3.2 Complexity . . . 22
3.3 SAT solvers . . . 22
3.3.1 DPLL Framework . . . 22
3.3.2 Conflict driven solvers . . . 22
3.3.3 Lookahead solvers . . . 23
3.4 Internals of a SAT solver . . . 24
3.4.1 Searching . . . 24
3.4.2 Propagation . . . 25
3.4.3 Learning . . . 26
3.4.4 Backtracking . . . 28
3.5 Solving an example . . . 28
3.6 Variations between solvers . . . 29
3.6.1 Preprocessing: resolution . . . 30
3.6.2 Searching: randomized restarts . . . 30
3.6.3 Learning: assignment stack shrinking . . . 30
3.6.4 Learning: conflict clause minimization . . . 30
3.7 MiniSat . . . 31
3.7.1 Conflict analysis . . . 31

4 Pseudo-Boolean **33**
4.1 Linear pseudo-Boolean constraints . . . 33
4.2 Complexity . . . 34
4.2.1 Expressiveness of pseudo-Boolean constraints . . . 35
4.2.2 Comparing problem classes . . . 35
4.3 Solving techniques . . . 36
4.3.1 Constraint operations . . . 36
4.3.2 Constraint propagation . . . 37
4.3.3 Conflict analysis . . . 38
4.3.4 Learning scheme . . . 38
4.4 Solvers . . . 38
4.4.1 Hybrid pseudo-Boolean solver Pueblo . . . 39
4.4.2 SAT based pseudo-Boolean solver MiniSat+ . . . 39

5 Facility location problem **41**
5.1 Description of variants . . . 41
5.2 Problem class . . . 42
5.3 General approximation techniques . . . 42
5.3.1 LP-rounding . . . 43
5.3.2 Primal-dual method . . . 43
5.3.3 Dual fitting . . . 45
5.3.4 Local search . . . 45
5.3.5 Greedy augmentation . . . 45
5.4 Approximation solving algorithms . . . 45
5.4.1 JMS algorithm . . . 46
5.4.2 MYZ algorithm . . . 46
5.5 ILP formulations of facility location variants . . . 47
5.5.1 Uncapacitated facility location . . . 47
5.5.2 Capacitated facility Location . . . 47

5.5.3 The linear-cost facility location problem . . . 48
5.5.4 The metric k-median problem . . . 48
5.5.5 Splittable demands . . . 49

6 MiniZSat **51**
6.1 Introduction . . . 52
6.2 Embedding the optimization function . . . 52
6.3 Bounding techniques . . . 53
6.3.1 Using a known upper bound . . . 54
6.3.2 Initial lower bound . . . 54
6.3.3 Dynamic lower bound . . . 58
6.4 Bound conflict analysis . . . 61
6.4.1 Bound conflict reduction . . . 63
6.5 Efficient traversal of the search space . . . 64
6.5.1 Search strategy . . . 65
6.5.2 Restart strategy . . . 66
6.6 Using the solver . . . 67
6.6.1 Problem definition . . . 69
6.6.2 Input format . . . 69
6.6.3 Translating from pseudo-Boolean definition . . . 69

7 Results **71**
7.1 Introduction . . . 71
7.1.1 MiniZSat variants . . . 71
7.1.2 Solver and running information . . . 71
7.1.3 Notation . . . 72
7.2 Facility location . . . 72
7.2.1 Lower bound techniques . . . 73
7.2.2 Solver comparison . . . 74
7.3 Random 3SAT . . . 74
7.3.1 Lower bound techniques . . . 74
7.3.2 Solver comparison . . . 76
7.4 Pseudo-Boolean evaluation 2007 . . . 78
7.4.1 Lower bound techniques . . . 78
7.4.2 Solver comparison . . . 80

8 Conclusion and Future work **83**
8.1 MiniZSat techniques . . . 83
8.1.1 Embedding the minimization function; MiniZSat vs MiniSat+ . . . 83
8.1.2 Bounding techniques . . . 84
8.1.3 Bound conflict analysis . . . 84
8.2 Application areas for MiniZsat . . . 84
8.3 Future work . . . 84

Chapter 1

Introduction

> *Computers can figure out all kinds of problems, except the things in the world that just don't add up.* — James Magary

In normal day life every person encounters a large number of problems, these may differ from choosing a pair of shoes to deciding which investment should be made. A large group of problem is very complex, hundreds of computers working day and night might not even find the solution.

Imagine a supplier to facilitate a large number of customers through a country. In order to deliver the products as cheap as possible, certain warehouses need to be placed from which products will be transported to the customers. The problem of choosing and positioning the warehouses is a complex problem.

The total costs for delivering the products are defined as the cost for the placement of the warehouses and the transportation cost. A supplier which needs to facilitate 20 customers and has 10 possible warehouse locations, already has billions of options to choose from. The number of options is defined by f^l, where f refers to the number of possible warehouse position and l the number of customers to server. It can easily be seen that this problem grows rapidly. This problem is also known as the *facility location problem.*

1.1 Problem classes

The facility location problem belongs to a certain problem complexity class called $\mathcal{NP}$-hard problems. This class of problems is currently only solvable in exponential time with respect to the input. Whether algorithms exist that can solve $\mathcal{NP}$ instances within polynomial time remains unknown.

Certain problems can be solved while using another problem. In theory all $\mathcal{NP}$-complete problems can be solved by using any $\mathcal{NP}$-hard problem. However solving a problem using another problem may not be effective, since specific problem information will be lost.

1.1.1 Integer linear programming problem

An important group of problems are the *optimization problems.* An optimization problem is the problem of finding the *best feasible* solution. In most cases multiple feasible solutions are available in an optimization problem, the goal is to find the best solution among them.

Linear programming problems are a group of optimization problems in which a linear objective function and linear constraints define the problem. A solution to the problem is an assignment of variables which comply to the linear constraints. The objective function represents the quality of the solution, e.g.

in the case of a minimization function the solution with the smallest value for the objective function is the best solution.

A special instance of the linear programming problem is *integer linear programming*. Integer linear programming is linear programming with the restriction that variables may only have integer values. Integer linear programming is a $\mathcal{NP}$-hard problem and finding the optimal solution may be a time consuming process. A large number of integer linear programming solvers exist and they have been successful on several problem instances.

There is a large group of problems which may be solved by using integer linear programming. One of them is the previously denoted facility location problem, which will also be the topic of a case study in this thesis.

1.1.2 Boolean satisfiability problem

An important problem often used to solve other problems with is the *Boolean satisfiability problem*. This problem is a *decision problem*, in which the question is whether an assignment of variables can evaluate a Boolean formula to `TRUE`. The Boolean formula consists of AND, OR, NOT-operators, binary variables and parentheses.

In recent years SAT solvers have gained significant increase in performance and applicability. An important application area for SAT solvers is formal verification. Verification of both applications and hardware design can be encoded in a Boolean satisfiability problem. This application area is of increasing interest, because applications and hardware become larger and more complex. Verification is essential in order to guarantee the correctness of hardware and software, especially for dependable companies such as banks and airplane aviation. The Boolean satisfiability problem is also frequently used in complexity theory. It was for example the first decision problem proven to be $\mathcal{NP}$-complete.

Performance of SAT solvers has been compared in SAT competitions [1]. A large number of solvers have been submitted for these competitions. One of the solvers performing very good in different categories in the latest editions is the solver MiniSat. MiniSat is a relatively simple and good extensible solver[2]. MiniSat will be further introduced in Chapter 3.

1.1.3 Pseudo-Boolean problems

Pseudo-Boolean problems are in a problem class between the integer linear programming problem and the SAT problem. Pseudo-Boolean problems have the same linear constraints as integer linear programming. The only difference between integer linear programming and pseudo-Boolean problems is that only binary variables are allowed for pseudo-Boolean problems.

The peudo-Boolean problem can be either a decision or an optimization problem. The difference is the absence of an objective function in the pseudo-Boolean decision problem. The pseudo-Boolean decision problem is closely related to the Boolean satisfiability problem, where only the constraints differ. The equivalent of a constraint in a SAT problem is a clause. A clause consists of an OR-combination of variables with or without negations. An example of a clause is $x_1 \vee x_2 \vee \neg x_3$, which has the following pseudo-Boolean equivalent $x_1 + x_2 - x_3 \geq 0$.

Pseudo-Boolean solvers are relatively new solvers. They are largely based on knowledge and techniques acquired from integer linear programming solvers and SAT solvers. As denoted earlier pseudo-Boolean solvers can solve SAT problems, however they are generally not designed to solve this problem and the solvers will therefore generally perform less on SAT problems. On the other hand there exist a large number of problems solvable in pseudo-Boolean representation, which may be more suited for the pseudo-Boolean solvers.

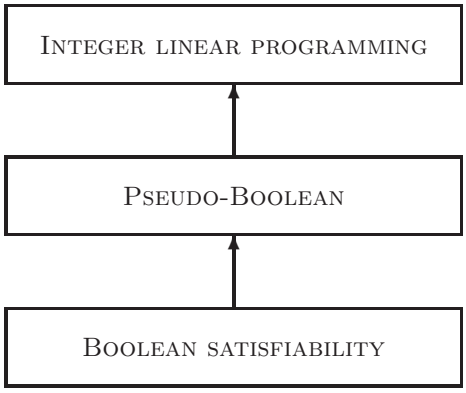


Figure 1.1: Comparing expressiveness of problem classes

1.1.4 Expressiveness

Figure 4.1 shows the relationships between the denoted problem classes. The integer linear programming problem is the most expressiveness, e.g. it has the most compact representation of a problem. Pseudo-Boolean problems are less expressive, because it only allows binary variables. A problem requiring integer variables should be rewritten to fit the pseudo-Boolean problem, which in general generates an increase in the size of the problem.

The Boolean satisfiability problem is the least expressiveness. Boolean satisfiability problems only allow binary variables and it has a simpler form of constraints (clauses). Translating one pseudo-Boolean constraint to a SAT representation may require a large number of clauses, which increases the size of the problem. The increase in size of the problem may in turn result in a decrease in performance.

1.2 Increasing performance of SAT based pseudo-Boolean solvers

A pseudo-Boolean decision problem can be converted to a SAT problem by rewriting the constraints. However the expressiveness of a pseudo-Boolean constraint is larger than the expressiveness of a SAT clause. The translation of a pseudo-Boolean clause to SAT could therefore generate a large number of clauses, which can in turn degrade the performance of a SAT solver.

MiniSat+ is a pseudo-Boolean solver that translates the pseudo-Boolean problem to a SAT problem [3]. MiniSat+ uses MiniSat to solve the translated problem. It is developed by Niklas Eén and Niklas Sörensson.

MiniSat+ can both solve the decision and the optimization variant of the pseudo-Boolean problem. In order to find the optimum solution, the solver first tries to find a solution. When the solver finds a solution, it adds an extra constraint which states that the objective function value should be smaller than the value of the solution found. A complete assignment is a new solution with a smaller cost than the current best solution. The algorithm stops when the problem is unsatisfiable, which means that the last found solution is the optimum solution.

The performance of MiniSat+ was reasonably good in the pseudo-Boolean evaluation [4] of 2006 and 2007. The pseudo-Boolean evaluation is created to assess the state of the art in the field of pseudo-Boolean solvers. MiniSat+ belongs to the best solvers in this evaluation, concerning the number of times it can find the optimum solution. This may indicate that the initial approach of using a SAT solver is in principle not a bad idea.

The question remains whether this approach can be improved. In this section a selection of possible improvements for a SAT based pseudo-Boolean solver will be presented.

Table 1.1: Extra clauses due to translation of the minimization function in MiniSat+.

	ORIGINAL			AFTER		
	clauses	**literals**	**ratio**	**clauses**	**literals**	**ratio**
5xp1.b	843	29,887	35.45	19,340	72,979	3.77
9symml	1,552	6,592	4.25	240,640	564,833	2.35
addm4	832	5,389	6.48	68,835	164,187	2.39
aim	260	679	2.61	6,908	16,243	2.35
bogr	420,966	998,490	2.37	420,970	998,501	2.37
circ10_3	135,760	393,530	2.90	151,169	448,557	2.97
domset	19,197	54,425	2.84	36,934	96,154	2.60
frb30-15-1	29,091	62,938	2.16	29,136	63,155	2.17
g15	7,778	18,732	2.41	7,874	18,967	2.41
market-split	11,613	29,956	2.58	N.A.	N.A.	N.A.
mps	19,873	46,623	2.35	21,179	49,678	2.35
opt-market	5,466	19,390	3.55	7,614	24,386	3.20
simp	117,844	284,458	2.41	122,960	296,426	2.41
ss97	163,148	532,482	3.26	N.A.	N.A.	N.A.
vtxcov	4,000	8,000	2.00	141,152	328,903	2.33
wnq	656,900	1,333,400	2.03	1,023,467	2,642,587	2.58
fac_20-40	4,480	11,200	2.50	44,860	155,436	3.46
3sat_450-150	450	1,350	3.00	94,296	220,685	2.34
3sat_600-150	600	1,800	3.00	92,039	215,434	2.34

1.2.1 Embedding the objective function

After MiniSat+ finds its first feasible solution, it adds a number of clauses ensuring that a new solution will be smaller than the first solution. This can be ensured by adding clauses which translate the objective function and state that this objective function value should be smaller than the value of the first solution.

Table 1.1 shows the number of clauses, literals and the literal - clause ratio in the original problem and after the addition of the clauses for the objective function. For the entries containing N.A. there was either no need for a translation of the minimization function or no solution could be found within the timeout of 1800 seconds used and therefore no results are available.

Certain instances in Table 1.1 only show a small increase (up to 115 %) in the number of clauses and literals, where other instances show a massif increase of almost 400,000 clauses or a relative increase of 200 times the original size! The number of terms in the minimization function plays an important role in the increase in clauses and literals. A small number of terms in the minimization function requires a small number of extra clauses and literals. The increase is also relatively small when the total number of original clauses is already large.

It is interesting to see that for certain instances the literal - clause ratio shows a large decrease after the translation of the objective function. This means that the added clauses are relatively small and therefore reasonably simple to maintain. The 3SAT instances for example show an original rate of 3, and a ratio of 2.34 after the translation, which suggest that a large number of small clauses (containing one or two literals) is added. The relatively simple new clauses ensure that the performance will not deteriorate too much.

A possible performance increase could be acquired by embedding the objective function in the solver

and not in the clauses. By maintaining the objective function value while solving a problem, the optimal solution can be found by discarding assignments which will never yield the optimal result. The advantage of this method opposed to embedding the objective function in the clauses is, that the total number of clauses will be (largely) reduced.

Embedding the objective function may especially be interesting for instances that show a large increase in size due to the translation of the minimization function in to clauses. On the other hand if the increase is small the effect may be minimal.

Another advantage may be related to the time to process a new solution. MiniSat+ requires a new translation of the objective function to guarantee that a new solution is better than the current solution. Embedding the objective function has no such requirement, it simply updates the best solution and continues the search for a better solution.

1.2.2 Bounding technique

Bounding is a technique which can find upper and lower bounds for the optimal solution in a subset of possible solution. Embedding the objective function actually corresponds to a form of upper bound technique. The upper bound in a minimization problem is defined by the best known solution. Assignments are discarded when their objective value is equal or larger than the upper bound, because an assignment equal or larger than the best known solution will never result in a the optimum solution.

The lower bound is the limit for each solution in the subset, which means that no solution in the subset will be smaller than this limit. If the lower bound is equal or larger than the upper bound, then the subset can be eliminated since it can not generate the optimum solution.

The idea while using bounding technique is to try and converge the lower and upper bound as soon as possible. Upper bounds can be improved by finding new improved solutions and lower bounds can be improved by investigating parts of the solution space. Improved upper and lower bounds decrease the size of the solution space to search through, because improved bounds lead to more assignment that can be discarded.

Bounding technique is extensively used in integer linear programming algorithms. All the current state-of-the-art solvers are based on this technique (see Section 2.4 and Section 2.5). This means that the SAT-based pseudo-Boolean solvers becomes closer to an ILP solver.

1.2.3 Bound conflict analysis

Whenever the lower bound equals or exceeds the upper bound, the current assignment can be discarded. In this thesis the state of the solver is referred to as in *bound conflict*. A bound conflict is caused by variables which increased the objective function value, the other variables play no role in the conflict. The bound conflict can be threated as a *logical conflict*, containing the variables which increased the function value in its reason. A logical conflict is 'the default' conflict of a SAT solver, in which a variable is supposed to be `TRUE` and `FALSE` at the same time.

The idea behind bound conflict analysis is to find the cause of the conflict and prevent the solver from running in to the same conflict again. The solver can prevent the recurrence of the conflict by learning from the conflict. Learning uses the cause of the conflict and it resolves this conflict by adding a clause which prohibits the assignment from reappearing. The addition of a clause in effect decreases the size of the search space the solver needs to explore. A decrease in the size of the search space in turn leads to better performance because the number of options to check also decreases.

It is therefore essential for the performance that the solver is as intelligent as possible and has good learning capabilities. The learning capabilities can be improved by reducing the bound conflict. A bound conflict can be reduced whenever certain variables in the conflict are non-essential. A variable in a bound conflict is non-essential if the variable can be removed and the bound conflict remains. This occurs whenever a variable has a coefficient smaller or equal to the gap between current solution value and best

known solution value. If for example the current solution value is 21 and the best known solution is 11, then a variable with a coefficient smaller or equal to 10 is not essential.

After the removal of the not essential variable the current solution value is still larger or equal to the best known solution and therefore the conflict remains. The removal of non essential variables in a bound conflict is useful because the conflict is caused by a smaller group of variables. This leads in turn to a stronger learnt constraint, which prohibits more assignments and it thereby decreases the size of the solution space to search through.

1.2.4 MiniZSat

This leads to a question which solver technique best suit a pseudo-Boolean solver. Should they be derived from an ILP solver or a SAT solver or both? In this thesis the particular focus is on the question whether a SAT based pseudo-Boolean solver can lead to good performance. This resulted in the development of a new solver based on MiniSat called MiniZSat. MiniSat was chosen as basis for its easy extensibility and the good performance it delivers.

The previously denoted improvements have been employed in the new solver MiniZSat. A detailed description of the implementation of the various techniques will be presented in this thesis. The advantages and disadvantages of the improvements have been compared and performance analysis is done. Comparison between MiniZSat and other solvers will be presented in order to compare the general performance of MiniZSat opposed to other solving techniques.

1.3 Outline

In order to gain understanding on the type of problems used in this thesis, several problem classes will be first introduced. In Chapter 2 integer linear programming (ILP) problems will be introduced. This problem class is closely related to linear programming which will also be introduced in this chapter. Chapter 3 introduces the Boolean satisfiability (SAT) problem. Pseudo-Boolean solvers are closely related to both integer linear programming and boolean satisfiability solvers. Pseudo-Boolean problems are therefore introduced after ILP and SAT problems in Chapter 4. In these introductions a description of the problem type and methods to solve these problems will be given. The focus lies on how to solve a problem and particularly the techniques used by solvers.

A case study on solving the facility location problem (deterministically) is also included in this thesis. The goal is to discover whether non-ILP solvers can be used effectively to solve facility location problems. In order to do so the first step is to analyze the currently available solving methods. An important part is the representation of the facility location problem in the pseudo-Boolean formulation. This allows a large group of solvers to handle the problem, including ILP solvers. The case study is available in Chapter 5.

The MiniZSat solver is presented in Chapter 6. The various improvements are introduced and carefully explained. The improvements are supplemented by results obtained from the solver in order to illustrate the performance changes due to the improvements.

The last chapter contains comparisons of the various MiniZSat variants. It also includes comparisons between various solvers on problem instances. Different problem instances, such as random 3 SAT with minimization function, facility location problem and instances from the pseudo-Boolean evaluation are used to compare performances. An overview and analysis of these results are available in Chapter 7. Chapter 8 gives a conclusion and summary on the conclusions drawn from the results.

Chapter 2

Integer linear programming (ILP)

In this chapter an introduction on integer linear programming is presented. In order to understand the concept of integer linear programming a short introduction into linear programming is in order which is given in Section 2.1. Both type of problems are *optimization problems*. An optimization problem is the problem of finding the *best feasible* solution. In most cases multiple feasible solutions are available in an optimization problem, the goal is to find the best solution among them. A more detailed description of integer linear programming is available in Section 2.2.

Integer linear programming is known to be in the complexity class $\mathcal{NP}$-hard. This is a class of problems which is very difficult to solve. In Section 2.3 complexity classes will be further introduced. Because it is such a difficult problem to solve, various research on solving techniques has been done.

An ILP solver is a program which can solve an ILP problem. The basic techniques used in ILP solvers will get an introduction in Section 2.4. Most of the ILP solver are based on three type of algorithms, which will be discussed in 2.5. The current state-of-the-art ILP solvers have a lot of options to specify. These options are available because each problem instance is different and may benefit from different types of approaches. A selection of options in ILP solver will be presented in Section 2.6. In the last section, Section 2.7, an introduction on two ILP solvers will be given.

2.1 Linear programming

Linear programming problems are optimization problems which consists of an objective function and linear constraints. Various optimization problems can be modeled using linear programming. The objective in such a problem is represented in the objective function. The objective function is defined as acquiring the maximum or the minimum of a specific function. The objective only holds under certain circumstances, which are represented in the problem constraints. Next to that, the variables are in most cases also bounded to non-negative values.

Definition The following notation is the standard form most commonly used:

$$\begin{aligned} \text{Maximize or minimize} & \sum_{j=1}^{n} c_j x_j \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for each } i \\ & x_j \geq 0 \quad \text{for each } j \end{aligned} \tag{2.1}$$

where x_j is a variable, and c_j, a_{ij}, b_i represent coefficients.

Both minimization or maximization problems can be handled by linear programming solvers, they are in fact interchangeable. A minimization problem can be rewritten in maximization form by multiplying the minimization function with -1. The solution will have the same values for the variables and the value for solution will be multiplied by -1. The same principle holds for the conversion from maximization to minimization problem. Unless stated otherwise, the minimization variant is used in this thesis.

Example. A production company delivers two types of products, product A and product B. The profit for product A is 5, and the profit for product B is 6. Both products require assemblage and wrapping time. Product A requires 4 hours on the assemblage line and 3 hours wrapping. Product B requires 6 hours assembling and 2.5 hours for wrapping the product. The production facility has a limited time available for assembling and wrapping. The assemblage line has 20,000 hours per year available for production and there are 10,000 hours available for wrapping of products. An overview of this information is presented in Table 2.1.

Table 2.1: Information matrix of the production company in the linear programming example

	Product A	**Product B**	**Available**
Assemblage	4	6	20,000
Wrapping	3	2.5	10,000
Profit	5	6	

The variable x_A and x_B represent the amount of products A and B. The objective for the production company is to gain as much profit as possible. The objective function is therefore defined as:

$$\text{maximize } 5x_A + 6x_B$$

The constraints on the time available for assembling and wrapping are defined in the following problem constraints:

$$4x_A + 6x_B \leq 20,000$$
$$3x_A + 2.5x_B \leq 10,000$$

The amount of produced products cannot be negative, which leads to the following constraints:

$$x_A \geq 0$$
$$x_B \geq 0$$

The production company will acquire the highest amount of profit when producing 1,250 products of A and 2,500 products of B with a total profit of 21,250. The problem can also be visualized in a graph, which is available in Figure 2.1. The graph is 2-dimensional because two variables have been used. The dimension depends on the number of variables used, if i variables are used, then the dimension is i. The constraints define the feasible region of the graph. The profit lines visualize how much profit is made. By moving the profit line parallel from a profit of 12,000 till 24,000 the optimum can be found. The optimum is found when the profit line is as high as possible and still in the feasible region.

Dual problem

The definitions previously given refer to the primal linear programming problem. A closely related problem is the dual linear programming problem. The definition of the dual linear programming problem is given in Definition 2.2. The *duality principle* states that an optimization problem may be solved by using the dual or the primal problem. Both problems will result in the same optimal solution.

The dual problem is often used for approximation algorithms. These algorithms are non deterministical algorithms which guarantee to give a solution which is at most α times as large as the optimum

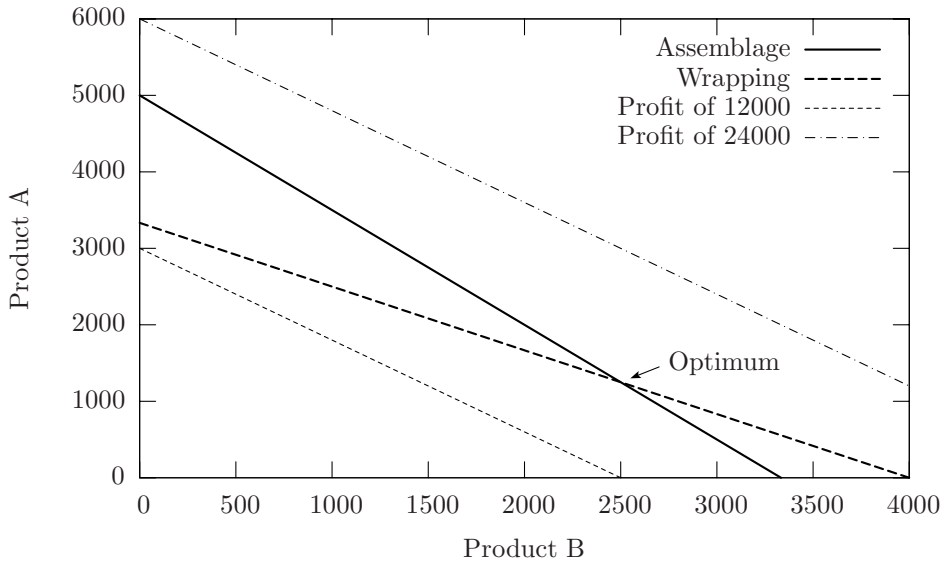


Figure 2.1: Graph of the linear programming example

solution (for a minimization problem). The dual problem could be used to indicate how close the current solution from the primal problem is to the optimum solution. A further discussion on the usage of primal dual techniques will be featured in Section 5.3.2.

Definition The dual problem of 5.1 is defined by:

$$\begin{aligned} \text{Min} & \sum_{i=1}^{m} b_i y_i \\ \text{Subject to:} & \sum_{i=1}^{m} a_{ij} y_i \leq c_j \quad \text{for each } j \\ & y_i \geq 0 \quad \text{for each } i \end{aligned} \tag{2.2}$$

where y_i is a variable, and c_j, a_{ij}, b_i represent coefficients.

2.1.1 Problem class

In complexity theory sets of computational problems are defined which can be solved by using a certain amount of computational resources. The most important classes in complexity theory are $\mathcal{P}$ and $\mathcal{NP}$. $\mathcal{P}$ is the class of problems that are *solvable* in polynomial time with respect to the input by a deterministic machine. $\mathcal{NP}$ is the class of problems that can be *verified* in polynomial time. From these definitions it is immediately visible that $\mathcal{P}$ is a subset of $\mathcal{NP}$, e.g. $\mathcal{P} \subseteq \mathcal{NP}$. The most important, still unanswered, question is whether $\mathcal{P} = \mathcal{NP}$. No prove has been found yet, which indicates that $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.

Linear programming problems are solvable in polynomial time. Various methods for solving these problems exist, each has different advantages and disadvantages. Some of the algorithms are effective in most cases, however they are unable to produce an answer within polynomial time for a few instances. On the other hand there are methods that perform better in a worst-case scenario, but perform less in other cases. Therefore a trade-off should be made while choosing an appropriate method for solving a linear programming problem.

2.1.2 Solving methods

A linear programming problem consists of a set linear inequalities, a set of variables and a linear maximization or minimization function. In graphical terms the number of variables define the number of dimensions. A variable represents one dimension. Each linear equation defines a plane in the n-dimensional space. This plane divides the space into two so called*half-spaces*. The combined half spaces define two spaces in the n-dimension, one is the feasible region and the other is the infeasible region. Points in the feasible region correspond with feasible solutions whereas the infeasible region has no feasible solutions.

Different outcomes exist for a linear programming problem;

- *One or more optimum solutions*; if a feasible and bounded region exists, at least one optimum solution will be found. It is possible to have multiple optimum solutions with the same function value. These optimal solution exists on a vertex of the polytope.
- *No feasible solution*; this situation occurs when two constraints contradict each other. For example $x \geq 1$ and $x \leq 0$, where no x can be found which satisfies both solutions. In this case there is no feasible region.
- *No optimal solution*; if the objective function is unbounded, the objective function value can increase to arbitrarily high values. For example maximize x, where $x \geq 0$.

Two general methodologies for finding the optimal solution exist:

- *Interior point method,* this method starts by selecting a point and it improves the solution by moving the point within the region.
- *Boundary search method*, this method tries to find the optimum solution by moving along the boundaries of the feasible region.

Interior point method

Interior point methods date back from the 60s, in which they were referred to as *barrier methods*. These methods are used to solve linear (and non-linear) convex optimization problems. Such a method moves through the interior of the convex, to reach the optimal solution asymptotically. The idea behind these methods is to encode the feasible set in barrier functions[1]. This idea has been studied in the early 60s by Fiacco-McCormick and others. This led to the Karmarkar's algorithm which solved LP problems in polynomial time. Karmarkar's algorithm determines the direction towards the optimum and scales it by a pre-defined factor. This process is repeated until the optimum is reached. [5]

Simplex method

The Simplex-method is a well-known technique for solving LP problems. It was created by George Dantzig in 1947 [6]. The Simplex-method travels along the boundaries of the feasible region in order to find the optimum solution, and can therefore be classified as a boundary search method. The simplex-method is an example of a method which has the advantage of being effective, however it has a disadvantage of being unable to solve certain problem instances in polynomial time. The structure of the Simplex method is defined in Algorithm 2.1.

[1] A barrier function is a function which has an increasing value near the boundaries of a feasible region.

Algorithm 2.1 Simplex method

Start along a vertex of the polytope
while the objective function does not decrease **do**
  Choose an adjacent vertex (such that the objective function does not decrease)
  if multiple vertices exist **then**
    use a pivot rule to determine the vertex to chose
  end if
end while
Solution is found

2.2 Description

Integer linear programming is a special case of linear programming in which all unknown variables need to be integers, e.g. $x \in \mathbb{Z}_+^n$. This new constraint makes the problem more complex compared to the linear programming problems. While in most practical situations this problem is $\mathcal{NP}$-hard, there are some cases in which it is possible to solve these problems effictively. The complexity of this problem will be further discussed in Section 2.3.

Definition The generic ILP formulation is [7]:

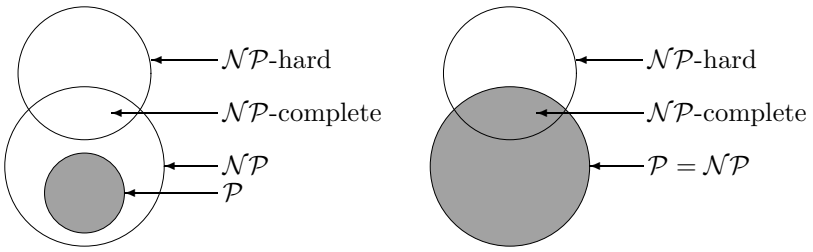
$$\begin{aligned} &\text{Max} \sum_{j=1}^{n} c_j x_j \\ &\text{Subject to:} \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for each } i \\ &x_j \in \mathbb{Z}_+^n \quad \text{for each } j \end{aligned} \tag{2.3}$$

When a problem consist of only some variables that need to be integer, then it is calles a mixed integer programming (MIP) problem. If a problem contains both integer and non-integer variables (a MIP problem) and it has only linear constraints, the problem concerned is of the type mixed linear integer programming (MILP). A special case of ILP is *0-1 integer programming* where unknown variables have to be 0 or 1, which will be further introduced in Chapter 4.

An example of an integer linear programming problem can be seen in Section 2.1. The example presented as linear programming problem in that section is actually an integer linear programming problem. The variables represent the number of products to be produced, which need to be integer. In order to convert the LP problem to an ILP problem, the following extra constraints should be added; $x_j \in \mathbb{Z}_+^n$. In this specific example the optimum solution was already integer and the solution is the same for the LP and the ILP problem.

2.3 Complexity

$\mathcal{NP}$-hard stands for *non-deterministic polynomial-time hard* and contains the problems where a polynomial-time many-one reduction exists for every decision problem in $\mathcal{NP}$. Whether it is possible to solve this problem class in polynomial time with respect to the input remains unknown. It is largely assumed that



(a) Complexity classes if $\mathcal{P} \neq \mathcal{NP}$ (b) Complexity classes if $\mathcal{P} = \mathcal{NP}$

these problems cannot be solved in polynomial time and solving these problems would therefore require a large amount of time.

$\mathcal{NP}$-hard contains the problems where a *polynomial-time many-one reduction* exists for every decision problem in $\mathcal{NP}$. A reduction from problem A to a problem B is a method that solves A based on solving B. A polynomial-time many-one reduction is a reduction that is a polynomial time computable transformation, which means that the reduction can be done in polynomial time. This shows that a problem is 'hard' enough to solve other $\mathcal{NP}$ problems, while using a polynomial-time reduction. In other words, each problem in $\mathcal{NP}$ is reducible to a $\mathcal{NP}$-hard problem. A problem is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard *and* the problem itself is in $\mathcal{NP}$. The set of $\mathcal{NP}$-complete problems is therefore a subset of the $\mathcal{NP}$-hard problem class. In Figure 2.2a and 2.2b n overview of the different classes is presented depending on whether $\mathcal{P} = \mathcal{NP}$.

ILP is known to be $\mathcal{NP}$-hard [8], which can be proven with a reduction from SAT to ILP. The SAT problem (see Chapter 3) was the first problem proven to be $\mathcal{NP}$-complete. In this reduction each clause is rewritten to a linear constraint, e.g. $(x_1 \vee ... \vee x_n)$ becomes $x_1 + ... + x_n \geq 1$. The values 0 and 1 of a variable in the ILP problem correspond with the values `FALSE` resp. `TRUE` in the SAT problem.

2.4 Solving techniques

The most important solving algorithms use a *LP relaxation* from the ILP problem. LP relaxations are obtained from a ILP problem, by dropping all the integral constraints, i.e., replacing $x \in \mathbb{Z}_+^n$ by $x \in \mathbb{R}_+^n$ [9]. The LP relaxations are LP problems and can therefore be solved by the techniques described in 2.1.2, such as the Simplex method.

In this section the most important techniques used in the algorithms of ILP solvers will be introduced.

2.4.1 Branching

Branching is an important part of the algorithms used to solve ILP problems. Branching is the division of a problem S into two subproblems S_1 and S_2, where the union of these subproblems covers the problem S. Recursive application of this technique generates a tree structure of subsets of S. In these subsets a technique called *bounding* is used.

2.4.2 Bounding

Recall from the Introduction that bounding is a technique which can find upper and lower bounds for the optimal solution within a feasible region. The definition of the upper and lower bound depends on the type of problem: maximization or minimization problem. For a minimization problem the upper bound defines the best known solution. The upper bound is global, in other words it holds for all the subsets.

The lower bound defines the smallest possible solution from the subset, e.g. no solution in the subset will be smaller than the lower bound. The lower bound can therefore differ per subset. Feasible regions can be eliminated when the lower bound is equal or larger than the upper bound, then the feasible region will not generate the optimum solution.

Bounding for a maximization problem works similarly; the definition of lower and upper bound are defubed opposite to the minimization case and subsets should be eliminated when the upper bound is smaller or equal to the lower bound.

2.4.3 Cutting planes

The objective of the cutting plane technique is to remove current fractional solution from the feasible region of the LP relaxation. This can be done by adding an extra constraint which prohibits the current fractional solution and allows a new iteration of the LP relaxation. This technique was introduced by Gomory [10] in 1958. The constraint is also referred to as a *cut*. The addition of the cut should not influence the feasible region of the ILP. This process can be repeated and it thereby strengthens the LP relaxation, e.g. the LP relaxation becomes 'closer' to the ILP problem.

2.5 Solving algorithms

Three solving algorithms for solving an ILP problem will be introduced, all based on the branching principle.

2.5.1 Branch and bound

The branch and bound algorithm uses a branch method which recursively divides the solution space into smaller regions. The next step would be to 'eliminate certain regions', by determining whether a region is not feasible or is unable to generate the optimum solution. The algorithm tries to eliminate regions until all regions have been solved or contain a feasible solution. The optimal solution is equal to the smallest feasible solution found (in the case of a minimization function).

The idea is to eliminate the subregions that are either infeasible or will not yield the optimal solution. Using the lower and upper bound, a technique called *pruning* is used. Pruning is the removal of a subregion when it has a lower bound which is greater than the global upper bound. The subregions which can give a feasible solution are then solved by determining their minimum value.

An example of a Branch and bound algorithm for an ILP problem is given in Algorithm 2.2 [9]. The best known solution in the Algorithm is denoted with z^{best}, the corresponding assignment is denoted with x^{best}. The problem starts with the best known solution as unknown; $z^{best} = \infty$ and $x^{best} = \emptyset$. In the Algorithm the subproblem k is referred to as $MIP(k)$.

In Step 4 of Algorithm 2.2, the pruning ensures that when no optimal solution can be found in $MIP(k)$, e.g. $z^k \geq z^{best}$, it will be discarded by continuing with Step 1. On the other hand, if $z^k < z^{best}$ and the solution is fractional, $MIP(k)$ must be further investigated by branching. This is done by adding subproblems $S^{k(i)}$ to L. A simple branch strategy is to branch on one fractional variable x_i^k and creating two subproblems L_1 and L_2; for subproblem L_1 add the constraint $x_i \leq \lfloor x_i^k \rfloor$, and add $x_i \geq \lceil x_i^k \rceil$ as a constraint to subproblem L_2. For example: $x_5 = 3.16$, then the extra constraint for S_1 is $x_5 \leq 3$ and for S_2 is $x_5 \geq 4$. These subproblems replace the original problem $MIP(k)$ in L.

2.5.2 Branch and cut

Branch and cut is a hybrid of branch and bound algorithm and cutting planes technique. The first step of the algorithm is to solve the LP-relaxation of the MIP using the simplex method. When the solution

Algorithm 2.2 Branch and bound algorithm

0. Initialize

   $L = \text{MIP}$, $z^{best} = \infty$, $x^{best} = \emptyset$

1. Terminate?

   if $L = \emptyset$ **then**
   the solution x^{best} is optimal. **end if**

2. Select

   Choose and delete a problem $MIP(k)$ from L.

3. Evaluate

   Solve the LP relaxation $LP(k)$ of $MIP(k)$.
   if $LP(k)$ is infeasible **then**
   it can be eliminated, therefore go to *Step 1*
   else
   let z^k be the objective function value of $LP(k)$ and x^k be its solution
   end if

4. Prune

   if $z^k \geq z^{best}$ **then**
   go to *Step 1*
   else if x^k is not integer **then**
   go to *Step 5*
   else
   let $z^{best} = \min\left\{z^{best}, z^k\right\}$, $x^{best} = \min\left\{x^{best}, x^k\right\}$ and go to *Step 1*
   end if

5. Branch

   Divide the feasible set S^k of $MIP(k)$ into smaller sets $S^{k(i)}$ for $i = 1, ..., q$, such that $\cup_{i=1}^{q} S^{k(i)} = S^k$ and add subproblems $MIP(k(i))$ for $i = 1, ..., q$ to L. Go to *Step 1.*

Algorithm 2.3 Branch-And-Cut algorithm of GLPK

0. Initialize

   L = $P_0 \leftarrow$ MIP problem, $z^{best} = \infty \leftarrow$ best known solution

1. Subproblem selection

   if L is $\emptyset$ **then**
   Goto *Step 8*
   else
   select a subproblem: $P \in L$

2. Solving LP relaxation of P: P_{LP}

   if P_{LP}has no primal feasible solution **then**
   Goto *Step 7*
   else
   Set z^*_{LP} optimal value for P_{LP}
   if $z^*_{LP} > z^{best}$ **then**
   Goto *Step 7*

3. Add 'lazy' constraints

   If 'lazy' constraints exist which violate x^*_{LP} (optimal solution for P_{LP}) **then**
   add these constraints and goto *Step 2*

4. Check for integrality

   If x^*_{LP} adheres to variable constraints; required integer variables have integer values **then**
   set $z^{best} = z^*_{LP}$ as best known solution and goto *Step 7*

5. Add cutting planes

   If there exist cuts that violate x^*_{LP} **then**
   add these cuts and goto *Step 2*

6. Branching

   Create two subproblems based on a branching variable x_i. One subproblem with added constraint $x_i \leq \lfloor x^*_i \rfloor$ and the other with $x_i \geq \lceil x^*_i \rceil$. **Remove** the subproblem P since it is included within the new subproblems and goto *Step 1*

7. Pruning

   Remove all subproblems, including current subproblem P, whose local bound $z^* \geq z^{best}$. Bound solution z^* indicates that only subproblems rooted from P cannot generate better integer feasible solution than the one found. and goto *Step 1*

8. Termination

   if z^{best} is ∞ **then**
   problem has no integer feasible solution
   else
   return best known solution x^{best} with value z^{best}.

of the LP-relaxation contains variables which have a non-integer value while it should have been integer, a cutting plane technique is used. The cutting plane algorithm tries to find linear constraints which are satisfied by all feasible integer points, but violated by the current fractional solution. The cutting plane algorithm is repeatedly used until an integer solution (which is optimal) is found or no more cutting planes can be found.

The branch and cut algorithm continues with the branch and bound part when no more cuts can be added. The problem is branched into two subproblems S_1 and S_2. After the branch, the process is repeated for both of the subproblems where the simplex method and the cutting plane technique is used to find an integer solution.

A description of a Branch and cut solver is given in Algorithm 2.3.

2.5.3 Branch and price

Branch and price is a hybrid of branch and bound algorithm and column generation methods. It is often used for MIP problems with a large number of variables. The idea is to leave out certain constraints, because it is then possible to solve more efficiently. Next to that, most of the constraints will not be binding in an optimal solution and are therefore not essential.

The basic LP problem is acquired from the MIP problem by removing certain constraints. When a resulting optimal solution of the basic LP problem is infeasible for the MIP, a subproblem is solved. This subproblem, also referred to as the *pricing problem*, is a separation problem of the dual LP and is used in order to identify constraints which need to enter the basic problem. After the addition of these constraints, the LP problem is solved again. Branching occurs when there are no constraints left which need to enter the basic problem. [11]

2.6 Search options

The development in the integer-programming technology gained large performance advances in the last years. A report of Bixby et al [12] shows an increase of 2360 times of the performance in 1988 in fourteen years based solely on the speedup in programming code (of CPLEX). Next to the increase in programming code, the increase in computing power was 800 times in that same period [12]. These speedups were made possible due to new ideas and techniques developed in these years. This section will shortly describe some techniques and their configuration which have a large impact on the performance of ILP solvers.

The branch-and-bound methods terminate when the global lower bound and global upper bound are equal. This makes it important to focus on decreasing the global upper bound and on increasing the global lower bound (in case of a minimization-problem) [9]. This can be done by finding improved feasible solutions, which lowers the global upper bound. Another option is to increase the global lower bound by finding a solution in the node with the smallest lower bound.

Different techniques use varying approaches to reach their goal. Because each problem instance is different, there is no best method. Selecting an appropriate approach depends on the problem structure and the specific instance in use. In some cases method A may function better than B, where as in another problem instance may improve while using method B. In such a situation the solver usually embeds a hybrid between these two approaches in an attempt to benefit from both methods. In order to acquire insights in the most important settings of ILP solvers, the following techniques will be further discussed in this section:

1. Node selection
2. Branching
3. Cutting planes

4. Preprocessing
5. Primal heuristics

These are however complex techniques which will not be discussed in detail. For a more detailed description, the reader is referred to [9].

2.6.1 Node selection

There are two popular node selection methods; best-bound search and diving search. The best-bound search focuses on the increase of the global lower bound. The global lower bound is defined by the node(s) with the lowest LP relaxation value. An increase of the global lower bound is possible by improving the LP relaxation at these nodes.

The diving search tries to decrease the global upper bound. The global upper bound is defined by the best feasible solution. This selection mode tries to find better feasible solutions by 'diving' in the search tree, since feasible solutions are typically found deep in the tree [9]. In this case there remains the question 'where to dive'. Several selection schemes exist, which try to predict where good feasible solutions can be found.

Well-known MIP-solvers, such as CPLEX and LINDO, have a default mode for node selection, which is usually a hybrid between best-bound and diving search.

2.6.2 Branching

Branching is an important part of the algorithms used to solve ILP problems. Branching strategies state which settings the solver may use for branching and it defines the selection criteria for selecting the best branch variable. In order to solve the problem as fast as possible, it is essential that in a branch-and-bound problem the lower and upper bound converge as fast as possible. The selection criteria for a branch should therefore be focused on improving these bounds. Methods exist that predict which fractional variables will improve the lower bound by the largest margin[9].

2.6.3 Cutting planes

The generation of a cut is a time consuming process and this technique should therefore be carefully deployed. The generation time for cuts should not exceed the speed-up gained from the improved LP relaxation. A selection of the different aspects which play a role in this trade-off are; occurrences of cuts, time spent on the creation of a cuts, type of cuts and removal of cuts.

2.6.4 Preprocessing

Preprocessing techniques are focused on a decrease in the size of the feasible region. If the size of the feasible region can be reduced, the time to find the optimum solution decreases in most cases. Preprocessing techniques are manipulations of the problem before the branch algorithm starts. These techniques are allowed as long as one optimum solution remains in the feasible region.

Preprocessing techniques usually try to remove unessential constraints, simplify constraints by reducing coefficients and strengthening variable bounds. All these techniques try to simplify the LP relaxation and let the feasible region defined by the MIP formulation remain in tact.

2.6.5 Primal heuristics

Primal heuristics are techniques targeted to find good upper bound solutions in short time spans. A good upper bound would largely reduce the running time, because the lower and upper bound will converge

earlier on. The earlier these solutions can be found, the more effective they can be, since they will immediately decrease the search space.

2.7 Solvers

By studying ILP solvers, the reader can be shown how theory becomes practice. There are various solvers, commercial as well as noncommercial. In order to en light the possibilities of these solvers, this section describes a commercial solver, CPLEX, and a noncommercial solver, GLPK.

2.7.1 Commercial solvers

The following solvers are state-of-the-art [9]: CPLEX, LINDO and Xpress-MP. Since all these solvers use the branch and cut algorithm and similar techniques to find a solution, only one solver is used to describe the basic functionality of the state-of-the-art solvers: CPLEX.

CPLEX uses a very general branch and cut algorithm to solve ILP (and MIP) problems. Within this branch and cut algorithm, CPLEX solves series of subproblems. The subproblems are ordered within a tree, where the complete tree represents a relaxation of the original MIP problem. CPLEX uses cuts when the relaxation contains one or more fractional variables in the result. If this new relaxation, containing the new cut, still generates a solution with one or more fractional variables, then CPLEX branches on one of the fractional variables. Two subproblems are created both with a different restriction on the fractional variable. The subproblems can have three different outcomes:

- All-integer solution
- Infeasible solution
- Fractional solution

In the case of a fraction solution, the previously described procedure is repeated. Nodes are cut off when the value of the objective function for the subproblem in the node is worse than the cut off value. CPLEX uses the best integer solution found as the cut off value. Depending on the type of problem (minimization or maximization) the solutions are discarded when the objective function value is larger or smaller than the cut off value.

In the event of finding an integer solution, CPLEX performs the following actions:

- it makes the solution the incumbent solution (feasible solution with the smallest objective value)
- it sets the cut off value equal to the function value of the solution
- it prunes all subproblems that are cut off due to the new cut off value

The branch and cut algorithm of CPLEX may choose to continue exploring a node or backtrack. Backtracking explores another part of the branch tree. The chosen option depends on various (user predefined) parameters. In the event of backtracking there is a parameter that determines which sort of node will be selected for further investigation. The selection of a new node can be based on the following search types:

- best bound; node with the best objective function value will be selected.
- best estimate; node selection is based on the progress toward an integer solution.
- depth first; nodes which are at the largest depth in the tree will be selected.

Next to this algorithm, CPLEX uses by default a preprocessor and heuristics. The preprocessor tries to reduce the size of the integer part in order to render the relaxations more useful and to decrease the overall size. Heuristics are periodically applied in order to find good solutions quickly.

This section is based on [13].

2.7.2 Noncommercial solvers

Commercial solvers are expensive software products. This is the primary reason for the existence of noncommercials solvers. A number of noncommercial solvers such as GLPK, lp_solve, ABACUS, and SYMPHONY exist. In general the noncommercial solvers do not perform as well as their commercial competitors [14]. Some of them however remain useful, since open-source software is more adaptable than commercial software.

In this thesis the solver GLPK is used, which will now be further introduced. For a more detailed description of the other solvers the reader can be referred to [14].

GLPK

GLPK stands for GNU Linear Programming Kit and contains a set of routines for solving LP and MILP problems. GLPK contains a stand-alone LP/MIP solver which uses the branch-and-cut algorithm.

Open source solvers, such as GLPK, are largely dependent on the time developers are willing to work on the project. In the last years there have been several new versions of GLPK with new features. GLPK version 4.2 shows no cut-generation, no preprocessing, and no primal heuristics[14]. The latest version (4.23 October 2007) however, does have these functionalities. Since GLPK is based on the branch method and GLPK uses cut-generations, it is therefore based on the branch-and-cut algorithm. A description of the algorithm used in GLPK is presented in Algorithm 2.3 [15]. In the algorithm *lazy constraints* are essential constraints which do not exist in the current MIP problem.

Chapter 3

SAT

This chapter introduces the Boolean satisfiability problem, also known as the SAT problem. A description of the problem is given in Section 3.1. Section 3.2 continues with the complexity of this problem. The extensive usage in certain industrial areas created a large number of SAT solvers. Most of the current SAT solvers are based on a type of DPLL framework. An introduction on these types of frameworks will be given in Section 3.3. In order to find a solution to this problem various techniques have been developed. The most important techniques will be discussed in the internals of a SAT solver in Section 3.4. In order to clarify the usage of these techniques Section 3.5 introduces an example solved by a SAT solver. This chapter closes with a short overview on the variations between solvers in Section 3.6.

3.1 Description

The Boolean satisfiability problem (SAT) is a decision problem; is there a satisfying assignment for a given problem. Problems are presented in propositional logic, which contains the following elements: variables, parenthesis and the AND-, OR-, NOT-operators. A problem is satisfied when an assignment of variables renders the complete expression `TRUE`.

SAT problems are defined in a format called Conjunctive Normal Form (CNF). CNF exist of conjunctions of clauses, where the clauses are a disjunction of literals. The SAT problem deals with the question whether a given CNF formula is satisfiable.

Example. An example of a propositional formula is the following formula:

$$(\neg A \vee \neg B) \wedge (A \vee \neg C) \wedge (B \vee C) \tag{3.1}$$

Two assignments that satisfy this formula exist: $\neg A, B, \neg C$ and $A, \neg B, C$.

Definition A CNF formula is defined as follows:

$$\begin{aligned} F &= c_1 \wedge c_2 \wedge ... \wedge c_m \\ c_i &= l_1 \vee l_1 \vee ... \vee l_n \end{aligned} \tag{3.2}$$

where F is a CNF formula, which consists of clauses, represented by c_i. Literals are denoted by l_i, their corresponding variables are denoted by x_j. The $\wedge$ and $\vee$ symbols represent the AND and OR operators.

A literal is the positive or negative form of a variable x_i. In order to satisfy a CNF formula, each individual clause needs to be satisfied. Satisfying a clause requires at least one literal to be satisfied. A

positive literal l_i is satisfied when its value is `TRUE` (1), and unsatisfied if the value `FALSE` (0) is assigned. On the other hand the negative literal $\neg l_i$ is satisfied when its value is `FALSE`, and unsatisfied with the value `TRUE`. The variable x_i is the variable which can take upon values 0 (`FALSE`) or 1 (`TRUE`) and corresponds to the literals l_i and $\neg l_i$. Only if $x_i = 1$, then l_i is `TRUE`, and iff $x_i = 0$, then $\neg l_i$ is `TRUE`.

3.2 Complexity

The first problem to be proven $\mathcal{NP}$-complete was the SAT problem. This means that it is currently believed that solving a SAT problem has exponential worst case complexity [16]. This does not mean that all the instances of the SAT problem need exponential time to find the solution. Various researches present classes of propositional formula's which can be solved in polynomial time [17, 18]. Next to that 2-SAT problems are solvable in polynomial time and are therefore in $\mathcal{P}$. 2-SAT problems contain at most 2 literals in one clause.

3.3 SAT solvers

A SAT solver is a solver which can determine whether a given SAT formula is satisfiable, e.g. it can solve a SAT problem. Most SAT solvers use the DPLL procedure as their framework. There exist basically two types of SAT solvers based upon the DPLL framework; *conflict driven* solvers and *lookahead* solvers. Both of these types will be further discussed in this section.

3.3.1 DPLL Framework

Most SAT solvers use the Davis-Putman-Logemann-Loveland procedure(DPLL) [19, 20] as framework. DPLL is a depth-first backtracking system, where the algorithm picks a variable and assigns a value to it at each iteration. The assigned value of the variable might implicate values for other variables. By removing the satisfied clauses and the false literals, the original problem can be simplified. This process is repeated until no variables remain unassigned or when an empty clause appears. An *empty clause*, or *conflicting clause*, is a clause without literals. When such a clause is detected, the procedure backtracks and tries the other value of the variable that caused the conflicting clause.

3.3.2 Conflict driven solvers

In Algorithm 3.1 [21] an overview of the DPLL procedure for a conflict driven solver is given. The procedure starts with all the variables unassigned. In the function `assignVar()` a variable is chosen according to some predefined heuristic and a value is assigned to the variable. The procedure of assigning a value to a variable based on some heuristic, is called the *decision procedure*. Each assigned variable has an associated *decision level*. A new decision level is introduced when a decision is made by the solver. The decision level therefore represents the number of decisions made that lead to the current assignment.

In `propagate()` the procedure assigns values to variables which have been implicated by the current decision. This technique is referred to as Boolean Constraint Propagation (BCP) and occurs when a *unit clause* exists. A unit clause is a clause where all literals except one have been assigned to `FALSE`. In order to satisfy the problem, the remaining literal must satisfy the clause. The variables assigned during the `propagate()` have the same decision level as the current decision level, since the number of decision remains the same.

If the problem contains a clause without literals, then the current set of assignments cannot generate a satisfying solution. The procedure therefore needs to undo decisions by backtracking. First the level to backtrack to (*btlevel*) is acquired in `analyze()`. The decision level starts with 1, where decision level 0 is used for assignments which are implicated by the CNF formula. If the solver is in conflict and the

btlevel is 0, it means that no assignment can be found that satisfies the problem and the problem is unsatisfiable. A *btlevel* larger than 0 leads to backtracking of the solver to this decision level by undoing the assignments with a higher decision level than *btlevel* (dec level > *btlevel*).

The process ends either with a *btlevel* of 0 after `analyze()` or when no unassigned variables remain in the problem. The latter means that the problem is satisfiable and the current assignment is a solution.

Algorithm 3.1 DPLL algorithm for conflict driven solvers

```
while nr of unassigned variables > 0 do
  assignVar()
  status = propagate()
  if status is CONFLICT then
    btlevel = analyze()
  end if
  if btlevel is 0 then
    return UNSAT
  else
    backtrack(btlevel)
  end if
end while
return SAT
```

3.3.3 Lookahead solvers

Several techniques have been proposed in order to make intelligent heuristics for selecting literals to assign. Different approaches have been developed which resulted in several heuristics. Some of them will be described in Section 3.4.1, one of them is the lookahead procedure which will be introduced in this section.

Lookahead solvers are SAT solvers which try to make an appropriate decision on the variable to select. One of the first solvers based on the lookahead procedure is satz [22]. The lookahead-procedure consists of calls to a `UnitPropagation` method. For each free variable x_i a call to `UnitPropagation`$(F \cup \{x_i\})$ and `UnitPropagation`$(F \cup \{\neg x_i\})$ will be made. These calls are sometimes referred to as the *lookaheads* on x_i and $\neg x_i$. A literal causing a conflict in the lookahead is referred to as *failed literal*. The occurrence of a failed literal means that using the given literal, no satisfying assignment can be found for the given problem while using the current assignment.

A problem is unsatisfiable when both of the calls to the lookahead procedure for a variable x_i result in failed literals. It is unsatisfiable since both x_i and $\neg x_i$ cannot satisfy the problem. When only one of the two calls results in a failed literal, the solver needs to set the variable to the complement of the failed literal. If both of the lookaheads do not result in a conflict, then the solver will use a method `diff` to calculate a weight of x_i and $\neg x_i$ (where $w(x_i)$ denotes the weight of x_i). Several approaches for the `diff` function have been proposed, such as: weighted occurrence of literals in clauses and the number of satisfied clauses caused by the unit propagation.

A heuristic function $H(x_i)$ is used to determine the variable to branch on; the variable with the highest $H(x_i)$ is used as branching variable. The heuristic function $H(x_i)$ is uses the weights $w(x_i)$ and $w(\neg x_i)$. A complete overview of the DPLL algorithm is given in Algorithm 3.2 [22].

The idea behind the lookahead procedure is to find an effective branching variable. The selection of an effective variable may lead to a fast solution, since a lot of clauses can be satisfied by setting one variable. The greatest disadvantage is that the lookahead procedure is very expensive.

Algorithm 3.2 DPLL(F), algorithm for lookahead solvers

```
if F is empty then
  return SAT
end if
F = UnitPropagation(F)
if F contains empty clause then
  return UNSAT
end if
x = max H(x_i) for all x_i's
if DPLL(F ∪ {x}) is SAT then
  return SAT
else
  return DPLL(F ∪ {¬x})
end if
```

Most notable differences between lookahead and conflict driven solver are the learning techniques and the variable selection heuristics. Conflict driven solvers learn from conflicts where as lookahead solvers try to learn while looking ahead. The lookahead solvers also employ more sophisticated variable selection heuristic in order to find an effective branching variable and find a solution fast. Another notable difference is the implementation methods used to implement the DPLL framework, conflict driven solvers are programmed iteratively and lookahead solvers are recursive algorithms.

The focus in this thesis is on the usage of conflict driven solvers, and therefore no in-depth comparison between both type of solvers will be given. In the remaining of this thesis whenever referring to a SAT solver, the conflict driven version of the SAT solver is meant unless stated otherwise.

3.4 Internals of a SAT solver

In general SAT solvers consist of the following elements:

- Search-engine
- Propagation methods
- Learning mechanisms
- Backtracking procedures

In this section these elements of a SAT solver will be reviewed more closely.

3.4.1 Searching

The search starts by heuristically made assignments. The assignments are stored in a stack (trail) in the order they were made. The trail is separated per decision level, which results in a division into groups of assignments which are the consequence of one assumption. For example: if x_1 is assumed `TRUE`, and as a consequence x_4 should have the value `FALSE` assigned, then both assignments are of the same decision level. This results in the following property; the number of decision levels equals the number of decisions made.

The solver defines a mechanism for making assumptions, usually this is based on some predefined selection criteria. This is also referred to as the *decision strategy*. The criteria define the way in which the solver searches the solution space. A criteria could be the number of times a variable has been used as

assumption before. In some solvers it is also possible to define whether the solver should set the variable initially to `FALSE` or `TRUE`.

The idea is to use a search strategy that effectively proceeds through the search space. The strategy should result in an answer as quick as possible, whether the problem is satisfiable or not. Different approaches can be taken and sometimes randomness is used, the goal however remains: find the answer as soon as possible.

Searching, among others, is one of the features that vary the most between SAT solvers [23]. The decision strategy may have a large impact on the performance of the solver. Most of these strategies use the current state of the partial assignment and try to select a literal that will satisfy as much (small) unsatisfied clauses as possible.

The 'simple' strategies, such as Dynamic Largest Individual Sum (DLIS), just select the literal with the highest number of occurrences in unresolved clauses. The BOHM heuristic selects a literal based on the occurrence in the smallest (unsatisfied) clauses. The Maximum Occurrences on Minimized sized clauses (MOM) method is a similar method, but uses a different method to count the number of occurrences in small clauses.

Another part of decision strategies is the use of randomness in making assignments. The usage of randomness prevents the solver from making too many bad decisions. The impact of the branching heuristic is further investigated in [24].

As denoted earlier the basic idea for lookahead solvers is to choose the best branching variable opposed to a near best selection for conflict driven solvers. Decision strategy is therefore even more important for lookahead solvers.

3.4.2 Propagation

Boolean Constraint Propagation (BCP) in general identifies unit clauses and creates the necessary implications[21]. Recall that a clause becomes unit when all literals except one have the value `FALSE` assigned, which indicates that the remaining literal should satisfy the clause. A unit clause implicates the value of one unassigned variable in order to satisfy the clause, this is an *implication*.

The step of propagation is repeated until either there are no variables left to propagate or there is a conflict. In the first case the search for a solution can continue, in the second case a variable should have the value `TRUE` in one clause while in another clause it should have the value `FALSE`. The latter case requires conflict analysis which will be introduced in Section 3.4.3.

Watched literals

In order to identify unit clauses a technique called *watch-lists* is often used [25, 26]. A watch-list is a set of certain literals within a clause. This watch-list is a subset of the clause and is used to efficiently detect unit clauses. The idea is to watch the status of these literals; if one of the watched literal changes, the clause is inspected. This may result in a unit clause being identified or an update of the watch-list.

The watch-list should be at least a set of size 2, because the solver will then be able to identify unit clauses. In the case of one literal in the watch list, the clause may already be unit even though no changes in the watch-list have been detected. This can happen when all the unwatched literals have a value assigned and do not satisfy the clause. In this case the literal being watched is the only unvalued literal and the clause is therefore unit which has not been identified.

Whenever one of the two literals in the watch set will be set to `FALSE`, inspection of the clause is required. Two cases may appear while inspecting the clause [25]:

1. *The clause is not unit*; within the clause there will be two or more unassigned literals. At least one of them is currently in the watch set. In order to maintain a correct watch set, the assigned literal in the watch is replaced by the unassigned literal not available in the watch list.

2. *The clause is unit*; all the literals from the clause have been assigned except for one. This literal should be by definition in the watch set, since the watch set contains two unassigned literals from the clause. In this case the solver must set the remaining unassigned literal in the watch set in order to satisfy the clause.

Note that when one of the literals in the watch set satisfy the clause, no action should be taken since the clause is satisfied and there is therefore no need to update the watch set.

3.4.3 Learning

Learning occurs when a conflicting clause is identified. Learning involves analysis of the assumptions and implications made. The goal is to add a new clause which prohibits the particular variable assignment that caused the conflict and possibly eliminate other assignments which can not lead to a solution. This new clause is referred to as the learnt clause and must, by construction, be implied by the original problem constraints [27]. The learnt clause is the result of conflict analysis.

Conflict analysis

Conflict analysis is the analyzation of the current assignment(s) in the problem, in order to find the cause of the conflicting state and to resolve this conflict. Conflict analysis consists of three phases:

1. *Analyzation*; analyze the current implications in order to find the assignments that are involved in the conflicting state. This phase results in a conjunction of literals which can be seen as the cause of the conflict.

2. *Add learnt clause*; the analyzation phase results in a conflicting clause that contains a certain conjunction of literals which caused the conflict. Since the solver needs to resolve this conflict and ensure that this particular conflict will not appear again, the solver adds a clause that prohibits this particular conjunction of literals. This can be done by transforming the conjunction into a disjunction and negating the literals. The created clause can also be referred to as the *learnt clause*, the *conflict-induced clause* or the *conflict clause*.

   For example: the disjunction is $x1 \wedge \neg x3 \wedge \neg x9$, the learnt constraint becomes $\neg x1 \vee x3 \vee x9$. This means that while solving either $\neg x_1$, x_3, or x_9 should hold, otherwise the same conflict could occur.

3. *Backtrack*; after addition of the learnt clause the solver needs to continue in order to find a solution. The solver needs to 'undo' certain assignments, since the current assignments result in a conflict.

Learning schemes

In a learning scheme is defined how a SAT solver learns from a conflict. Different learning schemes are based on the various conflict clauses which can be distilled from a conflict. In order to find the cause of the conflict an implication graph is used. An implication graph is a visual representation of the elements involved in the conflict. In an implication graph the following holds:

- A vertex corresponds with a variable assignment. The text on the vertex indicates which variable assignment has been made at which level. For example: $x_9 = 1@3$ indicates that variable x_9 has been set to `TRUE` at decision level 3.

- The predecessors of a vertex v are responsible for the variable assignment v represents, which is shown by the edges leading to the vertex v. A vertex which has no predecessor is a decision assignment, an assignment with no reason. The description along the edge shows which clause is responsible for the assignment.
- A conflict is denoted by a bi-directional edge with conflicting assignments attached to its vertices. It indicates that certain assignments force upon the value `TRUE` for a variable, whereas other force the value to `FALSE`.

In an implication graph (see for an example Figure 3.1) the various conflict clauses correspond to the different conflicting cuts in the implication graph. A conflict cut is a cut[1] with on one side of the cut both of the conflicting literals and on the other side the decision variables [21].

The unique implication point

Each conflict cut can be used in order to find a satisfying assignment, however some cuts tend to generate satisfying assignments faster than others. Some well known solvers, such as GRASP [28] and MiniSat [27, 2], base their conflict analysis on a *unique implication point (UIP)*. A UIP is a vertex at the current decision level that *dominates* both of the vertices involved in the conflicting variable [29]. A vertex a is said to dominate vertex b, iff any path from the decision variables of the decision level of a to b, needs to go through a [29]. A UIP is the cause of the conflict at the current decision level. Multiple UIPs can co-exist in a particular situation, the UIPs are then ordered in occurrence from the conflict to the decision level. It should be clear that the decision variable (of the current decision level) is always a UIP.

In certain solvers a UIP is used in the learning scheme. The conflict clause is created while using a cut where all the implications after the UIP are put on the *conflict side*. The other partition, the *reason side*, contains all the decision variables, the UIP and the implications that lead to the UIP. The first UIP cut corresponds to the cut where the first UIP is used as last part of the reason side.

An example of a first UIP cut is visible in Figure 3.1. After the creation of this first UIP cut, the corresponding learnt clause is added. As a result backtracking will take place, where a UIP will become unit in the learnt clause and as a consequence a UIP will be flipped. Changing the UIP can result in the flipping of variables that have been set at the same decision level before the first UIP has been set. In Figure 3.1 the literal x_4 is the first UIP. The last UIP cut is also available in the same figure. The last UIP corresponds with x_1.

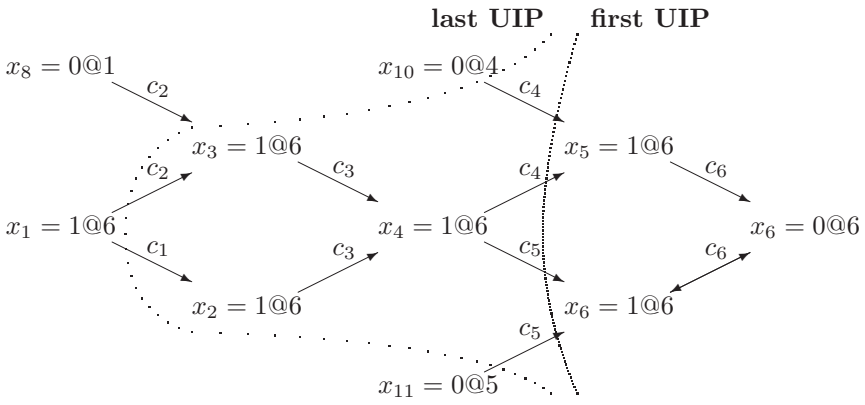


Figure 3.1: First vs last UIP cut in an implication graph

[1] A cut is a partition (division) of all the vertices of a graph into two sets.

Effects on learning

The effects of learnt clauses are divers: learnt clauses can guide the backtracking process and speed up future conflicts by caching the reason for conflict[27]. While a learnt clause prevents only a number of inferences, a set of learnt clauses can have an accumulated learning effect. On the other hand, a larger set of learnt clauses increases the propagation time, therefore the set of clauses is periodically reduced.

3.4.4 Backtracking

Backtracking is a search strategy that uses a depth-first search through the possible solutions. SAT solvers use the backtracking technique in order to continue the search after a conflict has been detected. Two forms of backtracking in SAT solvers exist, namely: *chronical* and *non-chronical backtracking*. Chronical backtracking restricts the backtracking after a conflict to one decision level, where non-chronical backtracking can backtrack over multiple decision levels. If the current decision level would be d, $\beta = d-1$ would be chronical backtracking, where as $\beta < d-1$ refers to non-chronical backtracking (see Equation 3.3).

If the learnt clause is known, it can easily be shown that the solver will not find a satisfying solution until there has been backtracked to the highest decision level of this conflict-induced clause. The conflict remains when all the literals of the conflicting clause maintain their values. Backtracking needs to be done to at least one level further than the highest level among literals from the conflicting clause, which should make the conflict-induced clause unit. This can be done by backtracking to the second to last decision level of the conflict-induced clause. The unit clause may then guide the solver to a new solution. [27, 28]

The backtracking level can be calculated according to [28]:

$$\beta = \max \{\delta(x) | (x, \nu(x)) \in A_C(\kappa) \text{ and } \delta(x) \neq \text{ current decision level}\} \tag{3.3}$$

where β is the backtracking level and the 'max' function should return the second to last decision level among the literals. The decision level of x is defined by $\delta(x)$. The vertex where the value of x is set to $\nu(x)$ is defined in the formula by $(x, \nu(x))$. The conflict (vertex) is defined as κ and the antecedent assignment of $A(\kappa)$ as $A_C(\kappa)$ (e.g. the set of assignments that imply κ, the reason of κ).

Example. An example of a conflict-induced clause could contain four literals of various decision level, e.g. $x_1@1, x_2@2, x_3@4$ and $x_4@6$. In this case backtracking should be done to decision level 4 in order to make the conflict-induced clause unit and x_4 should be set.

3.5 Solving an example

In order to clarify the techniques mentioned in the previous sections, an example will now be introduced.

Example. In this Example the methods previously explained will be used in order to solve a SAT problem. The SAT problem consists of the clauses defined in Table 3.1.

The solver makes the following heuristic truth assumption: *1)* $x_{12} = 1$. The first unit clause, clause c_8, appears and the solver propagates the value of x_8 to 0. The solver continues with the following decisions: *2)* $x_9 = 0$, *3)* $x_{10} = 0$, *4)* $x_{11} = 1$, and *5)* $x_1 = 1$. The last assignment results in the following new propagations: *5a)* $x_2 = 1$, *5b)* $x_3 = 1$, *5c)* $x_4 = 1$, *5d)* $x_5 = 1$, *5e)* $x_6 = 1$, *5g)* $x_7 = 0$, and *5h)* $x_6 = 0$. This last assignment cannot be made since it is in conflict with assignment 5e. An overview of the assignments is also available in Table 3.2. The next step for the solver would be to start the conflict analysis.

The implication graph of the current conflict is shown in Figure 3.2. The first UIP cut corresponds with $\neg x_4 \vee x_9 \vee x_{10}$. The highest decision level after the current decision level is 3, therefore the solver will use a

$c_1 = \neg x_1 \vee x_2$
$c_2 = \neg x_1 \vee x_3 \vee x_8$
$c_3 = \neg x_2 \vee \neg x_3 \vee x_4$
$c_4 = \neg x_4 \vee x_5 \vee x_9$
$c_5 = \neg x_4 \vee x_6 \vee x_{10}$
$c_6 = \neg x_5 \vee \neg x_7 \vee \neg x_{12}$
$c_7 = \neg x_5 \vee \neg x_6$
$c_8 = \neg x_8 \vee \neg x_{12}$

Table 3.1: Clauses from the SAT solving example

Dec. **level**	**Decision**	**Propagations**
1	x_{12}	$\{\neg x_8\}$
2	$\neg x_9$	$\{\}$
3	$\neg x_{10}$	$\{\}$
4	x_{11}	$\{\}$
5	x_1	$\{x_2, x_3, x_4, x_5, x_6, x_7, \neg x_6\}$

Table 3.2: Assignment stack of the SAT solving example per decision level

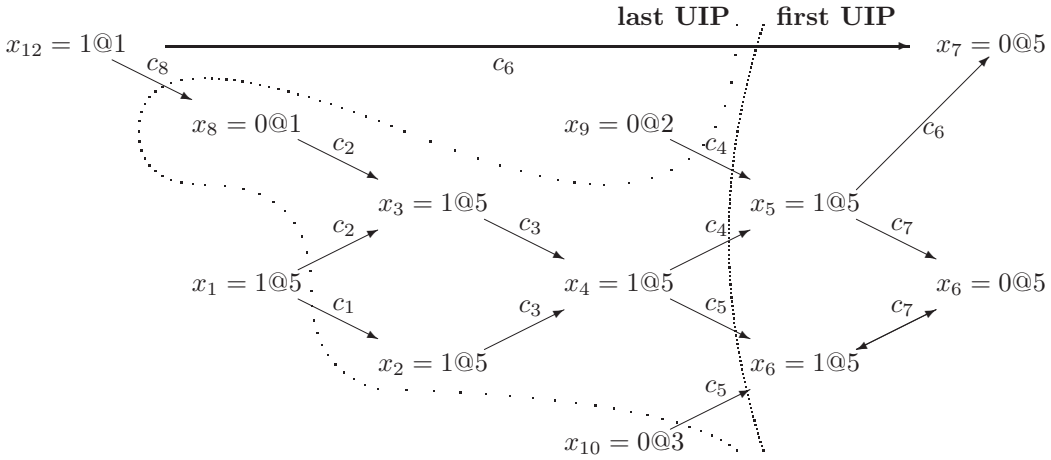


Figure 3.2: Conflict in an implication graph of an example at step $5h$

non-chronical backtrack to level 3 and undo all the decisions afterwards. At decision level 3 the newly learnt clause is unit and therefore $x_4 = 0$ (the first UIP) will be set at decision level 3 and the solver will continue.

The solver could continue with : *4)* $x_2 = 0$, which leads to *4a)* $x_1 = 0$. Another assumption could be made by the solver: *5)* $x_7 = 1$, which leads to *5a)* $x_5 = 0$. All clauses are now satisfied and the current assignment is therefore satisfying. Some of the variables have not been set to a value; these variables may take upon either of values, the current assignment will always satisfy all clauses.

3.6 Variations between solvers

There are various SAT solvers, most of today's SAT solvers use the DPLL framework. They are based on the frameworks denoted in Section 3.4 and they vary on the elements mentioned in Section 3.3. In this section a selection of aspects on which solvers differ are examined in more detail.

3.6.1 Preprocessing: resolution

Resolution is a technique which enables the removal of variables from a problem. Given two clauses $C_1 = \{x, a_1, ..., a_n\}$ and $C_2 = \{\neg x, b_1, ..., b_m\}$, the clause $C = \{a_1, ..., a_n, b_1, ..., b_m\}$ combines both C_1 and C_2. This can be proven by the fact that either $a_1, ..., a_n$ or $b_1, ..., b_m$ needs to satisfy their clause, since x can only satisfy one clause. Clause C is said to be the *resolvent* of the original clauses by performing resolution on x. The notation is $C = C_1 \otimes C_2$.

Resolution can also be performed on more than two clauses. Take S_1 as the set of clauses containing x, and S_2 as the set of clauses containing $\neg x$. By pair-wise resolving all the clauses from S_1 with every clause in S_2 the variable x can be eliminated via a new set of clauses S. The formal notation of $S = S_1 \otimes S_2$ is defined as:

$$S_1 \otimes S_2 = \{C_1 \otimes C_2 \,|C_1 \in S_1, C_2 \in S_2\} \tag{3.4}$$

3.6.2 Searching: randomized restarts

Restarts were first discussed in [30] and later on in [31]. The suggestion was made that randomized restarts may help proving satisfiability and unsatisfiability. A restart resets the solvers state to decision level 0 and threats the learned clauses as initial clauses. The restart strategy is defined by using a cutoff parameter which defines the space that can be searched between restarts. By gradually increasing the cutoff parameter the search space expands between restarts. This is done in order to be able to prove unsatisfiable instances.

Randomized restarts also help to eliminate *heavy-tail behavior* [30]. Heavy-tail distribution is a type of distribution, which has a relatively large number of solutions in the tail compared to a normal distribution. Researchers have shown that runtime distribution of complete backtrack search methods exhibit heavy-tail behavior [32]. This means that SAT solvers using one long run may experience large variance in solving times and they may even be unable to solve certain instances before a timeout occurs. The usage of (rapid) randomized restarts largely eliminate heavy-tail behavior and it helps solving a larger number of instances [33].

Various restart strategies have been suggested. The efficiency of a strategy largely depends on the runtime distribution of the instances. Most state-of-the-art SAT solvers use a restart strategy in which an initial cutoff value is used which increases linearly [23].

3.6.3 Learning: assignment stack shrinking

This is a technique which first appeared in Jerusat [34]. This technique is activated when the solver generates a conflict clause C that exceeds a threshold on the length of the clause. The idea is to cause the same conflict with fewer decisions. This largely reduces the conflict clause which should make a 'stronger' learnt clause.

The shrinking process starts with a conflict and backtracking to a certain decision level. It continues with assigning literals according to a scheme, usually by assigning literals close to the conflict. This results in the same conflict, however it tends to reduce the number of decisions and implications since the unnecessary elements in the conflict may have been removed.

3.6.4 Learning: conflict clause minimization

Conflict clause minimization first appeared in MiniSat [2]. The idea is to make learnt conflict clauses stronger by minimizing them. A smaller conflict clause further reduces the search space. The conflict clause can be reduced by identifying literals of clause C that are implied to `FALSE` while the rest of the literals of C are set to `FALSE`.

In MiniSat this is done by inspecting the reason for each literal p in clause C. The algorithm tries to *self-subsume* C by the reason clause for p and if successful the literal p can be removed from the clause C [2]. This process is repeated until all literals in C have been inspected.

A clause c_1 is said to *subsume* a clause c_2, when $c_1 \subseteq c_2$. The subsumed clause, c_1, can be removed from the problem since it is embedded in clause c_2. Another sort of subsumption is the *self-subsumption*. This subsumption may occur when a clause c_2 almost subsumes another clause c_1 except for one literal x, which occurs with the opposite sign in c_2. For example the following clauses are defined $c_1 = \{x, a, b\}$ and $c_2 = \{\neg x, a\}$. Resolving on x will produce $c_1^{'} = \{a, b\}$, which subsumes c_1. By adding $c_1^{'}$ the clause c_1 can be removed. [35]

3.7 MiniSat

In this section the SAT solver MiniSat is the central topic. MiniSat is developed by Niklas Eén and Niklas Sörensson. MiniSat is a conflict driven solver based on the DPLL framework. It uses advanced techniques such as randomized restarts and conflict clause minimization. It has shown good performances in recent SAT competitions. In order to acquire insights on the techniques used in the new solver MiniZSat, it is necessary to understand the basic functionality of MiniSat.

Most of the techniques employed in MiniSat have been discussed earlier on in this chapter. However it is interesting to take a glimpse at a partition of the implementation used by MiniSat. The conflict analysis is of specific interest, because MiniZSat makes extensive use of it and extends the solver with a new sort of conflict which partly uses the conflict analysis. This section will continue with a description of the conflict analysis as employed by MiniSat.

3.7.1 Conflict analysis

Conflict analysis in MiniSat finds the Unique Implication Point (see Section 3.4.3). In order to do so MiniSat uses a trail in which all literal assignments are kept. This trail contains the assignments in the order of which they have been made. The conflict analysis mainly exists of a backwards traversal of this trail. An important property of this trail is that the *reason of* p is always before p in the trail. The reason of literal p is the combination of literals which caused literal p to get its value assigned. Note that a decision variable p will have no reason.

In order to find the UIP the conflict analysis uses a counter to keep track of the number of references on the current decision level. The counter will increase if an unaccounted literal of the current decision level appears in the reason of a literal under 'investigation'. If the reason of literal x_3 is $x_1 \wedge \neg x_2 \wedge x_4$ and literals x_2 & x_4 are assigned at the current decision level and unaccounted for, then the counter is increased with 2. The counter is decreased with 1 when a literal has been investigated. If the total number of references is 1 at some point, the current literal in the trail under inspection is the first UIP.

This can be seen as follows: the conflict analysis only inspects those literals relevant in the conflict, which are those literals that occur in the reason of a conflict. Because the reason of a literal p is always before p in the trail, a backwards traversal of the trail will result in the investigation of those literals in the reason. The combination of increasing and decreasing ensures that all literals of the current decision level that appeared in the reason of a literal which were involved in the conflict, are investigated. If the counter is 1 and is decreased after investigation to 0, the analysis stops and the literal x under investigation is the first UIP.

The conflict analysis can also be seen as an analysis of the implication graph. The analysis updates a counter according to the number of edges starting from an assignment of the current decision level to the current literal under investigation. The analysis starts with the conflict and uses a backward traversal of the assignment stack, thus investigating all assignments of the current decision level involved in the conflict. The result is literal x, which has at most one literal from the current decision level in its reason.

This is caused by the fact that the counter will be reduced to 0 after investigation. It means that the conflict is dominated by the vertex under inspection. These facts combined ensure that literal x is in fact the first UIP. It is the first UIP because the trail is found in a backward traversal.

While finding the UIP, the conflict analysis also creates the conflict clause and it finds the highest decision level among literals from the conflict clause after the current decision level. This decision level is used for the backtracking process; the solver does a (non-) chronical backtrack to this decision level. In addition the conflict analysis will undo part of the trail, but not beyond the current decision level.

Chapter 4

Pseudo-Boolean

Pseudo-Boolean problems are a class of optimization problems based on standard integer linear programming (ILP) with linear constraints and a restriction on the values of the variables to either 0 or 1. In other words, it is an ILP problem with binary (0-1) variables. Sometimes this class of problems is also referred to as *0-1 programming*.

This chapter continues with linear pseudo-Boolean problems only. In Section 4.1 a definition of the linear pseudo-Boolean constraint will be given. As denoted earlier pseudo-Boolean problems are closely related to ILP problems. The relation in complexity and expressiveness of the problems will be discussed in Section 4.2. Pseudo-Boolean problems can be solved by ILP solvers, however the problem definition of pseudo-Boolean problem will not be used effectively in ILP solvers since they are not optimized for this type of problems. This created a new category of pseudo-Boolean solvers optimized for solving pseudo-Boolean problems. Several techniques used in pseudo-Boolean solvers will be reviewed in Section 4.3. Various strategies are used within pseudo-Boolean solvers. Two pseudo-Boolean solvers based on different strategies will be introduced in Section 4.4.1.

4.1 Linear pseudo-Boolean constraints

There is a default definition of a linear pseudo-Boolean constraint, which is shown below. Next to the linear pseudo-Boolean constraints there exists a set of non-linear pseudo-Boolean constraints in which products of variables may populate the constraint.

Definition A linear pseudo-Boolean constraint is defined by the following formula [36]:

$$\sum_i w_i x_i \ \triangleright k \tag{4.1}$$

where x_i is a binary variable (which should have value 0 or 1). Both w_i and k should be integers. The $\triangleright$ represents one of the relational operators $>, \geq, =, \leq$, or $<$.

Definition 4.1 does not define the complete set of linear pseudo-Boolean constraints, since every constraint which can be rewritten to this form is also a linear pseudo-Boolean constraint. For example, the following formula $1x_1 + 3x_3 - 5 \geq -2x_2$, which can be rewritten to $1x_1 + 2x_2 + 3x_3 \geq 5$, is also a linear pseudo-Boolean constraint. This example constraint is satisfied by setting at least $x_2 = 1,$ and $x_3 = 1$. Each pseudo-Boolean constraint can also be rewritten to a *normal form*.

Definition A linear pseudo-Boolean constraint in the normal form is defined by the following formula [36]:

$$\sum_i w_i x_i \geq k \tag{4.2}$$

Several transformations exist in order to transform a linear pseudo-Boolean constraint into normal form. A pseudo-Boolean constraint containing a negative coefficient can be transformed to the normal form by replacing x_i with x'_i, where $x'_i = 1 - x_i$. If a pseudo-Boolean constraint uses an operator other than $\geq$, it can be transformed by using the transformations denoted in Table 4.1. Constants within a constraint can be moved to the right-hand-side by algebraic manipulations. By definition the right-hand-side should be positive, otherwise the constraint will always be satisfied since all variables on the left-hand-side have positive coefficients.

Table 4.1: Transformation matrix from relational operator to $\geq$ operator.

Operator	**Action**
$=$	Add two constraints in which the operator $=$ is replaced by $\geq$ and $\leq$
$>$	Add 1 to the left-hand-side and change the operator to $\geq$
$\geq$	Nothing to do
$\leq$	Multiply both the left and the right-hand-side with -1 and change the operator to $\geq$
$<$	Subtract 1 from the left-hand-side and change the operator to $\leq$

A special type of pseudo-Boolean constraint is the *cardinality constraint*, which is a constraint where all the coefficients on the left are one. It is defined as Definition 4.2 where w_i equals 1, an example: $x_1 + x_2 + x_3 \geq k$. If all the coefficients are equal to 1 *and* the right-hand side is also 1, the PB constraint is equivalent to a CNF clause.

Definition A cardinality constraint is formulated as follows:

$$\sum_i l_i \geq k \tag{4.3}$$

For the LP problems holds that a minimization problem can be transformed into a maximization problem by multiplying with -1. This also holds for the pseudo-Boolean problem. Therefore only the minimization problems will be discussed in this thesis.

4.2 Complexity

Although pseudo-Boolean problems are a restricted variant of the Integer Linear Programming problem class, they share the same complexity class: $\mathcal{NP}$-hard. The same reduction as used in the reduction from SAT to ILP can be used for the reduction from SAT to a pseudo-Boolean problem (see Section 2.3), in order to prove it can solve $\mathcal{NP}$-complete problems. The pseudo-Boolean decisions problems were one of the 21 problems which Karp [37] proved to be $\mathcal{NP}$-complete, the optimization variant of pseudo-Boolean problems are however $\mathcal{NP}$-hard.

4.2.1 Expressiveness of pseudo-Boolean constraints

Pseudo-Boolean constraints have more expressive power when compared to CNF clauses. The expressiveness can be indicated by comparing the worst case complexity of certain 'easy' problems defined in CNF clauses and pseudo-Boolean constraints. This comparison is visualized in Table 4.2 [38]. In this Table it is shown that, the expressiveness of the linear pseudo-Boolean constraints is greater than the expressiveness of CNF clauses.

In the case of a single clause, the solver only needs to check whether the clause is empty. In order to determine whether a linear pseudo-Boolean constraint is satisfiable, the solver needs to verify the smallest and the largest value the left-hand-side can generate. If the value of the right-hand-side lies within these margins (including the boundaries), the constraint is satisfiable.

When two constraints are considered the complexity increases. When clauses are concerned, a solver needs linear time to determine satisfiability. This can be done by inspecting both of the clauses and find a literal for each clause, such that the chosen literals are not each others complements. Two linear pseudo-Boolean constraints are $\mathcal{NP}$ complete since the subset sum problem can be encoded in two linear pseudo-Boolean constraints. The subset sum problem is a problem given a set of integers, determine whether a subset of these integers can sum up to a pre defined integer. The subset sum problem is known to be $\mathcal{NP}$ complete and therefore the determination whether two linear pseudo-Boolean constraints are satisfiable is also $\mathcal{NP}$ complete. [38]

Table 4.2: Worst case complexity of PB and SAT problems containing 1 or 2 constraints

	Clauses	**Linear PB constraints**
satisfiability of 1 constraint	$O(1)$	$O(n)$
satisfiability of 2 constraints	$O(n)$	$\mathcal{NP}$-complete

Another indication on the expressiveness of the pseudo-Boolean constraints is the *pigeon-hole principle*. The pigeon hole principle is the principle that given two positive integers, n and m with $n > m$; if n items need to be placed into m cases, then at least one case contains multiple items. The principle is often illustrated by placing pigeons into holes, hence the name pigeon hole principle. In order to prove this principle an exponential number of steps are necessary when CNF clauses are used. If on the other hand cardinality (or linear pseudo-Boolean) constraints are in use, the problem can be solved in a polynomial number of steps. [36, 39]

The pigeon-hole principle can also indicate the difference between the expressiveness of cardinality constraints and CNF clauses. To prove the principle exponential steps are required while using CNF clauses. Cardinality constraints only needs a quadratic number of steps [40].

4.2.2 Comparing problem classes

In the previous subsection the expressiveness of pseudo-Boolean constraints and CNF clauses have been compared. A comparison of the expressiveness of several problem classes was given in the Introduction in Figure 1.1. Figure 4.1 presents the relations between the expressiveness of the constraints from the various problem classes. Integer linear programming constraints are the most expressiveness, since they allow integer variables. Pseudo-Boolean constraints only allow binary variables. The cardinality constraints also requires the usage of binary variables and it has another extra limitation; the constraint needs to be in the following form $\sum_i l_i \geq k$. CNF clauses are the same as cardinality constraints with the extra requirement that $k = 1$ in each constraint, e.g. $\sum_i l_i \geq 1$.

The relatively low expressiveness of CNF clauses, leads to a large number of CNF clauses for expressing one pseudo-Boolean constraints [41]. Translating a pseudo-Boolean constraint to a CNF clause may

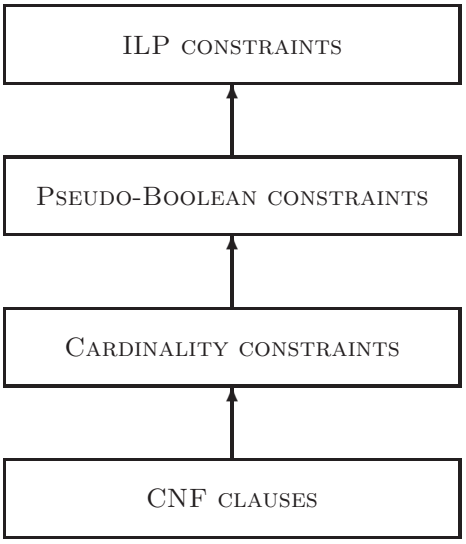


Figure 4.1: Comparing expressiveness of constraints and clauses

require an exponential number of clauses. Translating from a cardinality constraint to a clause using a naive translation, leads to $\binom{n}{k-1}$ clauses [36]. In these translation no new variables are used, there do however exist translation which use new variables that largely reduce the number of clauses required.

4.3 Solving techniques

In this section some techniques used in pseudo-Boolean solvers will be presented. These techniques are largely based on concept used in the related problem areas, ILP (see Chapter 2) and SAT (see Chapter 3).

4.3.1 Constraint operations

The equivalent of resolution for CNF is cutting plane for pseudo-Boolean problems. Cutting plane is the procedure in which a nonnegative combination of 2 constraints is used in order to eliminate a variable in the resulting constraint. Combining these two constraints leads to the following constraint (Where the antecedents are presented above the line and the consequence below the line) [42]:

$$\frac{\begin{array}{c}\lambda \times (\sum a_i x_i \geq k) \\ \lambda^{'} \times (\sum a_i^{'} x_i \geq k^{'})\end{array}}{\lambda \sum a_i x_i + \lambda^{'} \sum a_i^{'} x_i \geq k\lambda + k^{'}\lambda^{'}} \quad (4.4) \quad \frac{\begin{array}{c}1 \times (x_1 + 2x_2 + 3x_3 \geq 2) \\ 2 \times (\overline{x}_2 + 2x_4 + x_5 \geq 3)\end{array}}{x_1 + 3x_3 + 4x_4 + 2x_5 \geq 8}$$

It is also possible to round coefficients from a linear pseudo-Boolean constraint. This is due to the fact that $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$. *Saturation* uses rounding to saturate all coefficients a_i at k. This can be achieved by repeatedly multiplying the constraint with $(k-1)/k > \lambda > 1$. The value of λ ensures that k

remains the same value after multiplication and rounding up. A constraint containing a coefficient of 3 and $k = 2$, could saturate the coefficients by multiplying with $\lambda = 0.6$ $(0.5 > \lambda > 1)$, and rounding after wards.

$$\frac{\sum a_i x_i \geq k}{\sum \lceil a_i \rceil x_i \geq \lceil k \rceil} \qquad (4.5) \qquad \frac{3x_1 + x_2 + 2x_3 \geq 5}{\underline{x_1 + 1/3x_2 + 2/3x_3 \geq 5/3}} \; x_1 + x_2 + x_3 \geq 2$$

Cardinality constraint reduction creates a cardinality constraint from a linear pseudo-Boolean constraint. The reduction finds the absolute minimum number of variables that need to be satisfied in order to satisfy the pseudo-Boolean constraint. In Example 4.6 the constraint can not be satisfied by setting 1 or 2 variables. At least 3 variables are needed to satisfy the constraint $(3 + 2 + 1 \geq 6)$. The reduction of linear pseudo-Boolean constraint can be further investigated in [42, 43]

$$\frac{3x_1 + 2x_2 + x_3 + x_4 + x_5 \geq 6}{x_1 + x_2 + x_3 + x_4 + x_5 \geq 3} \qquad (4.6)$$

4.3.2 Constraint propagation

Pseudo-Boolean constraint propagation is similar to constraint propagation for SAT solvers. While in SAT solvers it is sufficient to watch two literals, within a PB constraint several literals need to be watched in order to identify propagations. A PB constraint is unit when

$$S_T + S_U < b + a^U_{max} \qquad (4.7)$$

where S_T represents the sum of coefficients of true literals in a pseudo-Boolean constraint. The sum of coefficients of unassigned literals in a pseudo-Boolean constraint is defined by S_U. The right-hand side of the pseudo-Boolean constraint equals b and a^U_{max} is the largest value of the coefficient of the unassigned literals in the pseudo-Boolean constraint.

The implied literal is x^U_{max} (the literal belonging to a^U_{max}), which is implied by the conjunction of false literals in the constraint.

Detecting unit literals in a pseudo-Boolean constraint can be done by watching a set of literals in such a way that, when a literal within the watched set is updated to `FALSE`, the constraint can still be satisfied. A straightforward technique would be to update S_T, S_F, and S_U, which requires to watch all literals in the constraint. Another technique is to watch the smallest set available, such that the watched sum (S_W) is:

$$S_W \geq b + a^U_{max} \qquad (4.8)$$

In the case of setting a watched literal to `FALSE`, one or more other non-`FALSE`, non-watched literals must replace the `FALSE`-literal in order to maintain the Equation 4.8. If this is not possible the constraint becomes unit and the watched literal with the largest coefficient is implied (x^U_{max}).

Example. The following example illustrates the described procedure to identify unit constraints.

$$6x_1 + 4x_2 + 3\overline{x_3} + 3x_4 + 2x_5 + 1x_6 \geq 9$$

The solver initially takes the following set as $L_w = \{x_1, x_2, \overline{x_3}, x_4\}$, with $S_w = 16 \geq 9 + 6$. If the solver would assign $x_3 = 1$, the set would need to be updated since $S_w = 13 < 9 + 6$. Therefore $\overline{x_3}$ needs to be replaced by another literal. The resulting L_w could be: $\{x_1, x_2, x_4, x_5\}$, where $S_w = 15 \geq 9 + 6$.

When the solver continues with the assignment of $x_2 = 0$, the set will be extended with the last unwatched literal; $L_w = \{x_1, x_4, x_5, x_6\}$ with $S_w = 11 < 9 + 6$. This results in the assignment of $x_1 = 1$ and a new watched set; $L_w = \{x_1, x_4, x_5, x_6\}$ with $S_w = 12 \geq 9 + 3$.

The solver could continue with the assignment of $x_6 = 0$, resulting in $S_w = 11 < 9 + 3$. This enforces x_4 to `TRUE`, which in turn also satisfies this constraint, since $x_1 = 1, x_4 = 1$ satisfies the constraint.

4.3.3 Conflict analysis

The conflict analysis of a pseudo-Boolean solver can be based upon the conflict analysis from a SAT solver (see Section 3.4.3). As is the case for SAT solvers, the pseudo-Boolean solver may use an implication graph in order to determine the reason of the conflict. The construction and analysis of this graph can be done in the same form as the analysis is defined for a SAT solver. The analysis should result in a conflict-induced clause corresponding to a cut in the implication graph. Analogous to the SAT solvers implication graph a unique implication point (UIP) can be found and used in the learning procedure.

Within the conflict analysis the pseudo-Boolean solver may also acquire a decision level where the solver can backtrack to. It is also possible to apply non-chronical backtracks, which is advantageous because the solver does not need to inspect part of the search tree that cannot generate a (optimum) solution.

4.3.4 Learning scheme

The learning scheme used with pseudo-Boolean constraints is largely based on the CNF learning scheme (see Section 3.4.3). The equivalent of clause resolution from the CNF learning scheme in the pseudo-Boolean learning scheme is cutting planes [38, 44]. Recall that cutting planes is an operation on linear constraints where a linear combination of a pair of pseudo-Boolean constraints is taken, in order to eliminate a particular literal (see Section 4.3.1).

Unlike the CNF learning scheme the result of the pseudo-Boolean learning scheme may not be a violated learned constraint. This is be due to the possibility of over-satisfaction[1] of a constraint. The reader is referred to [42] for procedures to ensure that the resolved clause remains violated. It should also be denoted that the described pseudo-Boolean learning scheme does not guarantee a unit constraint.

4.4 Solvers

In this section the following two pseudo-Boolean solvers will be introduced: Pueblo and MiniSat+. Pueblo is a hybrid between ILP and SAT solvers, whereas MiniSat+ is only based on a SAT solver. There is also a group of pseudo-Boolean solvers which are solely based on ILP techniques. In this group also solvers such as Galena [42], PBS [41], and OBPDP [45] can be found.

By definition an ILP solver can also solve pseudo-Boolean problems since a pseudo-Boolean problem is an integer linear programming problem with an extra constraint on the variables (e.g. $x \leq 1$). Therefore no introduction of a solver from this group will be given (the reader can use the description of Glpk as introduction, see Section 2.7.2).

This section continues with two solvers Pueblo and MiniSat+, which will be discussed in more detail.

[1] A constraint is over-satisfied when $S_T > b$.

4.4.1 Hybrid pseudo-Boolean solver Pueblo

There are various pseudo-Boolean solvers based on different concepts. A large number of the solvers are either based on ILP techniques or on SAT techniques. In this section the solver Pueblo is investigated, which is a hybrid of both techniques. Pueblo is developed by Hossein Sheini.

Pueblo is a solver based on MiniSat and it uses both ILP and SAT techniques to solve pseudo-Boolean problems. Pueblo uses a hybrid algorithm based on a combination of cutting plane techniques and implication graph analysis for conflict-based learning [46]. Via this combination the solver can profit from the cheap recognition of unit clauses and utilize the pruning power of the cutting planes.

The major disadvantage of learning pseudo-Boolean constraints is the high overhead of propagating through those constraints [42, 46]. This is mainly due to the search for the unvalued literal with the largest coefficient. A partial solution to this problem is weakening the learned constraints to cardinality constraints. The Pueblo solver tries to avoid a large loss in performance by watching more than the minimum number (but not all) of literals and by limiting the total number of learned pseudo-Boolean constraints [46].

4.4.2 SAT based pseudo-Boolean solver MiniSat+

MiniSat+ is a pseudo-Boolean solver which uses MiniSat as framework for solving the pseudo-Boolean problem. This is done by translating the PB-constraints and the objective function into clauses. This allows the actual problem to be solved by MiniSat.

Translating the pseudo-Boolean constraints

The first step in the translation process is the conversion into a single-output circuit. The second step is the translation of the circuits to clauses by a variation of the Tseitin transformation. MiniSat+ uses three approaches to generate a circuit from a PB-constraint [3]:

- Convert the constraint into a BDD
- Convert the constraint into a network of adders
- Convert the constraint into a network of sorters

During the translation different techniques are used to optimize the structure of the generated SAT problem. Structural hashing is used to reduce the size of the problem, where as extra variables are introduced to get a compact representation. The translation step also tries to preserve as many implications between PB-constraints.

For further details of the translation of the PB-constraints to SAT, the reader is referred to [3].

Translating the objective function

The minimization-function can be translated using the same approaches as denoted for the translation of the PB-constraints. Since the minimization-function usually contains a large number of pseudo-Boolean variables, the translation to SAT generates a large number of clauses. A minimal satisfying assignment can be found by iterative calls to the solver [3]. A high-level description of the algorithm used in MiniSat+ is given in 4.1.

If the problem requires a large number of iterations, a large number of constraints will be added. This can be very time ineffective for the overall performance. An option would be to replace the previous constraint ($f(x) < k$) with the new constraint, however by removing this constraint the pruning power of the learned clause will be reduced. Other mechanisms prevent this from hurting the solver too much [3].

Algorithm 4.1 MiniSat+ algorithm

Run solver to get initial solution $f(x_0) = k$
while problem is satisfiable **do**
  Add the constraint $f(x) < k$
  Solve the problem using MiniSat and update k
end while
return k as the optimum value

Chapter 5

Facility location problem

The facility location problem is an optimization problem that deals with the positioning of facilities. The problem defines a set of possible locations for the facilities, and a number of locations with certain demands needed to be served by these facilities.

A solution to this problem is a selection of facility locations and the assignment of demand locations to these facilities. The optimum solution is the solution with the smallest cost. The cost are commonly defined by the cost for opening a facility and the cost for transport of the demands. Different variants exist with slight difference in cost calculations, demand definitions and facility limitations. A description of the different variants is available in Section 5.1.

The complexity of the facility location problem depends on the variant in use and is presented in Section 5.2. The most important variant is $\mathcal{NP}$-hard and therefore difficult to solve. Both approximation and deterministic methods exist in order to solve a problem. The approximation algorithms will generate a good solution fast and the deterministic method will find the optimum solution in a longer time span. Approximation algorithms and techniques will be discussed in Section 5.3. Section 5.5 shows how the facility location problem can be solved deterministically.

5.1 Description of variants

Different variants of the facility location problem exist. There variants can differ based on cost calculations, whether demands are splittable, and whether facilities have unlimited capacities. The goal however always remains the same; serve all locations for the smallest possible cost. This is a selection of the most important variants [47]:

- **Uncapacitated facility location (UFL)**, is one of the problems most extensively studied in the literature. The cost are defined by the opening costs of the facilities and the transport cost from facility to locations.
- **Uncapacitated linear-cost facility location problem (ULFL)**, uses a different cost calculation method compared to UFL. The setup cost for a facility depends on the number of locations that will be served from this facility. The setup costs are dividable in fixed cost and mariginal cost. The fixed cost are the cost for opening the facility and the marginal cost are the extra cost for serving each location which is connected to the facility. This leads to the following cost function [48]:

$$f_i(k) = \begin{cases} 0, & k = 0 \\ a_i k + b_i, & k > 0 \end{cases}$$

- **Uncapacitated k-median problem (UKM)**, differs on two aspects from UFL; it has no initial setup cost for each facility, and there is a limit k on the number of facilities that can be selected. This results in a problem where k facilities should be selected to gain the minimal price for serving the demand from the locations. Since there is no initial setup cost for opening a facility, there will be exactly k open facilities in almost all cases.

In the previous mentioned problem variants, the default is to use facilities that have an unlimited capacity and they have demands from the locations which are unsplittable. There exist problem variants that do not have unlimited capacity; they have a bound M on the total demand which can be transported from a facility. They are denoted by a 'C' in the abbreviation, for example UFL is the uncapacitated variant, where CFL is the capacitated variant.

In the case of capacitated location problems it is possible to have splittable demands[1]. In this case the demand of each location can be split between multiple facilities, in contrary to the unsplittable demands where only one facility can serve the demand of a location. These versions are notated with an extra 'S' or 'U' depending on either splittable or unsplittable demands. For example the CFLS, denotes the capacitated facility location problem with splittable demands.

An overview of the different variants of the facility location problems is presented in Table 5.1.

Table 5.1: Overview of the facility location variants

Variant	**Capacitated?**	**Splittable?**	**Cost calculation**	**Maximum number of facilities**
UFLU	No	No	transport and facility cost	No limit
ULFLU	No	No	linear in connected nr of facilities	No limit
UKMU	No	No	transport cost only	k
CFLS	Yes	Yes	transport and facility cost	No limit
CFLU	Yes	No	transport and facility cost	No limit
CLFLS	Yes	Yes	linear in connected nr of facilities	No limit
CLFLU	Yes	No	linear in connected nr of facilities	No limit
CKMS	Yes	Yes	transport cost only	k
CKMU	Yes	No	transport cost only	k

5.2 Problem class

UFL is known to be an $\mathcal{NP}$-hard problem [8]. The other problems can be proven to be of the same problem class. There is however one exception: in the case of a capacitated version with splittable demands (CFLS), given a set S of open facilities, an optimal solution can be given in polynomial time, by solving a special instance of the transportation problem [49].

5.3 General approximation techniques

Since the facility location problem is known to be $\mathcal{NP}$-hard, the solving methods currently used are often approximation algorithms or heuristics. This is of course due to the fact that solving $\mathcal{NP}$-hard problems deterministically takes noticeably longer. On the other hand, while being an approximation algorithm it will not guarantee to find the best solution. Often it may not be necessary to yield the best solution, a

[1] This is not possible in the uncapacitated version.

near best solution is adequate. However in some cases the extra running time might earn back the extra investment in finding the optimal solution while using deterministic methodology.

In this section some approximation techniques will be introduced. These techniques are used in approximation algorithms for the facility location problem, which will be introduced in Section 5.4.

5.3.1 LP-rounding

As denoted in Chapter 2, linear programming problems are reasonably easy to solve. On the other hand integer linear programming problems are much harder to solve. Several methods exist which solve ILP problems by removing the integer constraints, and the creation of a LP relaxation. This LP problem can be used to find guidance to the ILP solution. The given solution of the LP relaxation is by definition better or equal to the ILP solution, since it also allows fractional variable values. The resulting solution from the LP relaxation can still assist in the search for the optimum ILP solution. The next step would be to acquire a valid ILP solution from the LP solution, which can be done by rounding the variable values to integer values. This method is called *LP-rounding*.

5.3.2 Primal-dual method

The first primal-dual method for LP problems was formulated by Dantzig, Ford and Fulkerson [50]. The primal-dual method makes use of the fact that each LP problem has a dual LP problem. Recall from Section 2.1 the definitions of the primal and dual problem:

Definition The primal LP problem is defined by

$$\begin{aligned} \text{Max} & \sum_{j=1}^{n} c_j x_j \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for each } i \\ & x_j \geq 0 \quad \text{for each } j \end{aligned} \tag{5.1}$$

Definition The dual problem of 5.1 is defined by:

$$\begin{aligned} \text{Min} & \sum_{i=1}^{m} b_i y_i \\ \text{Subject to:} & \sum_{i=1}^{m} a_{ij} y_i \leq c_j \quad \text{for each } j \\ & y_i \geq 0 \quad \text{for each } i \end{aligned} \tag{5.2}$$

There are various relationships between these two problems. For example: the dual of the dual problem results in the primal problem. Two duality theorems state the relationship between the function values of the primal and dual problem. The *weak duality theorem* states that the function value of the dual at any feasible solution is always greater than or equal to the function value of the primal at any feasible solution. The *strong duality theorem* states that if there exists an optimum solution x^* of the primal, then there exists an optimum solution y^* for the dual problem with the same function value; $\sum_{i=1}^{m} b_i y_i^* = \sum_{j=1}^{n} c_j x_j^*$.

Given two feasible solutions for the primal and dual problem, $\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j$ indicates the gap between the two solutions, sometimes referenced to as the *duality gap*. The optimums are found when there is no gap, the duality gap can therefore indicate how far the current solution is off from the

optimum solution. This property is used in primal-dual methods to converge to an optimum or good solution.

Complementary slackness
Another relationship between primal and dual exist from their definitions; the number of constraints of the one problem is equal to the number of variables in the other problem and vice versa. This suggest that constraints in one problem are complementary to variables in the other problem. The *complementary slackness* states that there can not be slack in both a constraint and its associated variable. Slack means that the variable or constraints in non-binding, e.g. non zero. In other words, the associated variable of a non-binding constraint should be equal to zero. On the other hand if a variable has slack, its associated constraint should be binding. This allows to easily solve the dual, when the solution of the primal is known.

From the duality theorems the following equations can be derived:

$$c \times x \leq yAx \leq b \times y \tag{5.3}$$

and:

$$c \times x' = y'Ax' = b \times y' \rightarrow (c - y'A) \times x' \tag{5.4}$$

For each component j of $y'A$, there should hold $(c - y'A)_j * x'_j = 0$, since $(c - y'A) > 0$ (because y' is feasible), and $x'_j > 0$. This can only happen if either $(c - y'A)_j$ or x'_j is equal to 0. This is exactly what the complementary slackness theorem states.

Definition An approximation algorithm is used to find a good solution, which will probably not be the optimal solution for the primal or dual problem. This suggest the possibility to drop the dual complementary slackness conditions. The idea is to simultaneously find an approximate solution to the ILP problem and a feasible solution to the dual of the LP relaxation [51].

An example of a general primal-dual method is given in Algorithm 5.1 [51].

Algorithm 5.1 Primal-dual method

Initialize, dual: $y \leftarrow 0$, primal: infeasible
while no feasible x satisfying the (primal) complementary slackness conditions exists **do**
  Acquire direction of increase for the dual (toward a feasible primal)
end while
return solution x

where the *primal complementary slackness* conditions are: $x_j > 0 \rightarrow \sum_{j=1}^{n} a_{ij} y_i = c_j$.

The primal-dual method for approximation algorithms is defined similarly; within the complementary slackness conditions only the primal conditions are used opposed to both primal and dual conditions.

Another option for an approximation algorithm is to use a relaxed version of the complementary slackness conditions. A relaxed version of the dual complementary slackness conditions can be defined as follows: for all i, either $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j = \alpha b_i$. In other words, if $y_i \neq 0$ the constraint should not be binding, but it should not be too far from binding. These relaxed version of the dual complementary slackness conditions are also known as the *α-approximate dual complementary slackness conditions*.

If x and y are feasible solutions satisfying the primal complementary slackness conditions and the α-approximate dual complementary slackness conditions, then x is an *α-approximate solution*. An α-approximate solution is a solution with at most α times the optimum solution. An *α-approximation algorithm* is an approximation algorithm that will guarantee to find a solution which will be at most α times the cost of the optimum solution.

5.3.3 Dual fitting

A dual fitting algorithm uses the primal and dual LP relaxation of a certain problem. The algorithm iteratively updates the primal and dual, although it is not an actual primal-dual method. This is because the dual computed solution is in general infeasible. The value of the primal integral solution resulting from this algorithm, is bounded by the value of the dual. The important phase in the algorithm is the shrinkage of the dual by a certain factor (γ) and proof show that it is feasible, i.e. it fits the problem instance. The new dual is then a lower bound for the optimum value, where γ is the approximation guarantee of the algorithm. [52]

5.3.4 Local search

Local search is a meta heuristic for solving various difficult optimization problems. Local search algorithms start with a feasible solution and move to other (better) feasible solutions until an optimum is found. The algorithm iteratively moves along *neighbouring solutions*, which can only be done if a neighbor of the current solution can be defined. For example, a neighbor of a solution of a facility location problem is another solution with one different facility selected. A local search algorithm stops when it meets its time criterion, or it cannot improve its current solution over a number of iterations.

The resulting solution of a local search algorithm is therefore a local optimum, since it search within a neighborhood. The best solution found by the algorithm may therefore be far from the global optimum solution. The chance of finding a better solution may be increased by starting the local search algorithm from various starting solutions. In this case multiple neighbourhoods will be searched.

5.3.5 Greedy augmentation

Greedy augmentation is an iterative process. The idea is to pick a new node n, which maximizes the ratio of decreased cost to the cost of picking node n. The process terminates when there is no node selectable that decreases the total cost. [53]

The greedy augmentation process within facility location approximation algorithms use the following ratio [54]:

$$\frac{C_t - C^{'} - f_n}{f_n} \tag{5.5}$$

where C_t is defined as the cost at iteration t and $C^{'}$ as the cost while using facility n at iteration t. The cost of facility u are defined as f_n.

The goal is to pick the node with the largest ratio of decrease in cost to the increase in cost. This can improve the approximation factor of an approximation algorithm [54].

5.4 Approximation solving algorithms

In order to acquire a broad overview of the possible methods to solve the facility location problem. The first set of solving methods being reviewed, is the set of approximation algorithms which will be reviewed in this section. The other set, the deterministic based methods, will be discussed in Section 5.5.

In this section some of the techniques used in approximation algorithms to solve facility location problems will be introduced. This introduction will not be in depth, since the introduction will merely be an expansion on the knowledge of solving methods for the facility location problems. The main focus remains on solving the problem deterministically.

5.4.1 JMS algorithm

The JMS algorithm is an approximation algorithm with a performance guarantee of 1.61; it guarantees to find a solution which is at most 1.61 times as large as the optimal solution. The algorithm is a *bifactor* approximation algorithm, since it has a separate approximation number for the facility cost and the transport cost. The high level description of the JMS algorithm is presented in Algorithm 5.2 [51, 55].

Algorithm 5.2 JMS algorithm

Initialize, budget for each city is 0, each location is unconnected, and each facility is unopened
while an unconnected location exists **do**
  increase budget of unconnected locations, until:
  if total offer of an unopened facility i = its opening cost **then**
    open facility i, and for each location j which has an offer > 0 to facility i connect j to i
  end if
  if an unconnected location j and open facility i exist, where the budget of j = opening cost c_{ij} **then**
    connect j to i
  end if
end while
Solution is found.

5.4.2 MYZ algorithm

The MYZ algorithm is based upon the JMS algorithm and has an approximation factor of 1.52. It consists of two phases, the goal in the first phase is to pick those facilities which are very economical. This is obtained by a scaling factor, which increases the weight of the facility opposed to the transport cost in the objective function.

The second phase is a variant of the greedy augmentation described in 5.3.5. The goal in the second phase is to try to reduce the total cost by opening an unopened facility. In the second phase at iteration t, the cost of the facility equals $(\delta - t)f_i$. The first facility which is opened is the facility which has the lowest value for t (or the highest value for $\delta - t$). For an opened facility in phase two the following holds: $(\delta - t)f_i = C_t - C^{'}$, which means that the algorithm in fact does pick the facility with the highest ratio $\frac{C_t - C^{'} - f_n}{f_n}$ (see Section 5.3.5). A high level description of the MYZ algorithm is available in Algorithm 5.3. [51]

Algorithm 5.3 MYZ algorithm

Phase 1
Scale the opening costs of all facilities by a constant factor δ.
Run JMS algorithm to find a solution
Phase 2
while facility cost > original cost **do**
  Scale down the facility cost with 1.
  if a facility could be opened without increasing the total cost **then**
    Open this facility and connect the locations to it.
  end if
end while

5.5 ILP formulations of facility location variants

The previously denoted approximation algorithms guarantee to deliver a solution which is at most 1.52 times larger than the optimum solution. In certain cases this result is unacceptable and the optimum value must be acquired. This can be done by using deterministic methods, which guarantee to find the optimum solution although it may need a far greater amount of time.

In order to use deterministic methods the problem must be converted to a readable input format for a deterministic solver. The facility location can easily be notated in ILP formulation. Most of the variants define a cost minimization function, a set of constraints ensuring that each location is served and a set of constraints to guarantee that a facility is open when it is used to serve a location. The ILP definition of the facility location problem can then of course be used in every ILP solver. The pseudo-Boolean solvers can also be used, since all the integer variables are restricted to the values 0 or 1.

This section continues with the ILP formulation of the facility location problems. Several different variants of the facility location problem will be investigated.

5.5.1 Uncapacitated facility location

The facility location problems exist of a number of facilities (i), and a number of locations (j). Each facility has a nonnegative value f_i as the setup cost of facility i. Within the problem each location is connected to all facilities, this results in a fully connected graph. Each connection between location and facility also has a nonnegative value c_{ij} associated with it. In order to determine which facilities should be opened and which facility should serve which locations, the following binary variables are introduced: y_i and x_{ij}. The variables y_i denotes whether the facility i should be opened, and the x_{ij} variables denote whether location j is served from facility i. This results in the following ILP notation for the uncapacitated facility location problem [56, 57]:

$$
\begin{aligned}
min \sum_i y_i f_i + \sum_{ij} x_{ij} c_{ij} d_j & \\
x_{ij} \leq y_i & \quad \text{for each } i, j \\
\sum_i x_{ij} = 1 & \quad \text{for each } j \\
x_{ij}, y_i \in 0, 1 & \quad \text{for each } i, j
\end{aligned}
$$

where y_i represents whether facility i is chosen in the solution. Whether location j is assigned to facility i is defined by x_{ij}. The cost of facility i is represented by f_i, the demand at location j by d_j and the distance from facility i to location j by c_{ij}.

5.5.2 Capacitated facility Location

As denoted earlier, the capacitated facility location problem is similar to the uncapacitated facility location problem. The capacitated facility location problem has a constraint on the number of locations a facility can serve. The maximum number of locations a facility i can serve is denoted with q_i.

The definition of the CFLP variants are the same as the UFLP definition, with the addition of the following extra constraint:

$$\sum_j x_{ij} \leq q_i \text{ for each } \quad i$$

where q_i represents the number of locations facility i can serve. Alternatively this can be written in the following form where the demands of the location and the demand deliverable by the facility are taken

into account. The coefficients d_j represent the demand in location j and u_i is the demand deliverable by facility i.

$$\sum_j d_j x_{ij} \leq u_i q_i \text{ for each } \quad i$$

5.5.3 The linear-cost facility location problem

The linear-cost facility location problem differs from the uncapacitated facility location problem in the cost function of a facility. In the uncapacitated facility location problem the cost are f_i for opening a facility. The cost for a facility in the linear-cost facility location problem depend on the number of locations the facility serves. The cost increase linearly as the number of served locations grow. This leads to the following definition [48] : [2]

$$f_i(k) = \begin{cases} 0, & k = 0 \\ a_i k + b_i, & k > 0 \end{cases}$$

where k is the number of locations that facility i serves and $f_i(k)$ defines the cost of facility i when connected to k locations. The variable a_i represents the marginal costs for facility i (extra costs when a location is served by a facility) and b_i the fixed costs for facility i (startup costs in order to use a facility).

This can be rewritten to $y_i b_i + \sum_j x_{ij} a_i$. The formulation of the linear-cost facility location problem is identical to the uncapacitated facility location problem except for the minimization function. The definition of the minimization function should be altered into:

$$min \sum_i y_i b_i + \sum_{ij} x_{ij} a_i + \sum_{ij} x_{ij} c_{ij} d_j$$

5.5.4 The metric k-median problem

The metric k-median problem differs from the uncapacitated facility location problem on two aspects; it does not have initial opening costs for a facility and it has a limit on the number of open facilities. This structurally changes the problem by requiring a maximum number of open facilities. The limit on open facilities is equal to k, which should be part of the input.

As a basis for the ILP formulation of this problem the uncapacitated facility location problem can be used. In order to reflect the problem correctly, the minimization function from the UFLP should be altered and a new constraint should be added. The new minimization function should be:

$$min \sum_{ij} x_{ij} c_{ij} d_j$$

And the new constraint:

$$\sum_i y_i \leq k$$

where k is the maximum number of open facilities.

[2] It should be denoted that the costs should always be nonnegative.

5.5.5 Splittable demands

In the previous subsections variants which do not have splittable demands were discussed. Each variant can in principle also be used with splittable demands. This however leads to new definitions of the minimization function and the constraints. Extra constraints are required to ensure that each location acquires its demand.

For example the uncapacitated facility location problem with splittable demands has the following ILP formulation:

$$
\begin{aligned}
min \sum_i y_i f_i + \sum_{ij} x_{ij} \frac{c_{ij}}{d_j} & \\
\sum_i x_{ij} = d_j & \quad \text{for each } j \\
\sum_j x_{ij} \leq (\sum_j d_j) y_i & \quad \text{for each } i \\
0 \leq x_{ij} \leq d_i & \quad \text{for each } i, j \\
y_i \in 0, 1 & \quad \text{for each } i, j
\end{aligned}
$$

where d_j represents the demand at location j and c_{ij} the distance from facility i to location j.

Overview

Table 5.2 presents an overview of the various ILP formulations from the mentioned facility location variant in this section.

Table 5.2 shows that UFLU and LFLU both have the least constraints, LFLU however has a far more complex minimization function. This probably would make UFLU the most interesting variant to investigate in the search for an efficient method to solve a facility location problem deterministically. A possible alternative is the p-median variant, since it has only one extra (simple) constraint and it has a simpler cost calculation function compared to UFLU.

Since the UFLU variant is the most plain and most studied variant of the facility location family, the focus in this thesis is on this variant. When referred to a facility location problem, the UFLU variant is meant unless stated otherwise.

Table 5.2: Overview of ILP formulations of the facility location variants

Variant	**Min function**	**Constraints**	
UFLU	$min \sum_i y_i f_i + \sum_{ij} x_{ij} c_{ij} d_j$	$x_{ij} \leq y_i$	for each i, j
		$\sum_i x_{ij} = 1$	for each j
		$x_{ij}, y_i \in 0, 1$	for each i, j
CFLU	$min \sum_i y_i f_i + \sum_{ij} x_{ij} c_{ij} d_j$	$x_{ij} \leq y_i$	for each i, j
		$\sum_i x_{ij} = 1$	for each j
		$\sum_j x_{ij} \leq q_i$	for each i
		$x_{ij}, y_i \in 0, 1$	for each i, j
ULFLU	$min \sum_i y_i b_i + \sum_{ij} x_{ij} a_i + \sum_{ij} x_{ij} c_{ij} d_j$	$x_{ij} \leq y_i$	for each i, j
		$\sum_i x_{ij} = 1$	for each j
		$x_{ij}, y_i \in 0, 1$	for each i, j
UKMU	$min \sum_{ij} x_{ij} c_{ij} d_j$	$x_{ij} \leq y_i$	for each i, j
		$\sum_i x_{ij} = 1$	for each j
		$\sum_i y_i \leq k$	
		$x_{ij}, y_i \in 0, 1$	for each i, j
UFLS	$min \sum_i y_i f_i + \sum_{ij} x_{ij} \frac{c_{ij}}{d_j}$	$\sum_i x_{ij} = d_j$	for each j
		$\sum_j x_{ij} \leq (\sum_j d_j) y_i$	for each i
		$0 \leq x_{ij} \leq d_i$	for each i, j
		$y_i \in 0, 1$	for each i, j

Chapter 6

MiniZSat

In the Introduction (Chapter 1) several methods for improvement on SAT based pseudo-Boolean solvers were presented. These improvements have been used to create a new SAT based 'semi' pseudo-Boolean solver. The new solver is based on MiniSat (see Section 3.7) and its name is MiniZSat. Via a translation step, presented in Section 6.6.3, pseudo-Boolean problems can be transformed to accepted input by MiniZSat.

The following three suggestions were made in the Introduction in order to improve the performance of a SAT based pseudo-Boolean solver:

- Embed the optimization function in the solver.
- Use bounding technique to discard assignments which will not lead to the optimum solution.
- Extend the solver with bound conflict analysis.

Embedding the optimization function in the solver ensures that no extra clauses are necessary for a translation of the minimization function. This largely reduced the problem size in various situations, but may also have some negative consequences. A further introduction of this technique is featured in Section 6.2.

As a consequence of embedding the optimization function, an upper bound was necessary in order to discard assignments which will never yield the optimum solution. The upper bound is equal is to the best found solution. The current assignment defines the lower bound. Performance can be improved if the upper and lower bound converge as early as possible. This means that the upper bound should decrease and the lower bound should increase. The usage of both the upper and lower bound will be further explained in Section 6.3.

Whenever an assignment exceeds the upper bound, e.g. the lower bound is equal or larger than the upper bound, a bound conflict occurs. A bound conflict is a conflict which needs to be resolved in order to find the optimum solution. Bound conflict analysis is necessary in order to find the cause of the conflict and to learn from the conflict. There are several options to learn from a conflict which all have their advantages and disadvantages. The analyses and the methods to learn from a conflict will be presented in Section 6.4.

Another important aspect of a deterministic solver is the traversal of the solution space. A deterministic solver such as MiniZSat needs to traverse large parts of the solution space in order to find the optimum solution. The methods used to traverse the search space will therefore have a large impact on the performance of the solver. During the search for the optimum solution a number of solutions will be

found by MiniZSat. These solutions define the upper bound of the solver and will there by cut off certain parts of the solution space. It is therefore of great importance that a good solution is found as fast as possible. A discussion on traversal of the search space continues in Section 6.5.

As denoted earlier MiniZSat is not a complete pseudo-Boolean solver. The usage of a translation step by MiniSat+ ensures that all pseudo-Boolean problems can be accepted by MiniZSat. The input format accepted by MiniZSat and the translation step will be introduced in Section 6.6.

6.1 Introduction

The global approach used in MiniZSat is to use the MiniSat solver to solve the problem. While MiniSat is solving the problem, the cost of the current assignment is continuously being updated. When the first complete assignment is found the solver stores this assignment as best solution found. The solver continues with an improved upper bound; the best solution found. Whenever the cost of the current assignment exceeds the upper bound, the current assignment will be rejected.

MiniZSat starts with an upper bound set to MAX, which ensures that the first complete assignment will always improve the upper bound. The algorithm starts as MiniSat by assigning variables. Propagation is used in order to propagate unit clauses. The solver is in conflict whenever a logical conflict occurs or in the situation where the current solution is equal or larger than the bound solution. In the latter case a bound conflict or bound violation has occurred. A current solution which is larger or equal to the bound solution will not generate the optimal solution and therefore the solver should backtrack.

After a conflict the solver may backtrack or stop. The solver stops when a conflicting assignment at decision level 0 is found. The solver will then return the best found solution as the optimum solution or if no solution was found it returns unsatisfiable.

If no conflicts have been found and all the variables have values assigned, then *a* solution to the problem is found. This solution is per definition the best solution found, because otherwise a bound conflict would occur. The solver stores this assignment and sets the upper bound to the value of the current assignment. The solver continues the search for the optimum solution with the new upper bound. A high-level description of the structure of MiniZSat is given in Algorithm 6.1.

6.2 Embedding the optimization function

The approach MiniSat+ uses in order to solve pseudo-Boolean problems is divided in two steps. The first step is the search for *a* feasible solution. The next step is the translation of the optimization function in clauses. The translation uses the value of the solution found in the first step. The solver adds the clauses which represent the following constraint $f(x) < k$, where $f(x)$ is the optimization function and k is the value of the solution found in the first step. A complete assignment will then be a better solution which can in turn be used to make a new translation of the optimization function in clauses (with an updated value for k).

Depending on the complexity of the minimization function this may lead to the addition of a large number of clauses. The disadvantage of a large number of clauses is that the translated problem increases in size. As a result more watches need to be checked, more space is required and more propagation will occur.

An alternative to solve the problem is to use a translation of the pseudo-Boolean constraints to SAT clauses and embed the optimization function in the solver. The solver will update the cost of the current

Algorithm 6.1 MiniZSat algorithm

```
upper bound := MAX
solution := NULL
while true do
  assignVar()
  status = propagate()
  if current solution >= upper bound or status is CONFLICT then
    btlevel := analyze()
    if btlevel is 0 then
      if upper bound = MAX then
        return UNSAT
      else
        return solution
      end if
    else
      backtrack(btlevel)
    end if
  else if nr of unassigned variables is 0 then
    upper bound := current solution
    solution := current assignment
  end if
end while
```

assignment according to costs represented by the minimization function. The solver will in turn detect whether an assignment can not lead to the optimum solution. If such a situation occurs the solver will discard the current assignment and use backtracking in order to continue the search for the optimum solution.

The advantage of embedding the optimization function is the decrease in problem size, because the addition of extra clauses for the translation of the minimization function are not required. A small disadvantage is that the solver needs to keep track of the cost of the current assignment, however the overhead seems to be minimal from this procedure. Another disadvantage is cause by the absence of the extra clauses for the translation of the minimization function. These clauses may guide the solver to a better solution, because variables may be propagated by the extra clauses.

6.3 Bounding techniques

Recall that bounding finds a lower and upper bound for the optimal solution within a feasible region. The upper bound is globally defined as the value of the best solution found. The lower bound is the limit of a subset of the problem, e.g. it defines the lowest possible solution given a subset of the problem.

In MiniZSat the best known solution is maintained as the upper bound. The lower bound is defined as the (minimal) cost of the current assignment. Whenever the lower bound is equal or larger than the upper bound, the current assignment will not find a better solution with the current assignment. In this chapter different techniques will be introduced which can improve the bounds in use. An improved upper or lower bound may decrease the solution space to search through.

The usage of an upper bound for MiniZSat has been first introduced in Section 6.1. The usage of an upper bound obtained from an external source will be introduced in 6.3.1. Lower bound techniques will be discussed in Section 6.3.2 and Section 6.3.3.

6.3.1 Using a known upper bound

The execution time of an algorithm searching a solution space in an exhaustive manner largely depends on the size of the solution space. SAT-solvers use exhaustive searching techniques and can benefit largely when the search space is minimized. Generally speaking for SAT-solvers, the search space is minimized by using preprocessing techniques that simplify clauses.

In the case of a minimization problem, the search space can be reduced by finding an upper-bound. All branches which have a higher function value than the upper-bound are to be rejected since they will never find a solution with a value lower than the upper-bound.

For several problem instances good approximation algorithms exist. These approximation algorithms run quite fast and give a reasonable solution. This solution can in turn be used as an upper-bound for a solver, such as MiniZSat. This technique is not new and is extensively used in ILP solvers (see primal heuristics in Section 2.6.5). ILP solvers use a generic heuristic as a preprocessing step on all problems.

6.3.2 Initial lower bound

In this section a form of lower bound technique for the MiniZSat solver is described. MiniZSat ensures that all variables in the minimization function have positive coefficients and that only setting a variable to `TRUE` enlarges the function value. This ensures that a the function value of a partial assignment can only increase. A further discussion on this technique is available in the next section.

If a clause contains no negated literals, satisfying this clause requires setting at least one literal to `TRUE`. The function value will as a result always increase, if the coefficients of the variables in the clause are all larger than 0. The increase in function value will be larger or equal to the smallest coefficient among the variables from the clause.

However if negations appear in the clause, there will be a literal which can satisfy the clause by setting the value of the variable to `FALSE`. Since setting the literal to `FALSE` does not affect the function value, there is no direct increase in the function value.

Definition The *minimum cost of a clause* is defined as the minimum from all the coefficients of the variables in the clause or 0 if a negated literal is apparent in the clause.

Example. In this Example the minimum cost for the clauses c_1, c_2, c_3 and c_4 are $1, 2, 0$ and 0. In c_1, literal x_1 has the smallest coefficient associated: 1 and therefore the minimum cost for this clause is 1. In c_2 the smallest associated coefficient is 2 for literal x_2. Clause c_3 contains a negated literal($\neg x_1$); setting the variable x_1 to `FALSE` satisfies the clause and it does not increase the minimization function. The minimum cost are therefore 0. In the last clause, c_4, the minimum cost are also 0 since x_4 has no coefficient associated in the minimization function (the coefficients of this variable is 0).

$$
\begin{aligned}
&\min : 1x_1 + 2x_2 + 3x_3 \\
&\quad c_1 = x_1 \vee x_2 \vee x_3 \\
&\quad c_2 = x_2 \vee x_3 \\
&\quad c_3 = \neg x_1 \vee x_2 \vee x_3 \\
&\quad c_4 = x_2 \vee x_3 \vee x_4
\end{aligned}
\tag{6.1}
$$

Example. The following example shows how the minimum cost can be used to rewrite the minimization function. The problem consists of one clause $x_1 \vee x_2 \vee x_3$ with a minimum cost of 1. A solution to this problem

shall therefore always have a minimum cost of 1. This knowledge of the solution can be used to rewrite the minimization function, as is visualized in 6.2.

$$\begin{aligned} \min &: 1x_1 + 2x_2 + 3x_3 \\ \min(reduced) &: 1 + 1x_2 + 2x_3 \\ & x_1 \vee x_2 \vee x_3 \end{aligned} \tag{6.2}$$

The rewriting of the minimization function as shown in the previous examples, corresponds to a form of *lower bound* technique; it defines a value which bounds the solution. It means that no solution can be found with a value smaller than the lower bound. In the MiniZSat solver a higher lower bound may lead to bound conflict earlier on in the search and there by reducing the search space. For example: if the lower bound is 10 and the upper bound is 15, setting a literal with coefficient 6 would cause a bound conflict. If no lower bound was used, setting this literal would not cause a bound conflict and unnecessary assignments are made which cannot lead to an improved solution.

Since each clause needs to be satisfied in order to find a satisfying assignment, the minimization function can be altered in the following manner:

1. For all clauses
   (a) Add the minimum cost for a clause to the minimization function.
   (b) For all variables used in that clause: subtract the minimum cost for a clause from the coefficient of the variable.

Example

In order to show that this method works for *certain* problems, an example is shown using a simple facility location problem (see Chapter 5). The following instance is defined:

$$\begin{aligned} &\min\ 7y_1 + 2y_2 + 5x_1y_1 + 3x_2y_1 + 2x_3y_1 + 2x_1y_2 + 5x_2y_2 + 8x_3y_2 \\ &c_1 : x_1y_1 \vee x_1y_2 \\ &c_2 : x_2y_1 \vee x_2y_2 \\ &c_3 : x_3y_1 \vee x_3y_2 \\ &c_4 : \neg x_1y_1 \vee y_1 \\ &c_5 : \neg x_2y_1 \vee y_1 \\ &c_6 : \neg x_3y_1 \vee y_1 \\ &c_7 : \neg x_1y_2 \vee y_2 \\ &c_8 : \neg x_2y_2 \vee y_2 \\ &c_9 : \neg x_3y_2 \vee y_2 \end{aligned} \tag{6.3}$$

In clause c_1, either x_1y_1 or x_1y_2 needs to be selected. The minimum cost of this selection is 2 for variable x_1y_2, which means the objective function can be reduced to:

$$\min\ 2 + 7y_1 + 2y_2 + 3x_1y_1 + 3x_2y_1 + 2x_3y_1 + 5x_2y_2 + 8x_3y_2 \tag{6.4}$$

The process continues with clause c_2, where x_2y_1 is the smallest:

$$\min\ 5 + 7y_1 + 2y_2 + 3x_1y_1 + 2x_3y_1 + 2x_2y_2 + 8x_3y_2 \tag{6.5}$$

The last possible reduction is in clause c_3, in which x_3y_1 can be reduced:

$$\min\ 7 + 7y_1 + 2y_2 + 3x_1y_1 + 2x_2y_2 + 6x_3y_2 \tag{6.6}$$

The remaining clauses do not give the ability to further reduce the minimization function. The correct optimal solution $y_1, y_2, x_1y_2, x_2y_1, x_3y_1 = 1$, has a value of 16 in the original problem and the reduced minimization function (6.6) also has the same value for this solution.

The previous example shows that the facility location problem is suitable for the lower bound technique. However, the lower bound technique as described earlier is not suitable for all problems.

Problems

Unfortunately this lower bound technique does not always yield correct results. In the following examples some problems with this lower bound technique will be presented.

Example. This example shows that the lower bound technique may result in incorrect values for the optimum solution.

$$\begin{aligned} \min &: 1x_1 + 2x_2 + 3x_3 + 4x_4 \\ \min(\mathit{reduced}) &: 2 + 1x_3 + 3x_4 \\ c_1 &: \neg x_1 \\ c_2 &: \neg x_2 \\ c_3 &: x_1 \vee x_2 \vee x_3 \vee x_4 \\ c_4 &: x_2 \vee x_3 \\ c_5 &: x_2 \vee x_4 \end{aligned} \tag{6.7}$$

The optimum solution is $x_3 = 1, x_4 = 1$ with a cost of 6 in the reduced version, where it is 7 in the original problem. In this case the problem could have been avoided by using clause c_4 or c_5 in stead of c_3 for increasing the lower bound.

Example. The following example shows that the lower bound technique may even incorrectly mark a solution as optimal.

$$\begin{aligned} \min &: 1x_1 + 2x_2 + 3x_3 + 4x_4 \\ \min(\mathit{reduced}) &: 1 + 1x_2 + 2x_3 + 4x_4 \\ & x_1 \vee x_2 \vee x_3 \\ & x_2 \vee x_4 \\ & x_3 \vee x_4 \end{aligned} \tag{6.8}$$

The solver finds an optimal solution that has a function value of 4 which should be 5. The optimum solution is $x_2, x_3 = 1$ and in addition it may select x_1 'for free'. The solution $x_1, x_2, x_3 = 1$ will in this case also be marked as optimal, where it should not have been. Another optimal solutin with cost 5, x_1, x_4, is no longer marks as optimal, since the other optimal solution has a cost of 4. In this specific example *any* reduction on one of the clauses will yield an incorrect result.

Cause of the problems

From the previous examples it can be seen that this lower bound technique does not always yield correct results. Incorrect results appear when multiple literals satisfy one clause and they all have (positive)

coefficients in the original minimization function. Since the coefficients of multiple literals in the clause are reduced with an amount α and the amount of α is added only once to the current solution value, the selection of multiple literals yield an incorrect result. In the third clause of Example 6.7, multiple variables are set to `TRUE` which all have coefficients in the minimization function. In this example $\alpha = 1$, which is added to the minimization function once and used in the result twice. The actual result is therefore 1 too small; $1 - 2 * 1 = -1$.

Usability of the lower bound technique

In the examples (6.7 and 6.8) previously denoted, it was shown that the technique of maintaining a lower bound can not always be successfully used. It is essential that the solver knows which reductions of the minimization function are allowed before updating the lower bound.

A safe, but not very sophisticated, method would be to only allow the usage of the minimum cost of clauses which consists of variables appearing only in that clause. From the minimization perspective it can be proven that the solver will always find the optimal solution with the correct value, since the solver has the option to satisfy the clause by setting one literal. Other assignments will eventually lead to bound conflicts. In this situation the variable corresponding to the smallest coefficient in the clause, could automatically be set to `TRUE`. The clause can also be seen as a subproblem which can be solved separately.

A more sophisticated technique is to allow reduction of variables that appear only once in clauses positively, and arbitrarily multiple times in a clause with a negation. To use a clause in order to increase the lower bound all its variables should be in positive form and have a positive coefficient. This results in a situation where each variable which may be reduced, is reduced in at most one clause. The restriction on the number of positive occurrences of a variable ensures that only one literal in the clause used for reduction is required to satisfy the clause. In other words, other clauses in the problem can not force one of those variables to `TRUE`, because all of the variables in that particular clause only occur positively in that clause.

It is still remains possible to obtain incorrect (intermediate) function values, because over satisfaction of a clause used for reduction is still allowed. The minimization principle and the fact that only one literal in such a clause is needed to satisfy the clause, ensure that the optimum solution will have only one literal satisfying the clause. This technique can therefore guarantee that the optimum solution has a correct function value.

The drawback of this method is that extra checking is required in order to guarantee the conditions under which the reduction of the minimization function is allowed. This can be done in a preprocessing step.

MiniZSat method

MiniZSat performs a preprocessing step in which the previously mentioned lower bound technique, denoted as 'more sophisticated', is employed. The lower bound is improved when a clauses consists of positive literals with positive coefficients only and the variables only appear once in clauses positively, and arbitrarily multiple times in a clause with a negation.

MiniZSat also employs another *initial lower bound technique*. It is a simple and correct method which updates the lower bound while parsing the input, hence the name initial lower bound. The lower bound is updated whenever a clause only contains variables which have coefficients in the minimization function larger than zero and the clause consists of positive literals only. In that case the lower bound can be increased and watches should be added to those variables. Whenever one of these variables gets the value `TRUE` assigned; remove the watches and increase the value for the current assignment with *variable*

coefficient - watch value (where *watch value* = minimum cost of the clause). Via this way the increase of the lower bound is subtracted only once and therefore a correct function value is guaranteed.

In order to correctly place the watches a *shadow cost table* should be used, in which the coefficient of the variables are updated while placing the watches. Whenever a watch is placed, the coefficients of the variables in the clause are reduced by the value of the minimum cost of the clause. When at least one variable has a coefficient of 0 in the 'shadow cost table' or a negative literal is present in the clause, a watch may not be placed.

The most important advantages of this method are:

- a large number of clauses are suitable for updating the lower bound and
- the correct function value is guaranteed at *any* time while solving.

The drawback of this method is that it requires the solver to check for a watch whenever a literal is set. This means an extra check that may need to be performed millions of times. This is of course a loss in performance. Next to the check, the solver needs to update the watches when necessary.

Another problem is that bound conflict reduction is more difficult to deploy, which will be featured in Section 6.4.1. For certain variables the *normal* cost should be used, whereas in other cases the cost from the shadow cost table should be taken. This problem is solved by maintaining the cost for each variable in the bound trail.

In the current version it is only possible to use one watch per variable. In theory this number could be unlimited. This would add a lot of extra complexity to the solver and is therefore left out of the initial version of MiniZSat.

Example. In Figure 6.1 an example shows the working of the initial lower bound technique as deployed in MiniZSat. The example shows a small part of the complete problem. In the initial state the solution value is 1. When x_3 is set to `TRUE` ($x_3 = 1$) the solution value is increased by the coefficient of variable x_3, and reduced by the value of the watch. In this case the coefficient is 3 and the value of the watch is 1, which results in a value of $1 + (3 - 1) = 3$ for the current assignment (which equals the original coefficient). After this assignment the watches should be removed. If for some reason another variable of the clause gets the value `TRUE` assigned, the increase in function value equals the coefficient of the variable.

6.3.3 Dynamic lower bound

The principle of improving the lower bound can also be applied while solving a problem. This principle is referred to as *dynamic lower bound*.

Example. The following example shows a part of a larger CNF-formula and minimization function:

$$\begin{aligned} &1x_1 + 2x_2 + 3x_3 \\ &\quad x_1 \vee x_2 \vee x_3 \end{aligned} \tag{6.9}$$

which can initially be reduced to:

$$\begin{aligned} &1 + 1x_2 + 2x_3 \\ &\quad x_1 \vee x_2 \vee x_3 \end{aligned} \tag{6.10}$$

If this would be a part of a larger CNF-formula and at a certain moment x_1 would be set to `FALSE`, the minimization function could be changed into:

$$\begin{aligned} &2 + 1x_3 \\ &x_2 \vee x_3 \end{aligned} \tag{6.11}$$

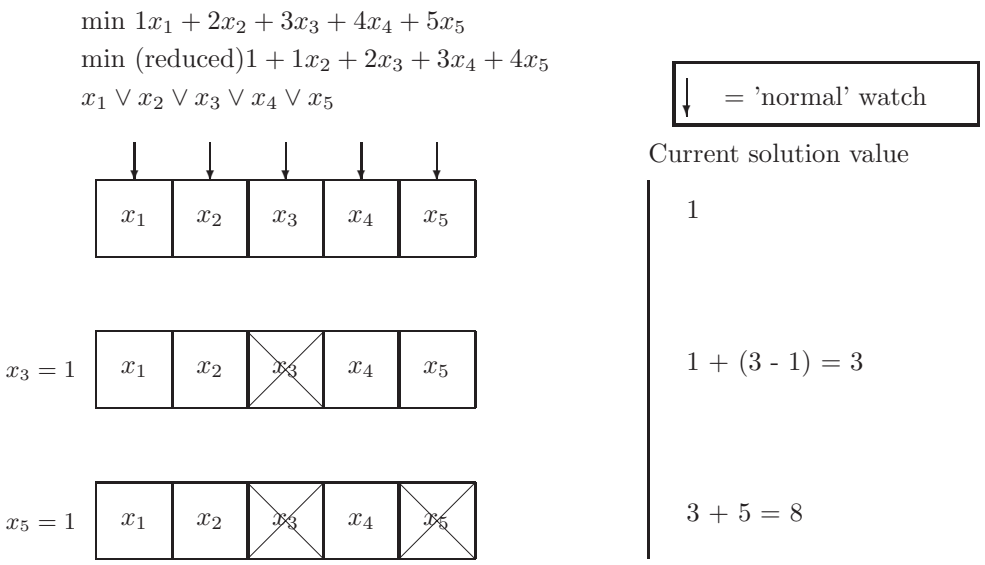


Figure 6.1: Example of the initial lower bound technique, as employed in MiniZSat

This reduction is allowed because the solver still needs to satisfy the clause, and when x_1 is `FALSE`, the new minimization function (6.11) represents the cost for doing so. This reduction is only allowed when x_1 has been set to `FALSE`, and if this assignment is undone, the minimization function needs to be restored into its original form.

This dynamic lower bound technique enables the solver to update the lower bound not only on clauses solely containing positive literals with positive coefficients associated. It is then also possible to mark other clauses for possible improvement on the lower bound. In principle each clause which contains two or more positive literals (belonging to a variable with a positive coefficient) in the minimization function may be suitable for a possible improvement on the lower bound.

A clause contains a set A positive literals with positive coefficients and a set B literals that either occur as negative literal or as positive literal with no positive coefficient. This means that there are at least $2^B - 1$ possibilities to satisfy this clause while the current function value will not increase. Only if all the literals B are *assigned* and do *not* satisfy the clause, then the lower bound may be increased with the smallest coefficient among variables related to the literals from A. Note that there is only a small chance if $|B|$ increases, e.g. the chance equals: $\frac{1}{2^B-1}$. Assuming that each literal has an equal chance of being selected, Table 6.1 shows the chance of being able to use the dynamic lower bound technique.

Disadvantages and problems

Several disadvantages appear while using a dynamic lower bound. The largest disadvantage is the extra overhead necessary in order to verify if and when the lower bound may be increased dynamically. Next to that certain steps need to be undone when the dynamic lower bound is no longer in effect, which adds

Table 6.1: Chance of activating the dynamic lower bound technique with varying number of literals

$\|B\|$	Chance
1	0.25
2	0.125
3	0.0625

additional overhead.

Another disadvantage that may appear is the effectiveness of the learning from a conflict involving a dynamically increased lower bound. Recall that a bound conflict is caused by all the literals increasing the function value. Without using the dynamic lower bound technique only `TRUE` literals may be part of the cause of the conflict. Using the dynamic lower bound, literals with value `FALSE` may also increase the lower bound and are therefore part of the cause of the conflict. As a result learnt constraints may be less strong and learnt constraints are more complex to reduce.

Example. This example shows that the usage of a dynamic lower bound may result in weaker learnt constraints. The following problem is part of a larger problem.

$$\begin{aligned} &1x_2 + 1x_3 + 2x_4 + 3x_4 \\ &x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_4 \end{aligned} \tag{6.12}$$

Currently $x_1, x_2,$ and x_3 have values assigned and do not satisfy the clause. The dynamic lower bound could is therefore equal to 2. If the upper bound is equal to 2, then the cause for the conflict would be $\neg x_1 \wedge \neg x_2 \wedge \neg x_3$. The learnt clause would be $x_1 \vee x_2 \vee x_3$, which could be weaker than the learnt clause obtained without using a dynamic lower bound.

The complexity of the complete algorithm also increases while using bound conflict reduction in combination with dynamic lower bound. The bound conflict reduction should be carefully deployed in order to maintain the conflict and acquire a learnt constraint as strong as possible.

MiniZSat method

In order to update the lower bound dynamically the solver needs to know when the lower bound may be increased. This can be done via a similar watch system as used to identify unit clauses (see the watched literals set 3.4.2). A dynamic watch should be placed on one of the literals in set B. When the dynamic watch on the literal gets a value assigned and does not satisfy the clause, two situations can occur:

1. the clause consists of at least one negative literal or a positive literal with no positive coefficient (e.g. $|B| > 0$) . Set the dynamic watch to one of the these literals and no reduction should be applied.

2. all the remaining literals are positive and have a coefficient larger than 0 (e.g. $|B| = 0$). Remove the dynamic watch, update the lower bound and add 'normal' watches (as described in the initial lower bound method).

Note: the initial version of MiniZSat only deploys the dynamic lower bound when $|B| = 1$, due to the extra complexity opposed to the small chance of increase in performance.

Example. In the following example:
$\min 1x_3 + 2x_4 + 3x_5 + 4x_6 + 5x_7$
$x_1 \vee x_2 \vee \neg x_3 \vee x_4 \vee x_5 \vee x_6 \vee x_7$

the set A consists of $\{x_4, x_5, x_6, x_7\}$ and set $B = \{x_1, x_2, x_3\}$.

The example continues in Figure 6.2.

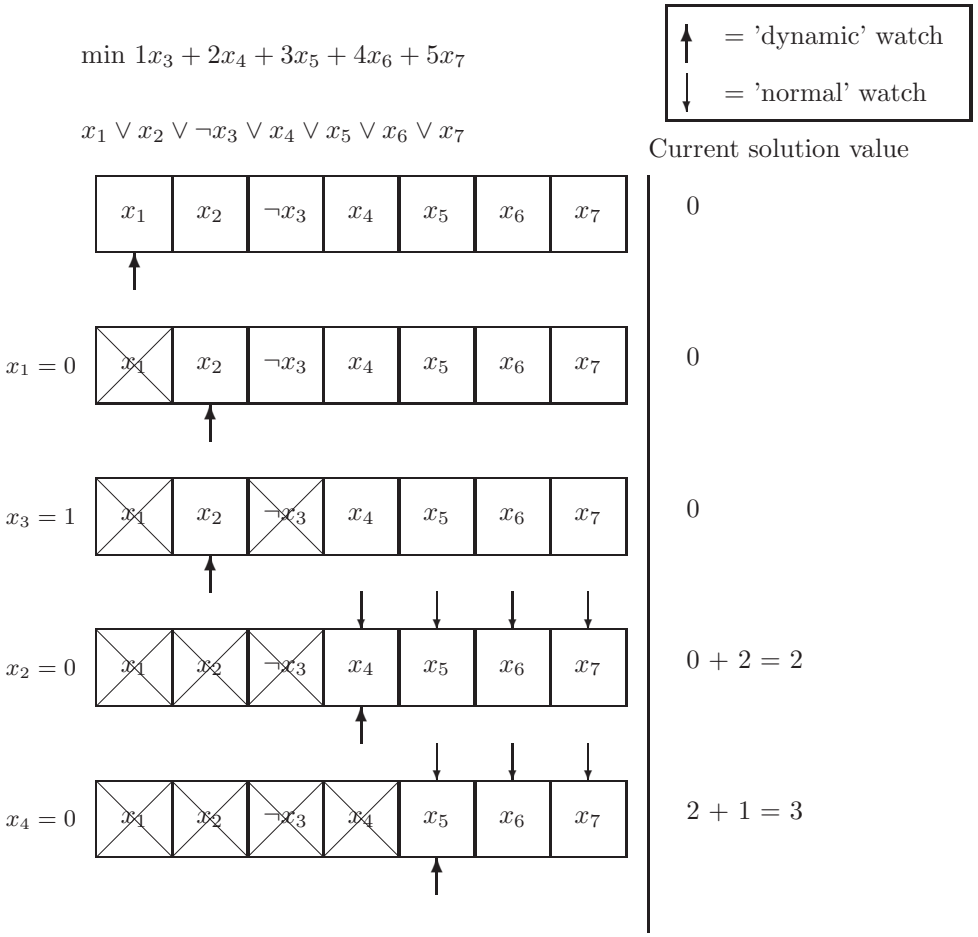


Figure 6.2: Example of the dynamic lower bound technique, as employed in MiniZSat

6.4 Bound conflict analysis

If a minimization function is available, another type of conflict can occur besides the logical conflict; the *bound conflict*. Recall that the bound conflict is a conflict which is caused by an assignment with a function value larger or equal to the upper bound. This is only a conflict when certain assumptions are made; all the coefficients minimization function should be positive and secondly, the starting solution is a solution with undefined values for each literal. The last assumption will always hold, because the solver starts with no variables assigned. Note: it is possible to start with pre defined values as long as they can not be undone.

These assumptions are necessary in order to maintain the following property: *new assignments can*

only increase the function value. Literals are associated with positive coefficients only, assigning a literal can therefore only enlarge, and not decrease, the function value. If assignments can only increase the function value, a partial solution with a function value equal or larger than the best known solution, will never yield the optimum solution. This can not occur since the function value of the partial solution can never generate a function value smaller than the best known solution.

The first constraint, all the coefficient in the minimization function should be positive, is not essential. Since negative coefficients can be made positive by rewriting the minimization function and certain constraints. This can be done by the following steps:

1. Summing all the negative coefficients in the minimization function and subtract this from the minimization function
2. Change all the negative coefficients into positive coefficient in the minimization function.
3. Negate the literals which previously had a negative coefficient.

Example. The following example shows the process of removing the negative coefficients from a minimization function and ensuring that only settings a variable to `TRUE` enlarges the value of the minimization function.

Original problem:
$\min -2x_1 - 3x_2 + 3x_3$
$x_1 \vee x_2 \vee \neg x_3$
$\neg x_1 \vee x_3$

1. Summing the negative coefficients leads to a sum of 5, the result of this step is therefore: $-5-2x_1-3x_2+3x_3$
2. Changing the negative coefficients leads to: $-5 + 2x_1 + 3x_2 + 3x_3$
3. Negate the literals leads to the result displayed below.

Resulting problem:
$\min -5 + 2x_1 + 3x_2 + 3x_3$
$\neg x_1 \vee x_2 \vee x_3$
$x_1 \vee \neg x_3$

This leads to the following new property: *only setting a variable to `TRUE` may increase the function value.* A bound conflict is therfore caused by variables that have been set to `TRUE`, since they increase the function value. Not all the literals in the selection may play an essential role in the (bound) conflict.

Definition A variable in a bound conflict is *not essential* if it has a coefficient smaller or equal to the cost of the total assignment minus the upper bound. The following equation must hold:

$$z_{curr} - k_j \geq z_{best} \tag{6.13}$$

where the cost of the current assignment is represented by z_{curr}, the cost of best known assignment by z_{curr} and the coefficient from the variable x_j by k_j.

A bound conflict may contain multiple variables which are not essential in the conflict. In that case the solver must make a selection among those variables to ensure that the conflict remains. The following example shows that not essential variables can be removed from the conflict clause and can therefore strengthen the conflict clause.

Example. For example an assignment has a cost of 19 and the upper bound is equal to 16, than a literal with coefficient 3 or smaller may be removed from the selection causing this conflict. If this literal would get the value `FALSE` assigned, the conflict remains because the cost of the assignment is still larger or equal to the upper bound. The literal is therefore not essential.

6.4.1 Bound conflict reduction

In order to easily maintain the cause (reason) for a bound conflict, all the literals increasing the function value will be placed on a stack in MiniZSat. Whenever the current assignment leads to a function value equal to or larger than the upper bound, the stack is used to find the cause of the conflict.

The smaller the reason for a conflict, the more effective the learnt constraint will be. A compact learnt constraint can eliminate a large part of the search space. Recall that a bound conflict is caused by certain variables which have the value `TRUE` assigned. The sum of the coefficients of these literals is larger than the upper bound, some of the literals may be non-essential. If a non-essential literal can be removed, then the bound conflict remains and it is caused by a smaller set of literals.

Example. A reduction of the bound conflict makes the reason for the conflict smaller. A smaller reason can in turn lead to better learning from a conflict. For example; if the reason for a conflict is

$x_1@1 \wedge x_2@3 \wedge x_3@6 \wedge x_4@7$

Normal backtracking would undo decision level 7 and $\neg x_4$ would be set at decision level 6. If the conflict could be reduced to $x_1@1 \wedge x_2@3 \wedge x_4@7$, the assignment $\neg x_4$ should be made at decision level 3 . In this case the reduction allowed the solver to backtrack multiple levels further and the learnt constraint is 'stronger' (e.g. it does not matter which value x_3 has).

A large number of options to reduce the number of literals in a bound conflict exist. The most straightforward method for a reduction of a bound conflict would be to loop through the conflicting clause from begin to end and remove literals when possible. The problem of selecting the best method for the removal of literals from a bound conflict is, that it is often unknown which learnt constraint will be the most effective in the future. This is partly due to the fact that it is unknown how the learnt constraint will accumulate with other learnt constraints.

A simple comparison method would be to count the number of literals in the bound conflict after removal of the non-essential literals or compare the function value of the conflicting clause. However it is hard to determine how the learnt constraints effects the search of the solver. Therefore a comparison is used on the complete running time of the different techniques.

The goal from a bound conflict reduction technique for MiniZSat is to find a reasonably fast method generating good results. In order to find a good method, the performance of the following techniques have been compared:

- *Ascending* (ASC): order the literals in the conflicting clause according to their associated coefficients in ascending order. Start at the begin and try to remove the smallest literals. The idea is to remove as much literals as possible.

- *Descending* (DSC): order the literals in the conflicting clause according to their associated coefficients in descending order, which is the opposite of the Ascending technique. Start at the begin and try to remove the largest literals. The idea is to remove the largest literals, since they might be 'faulty' decisions.

- *Forward* (FWD); walk through the conflicting clause from begin to end and remove literals when possible. This is the most simple method.

- *Reverse* (REV); walk through the conflicting clause from end to begin and remove literals when possible. The main advantage of this method would be that backtracking could be more efficient. This would be due to the fact that the latest chosen literals (the most recent decisions) can be removed. This allows for backtracking over multiple levels, which may not occur with the previous techniques.

Table 6.2: Comparison of various bound conflict reduction strategies on facility location problems with different sizes.

#	FWD	REV	ASC	DSC	NO
8	0.02	0.02	0.02	0.02	0.04
9	0.26	0.19	0.23	0.25	0.32
10	0.92	0.64	0.81	0.95	1.17
11	4.78	3.76	4.38	5.16	6.28
12	14.95	**9.04**	11.62	13.01	15.64
13	196.44	**126.26**	163.00	184.43	233.29
14	571.82	**480.47**	513.24	547.53	594.37

Table 6.3: Comparison of various bound conflict reduction strategies on random 3SAT instances with varying clause - variable ratios.

Ratio	FWD	REV	ASC	DSC	NO
3	79.31	71.71	94.24	114.09	75.66
3.1	50.64	46.78	56.58	79.89	49.79
3.2	16.90	14.51	21.20	22.20	15.28
3.3	8.91	7.54	11.34	12.05	7.73
3.4	3.78	3.01	4.45	4.67	3.34
3.5	1.54	1.45	1.83	1.87	1.47

Table 6.2 and Table 6.3 show the results of bound conflict reduction on facility location and random 3SAT problem instance. Both tables suggest that the reverse bound conflict reduction delivers the best results. The sorting variants (ascending and descending) perform reasonably well on the facility problem instances. This is due to the better constraints obtained by the sorting reductions, which lead to a lower number of decisions necessary to find the solution. On the other hand the sorting requires a large amount of extra time and therefore the sorting variants are not faster than the reverse technique. It should be denoted that the sorting methods are a rudimentary variant and performance could be gained by developing an optimized version.

Both the random 3SAT and facility location problem contain a large number of variables with different coefficients. If the difference in coefficients among variables is small, or worse all the coefficients are equal, the difference between the various bound conflict reduction techniques will deteriorate. This is due to the fact that sorting is less relevant when coefficients are equal.

The reverse technique has been used as default in the other benchmarks, due to the results from this test, the added possibility to backtrack over multiple levels and the fact that the overhead for this technique is minimal.

6.5 Efficient traversal of the search space

Efficient traversal of the search space is an important factor for the performance of a solver. An inefficient traversal of the search space may have negative consequences for the solving time. On the other hand an efficient traversal may increase the performance. A cause for the increase in performance may be a traversal leading to a good solution early on in the traversal of the search space. A good solution lowers the upper bound and a lower upper bound in turn leads to a smaller search space to investigate.

The traversal of the search space is defined by the search and restart strategy. The search strategy

defines how a solver traverses the search space. The restart strategy defines when a solver switches to another partition of the search space and which partition it will select. These strategies will be further inspected in this section.

6.5.1 Search strategy

Since some of the minimization problems, such as the facility location problem, are well structured, it might be useful to use a specific search strategy. The search strategy could use the properties of the structured problems in order to find the optimum solution as fast as possible.

MiniSat initially choses variables based on the appearance in the minimization function; MiniSat starts with the variables appearing *last* in the minimization function. The solver starts by setting these variables to `FALSE`. Later on it picks a literal based on the activity of a literal or randomly. The activity of a literal is based on the occurence of a variable in a recorded conflict clause. The activity of the other variables will be reduced. [27]

In order to solve problems faster, it might be wise to find a solution with variables that have small coefficients in the objective function. This may lead to a good starting solution since the solution may have avoided literals with large coefficients, which can then be used as an effective upper bound. The advantage of a good starting solution is that the solver will detect a bound conflict earlier on. The faster a solver runs into bound conflicts, the less options it has to try. Sorting the variables according to their coefficients and ensuring that the variables with small coefficients will be chosen first, may lead to better results.

Another option is to switch sorting techniques while solving. It might be advantageous to first find a good upper bound and then continue with selecting variables that have large coefficients in the minimization function in order to obtain bound conflicts as fast as possible. This would require to sort on descending coefficient first, and switch to ascending afterward.

The representation in tables use the following notation:

- N.S. denotes that *No Solution* could be delivered by the solver.
- M.O. means that the solver found a solution and ran *Out of Memory*. The solver may have given a result, however due to the shortage of memory no official output could be given.
- UNSAT refers to *not satisfiable*, which suggests that no feasible solution is available.
- sol column denotes the *best known solution*, obtained from the pseudo-Boolean evaluation. It does not always refer to the optimum solution.

It should be denoted that these methods will probably only results in substantial better performance when there are a large number of variables with *varying* coefficients (which is the same as for the bound conflict reduction techniques). If all variables would have the same coefficient or a similar coefficent, the sorting of these variables may have no effect.

In Table 6.4 a selection of a larger test set from the pseudo-Boolean evaluation has been used in order to compare sorting techniques. The selection is based on the variance of coefficients in the objective function, because no variance has no effect on the results. From the results it is visible that starting with an ascending sort and switch after 3 solutions delivers the best performance, although these variants are unable to deliver a solution for the bogr instance. Second is the variant without sorting. The variants using descending sort finishes last.

The results of the ordering on random 3SAT instances are available in Table 6.6, where no real winner can be denoted. The facility location problem on the other hand shows a preference for the ascending

Table 6.4: Comparison of various sorting techniques on a selection of instances from the pseudo-Boolean evaluation benchmarks.

bench	sol	No switching			Switching after 3 solutions	
		NO SORT	DSC SORT	ASC SORT	DSC SORT	ASC SORT
9symml	4517	**6035**	6100	**6035**	6047	**6035**
bogr	55	**60**	95	N.S.	95	N.S.
circ10_3	346	354	354	324	366	**312**
opt-market	1	**1**	**1**	**1**	3	5
simp	44	55	55	57	55	**53**
fac_20-40	6705	7202	26815	**6705**	26805	**6705**

Table 6.5: Comparison of various sorting techniques on facility location problem.

bench	NO SORT	DSC	ASC
10	0.64	83.32	0.58
11	3.80	221.87	2.27
12	9.07	566.26	**5.58**
13	128.38	T.O.	**51.81**
14	488.47	T.O.	**335.57**
15	T.O.	T.O.	T.O.

Table 6.6: Comparison of various sorting techniques on random 3SAT instances.

Ratio	NO SORT	DSC	ASC
3.0	71.71	74.31	79.70
3.1	46.78	45.26	41.22
3.2	14.51	15.33	16.94
3.3	7.54	7.59	8.89
3.4	3.01	3.03	2.97
3.5	1.45	1.38	1.39

sort. Overall the best performance will therefore be given while using ascending sort. In the following benchmarks no sorting is used, because the version using no sorting is able to solve more instances and sorting is not suitable for instances which have little or no variance in their coefficients.

6.5.2 Restart strategy

Often a minimization problem has only one optimum solution, which may be located anywhere in the solution space. A solver investing a certain partition of the solution space may be far from the optimum solution. Continuing the search in the partition may result in an improved local optimum, however it could be advantageous to restart and searching another partition of the solution space possibly closer to the optimum solution.

Certain solvers, such as ILP solvers, can predict the changes of finding the best solution in certain partitions of the solution space. This can guide the solver in selecting a partition to start the new search. A number of SAT solvers use randomized restarts to improve performance and avoid heavy-tail behavior (see Section 3.6.2).

MiniZSat is based on MiniSat which also uses randomized restarts. It uses a cut-off value for both the number of learnt clauses and the number of conflicts. Exceeding one of the cut off parameters causes a restart of the solver. Between restarts the search space expands because the cut off values increase.

Certain strategies to increase the cut off parameters may be effective. A possible strategy is to use a large number of restarts in order to search different parts of the solution space in order to find a good solution. This solution defines the upper bound and decreases the size of the search space. On the other hand a large number of restarts may make it more difficult to prove that a solution is the optimum solution, because the solver is unable to learn enough between restarts.

A number of strategies may be developed and the effect may differ per problem type and per instance.

It is therefore difficult to find the best restart strategy. The version using initial lower bound technique and restarts of MiniZSat has been used in order to compare restart strategies.

There are two parameters involved in the restart strategy of MiniZSat, the parameter which defines the number of conflicts allowed between restarts and the parameter which states the number of learnt constraints allowed between restarts. The parameters specify the new limit for the number of conflicts and learnt constraints. When one of the two limits has been reached, a restart will occur. The following restart strategies have been compared:

- **DEF(A)** is the default restart strategy from MiniSat. It uses an increase of 1.5 (50 %) for the number of conflicts between restarts and 1.1 for the number of learnt constraints between restarts.
- **B** is based on the default restart strategy. It tries to use more restarts in order to cover more partition of the solution space by decreasing the factors. The number of conflicts may increase only 1.05 between restarts and the number of learnt constraints is allowed to increase with a factor of 1.01 between restarts.
- **C** is also based on the default restart strategy and uses even more restarts than strategy B, by using factors of 1.005 and 1.001 for the increase in the number of conflicts and the number of learnt constraints, respectively.
- **CONFL** bases its increase on the percentage of bound conflicts from the total number of conflicts. A high percentage of bound conflicts leads to more restarts, and a low percentage to less restarts. Identified as a low percentage is $\leq 20\%$. The idea behind this strategy is to use less restarts if a relatively low number of bound conflicts occur, because the solver then needs more time to find *a* solution. On the other hand a large number of bound conflicts suggests a large number of feasible solutions. In that case it might be usefull to traverse a large number of partitions across the solution space in order to find a good solution among the feasible solutions.

The restart strategies have been used to compare performance on a selection of the pseudo-Boolean evaluation test set. The results from this comparison have been presented in Table 6.7. The table show varying results for the different restart strategies. Restart strategy C seems to be the most effective in finding the best solution within time. In combination with the initial lower bound technique it is able to present the best results for 11 out of 15 instances. The restart strategies B and CONFL follow with 7 times the best solution. However in general restart strategy B delivers better results than CONFL. The last place is for the default restart strategy. It is able to deliver the best solution in 6 out of 15 instances.

It should be denoted that the representation of the results may be misleading. The restart strategy C seems to perform the best, however the solver runs were performed using a time out. The results obtained within the time out do not guarantee that the specific strategy will be the fastest strategy in order to prove or find the optimum solution. This is illustrated in Table 6.8 in which the time required by the solver variant to prove the optimum of a random 3SAT instance is presented. The restart strategy C delivers the worst performance and CONFL delivers the best result.

This suggest that more research could be done in the future to find a good restart strategy. A balanced restart strategy should be able to find the best solution and prove it to be the optimum solution as fast as possible. Due to time restrictions no further investigation on restart strategies will be presented.

6.6 Using the solver

As denoted earlier on in this chapter, MiniZSat solver is not a native pseudo-Boolean solver. As input it accepts a SAT problem in DIMACS format in combination with a minimization function. However

Table 6.7: Comparison of various restart strategies on a selection of the pseudo-Boolean evaluation test set, TO = 1800 seconds.

bench	**opt**	**I.LB&RE**			
		DEF(A)	**B**	**C**	**CONFL**
5xp1.b	12	13	**12**	**12**	**12**
9symml	4517	5434	5320	**5032**	5418
addm4	165	200	193	**189**	199
bogr	55	**60**	61	62	**60**
circ10_3	346	350	**332**	338	356
domset	184	215	210	**207**	216
frb30-15-1	-30	-26	-26	-26	-26
g15	53	57	57	57	57
market-split	UNSAT	**UNSAT**	N.S.	N.S.	**UNSAT**
opt-market	1	7	**3**	**3**	11
simp	44	54	45	**44**	55
ss97	1159	N.S.	**14372**	19749	N.S.
vtxcov	1037	1197	1188	**1172**	1184
wnq	434	M.O.	M.O.	**4871**	M.O.

Table 6.8: Time comparison in seconds of various restart strategies on a random 3SAT instance.

bench	**RE**			
	DEF(A)	**B**	**C**	**CONFL**
3sat_450-150	151.62	92.89	576.71	**87.57**

all pseudo-Boolean constraints can be encoded in a SAT problem, which enables MiniZSat to solve pseudo-Boolean problems. The conversion of pseudo-Boolean constraints to SAT clauses can be done by MiniSat+.

This section continues with a formal definition of the problems accepted by MiniZSat and the input format it accepts. This section also includes a short introduction on the conversion step from pseudo-Boolean constraints to SAT clauses provided by MiniSat+.

6.6.1 Problem definition

The MiniZSat solver is technically not a pseudo-Boolean solver. It accepts as input a CNF formula with a minimization function. In this minimization function variables from the CNF formula can be used.

Definition

$$\begin{aligned} m &= min(\sum k_j x_j) && \text{for each } j \\ F &= \bigwedge c_i && \text{for each } i \\ c_i &= \bigvee l_j && \text{for each } j \end{aligned} \tag{6.14}$$

where m is the minimization function. Variables are represented by x_j and its coefficient by k_j. F is a CNF formula(a conjunction of clauses) and c_i is a clause (a disjunction of literals). The AND and OR operators are represented by $\bigwedge$ and $\bigvee$.

6.6.2 Input format

The format is based on the *DIMACS format*, with the addition of an extra line for the minimization function. A comment line in the DIMACS format needs to be preceded by a 'c'. The number of variables and clauses should be denoted on a line prefixed with 'p cnf', followed by the number of variables and the number of clauses. The following lines contain the clauses, in which a positive literal is denoted by the corresponding number and a negative literal is denoted by the negative corresponding number. A clause should end with '0'.

The extra line in the MiniZSat format starts with the letter 'm' and contains a combination of variables used in the CNF formula. Variables from the CNF formula can be referenced in the minimization function by prefixing a 'x'. For example, referencing variable 1 from the CNF formula can be achieved by using $x1$ in the minimization function.

Maximization problems are not supported as input. However as previously denoted (see Section 2.1 and Section 4.1), a maximization problem can easily be transformed into a minimization problem by multiplying the function with -1.

Example. The next lines show an example of the input format for MiniZSat.

```
c comment line
p cnf 5 2
m 1 x1 +2 x2 - 3x3 - 1x4 + 10x5
1 2 5 0
-3 -4 0
```

6.6.3 Translating from pseudo-Boolean definition

MiniSat+ has the ability to export the constraints of a pseudo-Boolean problem without the minimization function to CNF and thus translating the pseudo-Boolean constraints to clauses (see also Section 4.4.2

for further information on the translation process). In order to make a complete translation from pseudo-Boolean to the MiniZSat format possible, the minimization function needs to be translated as well. This can simply be done by taking the original minimization function from the pseudo-Boolean problem. If the export of MiniSat+ renames certain variables, care should be taken to generate a correct representation of the problem. In MiniZSat this can be achieved by using an extra input file, which contains the renamed variables.

Chapter 7

Results

7.1 Introduction

Due to time limitations each solver run used a time out. The time out value is either 900 or 1800 seconds, depending on the size of the problems and importance of the runs. All the the facility location problem instances and most of the random 3SAT instance were performed using a time out of 900 seconds. For random 3SAT and facility location problems the average of 5 instances is obtained. The instances from the pseudo Boolean evaluation test set were solved using a timeout of 1800 seconds, because this is the default for the evaluation results.

7.1.1 MiniZSat variants

The following variants of MiniZSat were tested. All of the variants use bound conflict reduction starting at the end (e.g. the method reverse denoted with 'REV', see Section 6.4.1).

- *DEF*, the default version of MiniZSat.
- *I.LB*, version using initial lower bound technique.
- *I.+D.LB*, version using initial and dynamic lower bound techniques.
- *RE*, the default version using restarts while solving.
- *I.LB & RE*, version using initial lower bound technique and restarts while solving.
- *I.+D.LB & RE*, version using initial and dynamic lower bound techniques and restarts while solving.

7.1.2 Solver and running information

All the solver runs were done on a Intel Core 2 Duo E6300 1.86GHz using 3 Gigabyte of memory running on Debian Linux 4.1. The following solvers were used:

- *MiniZSat version 1.0*, using MiniSat version 2.0 as basis.
- *MiniSat+ version 1.0*, using MiniSat version 1.13 as basis. MiniSat+ is available from `http://minisat.se/MiniSat+.html`.
- *Pueblo version 1.5* is available from `http://www.eecs.umich.edu/~hsheini/pueblo/`.

- *Glpk version 4.23*, although Glpk is not designed as a pseudo-Boolean solver, it can solve ILP problems with binary variables (which actually is a pseudo-Boolean problem). GlpPB is a pseudo-Boolean solver by Hossein Sheini [58] based on Glpk version 4.10. Because GlpPB did not output intermediate results and no source code was available, Glpk was used to solve instances. Glpk is available from `http://http://www.gnu.org/software/glpk/` and GlpPB is available from `http://www.eecs.umich.edu/~hsheini/pueblo/`.

7.1.3 Notation

Several notations are used through this chapter. The following notations are used within the presentation of the results:

- N.S. denotes that *No Solution* could be delivered by the solver.
- T.O. means that the solver found a solution and *Timed Out*. It could be that the optimum solution was found, but it could not be proven to be the optimum before the solver timed out.
- M.O. means that the solver found a solution and ran *Out of Memory*. The solver may have given a result, however due to the shortage of memory no official output could be given.
- N.A. refers to *Not Available*; due to circumstances no information could be given.
- A result entry containing parenthesis, states the number of instances that could be solved before timing out between parenthesis. For example 560.87 (3), means that the solver could solve 3 of the 5 instances and the average of solving these 3 instances is 560.87 seconds.
- # stands for *Number of*.
- UNSAT refers to *not satisfiable*, which suggests that no feasible solution is available.
- sol column denotes the *best known solution*, obtained from the pseudo-Boolean evaluation. It does not always refer to the optimum solution.

7.2 Facility location

In Section 5.5.1 a formulation of the uncapacitated facility location problem was given:

$$
\begin{aligned}
min \sum_i y_i f_i + \sum_{ij} x_{ij} c_{ij} d_j & \\
x_{ij} \leq y_i & \quad \text{for each } i,j \\
\sum_i x_{ij} = 1 & \quad \text{for each } j \\
x_{ij}, y_i \in 0,1 & \quad \text{for each } i,j
\end{aligned}
$$

The only complex constraint is $\sum_i x_{ij} = 1$, for each j. The translation of this pseudo-Boolean constraint to SAT generates a large number of clauses. The translation MiniSat+ delivers is not very suitable for MiniZSat, because the lower bound could not be used.

Another possible translation is the usage of one clause $\sum_i x_{ij} \geq 1$, for each j and additional clauses ensuring that only one of the literals will satisfy the clause. The number of additional clauses depends on the number of variables in the constraint. If the number of variables would be m, then the number of

additional clauses is $\frac{(m\times(m-1))}{2}$. The additional clauses are binary clauses prohibiting two literals to be both `TRUE` at the same time. The clause $\sum_i x_{ij} \geq 1$, for each j in turn ensures that at least one literal should be `TRUE`, therefore at most one literal can be `TRUE` in this clause.

Example. The following example shows how the pseudo-Boolean constraint $x_1 + x_2 + x_3 = 1$ will be translated to clauses.

$$x_1 \vee x_2 \vee x_3$$
$$\neg x_1 \vee \neg x_2$$
$$\neg x_1 \vee \neg x_3$$
$$\neg x_2 \vee \neg x_3$$

Table 7.1 shows the number of extra clauses generated per pseudo-Boolean constraint. This means that the problem size in CNF is significant larger than the pseudo-Boolean equivalent. Table 7.2 shows the difference in problem size. The fact that the number of clauses is larger than the number of pseudo-Boolean constraints may cause a decrease in performance. Although the additional clauses are relatively simple (binary), these clauses still need to be maintained and propagations should be made when necessary.

Table 7.1: The number of extra clauses generated per pseudo-Boolean constraint.

# variables in PB constraint	# addition clauses
5	10
10	45
20	190
30	435
40	780
50	1225
100	4950

Table 7.2: The number of constraints and clauses for varying facility location problems sizes.

# facilities	# locations	PB	CNF
5	10	60	160
10	20	220	1,120
20	40	840	8,440
30	60	1,860	27,960
40	80	3,280	65,680
50	100	5,100	127,600
100	200	20,200	1,010,200

7.2.1 Lower bound techniques

Table 7.3 presents the result of the lower bound variants and other solvers. In this subsection the focus is on the lower bound techniques, the comparison with other solvers will be presented in the following subsection.

From the table it is visible that the version using the initial and dynamic lower bound technique without restarts delivers the best performance. This version is able to solve instances, within the timeout of 900 seconds, up to a size of 39 facilities and 78 locations. This is considerably larger compared to for example the default version, which is only able to solver instances up to the size of 14 facilities and 28 locations.

Both of the lower bound techniques enable MiniZSat to find more solutions before the timeout. The cause for the increase in performance is the large increase of the lower bound. The problem contains a large number of clauses suitable for initially increasing the lower bound. These clauses contain almost all the variables from the minimization function, which is why the lower bound becomes close to the actual optimum solution.

The dynamic lower bound technique is also very effective for the facility location problem. Clauses suitable for the initial lower bound technique may also be used for the dynamic lower bound technique. The same large set of clauses is therefore able to increase the lower bound dynamically. The variant using both initial and dynamic lower bound without the usage of restarts, enables the solver to solve facility location problems up to 39 facilities and 78 locations within a timeout of 900 seconds.

The restart strategy on the other hand decreases the performance of the solver. This may be caused by the time required for a restart and the reduced learning effect from combined clauses. A restart may continue in another partition of the search space, in which the learned clauses may have less effect since they prohibit parts of another partition in search space. The number of learned clauses is also limited, which may decrease the accumulated effect of the learned clauses if the solver changes the partition to search through often.

7.2.2 Solver comparison

Facility location problem is the terrain of ILP solvers, such as Glpk. In Table 7.3 this is confirmed, where Glpk only needs 18 seconds to solve the largest instance opposed to all the other solvers which could not find the optimum solution. The great performance of Glpk on facility location instances is due to the nature of the facility location problem. The facility location problem has an enormous amount of feasible solutions and the difficulty in the problem is defined by the minimization function. The usage of LP relaxation and cuts enable the solver to quickly find the optimum solution.

The solver following Glpk, with a great distance, is MiniZSat. MiniZSat performs considerably less, although the variant using both initial and dynamic lower bound techniques without restarts can keep up with Glpk for small instances. Instances with less than 23 facilities can be solved by this MiniZSat variant within a second, although Glpk only needs 0.2 seconds. Time used by the MiniZSat variant rapidly increases after 23 facilities until the time out is reached for instances with 39 facilities.

The third place is a close call between MiniSat+ and Pueblo. They both deliver similar performance, which is at an appropriate distance from the MiniZSat version using initial and dynamic lower bound techniques without restarts. Only Pueblo was able to solve one instance with 13 facilities. Therefore Pueblo ends as third and MiniSat+ as fourth. However the performance gap between them is much smaller than the gaps between number one and two, and number two and three.

7.3 Random 3SAT

In the previous section the conclusion was drawn that LP solvers still remain the best choice for solving the facility location problem. This is partly due to the structure of the facility location, which is well suited for LP solvers. SAT-based solvers on the other hand, are very well suited for solving SAT problems. In order to compare solvers in this area, various random 3SAT with minimization functions were created.

A 3SAT problem is a problem consisting of clauses containing 3 literals. A random 3SAT problem is a randomly generated 3SAT problem. For this purpose the 3SAT problem instance generator *mkcnf.c* by A. van Gelder was adapted to create a pseudo-Boolean variant of the random 3SAT problem. In order to make it a pseudo-Boolean optimization problem, the generated random 3SAT problem has been extended with a random minimization function.

7.3.1 Lower bound techniques

Table 7.4 presents the results of the various lower bound techniques solving random 3SAT instances. These instances contain 180 variables and a varying number of clauses. This leads to a reasonably large

Table 7.3: Facility location comparison of MiniZSat variants and other solvers, TO = 900 seconds.

Size	**DEF**	**RE**	**I.LB**	**I.LB &RE**	**I+D.LB**	**I+D.LB &RE**	**Pueblo**	**MiniSat+**	**Glpk**
10	0.65	1.43	0.00	0.00	0.01	0.02	10.41	56.98	0.00
11	3.69	7.57	0.00	0.00	0.01	0.02	66.502(3)	131.07	0.00
12	9.15	24.66	0.00	0.00	0.01	0.07	398.09(1)	393.033(4)	0.00
13	126.07	152.89	0.01	0.01	0.02	0.09	717.43(1)	T.O.	0.00
14	201.45(3)	500.84(3)	0.00	0.01	0.02	0.14	T.O.	T.O.	0.00
15	T.O.	T.O.	0.01	0.01	0.04	0.27	T.O.	T.O.	0.00
16	T.O.	T.O.	0.02	0.05	0.04	0.51	T.O.	T.O.	0.00
17	T.O.	T.O.	0.06	0.04	0.08	1.23	T.O.	T.O.	0.00
18	T.O.	T.O.	0.32	0.28	0.12	1.80	T.O.	T.O.	0.20
19	T.O.	T.O.	0.23	0.44	0.13	10.09	T.O.	T.O.	0.00
20	T.O.	T.O.	5.15	6.09	0.33	4.69	T.O.	T.O.	0.20
21	T.O.	T.O.	6.12	23.91	0.30	11.22	T.O.	T.O.	0.20
22	T.O.	T.O.	46.72	39.11	0.94	27.39	T.O.	T.O.	0.20
23	T.O.	T.O.	86.82	214.52	1.85	49.12	T.O.	T.O.	0.20
24	T.O.	T.O.	293.25(4)	545.50(2)	2.77	203.44	T.O.	T.O.	0.20
25	T.O.	T.O.	542.794(2)	753.055(1)	6.15	93.51	T.O.	T.O.	0.60
26	T.O.	T.O.	164.04(1)	T.O.	6.40	136.22(3)	T.O.	T.O.	0.40
27	T.O.	T.O.	T.O.	T.O.	10.47	551.53(1)	T.O.	T.O.	0.80
28	T.O.	T.O.	T.O.	T.O.	28.35	T.O.	T.O.	T.O.	1.20
29	T.O.	T.O.	T.O.	T.O.	41.86	T.O.	T.O.	T.O.	2.00
30	T.O.	T.O.	T.O.	T.O.	54.70	T.O.	T.O.	T.O.	2.40
31	T.O.	T.O.	T.O.	T.O.	101.27	T.O.	T.O.	T.O.	2.80
32	T.O.	T.O.	T.O.	T.O.	119.77	T.O.	T.O.	T.O.	2.60
33	T.O.	T.O.	T.O.	T.O.	185.75	T.O.	T.O.	T.O.	5.60
34	T.O.	T.O.	T.O.	T.O.	323.53	T.O.	T.O.	T.O.	6.40
35	T.O.	T.O.	T.O.	T.O.	330.53	T.O.	T.O.	T.O.	5.80
36	T.O.	T.O.	T.O.	T.O.	416.46	T.O.	T.O.	T.O.	7.00
37	T.O.	T.O.	T.O.	T.O.	619.30(4)	T.O.	T.O.	T.O.	10.00
38	T.O.	T.O.	T.O.	T.O.	777.05(3)	T.O.	T.O.	T.O.	12.40
39	T.O.	T.O.	T.O.	T.O.	381.88(1)	T.O.	T.O.	T.O.	16.80
40	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	18.00

solving times for the low clause - variable ratios and even a few time outs (time out was set to 1800 seconds due to the large size of the problems).

Table 7.4 shows that the initial lower bound reduces the solving time for instances with a low ratio. Solving times for instances with ratios from 3.0 till 3.3 have been reduced by more than a factor of 2 while using lower bound techniques. This is due to the large number of feasible solutions available in random 3SAT instances with low clause - variable ratios. The initial lower bound increases the lower bound and may thereby reduce the number of feasible solutions.

The influence of the lower bound technique deteriorates while using higher ratios. 3SAT instances with ratios from 3.6 and onwards (till 4.5) do not benefit from the initial lower bound technique and performance is lost due to the overhead of the lower bound techniques. Although it should be mentioned that this overhead is minimal. The random 3SAT instances with higher ratios are relatively easy to solve, because the number of feasible solutions is lower than for instances with lower ratios. Therefore solving times are smaller and the variation between versions may be relatively large. These variations can be caused by different reasons. No conclusions will therefore be drawn for random 3SAT instances with high ratios.

The dynamic lower bound technique on the other hand is not effective for random 3SAT instances. The reason lies in the structure of the 3SAT problem: each clause contains 3 literals, which is not very suitable for deployment of the dynamic lower bound technique. A clause containing 3 literals is easily satisfied and due to the low number of literals a large number of propagations need to be made. The dynamic lower bound technique does not benefit from these actions, because the increase of the lower bound is useless when a logical conflict occurs or when a large number of literals are propagated. For example: setting a literal in a clause activates the dynamic lower bound technique and the lower bound is raised. After some propagations the specific clause is satisfied by another literal and the increase of the dynamic lower bound technique was useless. Another example would be a dynamic increase in the lower bound and the detection of a logical conflict after propagations. This conflict would also occur without the dynamic increase.

A possible improvement would be to activate the dynamic lower bound after all variables have been propagated. This avoids performance overhead on already satisfied clauses.

7.3.2 Solver comparison

It is interesting to see how other solvers cope with this problem, especially the non SAT based solvers. Since these solvers are not designed to solve (random 3-) SAT problems, they will probably perform less then MiniZSat. On the other hand MiniSat+ may be a good competitor because it uses MiniSat as underlying solver. Pueblo also uses SAT based techniques, which may give it an advantage over the integer linear programming solver Glpk.

The results of the comparison are available in Table 7.5. Random 3SAT problems containing 150 variables and varying ratios have been used to acquire these results. It is clearly visible that for (almost) all ratios every version of MiniZSat performs better than the other solvers.

Pueblo finishes second after MiniZSat. There is a large gap between MiniZSat and Pueblo, especially for the instances with low ratios solved by the MiniZSat variant using the initial lower bound version without restarts. For a ratio of 3, the initial lower bound version without restarts performs ten times better than Pueblo. This MiniZSat variant remains almost five times as fast as Pueblo for instances with ratios up till 3.7.

The third place is for the MiniSat+ solver, which is the first solver that could not find all the solutions within time. For none of the five problem instances with a ratio of 3.0 the optimum solution could be proven before reaching the time out. This is possibly due to the large number of feasible solutions which are available while solving random 3SAT instances with low clause - variable ratios. The minimization function therefore plays an important role, it decides which solution is better. After MiniSat+ found a

Table 7.4: Comparison of MiniZSat variants using different lower bound techniques on random 3SAT problems with 180 variables and varying clause - variable ratios, TO = 1800 seconds.

Ratio	DEF	RE	I.LB	I.LB&RE	I+D.LB	I+D.LB&RE
3.0	933.55 (4)	1350.87 (3)	**257.05**	447.80	428.53	894.90 (4)
3.1	1093.22 (4)	1626.00 (2)	**282.64**	555.38	441.04	1000.38 (4)
3.2	787.40	814.95	**315.18**	331.29	461.10	779.85
3.3	495.42	767.28	**174.80**	240.85	250.45	641.39
3.4	121.92	119.19	**93.01**	101.26	107.52	134.21
3.5	53.27	61.55	**36.71**	35.75	46.11	51.49
3.6	9.49	**8.35**	11.70	10.09	11.34	11.30
3.7	6.13	6.45	6.05	6.47	6.48	9.02
3.8	1.63	1.81	2.34	3.72	2.59	2.45
3.9	1.28	1.40	1.48	1.74	1.62	1.65
4.0	0.76	0.98	0.75	1.03	0.90	1.02
4.1	0.59	0.53	0.63	0.62	0.66	0.63
4.2	0.45	0.32	0.49	0.34	0.52	0.39
4.3	0.37	0.31	0.39	0.34	0.46	0.38
4.4	0.20	0.20	0.21	0.21	0.25	0.23
4.5	0.11	0.11	0.13	0.13	0.14	0.14

Table 7.5: Comparison of solvers on random 3SAT problems with 150 variables and varying clause - variable ratios, TO = 900 seconds.

Ratio	DEF	RE	I.LB	I.LB &RE	I+D.LB	I+D.LB &RE	Pueblo	MiniSat+	Glpk
3.0	71.71	131.62	**22.11**	27.70	29.79	47.99	284.78	T.O.	48.25
3.1	46.78	63.14	**16.36**	21.85	27.34	45.09	217.26	676.17	79.67
3.2	14.51	22.26	**10.28**	11.78	12.38	14.83	80.61	409.71	184.00
3.3	7.54	9.37	**6.14**	7.60	8.26	10.70	35.86	354.36	10.00
3.4	3.01	3.59	3.27	3.26	3.79	3.43	16.13	186.85	226.00
3.5	1.45	1.40	1.50	1.42	1.62	1.80	7.39	69.51	T.O.
3.6	0.70	0.74	0.73	0.75	0.82	0.82	3.99	50.03	T.O.
3.7	0.40	0.41	0.43	0.46	0.54	0.47	2.03	27.14	T.O.
3.8	0.25	0.22	0.27	0.26	0.31	0.25	0.99	27.81	T.O.
3.9	0.14	0.11	0.14	0.12	0.16	0.15	0.49	8.71	T.O.
4.0	0.11	0.08	0.11	0.09	0.12	0.12	0.43	3.27	T.O.
4.1	0.07	0.06	0.07	0.06	0.08	0.07	0.21	2.74	T.O.
4.2	0.06	0.04	0.06	0.05	0.07	0.05	0.15	1.71	T.O.
4.3	0.05	0.03	0.05	0.04	0.06	0.04	0.11	1.40	T.O.
4.4	0.04	0.03	0.04	0.04	0.05	0.04	0.11	1.60	T.O.
4.5	0.03	0.02	0.03	0.03	0.04	0.03	0.08	1.62	T.O.

new solution, it will need to add extra clauses in order to find an improved solution. This process may be repeated a large number of times due to the number of solutions available.

Another possible cause for a decrease in performance, is the translation process of the minimization function. In the Introduction the number of extra clauses needed for the conversion of the minimization function were discussed (displayed in Table 1.1). The increase in clauses is massif for random 3SAT problems, varying from 150 to 200 times the number of original clauses. This overhead reduces the number of decisions the solver can make and it increases the number of propagations necessary to satisfy all clauses. As a result the solver may need more time to find and prove the optimum solution.

The last place is for the Glpk solver. Glpk can easily solve instances with very low ratios, because it can prune large parts of the search tree with its effective bounding techniques. On the other hand Glpk has problems solving random 3SAT instances with high ratios, where it is difficult to find *a* solution satisfying the constraints. Finding the optimum solution becomes even more complicated due to the nature of the random 3SAT problem.

As denoted earlier Glpk uses LP relaxations of the problem to find lower bounds. These LP relaxations may not be effective since a clause $x_1+x_2+x_3 \geq 1$ may also be satisfied by taking $x_1 = 1/3, x_2 = 1/3, x_3 = 1/3$. This makes it difficult to use the result of the LP relaxation as effective input for the bounding process.

7.4 Pseudo-Boolean evaluation 2007

In this Chapter the results of a selection from the pseudo-Boolean evaluation benchmarks of the year 2007 will be presented. These benchmarks have been taken 'randomly' from the OPT-SMALLINT-LIN section. This is the set of problems which have an optimization function, use small integers and linear constraints. The problems are all from a different family of benchmarks in order to acquire some insights on how MiniZSat will compete in this benchmark set. It will also give an indication on how MiniZSat will perform in general.

The benchmarks used have been extended with three new instances; fac_20-40, 3sat_450-150 and 3sat_600-150. The first is an instance of the facility location and the last two of the random 3sat problem (with minimization function). The specific instances are added in order to gain a complete overview on the overall performance of the solvers.

7.4.1 Lower bound techniques

Lower bound techniques can not be employed on every problem. In some of the instances of the pseudo-Boolean evaluation benchmark the lower bound technique could not be used, because no clauses were suitable for the initial and/or the dynamic lower bound technique. Table 7.6 shows which instances may benefit from the initial or dynamic lower bound technique. Instances containing no clauses suitable for the specific lower bound technique (denoted by 0%) perform the same as without the lower bound technique employed.

The clauses suitable for initial lower bound are by definition also suitable for the dynamic lower bound technique. These specific clauses are however not added to the percentage of the dynamic classes. Only if the percentage for initial and dynamic are both 0, then the dynamic lower bound technique can not be used.

Table 7.7 presents the parsed starting solution value and the initial lower bound. It visualizes by which amount the lower bound is raised due to the initial lower bound technique. A negative value as parsed starting solution is caused by the transformation of negative coefficients in the minimization function (see Section 6.4). The starting solution may also be affected by the problem instance. The CNF can for example contain unit clauses which cause an increase of the starting solution.

Table 7.6: Problem instances suitable for lower bound techniques in MiniZSat, denoting the percentage of clauses suitable for the lower bound technique.

Bench	**Initial**	**Dynamic**
5xp1.b	0.36%	0%
9symml	3.26%	5.49%
addm4	11.66%	0%
aim	18.85%	0%
bogr	0%	0%
circ10_3	0%	0.13%
domset	0.02%	5.90%
frb30-15-1	0%	0%
g15	19.11%	0%
market-split	0%	0%
mps	0%	0.46%
opt-market	0%	1.13%
simp	0%	0%
ss97	0%	10.13%
vtxcov	0%	0%
wnq	0%	0%
fac_20-40	0.89%	0%
3sat_450-150	5.78%	4.22%
3sat_600-150	4.17%	5.17%

Table 7.7: Parsed solution and initial lower bound value obtained while parsing the problem instances.

Bench	**Parsed solution**	**Initial LB**	**sol**
5xp1.b	0	4	12
9symml	0	2,440	4,517
addm4	0	97	165
aim	0	49	100
bogr	45	45	55
circ10_3	0	0	346
domset	0	4	184
frb30-15-1	-450	-450	-30
g15	0	43	53
market-split	0	0	UNSAT
mps	0	0	34
opt-market	0	0	1
simp	0	4	44
ss97	-3,212	-2,545	1,159
vtxcov	0	0	1,037
wnq	0	132	434
fac_20-40	0	4,113	6,705
3sat_450-150	0	10,856	22,655
3sat_600-150	0	12,186	34,573

Table 7.8 presents a comparison of different MiniZSat variants using lower bound techniques. The table first indicates that the usage of restarts is recommended. Only two benchmarks perform less without restarts, 5xp1.b and opt-market. The restart strategy is able to deliver better results in seven instances. It is therefore recommended to use a restart strategy.

Comparing the lower bound techniques in Table 7.8 show only a small number of differences. This is due to the fact that a large number of instances can not profit from the lower bound techniques (see Table 7.6). The initial lower bound technique only delivers a worse result in one instance opposed to the version using no lower bound technique. For the versions without restarts the initial lower bound technique was able to deliver better performance for three instances, the variant using restarts and initial lower bound technique could deliver better performance for five instances.

The dynamic lower bound in combination with the initial lower bound technique did not deliver an increase of performance opposed to the version using initial lower bound technique only. The version using both lower bound techniques and restarts delivers an improvement on one benchmark and a worse result for two benchmarks. The version without restarts with both lower bound techniques deployed is able to deliver better results for three instances.

A special variant of the MiniZSat solver is included, the variant denoted with I.LB&RE+XX. This version is the same as the version with initial lower bound using restarts (I.LB&RE). The only difference is that this version runs on modified input instances. Extra clauses are added to the instances in an attempt to improve the possibility to employ the lower bound techniques. It also uses a modified version of the `simplify` method from MiniSat+, but originally it could not deliver the correct answer in combination with MiniZSat for all instances, therefore it was modified. The `simplify` method propagates unit clauses and removes unnecessary clauses. The performance of this variant is somewhat inconsistent. It delivers the best results for a number of instances and on the other hand it is unable to present the solution for market-split.

The versions with restarts enabled definitely perform better. The usage of the initial lower bound also improves the results. The dynamic lower bound show mixed results; improvements for the version without restarts and no improvement for the variant using restarts. The best version to solve these benchmarks is therefore the version using both restarts and the initial lower bound technique.

7.4.2 Solver comparison

The most important question remains; how does MiniZSat compete with other solvers on this selection of benchmarks from the pseudo-Boolean evaluation. Table 7.9 shows the results obtained from the various solvers. Two versions of MiniZSat are displayed: the version using initial lower bound technique and restarts and a hypothetical version which shows the best results from all the MiniZSat versions including different ordering and restart strategies.

The most important observation should be that certain instances favor a specific solver. The 9symml and addm4 are for example easily solvable by GLPK, where as it is unable to deliver a solution for aim. The instance aim is on the other hand easily solvable by all the other solvers. This is of course due to the nature of the pseudo-Boolean problems. Certain instances are pseudo-Boolean problems closer to a integer linear programming representation and others are closer to a SAT problem.

The solvers the highest number of best solutions on the test set are Glpk and the theoretical MiniZSat version using the best results. Both solvers are capable of delivering the best solution in 11 out of the 19 instances. On the other hand Glpk is unable to present a solution in 6 out of 19 instances. This underlines the fact that the performance of a solver such as Glpk largely depends on the type of problem instance.

Table 7.8: Comparison of different MiniZSat variants on a test set of the pseudo-Boolean evaluation, TO = 1800 seconds.

Bench	**sol**	**DEF**	**RE**	**I.LB**	**I.LB &RE**	**I+D.LB**	**I+D.LB &RE**	**I.LB &RE+XX**
5xp1.b	12	13	14	13	13	13	13	**12**
9symml	4517	6035	5476	6033	5434	6006	**5346**	5455
addm4	165	201	**198**	201	200	201	200	201
aim	100	100	100	100	100	100	100	100
bogr	55	60	60	60	60	60	60	**59**
circ10_3	346	354	350	354	350	350	352	**343**
domset	184	351	217	351	215	340	222	**214**
frb30-15-1	-30	-26	-26	-26	-26	-26	-26	-26
g15	53	70	61	65	**57**	65	**57**	**57**
market-split	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	N.S.
mps	34	34	34	34	34	34	34	34
opt-market	1	**1**	7	**1**	7	**1**	7	15
simp	44	55	54	55	54	55	54	**53**
ss97	1159	N.S.	N.S.	N.S.	N.S.	N.S.	N.S.	N.S.
vtxcov	1037	1201	1197	1201	1197	1201	1197	**1193**
wnq	434	N.S.	N.S.	N.S.	N.S.	N.S.	N.S.	N.S.
fac_20-40	6705	7202	7202	**6705**	**6705**	**6705**	**6705**	**6705**
3sat_450-150	22655	22655	22655	22655	22655	22655	22655	22655
3sat_600-150	34573	34573	34573	34573	34573	34573	34573	34573

MiniSat+ and Pueblo are both able to present the best solution in 8 out of the 19 instances. Interesting to see is that Pueblo is able to present a solution for all the instances. This makes it an all round performer suitable for a large group of problems. The good all round performance is the result of a combination of both ILP and SAT techniques. MiniSat+ on the other hand is also reasonably all round by solving all instances except for one.

MiniZSat comes in last while only able to present the best solution in 6 out of the 19 instances. It was unable to present a solution for only one instance, which makes it an all round performer on these benchmarks. MiniSat+ has the advantage opposed to MiniZSat that the number of conflicts are smaller. This is caused by the clauses containing the translated objective function which try to ensure that a new solution is automatically smaller than the best known solution.

An important note on this comparison is that the selection of solvers are the best competitors in the pseudo-Boolean evaluation. Together with bsolo they are in the top 5 of 12 solvers. This means that MiniZSat is not the best performer in this group, on the other hand it will certainly not the worst. The hypothetical version of MiniZSat also indicates that there is enough room for improvement on the performance of MiniZSat and that tweaking of certain parameters could gain a large increase in performance.

It should be denoted that the comparison is based on the solution it can present within the timeout. It does not however consider the time necessary to prove the optimality of a solution. MiniZSat version are sometimes able to present the optimum solution, however often they can not prove it to be optimum before the time out has been reached. MiniZSat can however prove the optimality of the newly added instances (fac_20-40 , 3sat_450-150, and 3sat_600-150), e.g. the facility location and random 3SAT instances.

Table 7.9: Solver comparison on a test set of the pseudo-Boolean evaluation, TO = 1800 seconds.

Bench	**sol**	**MiniZSat best**	**MiniZSat I+D.LB&RE**	**Pueblo**	**GLPK**	**MiniSat+**
5xp1.b	12	**12**	13	**12**	**12**	**12**
9symml	4517	5032	5434	5274	**4517**	5230
addm4	165	189	200	208	**165**	184
aim	100	**100**	**100**	**100**	N.S.	**100**
bogr	55	60	60	**55**	N.S.	61
circ10_3	346	**324**	350	346	N.S.	346
domset	184	207	215	211	**201**	216
frb30-15-1	-30	-26	-26	-14	-23	**-30**
g15	53	57	57	59	**54**	55
market-split	UNSAT	**UNSAT**	**UNSAT**	**UNSAT**	N.S.	**UNSAT**
mps	34	34	34	34	34	34
opt-market	1	**1**	7	**1**	**1**	**1**
simp	44	**44**	54	69	N.S.	**44**
ss97	1159	14372	N.S.	41649	**1049**	N.S.
vtxcov	1037	1172	1197	1188	**1056**	1200
wnq	434	**2791**	5403	5407	N.S.	5121
fac_20-40	**6705**	**6705**	**6705**	7921	**6705**	8644
3sat_450-150	**22655**	**22655**	**22655**	**22655**	**22655**	23469
3sat_600-150	34573	**34573**	**34573**	**34573**	36552	**34573**

This is likely to be caused by the incapabilities of the learning mechanism from MiniZSat. MiniZSat is unable to discard large parts of the solution space with its learnt constraints. This is due to the fact that a limited number of learnt clauses are allowed in order to prevent the problem from growing too large. Another reason is the effectiveness of a learnt constraint obtained from a bound conflict. This learnt constraint may not be very effective, because it only prohibits a small part of the solution space.

Chapter 8

Conclusion and Future work

From Chapter 7 it can be concluded that MiniZSat is a reasonable performer on a large set of benchmarks. MiniZSat is able to deliver a solution on a large selection of benchmarks. It performs extremely well in the random 3SAT instances and fairly well on the facility location problems. It is however outperformed by Glpk, MiniSat+ and Pueblo on most instances. MiniZSat would however still outperform a large number of other pseudo-Boolean solvers.

The results also show that there is room for possible improvements on the solver. If the best results of all the MiniZSat variants would be taken, then such a variant would have performed better than the other solvers. This leaves enough options for further development of the MiniZSat solver.

8.1 MiniZSat techniques

In the Introduction several techniques have been proposed in order to improve the performance of a SAT based pseudo-Boolean solver. These improvements have been analyzed in Chapter 6 and Chapter 7. A brief discussion on the results from these improvements will be presented below.

8.1.1 Embedding the minimization function; MiniZSat vs MiniSat+

Comparing MiniZSat with MiniSat+, MiniSat+ remains better in general as is visualized in the benchmarks of the pseudo-Boolean evaluation. MiniZSat on the other hand performs better for facility location problems and random 3SAT instances with a minimization function. This is partly due to the large increase in size of the problem for MiniSat+ and the lack of bounding mechanisms for MiniSat+.

MiniZSat may also be the best choice when the increase of the added clauses for the translated minimization function cripple the performance of MiniSat+ too much. This may occur when the number of clauses increase to more than 80,000. In certain cases the number of clauses remains smaller for MiniZSat and due to reduced overhead (propagations) it can thereby find a better solution than MiniSat+.

The main advantage of embedding the minimization function in the solver is the decrease in the size of the problem. This ensures that a large number of options can be verified. The largest disadvantage opposed to MiniSat+ is the inability to discard larger parts of the solution space. MiniSat+ is able to achieve this by the addition of the translated objective function. As a results MiniSat+ has a large number of propagations, where as MiniZSat has a large number of bound conflicts. This leads to time consuming conflict analysis and a learning process in order to prevent the conflict from reappearing.

8.1.2 Bounding techniques

The bounding techniques have been very effective in certain problem instances. The initial lower bound technique was very useful for all the benchmarks, and in specific the random 3SAT problems and the facility location problem. The dynamic lower bound technique was less successful; it could only improve the performance on facility location problems.

The initial lower bound technique allows the solver to converge faster to the upper bound. The increase in the lower bound causes a decrease in solution space to search through. Unfortunately the lower bound could not be employed on all benchmarks due to nature of the technique and the problem instances.

By adding bounding techniques MiniZSat took the SAT based solver MiniSat closer to an integer linear programming solver. MiniZSat may therefore deliver improved performance on problems closer to integer linear programming opposed to for example MiniSat+, of which the facility location problem is an example.

8.1.3 Bound conflict analysis

Bound conflict analysis allowed MiniZSat to improve the performance by strengthening the learnt constraint from a bound conflict. The stronger learnt constraints was able to discard extra parts of the solution space. Next to that it presented the opportunity to backtrack over more levels. These advantages of the bound conflict analysis enabled MiniZSat to deliver improved performances.

8.2 Application areas for MiniZsat

From the results obtained in Chapter 7 the only application area best suitable for MiniZSat is the random 3SAT problem with minimization function. It outperforms all other solvers by a large margin. However this is a theoretical problem and not very usable in normal day life.

Looking at the facility location problem, the performance increase compared to the original version and other SAT based solvers is very large. It still does not perform as well as the integer linear programming solver, but the gap is closing. There remain points for improvement for this specific problem although it seems that integer linear programming will remain the best solver to use for this sort of problems.

Problems which contain both properties of the facility location and the random 3SAT problems may perform very well on MiniZSat. These hypothetical problems would contain both restricting constraints and open constraints. The restricting constraints should make the problem difficult to solve for integer linear programming solvers. On the other hand the open constraints should enlarge the lower bound and ensure better performance from MiniZSat compared to other SAT-based solvers.

8.3 Future work

Several notes have been made in the previous chapters on possible improvements for MiniZSat. In this section a short summary on these improvements will be given.

- **Bounding techniques**; The bounding technique can be improved by the addition of support for multiple watches on a single variable. This could render more clauses suitable for the lower bound techniques.
- **Search strategy**; Different search strategies have been tested. Varying results have been gathered including large performance gains, however there was no clear conclusion on the best search strategy. More research should be done in order to create an appropriate search strategy for MiniZSat.

- **Restart strategy**; A number of restarts strategies were employed with MiniZSat. The different strategies showed the possibility of performance increase by using varying restart strategies. A balanced restart strategy is yet to be found and could largely increase the performance of MiniZSat.
- **Learning capabilities**; Learning is possibly one of the most important aspects to gain performance on in the future. Learning from bound conflicts is currently not very efficient, because MiniZSat is unable to discard large parts of the solution. Other improvements could be gained on the management of learnt clauses. No changes have been made to this part of the solver, this could however still be an interesting area to investigate.
- **Translation from pseudo-Boolean to MiniZSat format**; the translation of pseudo-Boolean constraints to clauses might not be very suitable for MiniZSat. In order to make efficient use of the lower bound techniques large clauses with positive coefficients should be added. The priority for the translation in MiniSat+ is to keep the problem small, which might lead to limited usage of the lower bound technique. A variant running on modified input instances already showed some interesting results.
- **General optimization of the solver**; optimizations could be made in the data structure and related aspects used by MiniZSat.

Bibliography

[1] D. L. Berre, O. Roussel, and L. Simon, "Sat competitions," 2007.

[2] N. Eén and N. Sörensson, "Minisat: A sat solver with conflict-clause minimization," *SAT '05*, 2005.

[3] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into sat," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, 2006. [Online]. Available: jsat.ewi.tudelft.nl/content/volume2/JSAT2_1_Een.pdf

[4] V. Manquinho and O. Roussel, "Pseudo-boolean evaluations," 2007.

[5] A. Forsgren, P. Gill, and M. Wright, "Interior methods for nonlinear optimization," *SIAM Rev.*, vol. 44, no. 4, pp. 525–597, 2002.

[6] G. Dantzig, "Programming in a linear structure," in *Comptroller*, 1948.

[7] K. L. Clarkson, "Las Vegas algorithms for linear and integer programming when the dimension is small," *Journal of the ACM*, vol. 42, no. 2, pp. 488–499, 1995. [Online]. Available: citeseer.ist.psu.edu/clarkson95la.html

[8] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[9] M. S. A. Atamtürk, "Integer-programming software systems," *Annals of Operations Research*, vol. 140, pp. 67–124, 2005.

[10] R. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Society*, vol. 64, pp. 275–278, 1958.

[11] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, "Branch-and-price: column generation for solving huge integer programs," *Operations Research*, vol. 46, pp. 316–329, 1998. [Online]. Available: citeseer.ist.psu.edu/barnhart96branchprice.html

[12] R. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling, "Mixed-integer programming: A progress report," in *The Sharpest Cut*, 2004, pp. 309–326.

[13] I. CPLEX Optimization, *ILOG CPLEX 9.0 Users Manual*, 2003.

[14] J. Linderoth and T. Ralphs, "Noncommercial software for mixed-integer linear programming," in *Integer Programming: Theory and Practice*. CRC Press Operations Research Series, 2005, pp. 253–303.

[15] G. Project, *GLPK Manual v.4.23*, 2007.

[16] S. Cook, "The complexity of theorem-proving procedures," in *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing.* New York, NY, USA: ACM Press, 1971, pp. 151–158.

[17] V. Chandru and J. Hooker, "Extended horn sets in propositional logic," *J. ACM*, vol. 38, no. 1, pp. 205–221, 1991.

[18] J. Franco and A. van Gelder, "A Perspective on Certain Polynomial Time Solvable Classes of Satisfiability," *Discrete Applied Mathematics*, vol. 125, pp. 177–214, 2003. [Online]. Available: citeseer.ist.psu.edu/franco98perspective.html

[19] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[20] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[21] Y. S. Mahajan, Z. Fu, and S. Malik, *Zchaff2004: An Efficient SAT Solver.* Springer, 2004, vol. 3542, pp. 360–375. [Online]. Available: http://www.gigascale.org/pubs/968.html

[22] C. M. Li and Anbulagan, "Heuristics based on unit propagation for satisfiability problems," in *IJCAI (1)*, 1997, pp. 366–371. [Online]. Available: citeseer.ist.psu.edu/li97heuristics.html

[23] C. Gomes, H. Kautz, A. Sabharwal, and B. Selman, *Satisfiability Solvers.* Elsevier, 2007, pp. 1–41.

[24] J. M. Silva, "The impact of branching heuristics in propositional satisfiability algorithms," in *EPIA '99: Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, 1999, pp. 62–74. [Online]. Available: citeseer.ist.psu.edu/article/marques-silva99impact.html

[25] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001. [Online]. Available: citeseer.ist.psu.edu/moskewicz01chaff.html

[26] A. V. Gelder, "Generalizations of watched literals for backtracking search," in *Annals of Mathematics and Artificial Intelligence*, 2004. [Online]. Available: citeseer.ist.psu.edu/642453.html

[27] N. Eén and N. Sörensson, "An extensible sat-solver [ver 1.2]." [Online]. Available: citeseer.ist.psu.edu/een03extensible.html

[28] J. Marques-Silva and K. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 1996, pp. 220–227. [Online]. Available: citeseer.ist.psu.edu/marques-silva96grasp.html

[29] L. Zhang, C. F. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in boolean satisfiability solver," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 2001, pp. 279–285. [Online]. Available: citeseer.ist.psu.edu/article/zhang01efficient.html

[30] C. P. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, Madison, Wisconsin, 1998, pp. 431–437. [Online]. Available: citeseer.ist.psu.edu/gomes98boosting.html

[31] L. Baptista and J. M. Silva, "Using randomization and learning to solve hard real-world instances of satisfiability," in *Principles and Practice of Constraint Programming*, 2000, pp. 489–494. [Online]. Available: citeseer.ist.psu.edu/baptista00using.html

[32] C. P. Gomes, B. Selman, and N. Crato, "Heavy-tailed distributions in combinatorial search," in *Principles and Practice of Constraint Programming*, 1997, pp. 121–135. [Online]. Available: citeseer.ist.psu.edu/gomes97heavytailed.html

[33] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," *Journal of Automated Reasoning*, vol. 24, no. 1/2, pp. 67–100, 2000. [Online]. Available: citeseer.ist.psu.edu/article/gomes99heavytailed.html

[34] A. Nadel, "Backtrack search algorithms for propositional logic satisfiability: Review and innovations," Master's thesis, Hebrew University of Jerusalem, 2002.

[35] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *Eighth International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 61–75.

[36] A. P. H. Dixon, M. Ginsberg, "Generalizing boolean satisfiability i: Background and survey of existing work," *Journal of Artificial Intelligence Research (JAIR)*, vol. 21, pp. 193–243, 2004. [Online]. Available: http://www.jair.org/media/1353/live-1353-2285-jair.pdf

[37] R. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Plenum Press, 1972, pp. 85–103.

[38] O. Roussel, "Pseudo-boolean and cardinality constraints," 2007, to appear in.

[39] A. Haken, "The intractability of resolution (complexity)," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1984.

[40] H. Dixon and M. Ginsberg, "Combining satisfiability techniques from ai and or," *The Knowledge Engineering Review*, vol. 15, pp. 31–45, 2000. [Online]. Available: citeseer.ist.psu.edu/article/dixon00combining.html

[41] I. Markov, K. Sakallah, A. Ramani, and F. Aloul, "Generic ilp versus specialized 0-1 ilp: an update," *iccad*, vol. 00, pp. 450–457, 2002.

[42] D. Chai and A. Kuehlmann, "A fast pseudo-boolean constraint solver," in *40th Design Automation Conference (DAC 2003)*, 2003, pp. 830–835. [Online]. Available: citeseer.ist.psu.edu/article/chai05fast.html

[43] P. Barth, *Logic-based 0-1 constraint programming*. Kluwer, 1995.

[44] V. Manquinho and J. M. Silva, "On applying cutting planes in dll-based algorithms for pseudo-boolean optimization," in *SAT*, 2005, pp. 451–458.

[45] P. Barth, "A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization," Max-Planck-Institut für Informatik," Research Report, 1995.

[46] H. Sheini and K. Sakallah, "Pueblo: A hybrid pseudo-boolean sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 165–189, 2006.

[47] M. Korupolu, C. Plaxton, and R. Rajaraman, "Analysis of a local search heuristic for facility location problems," in *In Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 24, 1998, pp. 1–10. [Online]. Available: citeseer.ist.psu.edu/korupolu00analysis.html

[48] M. Mahdian, Y. Ye, and J. Zhang, "Approximation algorithms for metric facility location problems," 2004, submitted to SIAM Journal on Computing. [Online]. Available: citeseer.ist.psu.edu/mahdian04approximation.html

[49] D. Shmoys, É. Tardos, and K. Aardal, "Approximation algorithms for facility location problems (extended abstract)," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997, pp. 265–274. [Online]. Available: citeseer.ist.psu.edu/shmoys98approximation.html

[50] G. Dantzig and H. Kuhn, *A primal-dual algorithm for linear programs.* Princeton University Press, 1956, pp. 171–182.

[51] V. Melkonian, "New primal-dual algorithms for steiner tree problems," *Computational Operational Research*, vol. 34, no. 7, pp. 2147–2167, 2007.

[52] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. V. Vazirani, "Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP," *Journal of the ACM*, vol. 50, pp. 795–824, 2005. [Online]. Available: citeseer.ist.psu.edu/537420.html

[53] A. Byrka and K. Aardal, "The approximation gap for the metric facility location problem is not yet closed," *Operations Research Letters*, vol. 35, pp. 379–384, 2007.

[54] M. Charikar and S. Guha, "Improved combinatorial algorithms for the facility location and k-median problems," in *IEEE Symposium on Foundations of Computer Science*, 1999, pp. 378–388. [Online]. Available: citeseer.ist.psu.edu/charikar99improved.html

[55] K. Jain, M. Mahdian, and A. Saberi, "A new greedy approach for facility location problems," in *In Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, 2002, pp. 731–740. [Online]. Available: citeseer.ist.psu.edu/jain02new.html

[56] S. Guha, "Approximation algorithms for facility location problems," Ph.D. dissertation, Stanford University, 2000.

[57] K. Al-Sultan and M. Al-Fawzan, "A tabu search approach to the uncapacitated facility location problem," *Annals of Operations Research*, vol. 86, pp. 91–103, 1999.

[58] H. Sheini, "Glppb v.0.2," 2006. [Online]. Available: http://www.cril.univ-artois.fr/PB06/papers/glpPB02.pdf