Incrementally encoding cardinality and pseudo-boolean constraints in SAT

J. Langerak

TU Delft

Incrementally encoding cardinality and pseudo-boolean constraints in SAT

by

J. Langerak

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday September 29, 2001 at 11:00 AM.

Student number:	4317327	
Project duration:	September 1, 2020 – September 29, 2021	
Thesis committee:	Dr. N. Yorke-Smith,	TU Delft
	Dr. E. Demirovi,	TU Delft, daily supervisor
	Dr. C. Hauff,	TU Delft

TU Delft

Abstract

Satisfiability solvers have been shown to be a powerful tool for solving constraint problems. These problems often contain pseudo-boolean and cardinality constraints. These constraints can either be encoded into SAT or handled by extending the solver with special propagators. Which method will perform better is often not known in advance. Abío et al. [1, 4] have already shown that adding the encoding during the search can be beneficial. In this thesis we will extend their work by looking how the encoding can incrementally be constructed during the search. We will develop a couple of methods that only encode the active parts of the constraints. In contrast to their work the full encoding of the constraint is not determined beforehand but instead is determined during the search. Furthermore, we will show that during the search the same subset of variables is active and therefore not all variables are needed for the encoding. Next, we will show that the order of the literals in the encoding has effect on the performance. However, this mostly affects the first part of the solve process, and therefore the effect on optimization problems is limited. Finally, we will show that an incremental encoding can lead to a smaller encoding while having similar results as the full encoding.

Preface

Before you lies my thesis, which will mark the end of my time studying at the TU Delft. I had a great time during my studies and was able to develop myself, both trough the study and through things you can do alongside your studies. This thesis is the product of everything I have learned the past years.

The goal of the thesis is to improve the handling of cardinality and pseudo-boolean constraints in a SAT solver. SAT solvers are powerful tool for solving a variety of constraint programming problems. A lot of these problems rely on cardinality and pseudo-boolean constraints. The best way to handle these constraints is not always known in advance and with thesis I try to improve this by incrementally constructing the encoding during the search.

The thesis done at the Algorithmics group at the Delft University of Technology. I would like to thank Emir for being the supervisor during the thesis. The feedback that I received was helpful and gave rise to new ideas to try. Finally, I want to thank my friends and family for their support through the years.

J. Langerak
Delft, September 2021

Contents

1 Introduction 1
- 1.1 Research questions . . . 1
- 1.2 Thesis contribution . . . 2
- 1.3 Outline . . . 2

2 Background 3
- 2.1 SAT problem . . . 3
- 2.2 DPLL algorithm . . . 4
- 2.3 CDCL algorithm. . . . 7
  - 2.3.1 Conflict analysis . . . 7
  - 2.3.2 Example . . . 9
- 2.4 Pseudo-boolean and cardinality constraints . . . 12
- 2.5 Generalized totaliser encoding (GTE) . . . 13
- 2.6 Propagation . . . 14
- 2.7 MAX-SAT problem . . . 14

3 Related research 17
- 3.1 Encode or propagate . . . 18

4 Methods 21
- 4.1 Incremental . . . 21
  - 4.1.1 Tree construction . . . 21
  - 4.1.2 Update encoding. . . . 21
  - 4.1.3 Encoding criteria. . . . 22
- 4.2 Top-down . . . 23
  - 4.2.1 Encoding criterium . . . 24
  - 4.2.2 Grouping. . . . 24
- 4.3 Bottom layers . . . 25
  - 4.3.1 Full Bottom Layers . . . 25
  - 4.3.2 Pairs . . . 26
- 4.4 Overview of the methods . . . 26

5 Benchmarks 29
- 5.1 MAX-SAT problems . . . 29
  - 5.1.1 Reducing number of distinct weights . . . 29
- 5.2 Pseudo boolean problems . . . 29
- 5.3 Curriculum-based timetabling . . . 30
  - 5.3.1 Concepts. . . . 30
  - 5.3.2 Hard constraints . . . 31
  - 5.3.3 Soft constraints . . . 31
  - 5.3.4 Encoding. . . . 31
- 5.4 Metrics . . . 34
  - 5.4.1 Normalization . . . 34
- 5.5 StarExec. . . . 34

6 Experimental results 35
- 6.1 Activity of the literals . . . 35
- 6.2 Literal order . . . 36
  - 6.2.1 Methods . . . 38
  - 6.2.2 Default order. . . . 38
  - 6.2.3 Weight . . . 38

6.2.4 Random order . . . 38
6.2.5 Count . . . 38
6.2.6 Activity . . . 38
6.2.7 Distance . . . 39
6.2.8 Results . . . 39
6.3 Usefulness of the auxiliary literals . . . 40
6.4 Size of the encoding . . . 43
6.5 Performance . . . 44

7 Conclusion 47
7.1 Recommendations . . . 48

Bibliography 49

A Appendix 53
A.1 Notation . . . 53
A.2 Normalized penalty CTT . . . 54
A.3 Normalized penalty MAX-SAT uw . . . 55
A.4 Normalized penalty MAX-SAW w=1 . . . 56
A.5 Normalized penalty MAX-SAW w=5 . . . 57
A.6 Normalized solve time PB10 . . . 58
A.7 Normalized solve time PB15 . . . 59
A.8 Normalized solve time d_n_k . . . 60
A.9 Normalized solve time Elffers . . . 61
A.10 Normalized solve time Elffers$_1$ . . . 62
A.11 Normalized solve time Elffers$_2$ . . . 63

1 Introduction

The satisfiability problem is a problem that through the years is extensively studied. It is the first problem that was proven to be NP-complete [21]. Despite that there exists no polynomial algorithm (unless P=NP), researchers have been able to create very efficient solvers. Especially conflict driven clause learning (CDCL) has proven to be a very powerful technique for solving a SAT problem. Now we are able to solve instances with hundreds of thousand variables and clauses. Furthermore, a lot of problems can efficiently be reduced to SAT instances and these problems can then be solved with the help of a SAT solver.

It has been shown that a SAT solver is a powerful tool for solving constraint programming problems. Among others it has successfully been applied to planning and scheduling problems [7, 17, 20, 24, 41, 46] and for solving software dependencies [42, 60]. Another area where it has been applied is Bounded Model Checking, for example hardware [15, 31] and Software [34] verification. Furthermore, is has been used for verifying product configurations [49, 58]. This are just a few examples of areas where SAT has been applied.

In order to use a SAT solver for solving a constraint problem, all the constraints must be converted to SAT clauses. However, not all constraints can be efficiently converted to a SAT encoding. Two of such constraints are the cardinality and pseudo-boolean constraints. These constraints can be used to specify how many literals are allowed to be true. For example, consider an employee scheduling problem, where at each shift exactly n employees must be scheduled. This can be specified as a cardinality constraint. These types of constraints appear a lot in real world problems. However, their SAT encoding is often very large, which will make solving the problem more difficult. An alternative approach is to extend the solver so that it can handle these constraints directly without having to encoding them, however for this approach there is a risk that the solver tries every possible violation of the constraint. This is of course not very efficient. For some problems encoding the constraint is the best option and for others it is better to extend the solver to handle the constraint directly.

Stuckey and Abío [1] have tried to combine these approaches by encoding the constraints during the search instead of at the start of the solve process. They have shown that this is a powerful technique that was often able to get a result similar to the best fixed strategy, where the fixed strategies are to either encode the constraints fully at the start or to never encode them. The encoding they partially added during the search was determined before the start of the solver. In the thesis we will investigate if there is an advantage for determining the encoding during the search instead of at the start of the solver. We will try different methods that combine the propagation and encoding method and we will evaluate what their effect is on the performance.

1.1. Research questions

The main objective of this thesis is to investigate if a partial encoding of cardinality and pseudo-boolean constraints can be beneficial for the solver. For pseudo-boolean constraints we will focus on the constraints with small weights because they can be handled in a similar way as the cardinality constraints. This is investigated by looking at the following sub-questions.

- **RQ1:** Is it needed to encode the full constraint? If this is not needed, then what are the important parts of the constraint to encode?

When constructing a partial encoding we have to decide which literals should be encoded. By analysing which literals are important we can learn which literals should be encoded and which can be handled by the propagator.

- **RQ2:** Has the order/grouping of the literals effect on the performance?

The advantage of the encoding over propagation is that it can utilize more general variables for the conflict analysis. A general variable can for example encode that only x_1 or x_2 can be set to True. This leads to more general learned clauses. Whether that general variable for x_1 and x_2 exists depends on the order of the input literals of the encoding. By placing related literals close to each other, one can increase the likelihood that the more general variables can be used. However, it is not known how big this effect is.

- **RQ3:** What is the effect of the incremental construction on the size of the encoding?

Larger problems tend to be more difficult to solve. Partial encodings should lead to a smaller size. However, if the final partial encoding has a similar size as the full encoding, then there might be no advantage for the partial encoding.

- **RQ4:** What is the effect of the incremental construction on the performance of the solver?

The main objective of the thesis is to improve the performance. This will be tested by looking at different types of benchmarks. Some of these will favour solvers that encode the constraints and others will favour the propagation method.

1.2. Thesis contribution

During the thesis several methods are proposed that construct a partial encoding. We will show that the active literals for a lot of problems are stable over time and therefore a partial encoding can be beneficial. We will show that for optimization problems a partial encoding can lead to a smaller encoding while having a similar or better performance as the full encoding. Several methods that modified the ordering were tested, but the differences between them were not significant.

1.3. Outline

The next chapter will provide the background information for the thesis. It will explain the core concepts of a SAT solver, furthermore it explains how the cardinality and pseudo-boolean constraints can be handled by a SAT solver. Chapter 3 looks at related work and discusses the work of Stuckey and Abío [1] in more detail. This will be followed by the chapter that discusses the methods that were tested during the thesis. Chapter 5 introduces the benchmarks that were used for the different tests. Furthermore, it discusses how the tests are performed and how the data is normalized for the different instances. In chapter 6 we present the results of the thesis. It starts with the experiments that were done to analyse what the important parts of the encoding are. These observations are the reason for some of the methods in chapter 4. The final chapter contains the conclusion and recommendations. A full overview of the different tested configurations can be found in the appendix.

2 Background

This chapter will explain the general concepts of the thesis and introduces some of the notations. The chapter starts with the formal definition of the SAT problem and sketches the main components of a modern SAT solver. Next, it introduces the pseudo-boolean constraints and explains how they are often handled by a SAT solver. The final section of the chapter introduces the MAX-SAT problem.

2.1. SAT problem

A SAT problem contains a set V of propositional variables and a set C containing clauses. A clause is disjunction of literals, where a literal is either a variable in V or the negation of a variable in V. If a literal l is v then the negation of l is $\overline{v}$ and if l is $\overline{v}$ its negation is v.

A clause is satisfied when there is at least one literal that evaluates to true for the given assignments of V. The goal of the problem is to find an assignment to all variables in V such that all clauses in C are satisfied. When such solution exists then the problem is satisfiable. If there exist no such solution, then the problem is unsatisfiable.

Suppose we have following SAT problem:

$$\begin{aligned} V &= \{x_1, x_2, x_3\} \\ C &= \{x_1 \vee x_2, \\ &\quad \overline{x_1} \vee x_3\} \end{aligned}$$

A satisfying assignment for this problem would be $\{x_1, x_2, x_3\}$. The first clause is True since x_1 is True and the second class is satisfied by the True assignment of x_3. An unsatisfying assignment would be $\{x_1, x_2, \overline{x_3}\}$. The second clause has no literal that evaluates to True.

An example of an unsatisfying SAT problem is the following problem:

$$\begin{aligned} V &= \{x_1, x_2\} \\ C &= \{x_1 \vee x_2, \\ &\quad \overline{x_1} \vee x_2, \\ &\quad \overline{x_1} \vee \overline{x_2}, \\ &\quad x_1 \vee \overline{x_2}\} \end{aligned}$$

If x_1 is set to True, then the first and last clause are satisfied. In order to satisfy the second clause, we have to set x_2 to True. However, to satisfy the third clause we have to set x_2 to False. Since we cannot set x_2 to both True and False, this will not lead to a solution.

If x_1 is set to False, then the second and third clause are satisfied. However, in order to satisfy the first clause, we have to set x_2 to True and for the last clause x_2 must be set to False. This is also not possible, thus there exists no assignment that satisfies all clauses. Therefore, the problem is unsatisfiable.

Basic constraints

When a problem is encoded as a SAT problem all the constraints must be converted to clauses. During the thesis we discuss certain encodings that add constraints to the SAT problem. Depending on the context we will either specify the encoding using clauses or with the help of simpler constraints. The following constraints can easily be converted to SAT clauses and will be used to describe more complex constraints later on.

$$\begin{aligned} l_1 \wedge l_2 &\iff \{l_1, l_2\} \\ \overline{l_1 \wedge l_2} &\iff \overline{l_1} \vee \overline{l_2} \\ (l_1 \implies l_2) &\iff \overline{l_1} \vee l_2 \\ (l_1 \iff l_2) &\iff \{\overline{l_1} \vee l_2, l_1 \vee \overline{l_2}\} \end{aligned}$$

These constraints can also be combined, for example the constraint $(l_1 \wedge l_2) \implies (l_3 \wedge l_4)$ can be converted to a SAT problem as follows:

$$\begin{aligned} &(l_1 \wedge l_2) \implies (l_3 \wedge l_4) \\ \iff &\overline{(l_1 \wedge l_2)} \vee (l_3 \wedge l_4) \\ \iff &\overline{l_1} \vee \overline{l_2} \vee (l_3 \wedge l_4) \\ \iff &\{\overline{l_1} \vee \overline{l_2} \vee l_3, \overline{l_1} \vee \overline{l_2} \vee l_4\} \end{aligned}$$

2.2. DPLL algorithm

A SAT problem can be solved using the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [23]. It is a backtracking based search algorithm that either finds a satisfying solution to the problem or is able to proof that it is unsatisfiable. The algorithm searches for a solution by incrementally assigning values to the variables in V. Thus during the search, a variable can be set to True, False or still be unassigned. Initially all variables are unassigned. The pseudocode is shown in algorithm 1.

Algorithm 1: The core idea of the DPLL algorithm.

```
1  function DPLL (P, A);
   Input  : P the SAT problem. A the current assignments. For the first call A is empty.
   Output: True if a solution is found. False if no solution is found.
2  while There exist a unit clause do
3      Assign a variable using unit propagation and add it to A
4      if Conflict detected then
5          return False
6      end
7  end
8  if All variables are assigned then
9      return True
10 end
11 Choose an unassigned variable $v$
12 return dpll($A \cup \{v\}$) or dpll($A \cup \{\overline{v}\}$)
```

The solver iterates trough three steps.

Propagation
The first step is propagation. During this step the solver checks if it can deduce new information using the current assigned variables and the clauses that are not yet satisfied. The deduction can lead to an assignment for a currently unassigned variable, this is called implication. For deduction most solvers rely on unit propagation. With this type of propagation, the solver checks if there are unit clauses. A unit clause is a currently unsatisfied clause with only one unassigned literal. In order to satisfy the unit clause, the unassigned literal must be set to true. The propagation step can be extended with more advanced reasoning steps [10, 40], however since unit propagation can be implemented very efficiently it often outperforms the more advanced reasoning steps. Therefore, most solvers rely only on unit propagation.

Furthermore, propagation can also detect a clause that can no longer be satisfied. This is called a conflict. If a conflict is detected, the propagation stops, and the solver backtracks. If the propagation does not find a conflict and is not able to do any more deductions, then the solver goes to the decision step.

Decision

In the decision step the solver selects a currently unassigned variable and gives it a value. The number of decisions that are currently made is called the decision level. Initially when there are no decisions made the level is 0. The selection of the variable is often done by selecting the variable with the highest activity. The solver stores for each variable its activity and each time a variable is part of a conflict its activity is increased. In order to give higher weights to recent conflicts the activity decays over time. Initially nothing is known about the activity and therefore the solver takes somewhat random decisions. There is a risk that the solver started with some bad decisions and therefore gets stuck in a part of the search tree that will not result in a solution. Therefore, modern solvers tend to restart the search process often. After each restart the solver has collected more data over the activity and should be able to make better decisions.

Backtrack

When a conflict is detected, the solver backtracks to the highest decision level where the opposing value for the decision variable is not yet tried. In algorithm 1 the backtracking step is the recursive return statement on line 12.

The solver terminates when it has found a satisfying assignment, or when it can no longer backtrack and has to conclude that the problem is unsatisfiable.

Next, we will study an example to get an idea how the algorithm works. Suppose we have the following clauses:

$$\begin{aligned} c_1 &= \overline{x_1} \vee x_2 \\ c_2 &= x_3 \vee x_4 \\ c_3 &= x_4 \vee x_5 \\ c_4 &= \overline{x_1} \vee \overline{x_4} \vee x_5 \\ c_5 &= \overline{x_1} \vee x_4 \vee \overline{x_5} \\ c_6 &= \overline{x_1} \vee \overline{x_4} \vee \overline{x_5} \end{aligned}$$

The search tree for the problem is shown in figure 2.1. Initially all variables are unassigned. Since there is no unit clause, propagation is not possible. Therefore, the solver has to make a decision. For example, it sets x_1 to True (figure 2.1a).

$$\begin{aligned} c_1 &= F \vee x_2 \\ c_2 &= x_3 \vee x_4 \\ c_3 &= x_4 \vee x_5 \\ c_4 &= F \vee \overline{x_4} \vee x_5 \\ c_5 &= F \vee x_4 \vee \overline{x_5} \\ c_6 &= F \vee \overline{x_4} \vee \overline{x_5} \end{aligned}$$

Now the first clause has become unit, thus x_2 must be True as well.

$$\begin{aligned} c_1 &= F \vee T \\ c_2 &= x_3 \vee x_4 \\ c_3 &= x_4 \vee x_5 \\ c_4 &= F \vee \overline{x_4} \vee x_5 \\ c_5 &= F \vee x_4 \vee \overline{x_5} \\ c_6 &= F \vee \overline{x_4} \vee \overline{x_5} \end{aligned}$$

It is not possible to do more unit propagation and therefore the solver makes a decision and sets x_3 to True (figure 2.1b).

$$\begin{aligned}
c_1 &= F \lor T \\
c_2 &= T \lor x_5 \\
c_3 &= x_4 \lor x_5 \\
c_4 &= F \lor \overline{x_4} \lor x_5 \\
c_5 &= F \lor x_4 \lor \overline{x_5} \\
c_6 &= F \lor \overline{x_4} \lor \overline{x_5}
\end{aligned}$$

Again, the solver has to make a decision and sets x_4 to True (figure 2.1c).

$$\begin{aligned}
c_1 &= F \lor T \\
c_2 &= T \lor T \\
c_3 &= T \lor x_5 \\
c_4 &= F \lor F \lor x_5 \\
c_5 &= F \lor T \lor \overline{x_5} \\
c_6 &= F \lor F \lor \overline{x_5}
\end{aligned}$$

Clause c_4 is now unit and can be used to propagate x_5 to True.

$$\begin{aligned}
c_1 &= F \lor T \\
c_2 &= T \lor T \\
c_3 &= T \lor T \\
c_4 &= F \lor F \lor T \\
c_5 &= F \lor T \lor F \\
c_6 &= F \lor F \lor F
\end{aligned}$$

However c_6 is now unsatisfied thus there is a conflict (figure 2.1d). Thus, the solver starts to backtrack. It has to backtrack to the decision where it had set x_4 to True, thus it reverts the assignment of x_5 and x_4. Next, the solver sets x_4 to False (figure 2.1e).

$$\begin{aligned}
c_1 &= F \lor T \\
c_2 &= T \lor F \\
c_3 &= F \lor x_5 \\
c_4 &= F \lor T \lor x_5 \\
c_5 &= F \lor F \lor \overline{x_5} \\
c_6 &= F \lor T \lor \overline{x_5}
\end{aligned}$$

After propagating x_5 there is again a conflict (figure 2.1f). This time the solver has to backtrack to x_3. Now it sets x_3 to False.

$$\begin{aligned}
c_1 &= F \lor T \\
c_2 &= F \lor x_4 \\
c_3 &= x_4 \lor x_5 \\
c_4 &= F \lor \overline{x_4} \lor x_5 \\
c_5 &= F \lor x_4 \lor \overline{x_5} \\
c_6 &= F \lor \overline{x_4} \lor \overline{x_5}
\end{aligned}$$

Next, it decides to set x_4. The True and False assignments for x_4 will both lead to a conflict (figure 2.1g). After finding both conflicts the solver has to backtrack to x_1 and set it to False.

$$\begin{aligned} c_1 &= T \vee x_2 \\ c_2 &= x_3 \vee x_4 \\ c_3 &= x_4 \vee x_5 \\ c_4 &= T \vee \overline{x_4} \vee x_5 \\ c_5 &= T \vee x_4 \vee \overline{x_5} \\ c_6 &= T \vee \overline{x_4} \vee \overline{x_5} \end{aligned}$$

Next, the solver decides to set x_2, x_3 and x_4 to True. This will lead to a solution (figure 2.1h). Thus, the problem is satisfiable.

$$\begin{aligned} c_1 &= T \vee T \\ c_2 &= T \vee T \\ c_3 &= T \vee T \\ c_4 &= T \vee F \vee T \\ c_5 &= T \vee T \vee F \\ c_6 &= T \vee F \vee F \end{aligned}$$

2.3. CDCL algorithm

Modern SAT solvers rely on the Conflict-Driven Clause Learning (CDCL) [57] algorithm to find a solution for a SAT problem. The algorithm is inspired by the DPLL algorithm and improves the backtracking step and learns new clauses during the search. By doing so it avoids making the same mistakes later in the search and it can skip certain parts of the search tree that would be visited by the DPLL algorithm.

Initially this algorithm behaves the same as the DPLL algorithm, but it starts to differ when a conflict is detected. Instead of immediately backtracking it analyses the current conflict and generates an explanation for the conflict. A trivial explanation would be all the current decision literals. If at a later point during the search they are all set again than we know that the same conflict will occur as is currently happening. For example, suppose that there is a conflict and the current decision literals are $\{x_1, x_2\}$, then an explanation would be $x_1 \wedge x_2$. In order to prevent the same assignment in the future we can construct a new clause that contains the complement of all literals in the explanation. For this example this would be the clause $\overline{x_1} \vee \overline{x_2}$ and by adding the clause to the problem the same mistake can no longer be made.

There are different strategies to construct a conflict clause. A conflict clause can contain both decision and implied literals. The most common used strategy is to find the first unique implication point, which will be discussed in the next section.

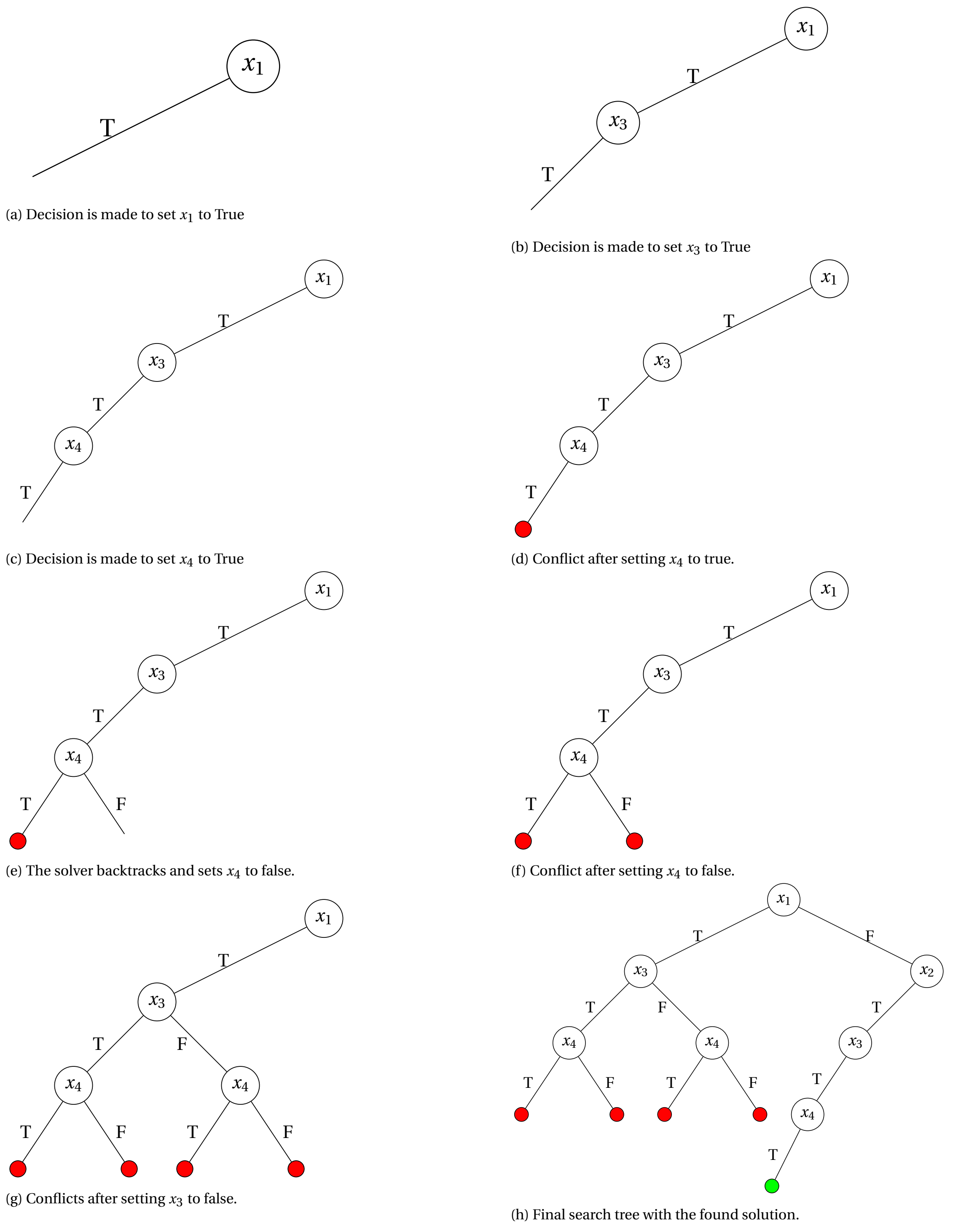
A conflict clause should satisfy three criteria. First of all, it should be logical implied by the constraints of the original problem. This ensures correctness. Secondly, the clause should be False by the current assigned variables. Finally, the clause should contain exactly one literal assigned at the current decision level. The final two criteria ensures that if we backtrack one level the clause becomes unit.

After learning a clause, the solver backtracks to the lowest decision level where the learned clause is still a unit clause. In the worst case this would be one level, however it is also possible to backtrack multiple levels. By backtracking multiple levels, the solver skips part of the search tree that does not contain a solution but would be visited by the DPLL solver. The pseudocode for the CDCL algorithm is shown in algorithm 2.

2.3.1. Conflict analysis

In the previous section we used all the decision variables for the conflict clause. Often, however, not all decision literals are involved in the conflict. Furthermore, it can also happen that an implied literal is a better explanation than the decision literals. This can be illustrated with the following example.

$$\begin{aligned} c_1 &= \overline{x_1} \vee \overline{x_2} \vee x_5 \\ c_2 &= x_1 \vee x_3 \vee \overline{x_5} \\ c_3 &= \overline{x_4} \vee \overline{x_5} \vee x_6 \\ c_4 &= \overline{x_4} \vee \overline{x_6} \end{aligned} \tag{2.1}$$



(a) Decision is made to set x_1 to True

(b) Decision is made to set x_3 to True

(c) Decision is made to set x_4 to True

(d) Conflict after setting x_4 to true.

(e) The solver backtracks and sets x_4 to false.

(f) Conflict after setting x_4 to false.

(g) Conflicts after setting x_3 to false.

(h) Final search tree with the found solution.

Figure 2.1: Search tree for the DPLL algorithm.

Suppose that the solver has decided to set x_1, x_2, x_3 and x_4 tot True. c_1 implies that x_5 is True and c_4 implies that x_6 is False. c_3 is now False, thus there is a conflict. Looking at the decision literals we can learn the clause $\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee \overline{x_4}$. However, x_3 has no effect on the conflict. Setting x_3 to False will also lead to a conflict, thus $\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}$ would also be a valid clause. Furthermore, $x_1 \wedge x_2$ imply x_5 and therefore $\overline{x_1} \vee \overline{x_2}$ can be replaced by $\overline{x_5}$. Thus, $\overline{x_5} \vee \overline{x_4}$ is also a valid conflict clause.

An explanation for a conflict can be found with the help of an implication graph. The implication graph for problem 2.1 is shown in figure 2.2. The green nodes are the decision nodes, the red node is the conflict

Algorithm 2: The core idea of the CDCL algorithm.

```
1  function CDCL ($P$);
   Input  : $P$ the SAT problem.
   Output: True if a solution is found. False if no solution is found.
2  while No solution has been found do
3      Apply unit propagation
4      if Conflict is detected then
5          if Current decision level is 0 then
6              return False
7          end
8          Learn a clause $C$ that explains the conflict and add it to $P$
9          Backtrack to the lowest decision level where $C$ is a unit clause
10     end
11     if All variables are assigned then
12         return True
13     end
14     Increase the decision level
15     Choose an unassigned variable $v$ and give it a value
16 end
```

and the black nodes are the implied nodes. The incoming edges from a node show by which combination of literals the node was implied. For example, x_5 is implied by the first constraint since both x_1 and x_2 are set to True. Thus in the implication graph there is an edge from x_1 to x_5 and an edge from x_2 to x_5. Both edges are labelled with the constraint that causes the propagation, which is in this case c_1.

The graph must be cut into two partitions, the reason and the conflict partition. The conflict partition contains the conflict node and the reason partition contains all decision nodes. The implied nodes can be in any partition. A conflict clause is generated by taking all literals in the reason partition that have an outgoing edge to a node in the conflict partition. Each cut can be a conflict clause and the one that will be found depends on the strategy of the solver. Most solvers use the strategy called first unique implication point [57], also called 1-UIP.

A unique implication point (UIP) is a node such that all paths from a decision variable to the conflict will go trough the node. In the given example x_5 is a unique implication point since all paths from x_1 to the conflict go through x_5. Furthermore, the decision variables themselves are also UIP. 1-UIP searches for the UIP that is the closest to the conflict and draws the cut after the found UIP. Thus, the UIP is part of the reason set, but any literal that is implied by the UIP is part of the conflict set. The pseudocode for the method is shown in algorithm 3. Each iteration the algorithm replaces the most recent set literal from the conflict explanation with the literals that implied that literal. It stops when it has found the closest UIP. The closest UIP is on all paths from the most recent decision literal to the conflict node. Therefore, the algorithm will at some point have only one literal from the current decision level in its explanation set. That literal has to be the closet UIP.

2.3.2. Example

To show the effect of the CDCL algorithm we will apply it on the same example as the DPLL algorithm.

$$
\begin{aligned}
c_1 &= \overline{x_1} \lor x_2 \\
c_2 &= x_3 \lor x_4 \\
c_3 &= x_4 \lor x_5 \\
c_4 &= \overline{x_1} \lor \overline{x_4} \lor x_5 \\
c_5 &= \overline{x_1} \lor x_4 \lor \overline{x_5} \\
c_6 &= \overline{x_1} \lor \overline{x_4} \lor \overline{x_5}
\end{aligned}
$$

Initially the CDCL algorithm behaves the same as the DPLL algorithm. Thus, it will start by deciding to set x_1 to True, which implies that x_2 should be True as well. Next, it decides to set x_3 and x_4 to True. c_4 implies that x_5 should be True, however c_5 implies that x_5 should be False, thus there is a conflict (figure 2.4a). At this

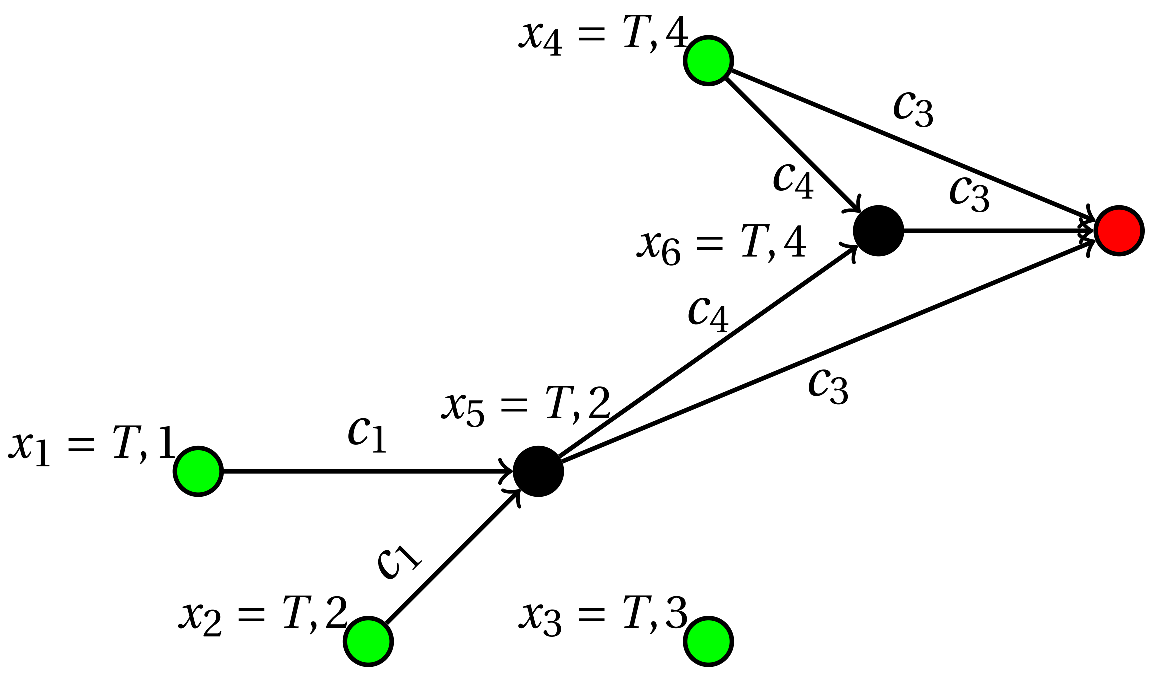
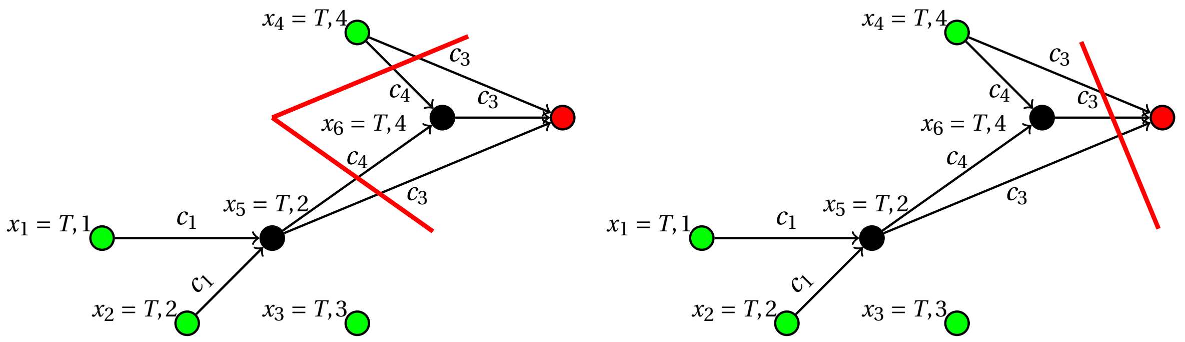


Figure 2.2: Implication graph of example 2.1 after setting x_1, x_2, x_3 and x_4 to true. Green nodes are the decision literals. Black node are the implied literals. The label of each node indicates the variable and the value that was set. The number behind the label refers to the decision level. The label on the implication edges refer to the clause that causes the implication. The red node is a conflict.



(a) Cut corresponding to the conflict clause $\overline{x_4} \vee \overline{x_5}$ (b) Cut corresponding to the conflict clause $\overline{x_4} \vee \overline{x_6} \vee \overline{x_5}$

Figure 2.3: Two possible cuts on the implication graph.

Algorithm 3: The 1-UIP algorithm.

```
1 function 1-IUP (C);
  Input  : C the literals that caused the conflict. C always contains literals that are true for the current
           assignment.
  Output: A learnt conflict clause.
2 Let d be the current decision level
3 while The number of variables in C assigned at level d is larger then 1 do
4     Find the most recent set literal l in C
5     Let I be the set of literals that implied l
6     In C replace l with I
7 end
8 Create the conflict clause by taking the disjunction of the negated literals in C
9 return the conflict clause
```

point the CDCL algorithm starts to behave differently. The conflict is caused by $x_1 \wedge x_4 \wedge \overline{x_5}$. Using the 1-IUP algorithm the solver learns the clause $l_1 = \overline{x_1} \vee \overline{x_4}$. Next, we have to backtrack to the earliest decision where the learned clause has become unit. This is at decision level 1, at this point only x_1 is set (figure 2.4b).

$$\begin{aligned}
c_1 &= F \vee x_2 \\
c_2 &= x_3 \vee x_4 \\
c_3 &= x_4 \vee x_5 \\
c_4 &= F \vee \overline{x_4} \vee x_5 \\
c_5 &= F \vee x_4 \vee \overline{x_5} \\
c_6 &= F \vee \overline{x_4} \vee \overline{x_5} \\
l_1 &= F \vee \overline{x_4}
\end{aligned}$$

Now $\overline{x_4}$ can be propagated using l_1. Using c_3, x_5 can be set to True. However, this will lead to a conflict in c_5 (figure 2.4c). This conflict is caused by $x_1 \wedge \overline{x_4} \wedge x_5$ and after applying 1-UIP we learn the clause $l_2 = \overline{x_1}$. This clause is already a unit clause when no decision is made. Thus, we backtrack all decisions.

$$\begin{aligned}
c_1 &= \overline{x_1} \vee x_2 \\
c_2 &= x_3 \vee x_4 \\
c_3 &= x_4 \vee x_5 \\
c_4 &= \overline{x_1} \vee \overline{x_4} \vee x_5 \\
c_5 &= \overline{x_1} \vee x_4 \vee \overline{x_5} \\
c_6 &= \overline{x_1} \vee \overline{x_4} \vee \overline{x_5} \\
l_1 &= \overline{x_1} \vee \overline{x_4} \\
l_2 &= \overline{x_1}
\end{aligned}$$

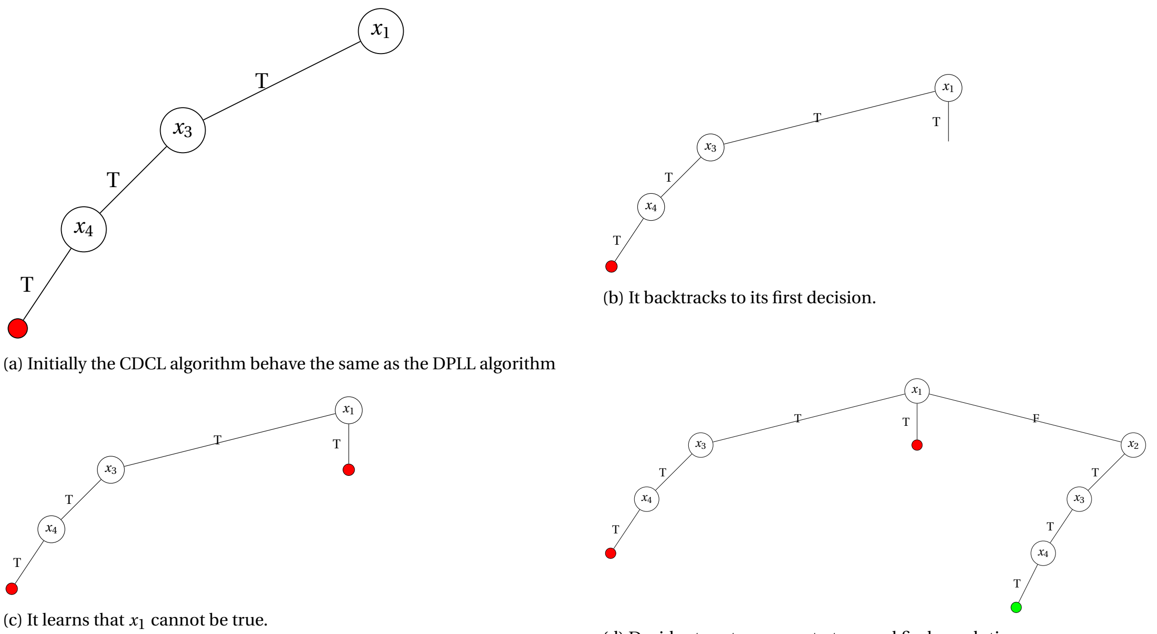
Now we do not have to make a decision but instead we can use l_2 to set x_1 to False.

$$\begin{aligned}
c_1 &= T \vee x_2 \\
c_2 &= x_3 \vee x_4 \\
c_3 &= x_4 \vee x_5 \\
c_4 &= T \vee \overline{x_4} \vee x_5 \\
c_5 &= T \vee x_4 \vee \overline{x_5} \\
c_6 &= T \vee \overline{x_4} \vee \overline{x_5} \\
l_1 &= T \vee \overline{x_4} \\
l_2 &= T
\end{aligned}$$

Next the solver decides to set x_2, x_3 and x_4 to True.

$$\begin{aligned}
c_1 &= T \vee T \\
c_2 &= T \vee T \\
c_3 &= T \vee x_5 \\
c_4 &= T \vee F \vee x_5 \\
c_5 &= T \vee T \vee \overline{x_5} \\
c_6 &= T \vee F \vee \overline{x_5} \\
l_1 &= T \vee F \\
l_2 &= T
\end{aligned}$$

At this point all clauses are satisfied and we can complete the assignment by giving x_5 an arbitrary value. Figure 2.4 shows the search tree for the CDCL algorithm. If we compare the algorithm with the DPLL algorithm we can see that there were much less conflicts needed to find the solution.



(a) Initially the CDCL algorithm behave the same as the DPLL algorithm

(b) It backtracks to its first decision.

(c) It learns that x_1 cannot be true.

(d) Decides to set x_2, x_3 x_4 to true and finds a solution.

Figure 2.4: The search tree for the CDCL algorithm.

Finally on real world problems there occur a lot of conflicts during the search. This will generate a lot of learned clauses and not all of them will be useful. At some point all the extra clauses will start slowing the solver down. Therefore, the solver regularly restarts and removes some of its learned clauses.

2.4. Pseudo-boolean and cardinality constraints

The SAT problem can be used to model a lot of different problems. However not all constraints can easily be converted to clauses. A cardinality constraint is one of those difficult constraints. This constraint can be used to specify how many literals should be at least or at most True. This can be written as follows:

$$l_1 + l_2 + .. + l_i \# n \tag{2.2}$$

Where $\# \in \{\leq, \geq, =, <, >, \neq\}$ and a True value for l_i is represented by 1 and False by 0.

A pseudo-boolean (pb) constraint is a generalization of the cardinality constrain and adds a weight to each literal. Thus it can be written as follows:

$$\sum w_i l_i \# n \tag{2.3}$$

These types of constraints are very useful for scheduling and timetabling problems. For example, suppose the problem where we have to allocate a timeslot for M meetings, but we have only enough rooms to schedule n meetings at the same time. Let T be the number of timeslots. For each meeting m and timeslot t, we create a variable $E_{m,t}$. If $E_{m,t}$ is set to True, then the meeting m is scheduled at time t. Finally, each meeting should be scheduled only once. The problem can be specified as follows:

$$\forall t \in T : \sum_{m=1}^{M} E_{m,t} \leq n \tag{2.4}$$

$$\forall m \in M : \sum_{t=1}^{T} E_{m,t} = 1 \tag{2.5}$$

The first constraint specifies that only n meetings can be scheduled at the same time and the second that all meetings must be scheduled once.

During the thesis the focus will be on pseudo-boolean constraints that specify the maximum allowed value, thus of the form $\sum w_i l_i \leq n$.

2.5. Generalized totaliser encoding (GTE)

There are different ways to handle pseudo-boolean constraints with SAT solvers. A common approach is to convert the constraint to a SAT encoding. There are a number of different encodings proposed in the literature. The method that is used during the thesis is the generalized totaliser encoding (GTE) [35]. The original totaliser [12] was specified for cardinality constraints. With GTE it is for the literals possible to have weights and therefore it can be used to encode both pseudo-boolean and cardinality constraint.

The GTE can be viewed as a binary tree where each node sums its children, this is shown in figure 2.5. Each node introduces for each possible subsum w of its children an auxiliary variable a_w. If the sum of its

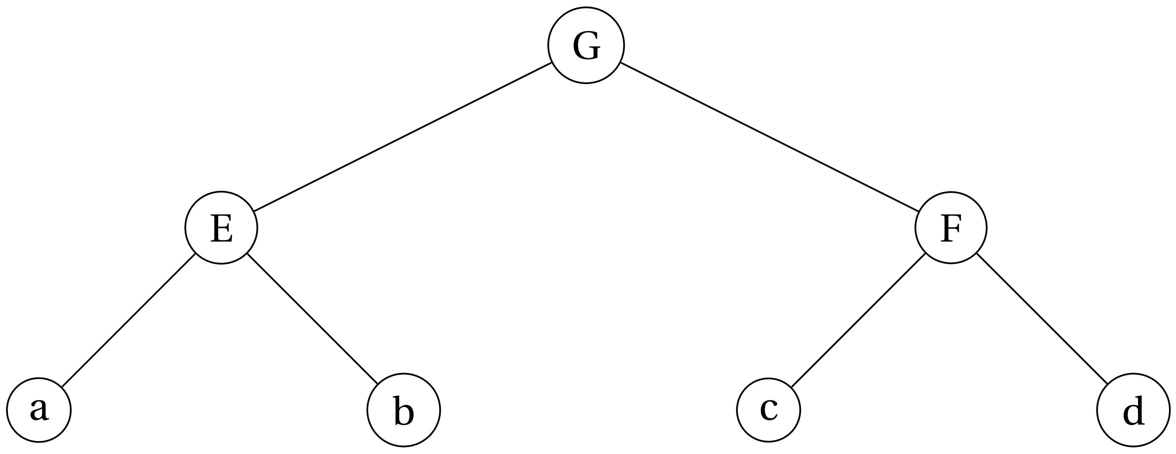


Figure 2.5: Generalized encoding for $a+2b+3c+3d \leq 4$

children is equal to w then a_w must be propagated to true. The notation P_i is used to refer to the literal that represents the subsum with value i in node P. For any node P with children L and R the following clauses are added:

$$\forall l_{w1} \in L.values, \forall r_{w2} \in R.values, w_3 = w_1 + w_2, p_{w3} \in P.values : \overline{l_{w1}} \vee \overline{r_{w2}} \vee p_{w3} \tag{2.6}$$

$$\forall l_{w1} \in L.values, p_{w1} \in P.values : \overline{l_{w1}} \vee p_{w1} \tag{2.7}$$

$$\forall r_{w1} \in R.values, p_{w1} \in P.values : \overline{r_{w1}} \vee p_{w1} \tag{2.8}$$

The first clause makes sure that when one child sums to w_l and the other node to w_r then the parent node should set the node with weight $w_l + w_r$ to True. The other two equations are needed for when one of the children sums to 0 and the other to w. In that case the parent should set the sum node with weight w.

In order to enforce the threshold v of the constraint, the following clauses are added for the root node.

$$\forall s_w \in S_1.values \wedge w > v : \overline{s_w} \tag{2.9}$$

For cardinality constraints it is enough to only add $\overline{s_{v+1}}$.

Furthermore, it is possible to optimize the encoding a little bit by reducing the number of clauses generated by equation 2.6. Values in L and R that are larger than the threshold v can be skipped for equation 2.6. This is called k-optimization.

As an example, we will look at the encoding for the following constraint: $a+2b+3c+3d \leq 4$. The tree for this constraint is shown in figure 2.5. The tree introduces three extra nodes E, F and G. The following clauses must be added:

$$
\begin{array}{rcc}
 & \overline{E_1} \lor G_1 & \\
\overline{a} \lor E_1 & \overline{E_2} \lor G_2 & \\
\overline{b} \lor E_2 & \overline{E_3} \lor G_3 & \\
\overline{a} \lor \overline{b} \lor E_3 & \overline{F_3} \lor G_3 & \overline{G_5} \\
 & \overline{F_3} \lor G_6 & \overline{G_6} \\
\overline{c} \lor F_3 & \overline{E_1} \lor \overline{F_3} \lor G_4 & \\
\overline{d} \lor F_3 & \overline{E_2} \lor \overline{F_3} \lor G_5 & \\
\overline{c} \lor \overline{d} \lor F_6 & \overline{E_3} \lor \overline{F_3} \lor G_6 &
\end{array}
$$

The encoding performs well when the number of distinct weights is low [35]. In the best-case all weights are equal and in that case the encoding would generate $O(n \log_2 n)$ auxiliary variables and it would generate $O(n^2)$ clauses. If k-optimization is applied only $O(nk)$ auxiliary variables are needed. In the worst-case all weights are unique. For example, if the weights for $l_0..l_n$ are $2^0..2^n$ then all possible combinations of weights will be unique and an exponential number of auxiliary variables must be added.

2.6. Propagation

A drawback of converting the constraint to a SAT encoding is that it adds a lot of extra literals and clauses. Some of these constraints are rarely violated by the solver and therefore converting them to a SAT encoding is not always an efficient solution.

Alternatively, one can extend the SAT solver with a propagator to handle these types of constraints directly [52]. In this case the propagation step of the solver is extended. Besides normal unit propagation on the clauses it should now also do propagation and conflict detection using the pseudo-boolean constraint. Furthermore, the conflict analyses step should be modified such that it can also construct the implication graph when literals are propagated using a pseudo-boolean constraint.

Propagation for the constraint $\sum w_i l_i \leq v$ can be done as follows. First it computes the sum s of the current assignment, thus $s = \sum w_i l_i$. Unassigned literals are not added to the sum. Next it computes how much slack it has with regard to v, $slack = v - s$. Finally, is sets all unassigned literals l_i with $w_i > slack$ to False.

When the conflict analyser must know which literals implied $\overline{l_i}$, the propagator can return all the literals that were set to True when it set l_i to False.

So suppose we have the constraint $a + 2b + 3c + 3d \leq 4$ and both a and b are set to True. The pseudo-boolean propagator can now be used to set c and d to False. When it has to provide an explanation for setting c to False, it returns the clause $\overline{a} \lor \overline{b} \lor \overline{c}$. This clause encodes the constraint $a \land b \implies \overline{c}$. Before setting c to False, this clause would have been a unit clause a could therefore have been used by the unit propagator to set $\overline{c}$. However, since the clauses is not explicitly added to the problem the unit propagator is not able to do this and therefore this was done by the pseudo-boolean propagator.

Finally, the propagator must be able to detect a conflict. A conflict occurs when $s > v$. Again, it must be able to generate a clause explaining the conflict. So suppose we have again the constraint $a+2b+3c+3d \leq 4$ and both c and d are set to True. There is now a conflict and the clause that explains the conflict is $\overline{c} \lor \overline{d}$.

The propagator will never introduce new auxiliary variables. During the conflict analysis it has to provide clauses, which will become part of a learned clause. In the best-case scenario, the constraint is never involved in a conflict and no clause is ever generated by the propagator. However, in the worst-case, the solver tries all possible violations of the constraint and learns for each combination that it is not allowed. There are $\binom{n}{k}$ combinations, which are much more than the number of clauses that will be added by the GTE.

2.7. MAX-SAT problem

The maximum satisfiability problem is an optimization variation on the SAT problem. The goal is now no longer to find a satisfying solution, but to find a solution that satisfies as most clauses as possible. In the best-case scenario it finds a solution that satisfies all clauses, however for most MAX-SAT problems such a solution does not exists.

There are two variations to this problem. With weighted MAX-SAT there is a weight associated with each clause. The penalty of a solution is the sum of the weights of the unsatisfied clauses. The goal is now to minimize the penalty.

The second variation is partial MAX-SAT. For this problem the constraints are divided in hard and soft constraints. The hard constraints must be satisfied. If they cannot be satisfied, then the problem is unsatisfiable. The soft constraints do not have to be satisfied. The goal is to find a solution that satisfies all hard constraints and as most soft constraints as possible. The partial MAX-SAT problem can also be weighted by giving each soft constraint a weight.

A MAX-SAT problem can be optimized by solving multiple SAT problems. Each iteration a SAT problem is constructed that specifies that the penalty should be smaller or equal to k. When a solution is found, then the penalty for the found solution is an upper bound for the optimal solution. If the SAT problem is unsatisfiable then k is a lower bound for the optimal solution. This method is often referred to as linear search or iterative SAT solving.

The SAT problem can be constructed by adding to each soft clause c_i an auxiliary variable p_i. By doing this, p_i can only be set to False when c_i is satisfied. If it is not possible to satisfy c_i, then p_i is set to True and therefore it still possible to find a solution for the SAT problem. The maximum allowed penalty is specified with the help of the pseudo-boolean constraint in equation 2.10.

$$\sum w_i p_i \leq k \tag{2.10}$$

This SAT problem is only satisfiable when there exists a solution with a penalty smaller or equal to k.

The optimal solution can be found with algorithm 4.

Algorithm 4: Linear search algorithm for finding the optimal MAX-SAT solution.

```
1  function MAX-SAT ($HC, SC$);
   Input: $HC$ the hard clauses of the problem. $SC$ the soft clause of the problem
2  $Upper\_bound = \sum_{s \in S} weight(s)$
3  $Lower\_bound = 0$
4  while $Upper\_bound > Lower\_bound$ do
5      Choose a $k$ such that $Lower\_bound \leq k < Upper\_bound$
6      Let $S$ be the SAT problem where the maximum penalty is $k$
7      Solve $S$
8      if $S$ is satisfiable then
9          $Upper\_bound = k$
10     else
11         $Lower\_bound = k + 1$
12     end
13 end
14 return the last found solution.
```

Algorithm 4 constructs in each iteration a new SAT problem. When using a CDCL solver this is not very efficient since you have to relearn certain clauses each iteration. However, it is possible to implement the algorithm in such way that the solver can be reused. By reusing the solver, the learned classes and the activity heuristics are kept. This should improve the performance.

There are two ways how a solver can be reused. When a satisfying solution is found we can always add more constraints to make the problem more difficult. All clauses that were learned on the easier problem should still hold for the more difficult problem. Thus, if it finds a solution with penalty k, it is allowed to add a constraint that sets the maximum penalty to a value smaller than k. This can be done until the problem becomes unsatisfiable. When the problem becomes unsatisfiable the previous solution was the best solution. With this method the upper-bound is lowered each iteration.

Reusing the solver when increasing the lower-bound l is a bit more difficult. The lower-bound can be tested by adding the constraint that specifies that the penalty should be at most l. The solver either finds a solution with the optimal penalty, or it finds that no such solution exists and can increase the lower-bound. Removing the optimization constraint will lead to some learned clauses that are no longer implied by the problem and therefore the solver can not be reused. There are ways around this problem, however they are more complicated than the upper-bound approach.

With the GTE this problem can be solved with the help of assumptions. Instead of setting the maximum of the constraint to l we set it to the upper-bound. Next, we reduce the maximum of the constraint by setting assumptions. We let the solver assume that all sum literals larger than l are set to False. This can be seen as some forced decision variables and the solver is not allowed to backtrack past these variables. Only when it finds that no solution exists, it is allowed to undo the assumptions. When the solver has to provide an explanation for a conflict these assumptions are part of the reason set in the implication graph and therefore, they can become part of learned clauses. When the assumptions are undone the learned clauses are still valid and therefore it is possible to reuse the solver.

3 Related research

Different methods for encoding cardinality and pseudo-boolean constraints have been proposed. These methods can be categorized into three different categories [26], Binary Decision Diagrams (BDD), Sorting networks and Adders.

Asín et al. [8] introduces Half Sorting and Half Merging networks, which are similar to sorting networks based on odd-even merges [14, 26] but need less clauses. Furthermore, they reduce the size of the encoding by using the observation that for most cardinality constraint the maximum allowed value k is much smaller than the number of literals in the constraint.

Codish and Zazon-Ivry [18] used a pairwise sorting network [53] to construct the encoding for a cardinality constraint. They show that the pairwise network has better propagation properties than the networks based on the odd-even sorting networks.

A generalization of the 2-Odd-Even [14] selection network was proposed by Karpiński and Piotrów [37, 38]. They have shown that a 4-Odd-Even network needs a smaller encoding than the 2-Odd-Even network and that they are competitive with other state-of-the art encodings.

Eén and Sörensson [26] decomposes the encoding for a pb-constraint into a number of sorting networks. These networks construct a number in a mixed radix base, where each network represents one digit. The choice of good radix bases is further improved by Codish et al. [19, 29].

Bailleux et al. [13] uses a binary representation for the weights to avoid an exponential blow-up of the encoding. They used multiple totalisers to sum the digits. Paxian et al. [54] optimizes the method further and shows how this encoding can be reused for different bounds when optimizing a weighted MAX-SAT problem.

Ogawa et al. [50] reduced the size of the totaliser by applying a similar idea as mixed radix encoding. Later this technique was applied to the generalized totaliser by Zha et al. [61, 62].

Instead of an encoding based on a network it is also possible to use a Binary Decision Diagram (BDD) to create an encoding. An advantage of the BDD methods is that they do not depend on the size of the weights, which make them very useful for pb-constraints. However, there are some pb-constraints for which no polynomial BDD exists [3, 33]. Abío et al. [3] has shown how a Reduced Ordered BDD (ROBDD) constraint can be constructed for pb-constraints. Later on, they extended the algorithm to also work with linear integer constraints [2]. Sakia et al. [56] extended the algorithm for constructing ROBDD to also work on constraints in band form. This are constraints that specifies both the minimum and maximum allowed value.

Abío et al. [5] introduced an encoding for linear constraints that utilized implication chains. Using these chains, a Multi Decision Diagrams (MDD) is constructed for the constraints. With the help of implication chains, they were able to generate smaller encodings with stronger propagation properties.

Martins et al. [45] proposed several techniques how a cardinality constraint for MAX-SAT can be modified so that the solver can be reused. They applied these techniques to the totaliser and show how it can be used to incrementally increase the lower-bound for the problem. Karpiński and Piotrów [39] proposed a technique to improve the reusability of the pb-encoding of an optimization constraint. They used a mixed-radix encoding but added a constant to both sides of the constraint such that the representation of the right side of the constraint contains only one non-zero digit and therefore needs only one constraint to specify the limit. With the help of assumptions, they were able change the bound without adding new clauses or variables.

Paxian et al. [55] introduced two prepossessing techniques to improve the weighted partial MAX-SAT problem. The first technique they introduced is Generalized Boolean Multilevel Optimization (GBMO), where the problem is divided into simpler sub-problems based on the weights. They start by solving the sub-problem with the largest weights. The second technique they added to their solver searches for unsatisfiable soft clauses and removes them from the problem. These techniques make the problems less complex and therefore easier to solve.

A drawback of CDCL is that the method of reasoning is quite weak. Another approach for finding a solution for pb-constraints is to use a pb-solver based on cutting planes [22, 30]. From a theoretical point, cutting planes are more powerful since they can utilize stronger resolution. However, in practice CDCL can be implemented much faster and therefore CDCL based solvers often outperform solvers based on cutting planes. Chai and Kuehlmann [16] show how the conflict driven methods can be adapted to generate cutting planes proofs. Elffers and Nordström [27] improved the method of Chai and Kuehlmann [16] by using the division rule [22] instead of the saturation rule.

A drawback of these cutting planes is that they perform poorly when the input is given in conjunctive normal form (CNF). Thus when the problem has already converted all pb-constraints to a SAT encoding. Elffers et al. [28] tried to mitigate this by recovering the pb-constraints from the CNF encoding during the search.

An important aspect for a SAT solver is the decision heuristic that selects the variables and decides on its value. Nadel [48] improved this step for MAX-SAT problems by treating the target and non-target variables different. For non-target variables the decision process uses values that worked in the previous solution and for the target variables it uses an optimistic approach and sets them to the value that would not increase the penalty. Furthermore, he proposed a method to solve MAX-SAT problems by reducing it to an Optimization Modulo Bit-Vectors problem [47, 48]. With that method it is possible to reuse the solver after changing which constraints are hard and which are soft. He shows that this is useful for Boolean Multilevel Optimization problems, which is a generalization of MAX-SAT where there are multiple objective functions.

The restart policy of a solver has also effect on its performance. A commonly used restart policy is Luby [44] which restarts after a fixed amount of time. The time till the next restart changes after each restart. More recent approaches try to utilize a heuristic to determine the optimal restart moment. The authors of the GLUCOSE solver [51] have shown that a restart policy based on the LBD score of clauses performs good. They restart the solver when the LBD score of the recent n learned clauses is a factor higher than the average LBD of all clauses. Haim and Walsh [32] uses a machine-learning algorithm to select between different restart strategies. Liang et al. [43] uses machine learning to predict the quality of the next clause that will be learned and restarts the solver if the predicted quality is below a threshold.

Oh [51] has analysed the difference between what solver configuration works well for SAT and for UNSAT instances. They proposed a hybrid method that exploits the differences between SAT and UNSAT and by doing so they were able to improve the solver for both instances.

Audemard et al. [9] introduces a solver that uses a restriction of extended resolution. They detect when extended resolution can be applied to consecutive learned clauses. If the solver has learned two clauses $\overline{l_1} \vee a$ and $\overline{l_2} \vee a$, then it introduces the variable $z \iff l_1 \vee l_2$ and replaces the learned clauses with the clauses $\overline{z} \vee a$ and $\overline{l_1} \vee \overline{l_2} \vee z$. The variable z will also be used the replace $l_1 \vee l_2$ in new clauses.

3.1. Encode or propagate

The methods that are closest to the thesis are the methods proposed by Abío et al. [1, 4]. Stuckey and Abío [1] introduced a method that combined the encoder and propagator. Constraints would initially be handled by the propagator and overtime certain parts of the constraint would be encoded. For the cardinality constraints they relied on a sorting network that was build using 2-comparators. The comparator $2comp(x_1, x_2, y_1, y_2)$ has two input (x_i) and two output (y_i) literals. The outputs are defined as follows: $y_1 = x_1 \vee x_2$ and $y_2 = x_1 \wedge x_2$. A resulting property of this is that $x_1 + x_2 = y_1 + y_2$. Initially the constraint $x_1 + x_2 + x_3 + .. + x_n \leq k$ is fully handled by the propagator. At some point it will add the encoding for $2comp(x_1, x_2, y_1, y_2)$ to the SAT problem and then it replaces the cardinality constraint with $y_1 + y_2 + x_3 + .. + x_n \leq k$. It continues doing that until the full encoding is added. When the full encoding is added the propagator is no longer needed.

For each literal in the constraint, they keep track of the activity. The activity is increased each time that the literal was involved in a conflict. Each time the solver restarts it checks the activities. If the activity of x_i exceeds a threshold, then there are three possibilities:

- x_i is not the input to a 2-comparator. This means that it is an output of the full cardinality encoding. In this case nothing is done.
- x_i is the input for the comparator $2comp(x_i, x_j, y_k, y_l)$ and x_j already exists. This means that x_j is already generated by the decomposition of one of the other literals or is part of the original constraint. In this case the comparator is encoded and both x_i and x_j are replaced by y_k and y_l in the constraint.
- x_i is the input for the comparator $2comp(x_i, x_j, y_k, y_l)$ and x_j does not yet exist. In this case the method selects all literals in the current constraint that can result into x_j after applying one or more decomposition steps. A decomposition step is applied to all comparators that have both inputs in the selected literals. Each time this is done the encoding becomes closer to the point where x_j must added to the encoding.

In order to focus on the recent conflicts, the activity is halved after each restart.

For pb-constraints they use the BDD of the constraint. It starts by generating the BDD for the constraint, but they do not yet encode it. This method keeps an activity per constraint and when the activity exceeds a threshold, it will encode the bottom layer of the BDD. After encoding a layer, the activity is set to 0. Again, it halves the activity during restarts. They have shown that these methods lead to a smaller encoding and are able to perform close to or better than the best of propagation or encoding.

Abío et al. [4] show that a similar result can be achieved by adding the full encodings instead of adding partial encodings. However, in both papers they only used encodings that were determined at the start of the solve process. It is unclear whether the performance of the decomposition can be improved when the encoding is determined during the solve process instead of at the start. This will be investigated further in this thesis.

4 Methods

This chapter discusses the methods that are introduced during the thesis. The first three approaches are variations on the methods described by Abío et al. [1, 4]. They use a different encoding scheme and decide on the ordering of the literals during the search. All methods are implemented as an extension to the pumpkin solver. Pumpkin is a solver that is currently being developed at the TU Delft by Emir Demirović. Some of the proposed methods rely on observations that will be done in the experimental section (chapter 6).

4.1. Incremental

Adding the encoding for a constraint will add a large number of extra literals and clauses. However, as will be shown in section 6.1, not all literals are heavily used by the solver. Furthermore, changing the order of the literals can help the solver finding its first couple of solutions, this will be discussed in section 6.2. The incremental method is trying to utilize those observations by determining during the search which literals should be encoded and in what order.

The encoding is incrementally constructed during the search. The idea behind the incremental approach is to add the literals to the bottom layer of the tree. If the bottom layer is full, then the size of the tree is increased by introducing a new root node. The encoding of the tree is similar to the generalized totaliser. Thus the leaf nodes are the literals from the constraint and all other nodes encode the sum of the literals in its subtree. The root node encodes the sum of all the literals that are added to the tree. Until all literals are added to the encoding a propagator is used to enforce the constraint. However as more literals are added to the encoding, more conflicts will be detected by the encoding instead of the propagator. We will first discuss how nodes are added to the tree. Next, we will discuss the SAT clauses that must be added after adding a node. Finally, we will discuss the criteria for adding a literal to the encoding.

4.1.1. Tree construction

Initially there is no tree, for the first literal a leaf node is constructed and that leaf node becomes the root of the tree. For any other literal there are two possibilities; There is either a node with an unassigned right child, or the tree is full.

If the tree is full then create a new root node and assign the current tree as its left child. Next use algorithm 5 to add the newly added literal at the same depth as the other leaf nodes. L is the node that represents the literal that must be added and R is the root. d is the height of the tree.

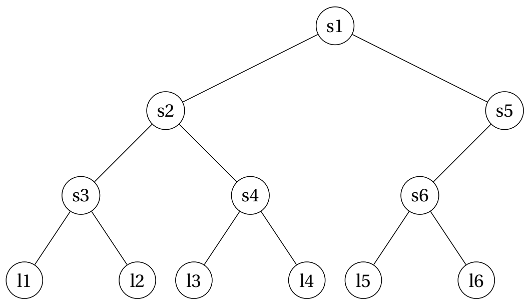
If the tree is not full, then find the lowest node without a right node and use algorithm 5 to add the leaf to the tree. R is now the lowest node without a right node. Figure 4.1 shows how the tree is grown.

4.1.2. Update encoding

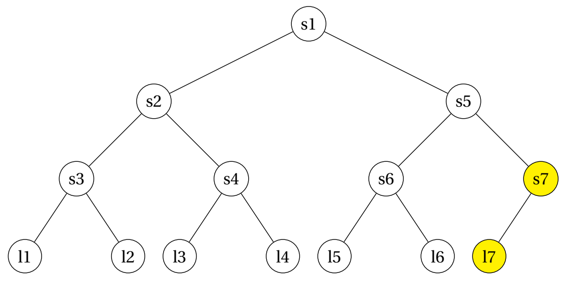
After the tree is updated the SAT encoding must be updated. Updating is done by starting at the newly added leaf node and propagating the newly added values to the parents until the root node is reached. The algorithm is described in algorithm 6. Initially N contains only the newly added literal L, P is the parent of L and C is the other child of P or empty if L is the only child of P.

Algorithm 5: Add a leaf at the bottom of the tree.

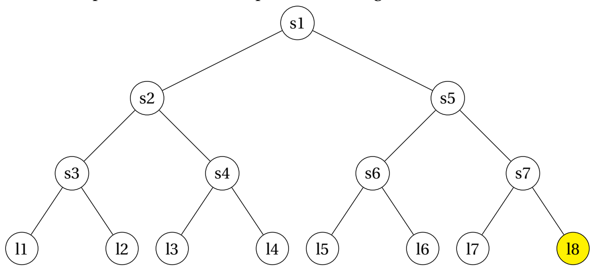
```
function Add leaf ($L, R, d$);
Input: $L$ node to add. $R$ node to which $L$ must be added as descendent. $d$ the depth of the other
       leaves. $c$ depth of $R$.
if $c+1=d$ then
    Add L as right child to $R$.
else
    Create a node $N$ and make this the right child of $R$
    $R \leftarrow N$
    $c \leftarrow c+1$
    while $c+1<d$ do
        Create a node $N$ and make this the left child of $R$
        $R \leftarrow N$
        $c \leftarrow c+1$
    end
    Add L as left child to $R$
end
```



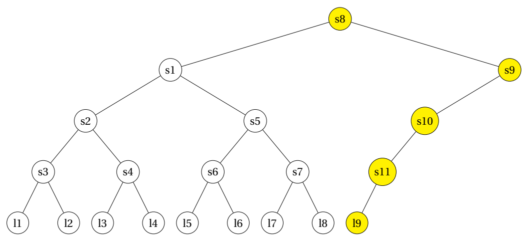
(a) At some point the tree for the partial encoding



(b) The tree after adding l7



(c) The tree after adding l8



(d) The tree after adding l9

Figure 4.1: The construction of the encoding, yellow are the newly added nodes.

4.1.3. Encoding criteria

In order to decide when a literal must be added to the encoding, the propagator keeps track of how often each constraint and literal is used during conflict analyses. The count of a constraint is increased when it causes the conflict and when it has to give an explanation for a propagated value. For the literals the count is increased if the constraint caused the conflict and the literal was set to true. Furthermore, the count of a literal is also increased when it was used in an explanation made by the constraint; either as cause or as propagated literal. If the count exceeds a certain threshold the literal or constraint will be scheduled for encoding. During a restart the solver encodes the scheduled literals. The reason for doing this during a restart is that it is difficult to add the encoding during the search. SAT solvers tend to restart often and therefore it is not a problem to only add the encoding during restarts.

We considered two different criteria, we will refer to them as the Incremental and Dynamic method. The Dynamic method adds the full encoding of the constraint when its criterion becomes true. Its criterion is

Algorithm 6: Update the encoding of the tree after adding a new node to the tree.

```
1  function Update encoding ($N, C, P$);
   Input: $N$ a list of new literals in the updated child. Let $N_i$ be the literal that encodes the value $i$. $P$ the
          parent of the updated child. $C$ the child of $P$ that was not updated. $C$ is empty if $P$ has only one
          child.
2  Let $U$ be an empty list that will keep track of the newly added values.
3  foreach $v \in N$ do
4      if $p_v \notin P$ then
5          add $p_v$ to $P$ and $U$
6      end
7      Add the clause $\overline{N_v} \vee p_v$
8      foreach $w \in C$ do
9          $x \leftarrow w + v$
10         if $p_x \notin P$ then
11             add $p_x$ to $P$ and $U$
12         end
13         Add the clause $\overline{N_v} \vee \overline{C_w} \vee p_x$
14     end
15 end
16 if $P$ has parent and $U$ is not empty then
17     $P_2 \leftarrow$ parent of $P$
18     $C \leftarrow$ the child of $P_2$ that is not $P$ or empty.
19     Update_encoding($U$, $C$, $P_2$)
20 end
```

defined as follows:

$$nr\_lits \cdot maximum\_of\_constraint \cdot d < constraint\_count \tag{4.1}$$

d is a parameter that was set to different values and we found that a value of 10 worked well. The appendix shows results for different values of d. $constraint\_count$ is the count for the constraint. The size of the full encoding depends mostly on the number of literals and the maximum value of the constraint. Thus the encoding is added when the number of generated conflict clauses exceeds a certain fraction of the number of clauses in the full encoding. A constraint that exceeds the threshold is likely to generate a lot more conflicts and therefore it is better to add the encoding.

The incremental method adds only some of the literals to the encoding. A literal is added to the encoding if the following criterion is met:

$$weight \cdot maximum\_of\_constraint \cdot d < literal\_count \tag{4.2}$$

Again, d is a parameter and the results for different values can be found in the appendix. This criterium worked well with a value of 0.5 for d. Literals with a large weight are more likely to cause a conflict. By adding the weight of the literal to the criterium, they are more likely to be encoded.

The dynamic method adds all literals sorted on weight and therefore should be similar to Abío et al. [4]. With the incremental method the literals are ordered on the moment they triggered the encoding criterium. This should place active literals closer to each other. When multiple literals are encoded at the same time, they are ordered on their activity.

4.2. Top-down

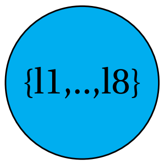
A problem with the previous method is that for optimization problems there are a lot of literals that initially behave in the same way, thus participating in the same conflicts. All these literals are added at the same time to the encoding and since they behaved the same there is no way of deciding on their order based on their history. So when they are added, their ordering is a bit arbitrary and does often not improve the encoding.

A method that might solve this problem is the top-down method. The idea behind this method is that we start with a propagator for the root node. At some point it is decided that the root node must be encoded.

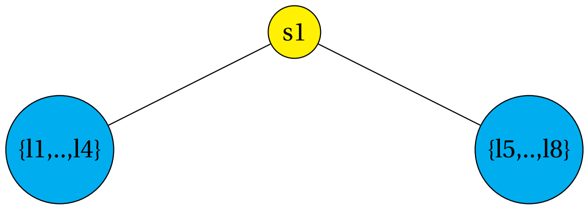
Now the literals are divided into two groups and each group is added as a child node to the root node. These child nodes can now be handled by a special propagator that propagates the sum of the input literals to a set of output literals. These two sets of output literals are summed in the root node with the help of the GTE. The propagator for the root node is no longer needed. Later the child nodes can be split and encoded as well. The idea is that the ordering is slowly constructed by grouping literals in smaller and smaller groups. The final position of a literal is decided much later than with de incremental method. Figure 4.2 shows how the tree is grown and the pseudo-code is shown in algorithm 7.

For this method we have to create a new propagator. The propagator has a set of weighted input literals, which are a subset of the literals of the pseudo-boolean constraint. Furthermore, is has a set of weighted output literals that represents the possible sums of the input literals. The propagator should simulate the part of the tree that is not yet encoded. To achieve this the propagator should propagate true values of the input to the output literals and false values of the output to the input.

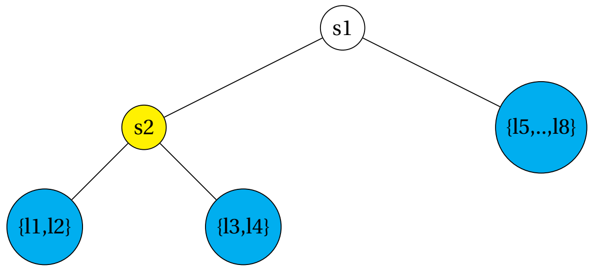
This means that if the current sum of the input literals is s, then all output literals with $weight \leq s$ should be set to true. Further let m be the smallest weight of an output literal that is set to False. The propagator should set all unassigned input literals to False with $weight \geq m - s$. Note that the propagator is slightly stricter than the full GT encoding. However, anything that is learned with the stricter propagation holds also for the GT encoding and therefore this is not a problem.



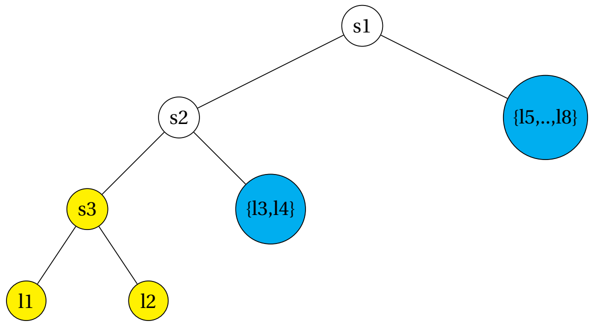
(a) Start with the root propagator



(b) Create the children of the root and replace the root propagator with an encoded node.



(c) Create the children of $s2$ and replace the propagator with an encoded node.



(d) Create the children of $s3$. We have reached the bottom of the tree.

Figure 4.2: Construction of the encoding using the propagator for the subtrees. Blue nodes are propagators. Yellow nodes are newly encoded nodes.

4.2.1. Encoding criterium

The top-down method keeps for each propagator track of how often it provided an explanation during the conflict analysis. When the count exceeds a threshold, then the propagator is replaced by an encoded node. The threshold is defined as follows:

$$min(maximum\_of\_constraint, nr\_literals) \cdot d < propagator\_count \tag{4.3}$$

The size of an encoded node depends on the possible sums of the literals in the node. The number of possible sums is limited by the number of literals and the maximum allowed value. d is a parameter that controls how quick the encoding should be added.

4.2.2. Grouping

When a propagator is encoded, its literals are divided over two groups and each group is assigned to a new propagator. The grouping is done by sorting the literals first on weight and then on activity. Thus literals with the same weight and a similar activity will be close to each other. Next it will search for the largest difference

Algorithm 7: Split a node into two child nodes.

```
1  function Split node ($I, O$);
   Input: $I$ the set of literals that are summed by the node. $O$ the sum literals that store the sum of $I$ as a
          unary number.
2  if $|I| \leq 1$ then
3      return
4  end
5  Split $I$ into the two groups $I_1$ and $I_2$.
6  For each input group, create the possible sum literals $O_1$ and $O_2$
7  Create the constraints that propagates the sum of $I_1$ to $O_1$ and add it to the problem
8  Create the constraints that propagates the sum of $I_2$ to $O_2$ and add it to the problem
9  Create a node $N$ and make this the right child of $R$
10 foreach $l_w \in O_1$ do
11     Add clause $\overline{l_w} \vee O_w$
12 end
13 foreach $l_w \in O_2$ do
14     Add clause $\overline{l_w} \vee O_w$
15 end
16 foreach $l_w \in O_1$ do
17     foreach $l_x \in O_2$ do
18         Add clause $\overline{l_w} \vee \overline{l_x} \vee O_{w+x}$
19     end
20 end
```

in activity between two consecutive literals in the sorted list. In order to ensure a somewhat balanced tree, it will only search between the literals where the index in the sorted list is between 0.4 and 0.6 times the size of the list. After finding the two literals with the largest difference, it will split the list between these two literals. The resulting groups will be used to construct the next propagators.

4.3. Bottom layers

In the experiments of section 6.3, the literals of the encoding that are used in the explanations are analysed. These results show that the literals of the bottom layers are the literals that are mostly used in the explanations. In order to exploit this, two methods were tried.

4.3.1. Full Bottom Layers

With this method the full encoding for the bottom layers is added at the start. This means that the encoding is a set of disjoint trees, where the literals of the constraint are divided over the trees. Let the value of a node be the weight of the highest sum literal that is set to true. The sum of all root nodes equals the total number of literals set to true. So a propagator is used to sum all the root nodes. The structure of the method is shown in 4.3. Let s be the sum of all root nodes. There is a conflict when s is larger than v, where v is the maximum value of the constraint. Furthermore, the propagator propagates the maximum allowed subsum to the trees. This is done by first computing the $slack = v - s$. Next for each node i with value s_i all sum literals with a weight larger than $s_i + slack$ must be set to False.

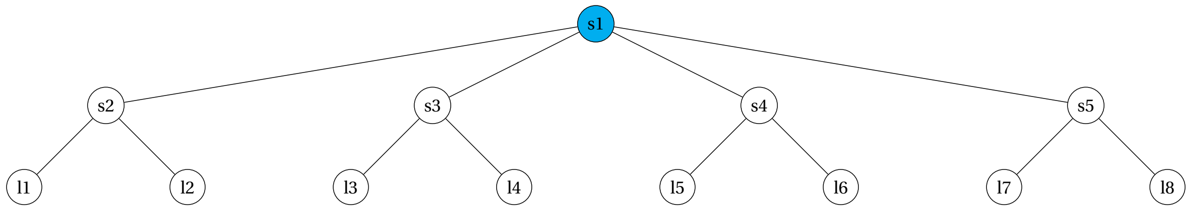


Figure 4.3: Structure of the fbl method. There are a number of subtrees that add two literals. A propagator is used to combine all the subtrees an enforce the constraint. The blue node is a propagator.

4.3.2. Pairs

The experiments in section 6.3 show that most conflicts contain the original constraint literals. However, it is preferable that the literals from the higher layers are used. In order to increase the chance of this happening, we will pair the literals with multiple other literals. Since each literal can appear in more pairs there is a higher chance that a suitable literal from layer 1 can be used in a conflict.

The idea of this method is to introduce auxiliary variables for literals that often appear together. When an explanation for a conflict must be provided these pairs are replaced by the auxiliary literals. The propagator is used to enforce the constraint and to do propagation. By introducing these literals, the explanations will use literals that represents subsums instead of the original literals. Since literals can appear in multiple pairs it is more likely that an explanation can use these auxiliary literals. This should lead to smaller explanations.

Algorithm 8 shows how the explanation is modified using the auxiliary variable for pairs and how it is decided which pairs should be added to the problem. Pairs are only added during a restart, the solver keeps track of the pairs that will be added with the next restart.

When a new pair is added to the solver, the solver will create an auxiliary variable s and a clause that forces s to true if the members of the pair are both true. Let p_1 and p_2 be the literals from the pair, the clause $\overline{p_1} \vee \overline{p_2} \vee s$ will force s to True if p_1 and p_2 are True.

Algorithm 8: In an explanation replace pairs of literals with auxiliary variables.

```
1  function Replace pairs $(L, D, S)$;
   Input: $L$ a subset of the input literals of the constraint. It is either the set of literals that caused the
          propagation of another literal, or the set of literals that violated the constraint. $D$ a dictionary
          that maps two input literals to an auxiliary literal. $S$ list of pairs that are not yet added, but will
          be with the next restart.
2  $O$ is the output set, initially empty.
   /* Replace the pairs                                              */
3  while $L$ has a pair of literals that are in $D$ do
4      Find a pair of literals $l_1$ and $l_2$ in $L$ that are also in $D$.
5      Let $a$ be the literal in $D$ the belongs to the pair $l_1$ and $l_2$.
6      Add $a$ to $O$ and remove $l_1$ and $l_2$ from $L$.
7  end
8  Add all remaining literals in $L$ to $O$.
   /* Ignore the pairs that are already scheduled                    */
9  while $L$ has a pair of literals that are in $S$ do
10     Find a pair of literals $l_1$ and $l_2$ in $L$ that are also in $S$.
11     Remove $l_1$ and $l_2$ from $L$.
12 end
   /* Schedule new pairs                                             */
13 Create pairs from the remaining literals in $L$ and add them to S.
14 return $O$
```

4.4. Overview of the methods

We will now give a brief summary of the methods and highlight the differences between them. This is also shown in table 4.1. The **encoder** and **propagator** refer to the methods that were discussed in the background. The encoder adds the full GTE at the start of the solve and the propagator will never add the encoding and will always use a propagator. The **dynamic** method will add during the search the full encoding for some of the constraints. Both the **incremental** and **top-down** methods will add a partial encoding for some of the constraints during the search. The encodings can be updated and it is possible that at some point the full encoding is added. These methods differ in their construction of the partial encoding. The **pairs** and **full bottom layers** method will also add a partial encoding, however these methods will never add the full encoding. The full bottom layers encodes at the start of the solve for all constraint the bottom layer. It will not extend the encoding during the search. The pairs method constructs pairs during the search and literals can be part of multiple pairs.

Method	**Encodes during the search**	**Can add the full encoding**	**Adds a partial encoding**
Encoder		X	
Propagator			
Dynamic	X	X	
Incremental	X	X	X
Top-down	X	X	X
Pairs	X		X
Full Bottom Layers			X

Table 4.1: Overview of the different methods

5 Benchmarks

During the thesis the solvers have been tested on three different types of benchmarks. The benchmarks are chosen to be divers and are expected to favour different methods. The MAX-SAT problems have one large constraint that must be optimized. Pseudo-boolean benchmarks have a lot of small constraints and the CTT is a real-world application that contains both types of constraints. For each benchmark we will discuss how it was obtained and how they were converted to a (MAX-) SAT problem.

5.1. MAX-SAT problems

The first set of problems are all MAX-SAT problems. They are obtained from the MaxSAT Evaluation 2020 competition [1] [11]. The problems from both the weighted (w) and unweighted (uw) incomplete track were used. The weighted set is both tested as unweighted and weighted. In the results MAX-SAT w=1 refers to the tests where the weighted problems were tested as if they were unweighted by setting all weights to 1.

For both sets there were a few problems that were too large to test. This was due to the limitation on storage on the StarExec server. Problems from the unweighted track that were larger than 300 mb were removed and for the weighted track problems larger than 70 mb were removed. The final set of unweighted problems contained 241 test instances and for weighted it contains 204 instances. The problems are specified by the WDIMACS format. These are SAT problems with one large cardinality/pseudo-boolean constraint that must be optimized. The expectation is that encoding the constraint is necessary to find a good solution. Therefore, this test set will favour the methods that add the GTE to the problem.

5.1.1. Reducing number of distinct weights

The size of the encoding depends on the number of distinct weights in the constraint. The weighted problems contained too many different weights to be solvable by encoding the full constraint. The number of weights is reduced by the techniques that are described by Joshi et al.[36]. The algorithm starts by sorting the weights and then it computes the difference between two consecutive weights. Next it groups the weights into n clusters by setting the boundaries between the $n-1$ weights with the largest differences. For each cluster a replacement weight is computed by fist taking the average and then dividing it by the smallest average of any of the clusters. Finally for each literal in a cluster its weight is replaced with the replacement weight of the cluster. For the tests 5 clusters were used. These tests are referred with MAX-SAT w=5.

5.2. Pseudo boolean problems

The pseudo-boolean instances were collected from the pseudo-boolean competition of 2010, 2015 and 2016[2]. From each competition the problems from the small int decision track were used. These are problems where all the weights are small integers and where the goal is to find a solution or to proof unsatisfiability. In the results pb10 and pb15 are used to refer to the instances of the 2010 and 2015 competition. The set from 2010 contains 137 instances and the set from 2015 contains 352 instances. From the 2016 benchmarks two sets

[1]Benchmarks can be found at `https://maxsat-evaluations.github.io/2020/`

[2]The can all be downloaded from `http://www.cril.univ-artois.fr/PB16/`

were used, Elffers and d_n_k, they are listed separately in the results. Elffers consists of 293 instances and d_n_k has 234 instances.

The problems were specified using the opb format. Each line in a problems instance specifies a pseudo-boolean constraint. The literals in the constraints can have positive and negative weights. However negative weights will make propagation and encoding difficult and therefore they had to be converted to positive weights. A negative weight can be converted to a positive weight by replacing the literal with its complement and adding the weight to the right-hand side of the constraint. This is shown in equation 5.1.

$$-x_1 + x_2 \geq 0 \iff \overline{x_1} + x_2 \geq 1 \tag{5.1}$$

Furthermore, the constraints could have the ≤, ≥, = operators. However not all methods supported all operators and therefore they had to be converted to the ≤ operator. The equal sign could be replaced by two constraints one that specified that is should be less or equal and the other that it should be at least the specified value.

$$x_1 + x_2 = 1 \iff \begin{array}{l} x_1 + x_2 \leq 1 \\ x_1 + x_2 \geq 1 \end{array} \tag{5.2}$$

The greater than or equal operator can be replaced with the at most operator by first replacing all literals with their complement. Next the maximum value for the sum of complements must be computed. This can be done by taking the sum of all weights and subtracting the minimum value of the original constraint.

$$2x_1 + x_2 \geq 1 \iff 2\overline{x_1} + \overline{x_2} \leq 2 + 1 - 1 \tag{5.3}$$

Finally, some of the pseudo-boolean constraints are actually just normal clauses. For example, the constraint $x_1 + x_2 \geq 1$ specifies that at least one literal must be true. This is the same as the clause $x_1 \vee x_2$. After converting all constraints to at most constraints they can be detected as follows. Let k be the maximum allowed value, w the smallest weight in the constraint and let s be the sum of all weights. If $s - w \leq k$ and $s > k$ than the constraint can be replaced by a normal clause. The clause contains all negations of the literals in the constraint.

$$2x_1 + 3x_2 + 4x_3 \leq 7 \iff \overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \tag{5.4}$$

These problems contain a lot of pseudo-boolean constraints. It is expected that not all the constraints are difficult to satisfy and therefore they do not have to be encoded. Thus, it is expected that these problems will favour the methods that do not add the full encoding of the constraints.

5.3. Curriculum-based timetabling

The Curriculum-based timetabling (CTT) [25] problem is a course timetabling problem. The goal of the problem is to create a weekly schedule for a university. The problem consists of a set of lectures belonging to specific curricula. Each lecture must be scheduled to a specific room and time slot. The problem has both hard and soft constraints and the goal is to minimize the penalty of the soft constraints. Some of the constraints can be specified as a clause and others will be specified as a pseudo-boolean constraint. The optimization constraint will also be a pseudo-boolean constraint. It is expected that the optimization constraint must be encoded. However, for the other constraints it might be better to not encode them, therefore it is expected that methods that add the encoding during the search perform better. This set of benchmarks is much smaller than the other sets and only contains 21 different instances[3].

The next sections will discuss in more detail the problem specification and how it was encoded.

5.3.1. Concepts

The problem contains the following concepts. Each problem has a number of **days** D and a number of **hours** H in a day. Together they form the **time slots** T. Each time slot t is a combination of a day d and an hour h. Depending on the context the notation t or $\{d,h\}$ is used to refer a time slot. The number of hours in a day is equal for all days.

Each **room** has a capacity for a limited number of students. The capacity is specified per room. The set of rooms is specified by R.

[3]They can be downloaded from `http://www.cs.qub.ac.uk/itc2007/`

A **course** is given by a specific **teacher**. The set of courses is specified by C and the teachers are specified by TE. Each course is followed by a given number of students. For each course a given number of lectures must be scheduled. Each lecture must be scheduled to a room and time slot. For each course a minimum number of working days is specified. This minimum states over how many days the lectures must be scheduled. Furthermore, for each course there can be a number of time slots where the course cannot be scheduled. A teacher can teach multiple courses therefore a teacher te can be seen as a set of courses.

A **curriculum** specifies a group of courses that are followed by the same students. Lectures from courses in the same curriculum should be scheduled to different time slots. A course can belong to multiple curricula. CU denotes the set of curricula. Each curriculum cu is a set of courses.

The problem has both hard and soft constraints.

5.3.2. Hard constraints

- Lectures: All the lectures must be scheduled. It is not possible to schedule two lectures of the same course to the same time slot.
- Room: There can be no two lectures that are scheduled to the same room at the same time slot.
- Conflicts: Lectures from courses that belong to the same curriculum or that are given by the same teacher must be scheduled to different time slots.
- Unavailable: A lecture cannot be scheduled in a time slot that was specified as unavailable for the given course.

5.3.3. Soft constraints

- Capacity: A course can be scheduled in a room that has not enough capacity. However, there will be a penalty of 1 point for every student that does not fit in the room.
- Minimum days: For each course there is a minimum number of days that must be used to schedule the lectures. Each day below the minimum results in a penalty of 5 points. It is allowed to use more days than the minimum to schedule the lectures.
- Compactness: Lectures from the same curriculum should be scheduled close to each other. Each lecture from a curriculum should be scheduled adjacent to another lecture of the curriculum within the same day. Each isolated lecture will result in a penalty of 2 points.
- Stability: The lectures from a course should be given in the same room. Each room that is used besides the first room will give a 1 point penalty.

5.3.4. Encoding

For the encoding we will introduce a number of variables.

- $S_{c,r,t}$ states that a lecture for course c is scheduled for time slot t in room r.
- $CT_{c,t}$ specifies that course c has a lecture scheduled for time slot t.
- $CD_{c,d}$ specifies that the course c has a lecture scheduled on day d.
- $CUT_{cu,t}$ is True when curriculum cu has a course scheduled for time slot t.
- $CR_{c,r}$ states that course c has at some point a lecture scheduled in room r.

Finally, penalties for the soft constraints will be added to P. These penalties are all weighted depending on the cost of the violation.

Consistency

The introduced variables are related to each other. In order to make sure that they are consistent with each order some constrains must be added.

The first relation will be between S and CT. When $S_{c,r,t}$ is set it means that the course c is scheduled for a given room r and time slot t. Since the course is scheduled for time slot t $CT_{c,t}$ should be set. This can be specified using the following clause.

$$\forall r \in R : \overline{S_{c,t,r}} \vee CT_{c,t} \tag{5.5}$$

Furthermore, $CT_{c,t}$ should only be set to True when there exists a room to which the course is assigned for that time slot. Therefore, we need the following clause:

$$\bigvee_{r \in R} S_{c,t,r} \vee \overline{CT_{c,t}} \tag{5.6}$$

Something similar can be done between the time slots $\{d,h\}$ that a course c is scheduled ($CT_{c,\{d,h\}}$) and the days that it is scheduled ($CD_{c,d}$).

$$\forall h \in H : \overline{CT_{c,\{d,h\}}} \vee CD_{c,d} \tag{5.7}$$

$$\bigvee_{h \in H} CT_{c,\{d,h\}} \vee \overline{CD_{c,d}} \tag{5.8}$$

The first clause specifies that when $CT_{c,\{d,h\}}$ is set to True then $CD_{c,d}$ should be True as well. The second specifies that if $CD_{c,d}$ is True than there should be at least one $CT_{c,\{d,h\}}$ set to True.

When a course c is scheduled to a time slot t ($CT_{c,t}$) then each curriculum cu that contain the course should have a lecture scheduled for time slot t ($CUT_{cu,t}$). Furthermore, when a curriculum is scheduled for time slot t, then there must exist a course in the curriculum that is scheduled for that time slot.

$$\forall c \in cu : \overline{CT_{c,t}} \vee CUT_{cu,t} \tag{5.9}$$

$$\bigvee_{c \in cu} CT_{c,t} \vee \overline{CUT_{c,t}} \tag{5.10}$$

The final relation that must be specified is between S and CR. When a course c is scheduled to a specific time slot t and room r ($S_{c,r,t}$), then the course is scheduled for that room. Furthermore, when a course uses a room, then it must be scheduled to it for a specific time slot.

$$\forall t \in T : \overline{S_{c,t,r}} \vee CR_{c,r} \tag{5.11}$$

$$\bigvee_{t \in T} S_{c,t,r} \vee \overline{CR_{c,r}} \tag{5.12}$$

Hard constraints
The lectures hard constraint consists of two parts. First of all, we have to specify that for a course only one lecture can be scheduled per time slot. This can be done with the following constraint.

$$\sum_{r \in R} S_{c,r,t} \leq 1 \tag{5.13}$$

Furthermore, for each course c there is a specified number of lectures k that must be scheduled.

$$\sum_{t \in T} CT_{c,t} = k \tag{5.14}$$

The room constraint specifies that there can be at most one course scheduled to a room at a specific time slot. This can be done with the following constraint:

$$\sum_{c \in C} S_{c,t,r} \leq 1 \tag{5.15}$$

The conflict constraint should be specified for the teachers and curricula. They are both a set of courses. The following constraint specifies that at most one course from curriculum cu can be scheduled at time slot t.

$$\sum_{c \in cu} CT_{c,t} \leq 1 \tag{5.16}$$

For the teacher te the same thing has to be done.

$$\sum_{c \in te} CT_{c,t} \leq 1 \tag{5.17}$$

An unavailability constraint specifies that course c can not be given at time t. This can be specified by adding the unit clause $\overline{CT_{c,t}}$.

Soft constraints
Each soft constraint will introduce one or more penalty variables P. Each penalty variable will be used in only one soft constraint. The superscript * is used to denote that a new variable should be created each time that the constraint is added. So for example, if constraint 5.18 is added for course c_1 and c_2. Then the penalty literals that are used for c_2 are not the same as were used for c_1. All penalty variables will be added to P with a specific weight.

The first soft constraint is minimum days. It specifies over how many days the lectures for course c must be spread. Let k_c be the minimum number of working days for course c. If $k = 1$ the constraint can be skipped. For each course we introduce $k-1$ penalty variables p_i. Each p_i is added to P with weight 1.

$$\sum_{d\in D} CD_{c,d} + \sum_{i=1}^{k} p_i^* \geq k \tag{5.18}$$

By adding the penalty literals to the constraint, the constraint becomes soft. When the soft constraint is satisfied then $\sum_{d\in D} CD_{c,d} \geq k$. In this case all penalty variables can be set to False. However, if $\sum_{d\in D} \leq k$ then some of the penalty literals must be set to True. The minimum number of penalty literals that must be set to True is $k - \sum_{d\in D}$.

Compactness of a curriculum cu states that each lecture in a curriculum should be adjacent to another lecture within the same day d. For this we have to specify that if a curriculum has a lecture at hour h, then there must also be a lecture at $h-1$ or $h+1$. If this is not the case a penalty p^* literal must be set. The penalty literal is added to P with weight 2. For all hours that are not the first or last hour of a day this can be specified as follows:

$$\forall h \in H, 1 < h < H : CUT_{cu,\{d,h-1\}} \vee \overline{CUT_{cu,\{d,h\}}} \vee CUT_{cu,\{d,h+1\}} \vee p^* \tag{5.19}$$

The first and last time slot of a day have only one adjacent time slot. For those time slots we have to specify the following constraints:

$$\overline{CUT_{cu,\{d,1\}}} \vee CUT_{cu,\{d,2\}} \vee p^* \tag{5.20}$$

$$CUT_{cu,\{d,h-1\}} \vee \overline{CUT_{cu,\{d,h\}}} \vee p^* \tag{5.21}$$

The idea is that when a course is scheduled at time h than $\overline{CUT_{cu,\{d,h\}}}$ is False. The constraint can than only be satisfied if an adjacent course is scheduled or the penalty is set to True.

The room capacity constraint must be specified for all combinations of rooms and courses where the number of students in the course is larger than the room capacity. For each combination a penalty literal with weight $students - capacity$ is added to P.

$$\forall t \in T : p^* \vee \overline{S_{c,t,r}} \tag{5.22}$$

However, depending on the room capacity and students per course this can lead to large weights for the penalty literals. This can make it difficult to optimize the problem. Testing showed that when this constraint was encoded as a soft constraint then most methods could no longer optimize the problem. Therefore, the constraint is encoded as a hard constraint. This can be done by adding the following unit clauses for each room r that has a smaller capacity than the number of students that follow c.

$$\forall t \in T : \overline{S_{c,t,r}} \tag{5.23}$$

For room stability we have to count in how many rooms a course c is given. This can be done by summing the variables $CR_{c,r}$. In order to specify this constraint as a soft constraint we have to introduce a number of penalty literals. Each penalty is added with weight 1 to P. The constraint can be specified as follows:

$$\forall c \in C : \sum_{r\in R} CR_{c,r} - \sum_{i=1}^{r-1} p_i^* \leq 1 \tag{5.24}$$

If the course is given in only one room, then all penalty literals can be set to False. For each extra room, one penalty literal must be set to True to satisfy the constrain.

Literals cannot have negative weights and therefore the constraint must be modified so that all the penalties have positive weights. This can be done in that same way as we handled the negative weights with the pseudo-boolean constraints. Thus the penalty literals are replaced with their complement and the sum of their weights, $r-1$, is added to the right-hand side. This will lead to the following constraints:

$$\forall c \in C : \sum_{r\in R} CR_{c,r} + \sum_{i=1}^{r-1} \overline{p_i^*} \leq r \tag{5.25}$$

Finally, the optimization constraint is specified by $\sum_{p\in P} weight(p)p \leq k$ Where k is the maximum allowed penalty that will be set during the search.

As noted by Achá and Nieuwenhuis [6] for most instances the solver works better when some of the soft constraints are encoded as hard constraints. Less soft constraints will lead to a smaller optimization constraint, which will make it easier for the solver to find a solution. However, by making some of the soft constraints hard it is possible that the optimal solution is no longer a valid solution. Even worse, it also possible that the problem becomes infeasible. Testing showed that by making the room capacity hard the performance was improved a lot and only one problem became infeasible. Making other soft constraints hard increased the number of unsatisfiable problems. Therefore, only the room capacity constraint was implemented as a hard constraint.

5.4. Metrics

For decision problems the performance was measured by looking at the time needed to find a solution or to determine unsatisfiability. For optimization problems the instances were run for a certain amount of time and then the penalties of the best found solutions were compared. For most tests the penalty was measured after 5, 10 and 15 minutes. However, the final penalty does not state how quick the solution was found.

For example, if we have two methods and they both find quickly an initial solution with a penalty of 1000. The first method is initially not able to optimize it, and only just before the 5 minutes has past it manages to find a solution with a penalty of 100. The second method optimizes the initial solution quickly to a penalty of 101. However, it is not able to optimize it further. If we only look at the final penalty the first method is slightly better. However, the second method optimizes quicker and might be the better choice. The speed of the optimization can be quantified by looking at the area under the penalty curve. Methods that optimize quicker will have a smaller area. For the first part of the penalty curve there is not yet a penalty since no solution is found. The penalty that is used for that part of the curve is twice the penalty of the worst initial solution of any method. Other values for the initial penalty were tested as well and they produced similar results.

5.4.1. Normalization

The tested methods were run on a large set of benchmarks. The resulting penalties and run times varied a lot from instance to instance. In order to compare the methods an aggregated metric must be computed over all instances. Due to the large variety of the magnitude of the scores, this cannot be done by taking the average. Therefore, the scores have to be normalized and then the normalized values can be averaged. Normalization is done using the same formula that is used in MAX-SAT competitions. This is shown in equation 5.26

$$normalized\_score = (\text{best score of any method for the problem } + 1)/(score + 1) \tag{5.26}$$

5.5. StarExec

All tests were run on the StarExec server [59]. The server is hosted by the University of Iowa and is designed to test logic solvers. One type of the solvers that can be tested are SAT solvers. Competitions within the SAT community are often run on the StarExec server. The server runs on an Intel Xeon CPU E5-2609 at 2.40GHz with a cache of 10240 kB. It runs CentOS Linux and at the time of testing it was using version 7.7 with kernel version 3.10.0-1062.4.3.el7.x86_64.

6 Experimental results

This chapter shows the results of the thesis. The first three sections look into the behaviour of the encoding and try to identify the import parts of the encoding. The final two sections show the results of the proposed methods. The code of all methods and tests is available at `https://github.com/JensLangerak/thesis`.

6.1. Activity of the literals

Before we start trying to improve the handling of the constraints with dynamic or incremental approaches it is necessary to get some insights into the inner workings of the solver. By looking what is happening we might get an indication which things are worth investigating and what will probably not work. One of the advantages of an incremental approach is that it can lead to a smaller encoding. This will only work if there are enough literals that are not very active and therefore do not have to be encoded. By looking at the activity of the literals we can see if this is the case. Furthermore, it would be interesting to see if all important literals are already active at the start of the search or that they can become active much later during the search. This can give an indication of when the literals should be added. Finally, if active literals change over time, then the related literals might also change over time, which would make finding a good order more difficult.

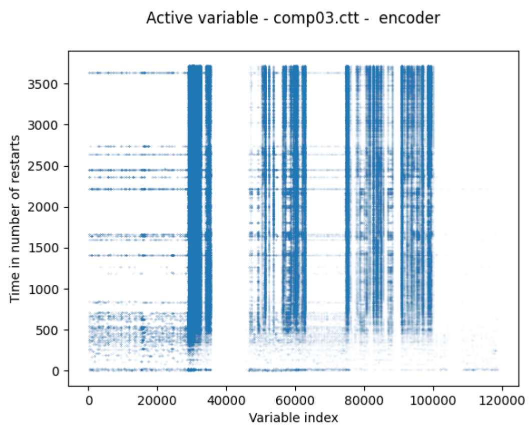
The activity of the literals is examined by creating scatter plots of the active literals over time. A literal is active when it is used during the conflict analyses step of the solver. Furthermore, the conflicts are grouped per restart of the solver. The reason for grouping the data per restart is that first of all logging individual conflicts would generate to much data and therefore some grouping is necessary. Furthermore, within a single run the solver tends to explore a specific part of the search tree and will therefore find similar conflicts. After a restart it will explore a different part of the search tree and this will probably lead to different conflicts. Therefore, the conflicts are grouped by restart.

Figure 6.1 shows the activity of the literals from two different problems. These two problems show the patterns that are also observed with a number of other instances. There are three things that are noticeable. First of all, during the majority of the search process there is constant group of variables active. These are the variables corresponding to the vertical lines in the plots. This indicates that it is not needed to encode all literals and that the literals that are related early in the search are probably also related later in the search.

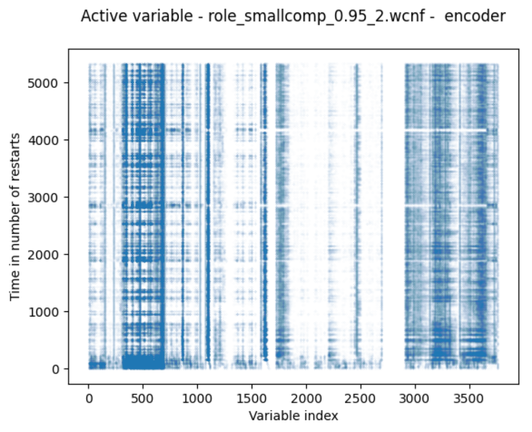
Secondly, at the start of the search there is a larger number of variables active. In the plot this can be seen by examining the active variables around restart 0. This is expected since at start the solver will need some time to find a solution for the less difficult parts of the problem. Furthermore, at the start the decision process will be somewhat random since the solver does not yet have good heuristics to base the decisions on.

Finally, with most problems, there are also some horizontal bands of very high or very low activity in the plots. Which suggest that there are short moments where the solver uses different variables to progress. However, it will always go back to the literals that were previously active. In the plots these can be seen as the horizontal lines.

In order to investigate if there are more instances where the number of active literals remains constant over time, we have collected the active literals for the unweighted MAX-SAT instances, pb10 and CTT problems. Figure 6.2 shows the fraction of literals that is used at both the start and end as fraction of the total literals that were used at either the start (b,d,f) or end (a,c,d). All histograms have a peak near the right of the plot. This



(a) CTT instance comp03.ctt



(b) MAX-SAT instance role_smallcomp_0.95_2.wcnf

Figure 6.1: Literals involved in conflict analysis for a CTT and MAX-SAT instance. On the x-axis are the ids of the variables and on the y-axis the number of restarts since the start of the solver.

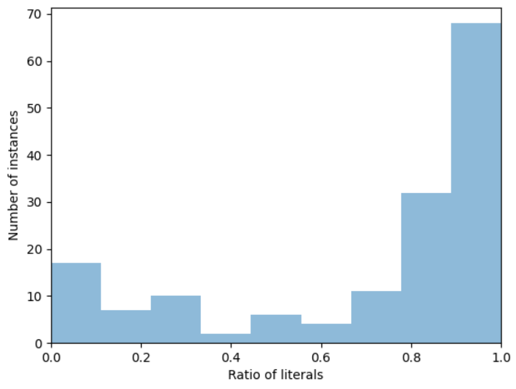
means that for most problem instances the majority of literals that are used, are used at the start and end of the solve process. Figure 6.2c has also a peak at the left of the plot. Thus for pb10 there are a lot of instances where the literals that are used at the end are not used at the start. pb10 contains decision problems and the other two sets contain optimization problems. This can indicate that for optimization problems the order has a larger impact than for decision problems.

Based on these observations we can conclude that not all variables are important and that it is worthwhile to investigate if they can be excluded from the encoding. Furthermore, the important variables can be identified early on during the search. However, the first couple of restarts are not representative for the rest of the search process.

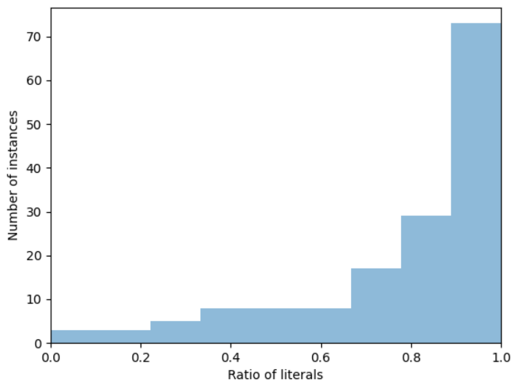
6.2. Literal order

The size of the GTE depends on the number of distinct possible sums. By keeping literals with the same weight together in the same subtree the number of auxiliary variables is reduced. This does improve the performance. However, there is not much known about the effect of the order of literals with the same weight. The hypothesis is that it is better when literals that are related are in the same branches of the subtree.

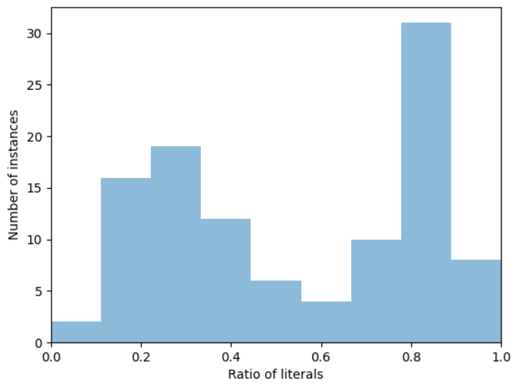
For example, suppose we have a constraint where at most 8 out of 16 literals must be True. Let x_1 and x_2 be two literals from the constraint. Now suppose that the constraint $x_1 + x_2 = 1$ is somehow implied by the other constraints from the problem. This constraint can be explicitly stated by the problem, but it is also possible that it is the consequence of a number of other constraints. If we place these literals at random positions of the tree, then it is not possible to utilize the fact that only one of them will be true in the construction of a conflict clause. However, if we place them in such way that they have the same parent. Thus there is a



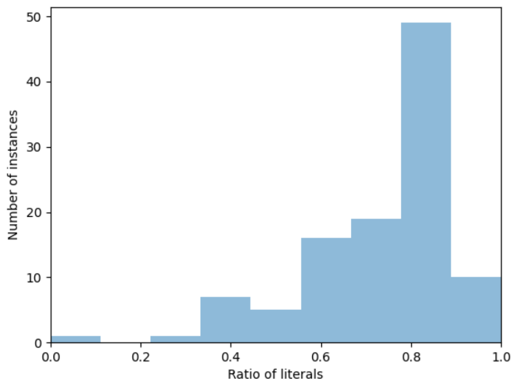
(a) MAX-SAT ratio of literals that were used at the end that were also used at the start.



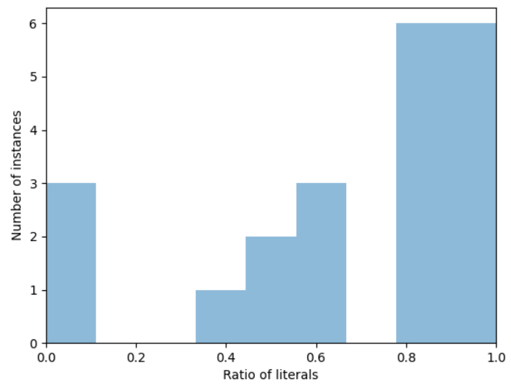
(b) MAX-SAT ratio of literals that were used at the start that were also used at the end.



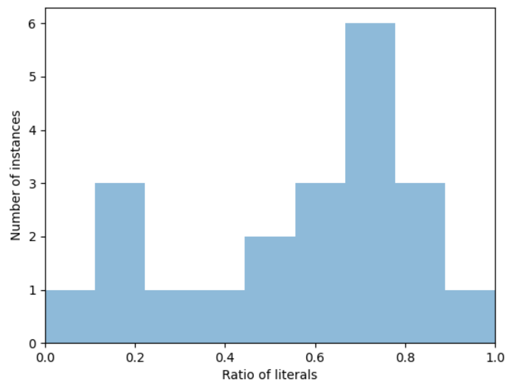
(c) pb10 ratio of literals that were used at the end that were also used at the start.



(d) pb11 ratio of literals that were used at the start that were also used at the end.



(e) CTT ratio of literals that were used at the end that were also used at the start.



(f) CTT ratio of literals that were used at the start that were also used at the end.

Figure 6.2: Ratio of literals that were used at both the start and end for the different benchmarks. For the literals that were used at the end it considers the literals that are used in the conflict analysis of the last 10% of conflicts. For the literals that were used at the start it skips the first 20% of conflicts and instead considers between 20% and 30%. The reason for skipping the first conflicts is that the solver behaves random during that time. Figure a, c and e show the number of literals that were used both at the start and end as ratio of the total number of literals that were used at the end. Figure b, d and f show the number of literals that were used both at the start and end as ratio of the total number of literals that were used at the start. a and b correspond is performed on the MAX-SAT w=1 problems. The results for pb10 are shown in c and d and the plots e and f correspond to the CTT problem.

node n in the encoded tree that holds the result of $x_1 + x_2$. If that node exists, then the conflict clause can use n_1 instead of either x_1 or x_2 and derive a more general clause. A similar advantage can also be found when the constraint $x_1 = x_2$ is implied. In that case random placement in the tree results in a conflict clause that contains both x_1 and x_2, placing them next to each other results in the usage of n_2, instead of x_1 and x_2. Depending on how many literals are related this can reduce the size of the learned clauses.

As already stated, the relatedness of literals is not always explicit specified by the constraints. Therefore, this is hard to detect in advance by just analysing the problem. Since we are going to construct the encoding during the search, we can use the time before adding the encoding to learn something about the related literals. When the literals are added to the encoding the solver should utilise the made observations to group the related literals together. However, it is not clear what a good measure for relatedness is. In order to investigate what methods can be used to improve the grouping of the literals an experiment has been conducted.

The experiment is conducted as follows. First a probing solver is constructed. The probing solver runs for some time and collects data about the relatedness. Depending on the probing method the constraints are either encoded or handled by the propagator. Next a new solver is created and for the new solver all constraints are encoded. The order of the literals depends on the heuristic collected by the probing solver. The performance of the new solver is compared with the other methods.

For problems with different weights, the literals are first ordered on the weight, literals with the same weight are ordered on relatedness. Initial testing showed that weight is more important than sorting on relatedness. This is done for all methods, except for the method that utilises the default order.

6.2.1. Methods

This section will discuss the methods for measuring the relatedness. The first three methods are control methods and the other methods try to improve the order.

6.2.2. Default order

This method uses the order that is provided by the problem definition. In the problem definition literals are often ordered on variable index. In practice when a problem is constructed variables are not created in a random order. Variables with a similar index often mean something similar or appear in the same constraint. For example, with the CTT benchmark there are constraints that added multiple penalty variables. These variables will have consecutive indexes. This means that the default order might already group related literals.

6.2.3. Weight

As already stated, it is often a good idea to order the literals on weights to reduce the size of the encoding. The weight method orders the literals on weight and for literals with the same weight, the default order is used.

6.2.4. Random order

Since the default order can be already good. We also tested with a random order of the literals. If the default order is already good, a random order will lead to worse results. The literals are first ordered on weight and literals with the same weight have a random order.

6.2.5. Count

The first methods that was tried based the order on how often a variable is used by the propagator in the conflict analysis step. This method used the propagator for the probing solver.

Each constraint records for each of its variables how often it was used in the conflict analysis step. Since the count is kept per variable, usage of both the positive and negative literal will increase the count. The count of a variable is increased in two cases. First, when the constraint causes a conflict then the count of all variables in the conflict are increased. The second case is when the conflict analyses found that the variable was implied by the constraint. The count for both the implied and reason variables are increased. The test solver orders the literals on the variable count. The idea is that literals that are often used are probably related.

6.2.6. Activity

A drawback of the count method is that it is possible that two literals have a similar count, but are used by the solver at different moments and therefore they are not related. This can be solved by looking at the activity. The activity is a metric that is used by the decision method. The activity of a variable is increased when it is encountered during the conflict analysis step. Furthermore, the solver will decay all activities over time. This is done before each conflict, by multiplying all activity scores with a decay factor smaller than 1. Due to the decay factor variables that appear in the same number of conflicts, but not the same conflicts, will have different values for the activity. If variables appear in the same conflicts, then they have a similar activity. Therefore, by sorting on activity, related variables should appear close to each other. This method has been tested with a probing solver that used the propagator and a probing solver that uses the GTE.

6.2.7. Distance

A drawback of activity is that it favours recent conflicts more than older conflicts. This is a good thing for the decision process, but for ordering the literals it might not be a good thing. It might be possible to mitigate this by looking in which conflicts the literals were involved.

This method records for each variable in which conflicts it was used. Next the distance $d_{i,j}$ between to variables i and j is computed by counting the number of conflicts where only i or j was involved, but not both. Next, we want literals with a low distance close to each other. One approach could be to partition the variables into clusters with the smallest average distance between variables. However, this is an NP-hard problem and will not be feasible when it is used during the solve process. Furthermore, the order of literals that appear in a lot of conflicts are much more important than the literals that are rarely used. So instead, we should focus on the grouping for the literals that appear in a lot of conflicts.

To do this the variables are first sorted on the number of conflicts they appear in. Next, we select the variable v that appears in the most conflicts and that has not yet been added to the encoding. Then we compute the distance to all variables that are not yet added. Next, we only select the variables where the distance is smaller than 0.1 times the number of conflicts v appears in. Finally, we first add v to the encoding and then the selected variables are encoded in order of their distance to v. These steps are repeated until all variables are added.

6.2.8. Results

All methods were run on the pb10, CTT, MAX-SAT weighted and MAX-SAT unweighted benchmarks. The probing solver was run for 10, 60 and 300 seconds. The results are discussed below.

The results for the pseudo-boolean benchmarks are shown in table 6.1. The default order and the default on weight perform much worse than the others. They are able to solve a similar number of problems however they need more time to find a solution and therefore they have a lower score. The random order performs much better than the default orders and is in fact similar to the worst alternative order. This is surprising and suggest that the default order is bad. Looking at the methods that utilised probing, it seems that the probing time does not play a large factor. Using the activity that is collected by a prober that uses the default encoding, seems to be the best method for determining order with this benchmark set. This is the "encoder activity" method and has by far the most problems that are solved the quickest. The order based on count does not perform as good as the other methods. Looking at how many problems were solved with a score of at least 0.9 we can see that it does less well than the other ordering methods.

With CTT and the two MAX-SAT benchmarks the results are much closer to each other. Table 6.2 shows the results when considering the area under the curve of the objective function. Table 6.3 considers the best found solutions after 5, 10 and 15 minutes. The average penalty score is similar among the different methods for the MAX-SAT benchmarks. The random method has the lowest score. For CTT all three control methods perform very well and ordering the literals on counts have a similar result as the control methods. The other methods are slightly worse.

However, when comparing the area, the results change. For both MAX-SAT benchmark sets the number of best solutions and with a score higher than 0.95 is lower for the default methods. For the weighted instances they have a lower average score, however for the unweighted the default methods have one of highest average score. That the number of best solutions is often lower for the default methods suggest that the other methods are quicker in finding a good solution. Table 6.4 shows the average normalized time to find the first optimized solution. This is second solution that is found and is the first solution that uses the objective constraint. The table shows that for the default and the weighted default order it takes much more time to find this solution. This will increase the size of area under the penalty curve and explains why these methods perform worse when looking at the area, but similar when considering the best found penalty. With CTT the control methods perform also worse when considering the area. Using the activity from the encoder probing solver, seems to be the solution that performs the best when considering the area under the curve.

Order method	Average score	Best	At least 0.95	At least 0.9	No solution
Default	0.40	0	0	0	83
Weight	0.39	0	0	0	83
Random	0.71	2	48	195	78
Encoder activity 10s	0.77	118	271	287	80
Encoder activity 60s	0.78	116	272	285	77
Encoder activity 300s	0.77	122	277	288	79
Propagator activity 10s	0.73	8	70	230	78
Propagator activity 60s	0.73	2	68	227	78
Propagator activity 300s	0.73	5	73	230	78
Propagator count 10s	0.71	5	56	199	78
Propagator count 60s	0.71	10	52	195	78
Propagator count 300s	0.72	6	55	200	78
Propagator distance 10s	0.73	5	70	232	79
Propagator distance 60s	0.72	7	67	227	80
Propagator distance 300s	0.73	10	69	231	79

Table 6.1: Results for different ordering methods on the pb10 benchmark set. Average score is the average normalized time to either find a solution or proof that the problem is unsatisfiable. A score of 1 is the best possible score and a score of 0 is the worst score. The order method lists if the probing solver used the encoding or the propagator and for how long the probing was done in seconds. Furthermore, it lists the heuristic that was used to order the literals.

Changing the order seems mostly beneficial for finding the first solutions. This means that decision problems benefit the most from the order since there you only have to find one solution. The proposed methods were better than the default on the pseudo-boolean benchmark, especially the order on the activity of the encoder performed well. However, they did not perform much better on the optimization problems. One explanation for that could be the number of literals in the pseudo-boolean constraint of the optimization constraint. The number of literals in the constraints of the pseudo-boolean benchmarks is much lower than the number of literals in the optimization constraint. When there are a large number of literals it is very likely that there are also large groups with similar values for the order heuristics. Improving the order of these large groups is difficult. With the pseudo-boolean benchmarks the groups of literals with similar heuristics were much smaller and therefore it is much clearer how they should be ordered. Of the proposed methods there was no method that was consistently better than the other methods.

The random order performed good and was sometimes better than the default order. The difference between the worst and random method was often larger than the difference between the best and random method. This suggest that a bad order has more effect than a good order. This can mean that a good order is an order that avoids inefficient groupings.

Finally, one thing that was noticed when a debugger was used to see how the order was made when the probing time was very short, was that there are a lot of literals that have appeared in exactly the same number of conflicts. This means that their count, activity and distance are all equal. This observation gave the idea for the top-down approach.

6.3. Usefulness of the auxiliary literals

When the encoding for a constraint is added a lot of auxiliary literals are created. Each literals represents a potential sum of a subset of the literals. With the help of these literals learned conflict clauses become more general. Using these literals, the solver can learn clauses that state that at most n literals of a subset can be true. Without these clauses the solver has to learn all possible violations of $n+1$ literals. Thus, in theory these literals can be very useful. But it is not clear how often they are used and if all layers are used. The layers closer to the root node encode more general constraints, but it is also more difficult to infer them. Table 6.5 shows how often each layer is used in a conflict. The analysis is done by looking at all learned clauses that contained at least one literal from an encoded constraint. This can be an auxiliary literal, but it can also be a literal from the constraint. Next for each layer the number of conflicts that contained at least one literal from that layer were counted. In the table leaf nodes are considered to be on layer 0. The leaf nodes are the literals from the original constraint. Layer 1 is the layer that sums 2 leaf nodes and so on. Layers 10 and higher are grouped together.

The table shows that the fast majority of literals are from the bottom layer. There is a large cap between

Ordering method	Average area score	Best	At least 0.95	At least 0.9
Default	0.45	1	25	52
Weight	0.45	0	21	52
Random	0.41	5	40	54
Encoder activity 10s	0.41	21	49	60
Encoder activity 60s	0.45	24	52	62
Encoder activity 300s	0.43	22	42	52
Propagator activity 10s	0.38	14	40	47
Propagator activity 60s	0.40	12	40	46
Propagator activity 300s	0.41	25	47	52
Propagator count 10s	0.41	4	41	56
Propagator count 60s	0.44	7	46	60
Propagator count 300s	0.47	27	58	68
Propagator distance 10s	0.42	16	43	53
Propagator distance 60s	0.42	24	49	59
Propagator distance 300s	0.40	18	39	49

(a) MAX-SAT unweighted benchmark set.

Ordering method	Average area score	Best	At least 0.95	At least 0.9
Default	0.45	0	6	10
Weight	0.45	0	6	11
Random	0.54	2	41	50
Encoder activity 10s	0.59	14	47	55
Encoder activity 60s	0.57	16	46	57
Encoder activity 300s	0.63	22	56	65
Propagator activity 10s	0.56	14	49	54
Propagator activity 60s	0.56	16	48	54
Propagator activity 300s	0.57	34	51	60
Propagator count 10s	0.54	0	39	50
Propagator count 60s	0.54	2	38	51
Propagator count 300s	0.55	20	47	54
Propagator distance 10s	0.53	18	48	54
Propagator distance 60s	0.53	18	42	52
Propagator distance 300s	0.53	18	42	49

(b) MAX-SAT weighted benchmark set.

Ordering method	Average area score	Best	At least 0.95	At least 0.9
Default	0.75	0	0	3
Weight	0.76	0	0	2
Random	0.80	0	3	6
Encoder activity 10s	0.82	2	5	10
Encoder activity 60s	0.85	4	7	11
Encoder activity 300s	0.87	6	10	12
Propagator activity 10s	0.81	1	3	6
Propagator activity 60s	0.75	0	0	4
Propagator activity 300s	0.82	1	5	11
Propagator count 10s	0.80	0	3	7
Propagator count 60s	0.80	0	3	7
Propagator count 300s	0.80	2	3	7
Propagator distance 10s	0.77	1	2	3
Propagator distance 60s	0.75	1	3	4
Propagator distance 300s	0.78	2	3	4

(c) CTT benchmark set

Table 6.2: The area under the curve of the penalty function using the different methods to order the literals. The average area score shows the average of the normalized areas. A score of 1 is the best possible score and a score of 0 is the worst score. The Ordering method lists if the probing solver used the encoding or the propagator and for how long the probing was done in seconds. Furthermore, it lists the heuristic that was used to order the literals.

Ordering method	Average penalty score	Best	At least 0.95	At least 0.9
Default	0.79	263	410	457
Weight	0.79	263	407	457
Random	0.72	245	388	431
Encoder activity 10s	0.77	200	373	438
Encoder activity 60s	0.78	199	373	444
Encoder activity 300s	0.78	205	365	451
Propagator activity 10s	0.79	214	345	437
Propagator activity 60s	0.79	217	354	434
Propagator activity 300s	0.78	233	325	409
Propagator count 10s	0.72	242	388	429
Propagator count 60s	0.76	256	398	445
Propagator count 300s	0.78	260	408	457
Propagator distance 10s	0.77	157	297	404
Propagator distance 60s	0.78	201	330	445
Propagator distance 300s	0.78	208	330	422

(a) MAX-SAT unweighted benchmark set.

Ordering method	Average penalty score	Best	At least 0.95	At least 0.9
Default	0.82	272	351	406
Weight	0.82	270	351	406
Random	0.79	266	340	391
Encoder activity 10s	0.82	250	354	419
Encoder activity 60s	0.82	249	357	415
Encoder activity 300s	0.82	247	350	394
Propagator activity 10s	0.82	256	363	410
Propagator activity 60s	0.82	252	355	417
Propagator activity 300s	0.81	255	360	417
Propagator count 10s	0.79	261	342	391
Propagator count 60s	0.82	271	349	403
Propagator count 300s	0.82	278	354	408
Propagator distance 10s	0.82	274	376	431
Propagator distance 60s	0.82	254	363	428
Propagator distance 300s	0.82	260	372	412

(b) MAX-SAT weighted benchmark set.

Ordering method	Average penalty score	Best	At least 0.95	At least 0.9
Default	0.87	30	34	41
Weight	0.87	32	35	41
Random	0.87	33	35	41
Encoder activity 10s	0.82	21	26	33
Encoder activity 60s	0.75	14	17	21
Encoder activity 300s	0.80	20	31	35
Propagator activity 10s	0.76	5	13	21
Propagator activity 60s	0.75	11	17	24
Propagator activity 300s	0.81	16	24	28
Propagator count 10s	0.87	31	34	41
Propagator count 60s	0.86	30	32	41
Propagator count 300s	0.86	30	33	41
Propagator distance 10s	0.74	12	14	20
Propagator distance 60s	0.76	12	19	26
Propagator distance 300s	0.76	17	20	23

(c) CTT benchmark set

Table 6.3: The penalty of the best found solution after 5, 10 and 15 minutes. The average area shows the average of the normalized areas. A score of 1 is the best possible score and a score of 0 is the worst score. The ordering method lists if the probing solver used the encoding or the propagator and for how long the probing was done in seconds. Furthermore, it lists the heuristic that was used to order the literals.

Ordering method	MAX-SAT w=5	MAX-SAT uw	CTT
Default	0.50	0.56	0.76
Weight	0.50	0.55	0.76
Random	0.60	0.57	0.82
Encoder activity 10s	0.65	0.65	0.85
Encoder activity 60s	0.64	0.66	0.83
Encoder activity 300s	0.61	0.67	0.87
Propagator activity 10s	0.64	0.69	0.84
Propagator activity 60s	0.64	0.69	0.87
Propagator activity 300s	0.65	0.72	0.88
Propagator count 10s	0.60	0.58	0.82
Propagator count 60s	0.63	0.62	0.83
Propagator count 300s	0.65	0.65	0.83
Propagator distance 10s	0.65	0.68	0.84
Propagator distance 60s	0.64	0.70	0.83
Propagator distance 300s	0.65	0.72	0.83

Table 6.4: The average normalized time to find the first optimized solution in the different benchmark sets. This is the fist solution that makes use of the optimization constraint. The best score is 1 and the worst score is 0, thus a higher score is better.

the leaf nodes and the first layers. For the other layers increasing the layer will lead to a small reduction of used literals. The big cap between the leaf nodes and the first layer can indicate that it is possible to improve the pairing of the literals. The pairs method tries to improve this by pairing literals with multiple other literals. This should make it more likely that an auxiliary literal from layer 1 can be used. The pb10 benchmark has also a lot of literals that come from layer 1. With the Full Bottom Layers method, we will test if these two layers are enough to improve the performance.

Layer	MAX-SAT uw	MAX-SAT w=5	CTT	pb10
0	47.20	46.83	83.32	57.39
1	10.54	7.20	5.52	25.99
2	9.68	6.65	6.05	8.16
3	8.54	7.06	3.30	4.97
4	8.41	6.90	1.57	3.77
5	5.81	6.67	0.24	2.10
6	4.09	5.93	0	1.48
7	2.92	5.34	0	0.48
8	2.67	4.28	0	0
9	1.85	3.31	0	0
10 or higher	4.37	6.98	0	0

Table 6.5: The table shows how often a literal from an encoded layer is used in a conflict. It shows for each layer the percentage of conflict clauses that contain a literal from that layer. Furthermore, only conflicts clauses that contained at least one literal from the encoding were considered and a conflict can contain literals from different layers. The bottom layer is layer 0 and contains the original literals of the constraint. All other layers contain auxiliary literals. The higher layers were not present in all problem instances. For those percentages only the instances that contained that layer were considered.

6.4. Size of the encoding

In this section we study the number of auxiliary literals and clauses that were added to the problem definition. We only consider clauses that are permanently added. Thus the learnt clauses from the propagator are not counted. Table 6.6 shows how many auxiliary literals were added and table 6.7 shows how many clauses were added for the encodings. The number after the dynamic, incremental and top-down method specify the value that was used for the parameter d. A lower value means that the method is more eager to encode the constraint. All scores are normalized by dividing the number of added literals/clauses by the number that was added by the encoder. Thus a score of 1 means that it has added as much as the full encoding. A lower score is better.

In almost all cases the encoder adds the most literals and clauses. There are two exceptions, the incremen-

Solver	CTT	MAX-SAT uw	MAX-SAT w=1	MAX-SAT w=5	pb10	pb15	Elffers	d_n_k
Encoder	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Propagator	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Dynamic 0.5	0.68	0.83	0.96	0.90	0.05	0.84	0.82	0.25
Dynamic 10	0.54	0.83	0.95	0.90	0.03	0.64	0.69	0.08
Incremental 0.5	0.62	0.58	0.61	0.73	0.05	0.82	0.82	0.27
Incremental 10	0.41	0.45	0.44	0.39	0.03	0.70	0.73	0.18
Top-down 0.5	0.62	0.51	0.63	0.59	0.06	0.82	0.78	0.27
Top-down 1000	0.42	0.40	0.41	0.37	0.00	0.17	0.42	0.04
Full Bottom Layers	0.13	0.16	0.12	0.11	0.14	0.38	0.32	0.33
Pairs	0.33	0.70	0.71	0.60	0.01	0.35	1.01	0.07

Table 6.6: Number of added auxiliary literals as fraction of the auxiliary literals from the encoder.

Solver	CTT	MAX-SAT uw	MAX-SAT w=1	MAX-SAT w=5	pb10	pb15	Elffers	d_n_k
Encoder	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Propagator	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Dynamic 0.5	0.42	0.74	0.85	0.78	0.05	0.84	0.82	0.24
Dynamic 10	0.38	0.74	0.85	0.78	0.03	0.64	0.69	0.08
Incremental 0.5	0.13	0.48	0.50	2.16	0.03	0.82	0.82	0.27
Incremental 10	0.09	0.36	0.34	0.28	0.02	0.70	0.72	0.18
Top-down 0.5	0.16	0.51	0.63	0.82	0.06	0.84	0.87	0.27
Top-down 1000	0.39	0.37	0.41	0.35	0.00	0.19	0.47	0.03
Full Bottom Layers	0.01	0.07	0.02	0.02	0.04	0.35	0.24	0.29
Pairs	0.01	0.07	0.07	0.06	0.00	0.22	0.28	0.04

Table 6.7: Number of added clauses as fraction of the added clauses from the encoder.

tal 0.5 method adds a lot more clauses than the encoder for the MAX-SAT w=5 benchmark. The incremental method does not group the literals on weight and therefore it needs more clauses. Furthermore, the pairs methods adds slightly more variables than the encoder for the Elffers benchmark set. With the exception for the w=5 benchmarks the incremental methods are smaller than the dynamic method for the optimization problems. For the pseudo-boolean problems the incremental method can sometimes be larger than the dynamic method. Most constraints in the pseudo-boolean benchmarks are small and therefore a partial encoding has much less effect.

The size for the top-down method is similar to the incremental method. It is smaller on the MAX-SAT w=5 benchmark and this can be explained by the fact that the top-down method groups literals with the same weight.

When the delay factor d is set higher, the method waits longer before a constraint is encoded. Therefore, a higher d value corresponds to a smaller size.

The pairs method produces only a fraction of the clauses that are produced by the other methods, however the number of literals is often not lower than the other methods. The Full Bottom Layers also adds only a fraction of the clauses of the other methods. In most cases it also adds fewer auxiliary literals.

6.5. Performance

In this section we discuss the penalties and required solve time for the different methods. Table 6.8 shows the average normalized penalty for the different methods on the benchmarks and table 6.9 the normalized area under the curve. The results for the pseudo-boolean benchmarks are shown in table 6.10.

The results show that the encoder achieves a lower penalty than the propagator on the optimization problems. The propagator seems to be stronger for most of the pseudo-boolean benchmarks. The Elffers benchmark is the only pseudo-boolean benchmark where the encoder scored better than the propagator. Inspecting the benchmarks in this set showed that it contained different types of benchmarks and one of the types was actually a MAX-SAT problem specified with pseudo-boolean constraints. Therefore, this set was divided into two sets, Elffers$_1$ and Elffers$_2$. Elffers$_2$ contains all MAX-SAT problems of the Elffers set, and Elffers$_1$ contains the remaining problems. This shows that the encoder was only better for the MAX-SAT problems in the set.

Despite that the encoder has a better penalty score for the optimization problems, the propagator has a similar or better area under the curve. This suggest that early on the propagator is not worse than the encoder.

The dynamic and incremental methods are close to each other. For the MAX-SAT problems they are similar to the encoder. On the unweighted sets (uw and w=1), the incremental method was the solution with the highest score. On w=5 both the dynamic and incremental method perform slightly worse than the encoder.

For the CTT benchmark the dynamic method is the best method and it is much better than the incremental method. It is also much better than the encoder and propagator. This can be explained by the fact that CTT has both an optimization constraint and a lot of small pseudo-boolean constraints. For this problem it is important to decide which constraint should be encoded. The area under the curve for the dynamic method is in general better.

For most of the pseudo-boolean problems these two methods are closer to the propagator. Which of these is better depends on the benchmark.

The performance of the top-down method seems to depend on the parameter that controls how quick an encoding is added. When it is eager to add the encoding (Top-down 0.5) it is often outperformed by both the incremental and dynamic method. For the pseudo-boolean problems the method performed well when it was configured to wait with adding the constraints (Top-down 1000). It was able to achieve the best score for two of the pseudo-boolean benchmarks and was able to outperform the propagator on all of them. However, this configuration did not perform well on the optimization problems. This method requires a lot of bookkeeping and that will slow the method down.

The performance of the pairs and full bottom layers method is mixed. The pairs method performed better than the propagator on the CTT and MAX-SAT uw benchmark, however it was not as good as the encoder. On the weighted MAX-SAT it was able to perform only slightly better than the propagator. On the pseudo-boolean problems it was slightly worse than the propagator. The full bottom layers method does not perform well and is for five of the benchmarks the worst method. It performed surprisingly well on the Elffers$_2$ set, only slightly worse than the encoder and much better than the other methods.

Interestingly the area under the curve for the full bottom layers method is among the best methods. This indicates that early on this can be useful, however the lowest layer does not contain enough information to be useful later on. Due to the extra work that is needed for theses methods they end up being worse than the propagator. Especially replacing the literals in the explanation with the pairs methods had large hit on the performance.

The dynamic, incremental and top-down methods worked for both the problems where the encoder was strong and for the problems where the propagator was strong. A conservative top-down approach is especially strong for the pseudo-boolean problems. The incremental method had for three of the benchmark sets the highest score and the top-down and dynamic method scored both for two sets the highest score. This shows that these methods can improve the performance, however the improvement is often small.

Solver	**CTT**	**MAX-SAT uw**	**MAX-SAT w=1**	**MAX-SAT w=5**
Encoder	0.73	0.79	0.83	**0.81**
Propagator	0.25	0.66	0.76	0.65
Dynamic 0.5	**0.77**	0.79	0.84	0.78
Dynamic 10	0.75	0.79	0.84	0.78
Incremental 0.5	0.69	**0.80**	**0.87**	0.77
Incremental 10	0.61	0.79	0.86	0.73
Top-down 0.5	0.71	0.72	0.83	0.76
Top-down 1000	0.61	0.71	0.77	0.69
Full Bottom Layers	0.22	0.75	0.78	0.58
Pairs	0.55	0.74	0.77	0.69

Table 6.8: Average normalized penalties for the benchmarks. A higher score is better.

Solver	**CTT**	**MAX-SAT uw**	**MAX-SAT w=1**	**MAX-SAT w=5**
Encoder	0.24	0.27	0.38	0.39
Propagator	0.26	0.30	0.42	0.33
Dynamic 0.5	0.59	0.29	0.42	0.38
Dynamic 10	**0.63**	0.29	0.42	0.39
Incremental 0.5	0.49	0.24	0.32	0.35
Incremental 10	0.61	0.22	0.32	0.31
Top-down 0.5	0.42	0.24	0.39	0.36
Top-down 1000	0.40	0.24	0.35	0.29
Full Bottom Layers	0.16	**0.50**	**0.48**	**0.45**
Pairs	0.37	0.22	0.32	0.28

Table 6.9: Average normalized area of the penalty curve for the benchmarks. A higher score is better.

Solver	**pb10**	**pb15**	**Elffers**	**d_n_k**	**Elffers$_1$**	**Elffers$_2$**
Encoder	0.12	0.41	**0.38**	0.37	0.17	**0.74**
Propagator	0.71	0.44	0.24	0.63	0.26	0.21
Dynamic 0.5	0.67	0.40	0.30	0.65	0.21	0.45
Dynamic 10	0.69	0.41	0.32	**0.69**	0.24	0.46
Incremental 0.5	0.69	0.39	0.28	0.59	**0.30**	0.24
Incremental 10	0.71	0.43	0.26	0.64	0.28	0.23
Top-down 0.5	0.69	0.36	0.19	0.59	0.17	0.23
Top-down 1000	**0.73**	**0.45**	0.25	0.66	0.27	0.23
Full Bottom Layers	0.40	0.32	0.34	0.27	0.13	0.70
Pairs	0.69	**0.45**	0.20	0.59	0.21	0.20

Table 6.10: Average normalized run times for the benchmarks. A higher score is better.

7
Conclusion

During the thesis we have looked at different methods to incrementally construct the encoding of a pseudo-boolean constraint. There is no method that outperforms all other methods on all benchmarks. However, there are some methods that perform more consistent. The encoder excelled on the optimization problems but was not good on the pseudo-boolean benchmarks. The propagator performs well on the pseudo-boolean benchmark but was not very good on the optimization problems. The dynamic and incremental method performed very similar to each other and the main advantage of the incremental method is its smaller encoding size. They both were able to outperform both the encoder and propagator on half of benchmarks. When the top-down method is very conservative with adding the constraints, it can work really well for the pseudo-boolean problems. However, that configuration does not work well for the optimization problems. The other two methods that were tested did not perform well.

Now we are able to answer the research questions that were proposed at the start of the thesis.

- **RQ1:** Is it needed to encode the full constraint? If this is not needed, then what are the important parts of the constraint to encode?

The results show that it is not always needed to encode the full constraint. We have seen that during the search the same group of literals stay active and the methods that apply a partial encoding performed well. The bottom layers of the encoding are the most used. However, when only these layers are added the solver will not perform better than the propagator, thus all layers are needed.

- **RQ2:** Has the order/grouping of the literals effect on the performance?

The order of the literals has effect on the performance, but it is difficult to improve the encoding. The biggest effect of the ordering is during the first part of the search. As time progresses its effect becomes less significant.

- **RQ3:** What is the effect of the incremental construction on the size of the encoding?

All methods were able to reduce the size of the encoding. On the MAX-SAT problems a partial encoding reduced the size compared to the dynamic method. On the pseudo-boolean problems, the size reduction of a partial encoding is limited. When working with weighted problems, the size of the incremental encoding can be larger due to the less efficient grouping of weights.

- **RQ4:** What is the effect of the incremental construction on the performance of the solver?

The methods that were develop during the thesis were able to improve the performance. However, the improvement is small. The difference between the incremental and dynamic method is also small and which method is better differs per benchmark.

The incremental method is the best performing partial encoding. Its advantage over the dynamic method is mostly that it creates a smaller encoding for MAX-SAT. For those problems it had a similar or slightly better performance than adding the full encoding.

7.1. Recommendations

The main advantage of the incremental method is the reduction in its size. However, for weighted problems it did not perform well. In the future this method can be extended to allow the grouping of literals with the same weights. Furthermore, the partial encoding can be applied to other encoding schemes. Since its effect on the performance is limited, a partial encoding should only be done when the size of the encoding is a concern. Furthermore, the best performing method differs per benchmark. It would be interesting to see if it is possible to predict during the search which method will be the best choice and choose that for the encoding.

Bibliography

[1] Ignasi Abío and Peter J Stuckey. Conflict directed lazy decomposition. In *International Conference on Principles and Practice of Constraint Programming*, pages 70–85. Springer, 2012.

[2] Ignasi Abío and Peter J Stuckey. Encoding linear constraints into sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[3] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012.

[4] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J Stuckey. To encode or to propagate? the best choice for each constraint in sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 97–106. Springer, 2013.

[5] Ignasi Abío, Valentin Mayer-Eichberger, and Peter J Stuckey. Encoding linear constraints with implication chains to cnf. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–11. Springer, 2015.

[6] Roberto Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with sat and maxsat. *Annals of Operations Research*, 218(1):71–91, 2014.

[7] Fadi A Aloul, Syed ZH Zahidi, Anas Al-Farra, Basel Al-Roh, and Bashar Al-Rawi. Solving the employee timetabling problem using advanced sat & ilp techniques. *J. Comput.*, 8(4):851–858, 2013.

[8] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[9] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[10] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. *AAAI/IAAI*, 2002: 613–619, 2002.

[11] Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Rubens Martins. Maxsat evaluation 2020: Solver and benchmark descriptions. 2020.

[12] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 108–122. Springer, 2003.

[13] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-boolean constraints into cnf. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 181–194. Springer, 2009.

[14] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[15] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320, 1999.

[16] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005.

[17] Kenneth Chin-A-Fat. *School timetabling using satisfiability solvers*. PhD thesis, Master's thesis, Technical University Delft, The Netherlands, 2004.

[18] Michael Codish and Moshe Zazon-Ivry. Pairwise cardinality networks. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 154–172. Springer, 2010.

[19] Michael Codish, Yoav Fekete, Carsten Fuhs, and Peter Schneider-Kamp. Optimal base encodings for pseudo-boolean constraints. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 189–204. Springer, 2011.

[20] José Coelho and Mario Vanhoucke. Multi-mode resource-constrained project scheduling using rcpsp and sat solvers. *European Journal of Operational Research*, 213(1):73–82, 2011.

[21] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[22] William Cook, Collette R Coullard, and Gy Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[23] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[24] Emir Demirović and Nysret Musliu. Maxsat-based large neighborhood search for high school timetabling. *Computers & Operations Research*, 78:172–180, 2017.

[25] Luca Di Gaspero, Barry McCollum, and Andrea Schaerf. The second international timetabling competition (itc-2007): Curriculum-based course timetabling (track 3). Technical report, Technical Report QUB/IEEE/Tech/ITC2007/CurriculumCTT/v1. 0, Queen's ..., 2007.

[26] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[27] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, volume 18, pages 1291–1299, 2018.

[28] Jan Elffers et al. A cardinal improvement to pseudo-boolean solving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1495–1503, 2020.

[29] Yoav Fekete and Michael Codish. Simplifying pseudo-boolean constraints in residual number systems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 351–366. Springer, 2014.

[30] Ralph E Gomory. An algorithm for integer solutions to linear programs. *Recent advances in mathematical programming*, 64(260-302):14, 1963.

[31] Aarti Gupta, Malay K Ganai, and Chao Wang. Sat-based verification methods and applications in hardware verification. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 108–143. Springer, 2006.

[32] Shai Haim and Toby Walsh. Restart strategy selection using machine learning techniques. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 312–325. Springer, 2009.

[33] Kazuhisa Hosaka, Yasuhiko Takenaga, and Shuzo Yajima. On the size of ordered binary decision diagrams representing threshold functions. In *International Symposium on Algorithms and Computation*, pages 584–592. Springer, 1994.

[34] Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[35] Saurabh Joshi, Ruben Martins, and Vasco Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In *International conference on principles and practice of constraint programming*, pages 200–209. Springer, 2015.

[36] Saurabh Joshi, Prateek Kumar, Ruben Martins, and Sukrut Rao. Approximation strategies for incomplete maxsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 219–228. Springer, 2018.

[37] Michał Karpinski and Marek Piotrów. Competitive sorter-based encoding of pb-constraints into sat. *Proceedings of Pragmatics of SAT*, pages 65–78, 2015.

[38] Michał Karpiński and Marek Piotrów. Encoding cardinality constraints using generalized selection networks. *arXiv preprint arXiv:1704.04389*, 2017.

[39] Michał Karpiński and Marek Piotrów. Incremental encoding of pseudo-boolean goal functions based on comparator networks. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 519–535. Springer, 2020.

[40] Michael Kaufmann and Stephan Kottler. Beyond unit propagation in sat solving. In *International Symposium on Experimental Algorithms*, pages 267–279. Springer, 2011.

[41] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa. Solving open job-shop scheduling problems by sat encoding. *IEICE TRANSACTIONS on Information and Systems*, 93(8):2316–2318, 2010.

[42] Daniel Le Berre and Anne Parrain. On sat technologies for dependency management and beyond. In *First Workshop on Software Product Lines (ASPL'08)*, pages 197–200, 2008.

[43] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 94–110. Springer, 2018.

[44] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[45] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 531–548. Springer, 2014.

[46] Gonçalo P Matos, Luís M Albino, Ricardo L Saldanha, and Ernesto M Morgado. Solving periodic timetabling problems with sat and machine learning. *Public Transport*, pages 1–24, 2020.

[47] Alexander Nadel. Solving maxsat with bit-vector optimization. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 54–72. Springer, 2018.

[48] Alexander Nadel. Anytime weighted maxsat with improved polarity selection and bit-vector optimization. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 193–202. IEEE, 2019.

[49] Alexander Nöhrer, Armin Biere, and Alexander Egyed. Managing sat inconsistencies with humus. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 83–91, 2012.

[50] Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Modulo based cnf encoding of cardinality constraints and its application to maxsat solvers. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 9–17. IEEE, 2013.

[51] Chanseok Oh. Between sat and unsat: the fundamental difference in cdcl sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 307–323. Springer, 2015.

[52] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[53] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2(02n03):205–211, 1992.

[54] Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted maxsat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 37–53. Springer, 2018.

[55] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted maxsat. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 556–577. Springer, 2021.

[56] Masahiko Sakai and Hidetomo Nabeshima. Construction of an robdd for a pb-constraint in band form and related techniques for pb-solvers. *IEICE TRANSACTIONS on Information and Systems*, 98(6):1121–1127, 2015.

[57] Joao P Marques Silva and Karem A Sakallah. Grasp-a new search algorithm for satisfiability. In *ICCAD*, volume 96, pages 220–227, 1996.

[58] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Detection of inconsistencies in complex product configuration data using extended propositional sat-checking. In *FLAIRS conference*, pages 645–649, 2001.

[59] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In *International joint conference on automated reasoning*, pages 367–373. Springer, 2014.

[60] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *29th International Conference on Software Engineering (ICSE'07)*, pages 178–188. IEEE, 2007.

[61] Aolong Zha, Miyuki Koshimura, and Hiroshi Fujita. A hybrid encoding of pseudo-boolean constraints into cnf. In *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 9–12. IEEE, 2017.

[62] Aolong Zha, Naoki Uemura, Miyuki Koshimura, and Hiroshi Fujita. Mixed radix weight totalizer encoding for pseudo-boolean constraints. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 868–875. IEEE, 2017.

A
Appendix

Here we list the score of all different configurations that were tested. The methods in bold were reported in the results section of the thesis.

A.1. Notation

- **d=***v*: Where v is a number. Specifies the value that was used for the parameter d, that controls how quick the encoding is added (see section 4.1.3).
- **m**: Methods with the *m* parameter have a maximum allowed value 1000 for the encoding criteria. For the dynamic method this gives the following encoding criteria:

  $$min(1000, weight \cdot maximum\_of\_constraint \cdot d) < literal\_count \tag{A.1}$$

  And for the incremental:

  $$min(1000, weight \cdot maximum\_of\_constraint \cdot d) < literal\_count \tag{A.2}$$

  This causes larger constraints to become more likely to be encoded.
- **h**: The *h* specifies that the counts are halved after each restart. This was also done by Abío et al. [1].
- **r**: The incremental approach uses both the encoder and propagator to enforce the constraint. The *r* species that if the propagator provides the explanation, then the literals that are already encoded should be replaced by the sum literals. This is done by removing all literals that are added to the encoding from the explanation and then the sum literal with the highest weight that is set to True from the root node is added to the explanation.
- For the incremental method two strategies were used to determine the order of literals when multiple literals were encoded at the same time.

  Distance uses the strategy described in section 6.2.7. **Activity** sorts the literals on their activity.

A.2. Normalized penalty CTT

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.73**	**21**	**24**	**29**	**36**	**13**	**3**
Propagator	**0.25**	**3**	**3**	**3**	**3**	**55**	**3**
Dynamic d=0.5	0.82	30	37	41	46	9	3
Dynamic d=1	0.75	24	36	38	44	17	3
Dynamic d=10	0.37	10	11	13	16	45	3
Dynamic m h d=0.5	**0.77**	**20**	**26**	**30**	**44**	**13**	**3**
Dynamic m h d=1	0.77	19	29	30	39	12	3
Dynamic m h d=10	**0.75**	**17**	**27**	**31**	**35**	**13**	**3**
Dynamic m d=0.5	0.76	29	32	33	39	12	3
Dynamic m d=1	0.75	24	28	33	39	13	3
Dynamic m d=10	0.76	25	34	36	39	12	3
Incremental Distance d=0.5	0.72	13	16	20	39	16	3
Incremental Distance d=1	0.70	12	15	22	32	18	3
Incremental Distance d=10	0.63	14	14	20	30	25	3
Incremental Distance m h d=0.5	0.62	12	15	24	30	24	3
Incremental Distance m h d=1	0.63	10	15	17	30	22	3
Incremental Distance m h d=10	0.51	9	13	19	21	34	3
Incremental Distance m d=0.5	0.72	13	16	20	39	16	3
Incremental Distance m d=1	0.71	11	16	21	36	18	3
Incremental Distance m d=10	0.71	15	17	21	36	18	3
Incremental Activity m h d=0.5	0.68	11	17	20	32	18	3
Incremental Activity m h d=1	0.66	14	15	27	32	22	3
Incremental Activity m h d=10	0.64	14	16	21	28	19	3
Incremental Distance r m h d=0.5	0.58	11	11	12	17	26	3
Incremental Distance r m h d=1	0.59	9	11	14	20	25	3
Incremental Distance r m h d=10	0.50	5	7	10	14	27	3
Incremental Activity r d=0.5	**0.69**	**10**	**15**	**21**	**32**	**18**	**3**
Incremental Activity r d=1	0.65	7	9	14	25	16	3
Incremental Activity r d=10	**0.61**	**6**	**12**	**16**	**22**	**20**	**3**
Top-down d = 0.5	**0.71**	**11**	**17**	**26**	**34**	**16**	**3**
Top-down d = 1	0.64	10	13	20	29	20	3
Top-down d = 10	0.31	5	6	7	10	51	3
Top-down d = 100	0.24	3	3	3	3	55	3
Top-down d = 1000	**0.61**	**12**	**13**	**17**	**23**	**19**	**3**
Full Bottom Layers	**0.22**	**3**	**3**	**3**	**5**	**58**	**3**
Pairs	**0.55**	**6**	**9**	**9**	**11**	**22**	**3**

A.3. Normalized penalty MAX-SAT uw

Solver	**Average score**	**Best**	**At least 0.95**	**At least 0.9**	**At least 0.8**	**Less than 0.5**	**No solution**
Encoder	**0.79**	**255**	**404**	**465**	**530**	**125**	**99**
Propagator	**0.66**	**100**	**208**	**246**	**339**	**200**	**36**
Dynamic d=0.5	0.73	187	327	389	459	170	48
Dynamic d=1	0.72	193	325	382	442	171	48
Dynamic d=10	0.68	136	279	337	406	196	60
Dynamic m h d=0.5	**0.79**	**220**	**375**	**435**	**516**	**119**	**60**
Dynamic m h d=1	0.79	221	374	432	515	119	60
Dynamic m h d=10	**0.79**	**222**	**373**	**431**	**514**	**118**	**60**
Dynamic m d=0.5	0.79	228	367	429	512	119	60
Dynamic m d=1	0.79	228	369	433	512	120	60
Dynamic m d=10	0.79	226	367	429	512	120	60
Incremental Distance d=0.5	0.78	177	349	413	509	122	36
Incremental Distance d=1	0.77	168	323	421	499	126	36
Incremental Distance d=10	0.73	163	312	380	444	158	36
Incremental Distance m h d=0.5	0.75	171	327	396	482	148	45
Incremental Distance m h d=1	0.74	180	325	395	473	153	45
Incremental Distance m h d=10	0.72	167	306	363	433	176	45
Incremental Distance m d=0.5	0.76	172	332	400	499	135	51
Incremental Distance m d=1	0.76	162	315	411	490	137	51
Incremental Distance m d=10	0.74	159	314	382	463	144	51
Incremental Activity m h d=0.5	0.77	206	345	422	501	138	42
Incremental Activity m h d=1	0.75	161	312	388	485	146	42
Incremental Activity m h d=10	0.72	166	314	368	445	168	42
Incremental Distance r m h d=0.5	0.78	165	341	424	517	119	42
Incremental Distance r m h d=1	0.78	193	348	430	530	120	42
Incremental Distance r m h d=10	0.78	181	351	431	527	120	42
Incremental Activity r d=0.5	**0.80**	**185**	**359**	**446**	**533**	**104**	**36**
Incremental Activity r d=1	0.79	200	347	428	526	112	36
Incremental Activity r d=10	**0.79**	**188**	**343**	**418**	**522**	**117**	**36**
Top-down d = 0.5	**0.72**	**150**	**283**	**359**	**442**	**171**	**48**
Top-down d = 1	0.71	144	281	349	437	174	48
Top-down d = 10	0.68	135	264	327	395	197	60
Top-down d = 100	0.70	154	276	332	399	183	60
Top-down d = 1000	**0.71**	**128**	**235**	**277**	**393**	**152**	**60**
Full Bottom Layers	**0.75**	**178**	**275**	**303**	**397**	**140**	**42**
Pairs	**0.74**	**128**	**268**	**345**	**451**	**131**	**42**

A.4. Normalized penalty MAX-SAW w=1

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.83**	**278**	**378**	**422**	**502**	**73**	**72**
Propagator	**0.76**	**157**	**230**	**288**	**390**	**116**	**30**
Dynamic d=0.5	0.84	249	329	377	477	73	30
Dynamic d=1	0.83	249	321	377	472	81	30
Dynamic d=10	0.76	228	296	323	415	122	60
Dynamic m h d=0.5	**0.84**	**265**	**376**	**410**	**491**	**65**	**60**
Dynamic m h d=1	0.84	268	378	410	490	65	60
Dynamic m h d=10	**0.84**	**265**	**375**	**411**	**491**	**65**	**60**
Dynamic m d=0.5	0.85	268	382	419	502	63	60
Dynamic m d=1	0.85	270	382	421	502	63	60
Dynamic m d=10	0.85	267	380	419	500	63	60
Incremental Distance d=0.5	0.86	248	351	393	498	56	30
Incremental Distance d=1	0.85	238	330	384	490	67	30
Incremental Distance d=10	0.83	218	311	361	466	75	30
Incremental Distance m h d=0.5	0.84	226	330	381	474	68	30
Incremental Distance m h d=1	0.83	230	309	368	465	74	30
Incremental Distance m h d=10	0.80	223	293	345	426	90	30
Incremental Distance m d=0.5	0.86	250	353	395	500	55	30
Incremental Distance m d=1	0.85	241	332	387	488	63	30
Incremental Distance m d=10	0.84	222	327	372	469	68	30
Incremental Activity m h d=0.5	0.84	239	326	383	478	67	30
Incremental Activity m h d=1	0.83	227	309	371	479	71	30
Incremental Activity m h d=10	0.81	223	301	349	436	93	30
Incremental Distance r m h d=0.5	0.86	256	348	397	492	58	30
Incremental Distance r m h d=1	0.86	241	342	406	495	60	30
Incremental Distance r m h d=10	0.85	228	324	386	492	63	30
Incremental Activity r d=0.5	**0.87**	**251**	**359**	**422**	**507**	**47**	**30**
Incremental Activity r d=1	0.86	237	346	408	504	52	30
Incremental Activity r d=10	**0.86**	**248**	**346**	**399**	**498**	**59**	**30**
Top-down d = 0.5	**0.83**	**243**	**333**	**388**	**459**	**74**	**30**
Top-down d = 1	0.83	233	324	392	467	80	30
Top-down d = 10	0.77	214	284	323	421	112	51
Top-down d = 100	0.75	230	287	321	398	136	51
Top-down d = 1000	**0.77**	**190**	**280**	**325**	**400**	**107**	**51**
Full Bottom Layers	**0.78**	**165**	**247**	**333**	**423**	**99**	**30**
Pairs	**0.77**	**187**	**298**	**336**	**411**	**107**	**69**

A.5. Normalized penalty MAX-SAW w=5

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.81**	**264**	**367**	**414**	**465**	**88**	**84**
Propagator	**0.65**	**141**	**217**	**256**	**321**	**207**	**84**
Dynamic d=0.5	0.76	225	305	354	429	133	63
Dynamic d=1	0.75	226	297	349	410	137	63
Dynamic d=10	0.69	213	266	294	358	184	84
Dynamic m h d=0.5	**0.78**	**235**	**337**	**378**	**443**	**108**	**87**
Dynamic m h d=1	0.78	236	339	379	444	108	87
Dynamic m h d=10	**0.78**	**235**	**338**	**379**	**441**	**108**	**87**
Dynamic m d=0.5	0.79	246	352	393	459	100	87
Dynamic m d=1	0.79	245	352	393	457	100	87
Dynamic m d=10	0.79	241	352	392	459	100	87
Incremental Distance d=0.5	0.77	228	317	353	423	118	69
Incremental Distance d=1	0.75	215	301	344	419	132	75
Incremental Distance d=10	0.72	198	273	330	386	151	78
Incremental Distance m h d=0.5	0.75	216	302	339	396	134	72
Incremental Distance m h d=1	0.74	219	289	336	401	144	75
Incremental Distance m h d=10	0.72	200	273	328	388	162	78
Incremental Distance m d=0.5	0.77	231	321	359	425	116	69
Incremental Distance m d=1	0.76	217	306	356	425	127	75
Incremental Distance m d=10	0.74	210	292	346	405	143	78
Incremental Activity m h d=0.5	0.74	218	294	351	404	137	78
Incremental Activity m h d=1	0.73	216	291	335	399	146	78
Incremental Activity m h d=10	0.71	196	265	313	381	164	78
Incremental Distance r m h d=0.5	0.76	216	299	352	416	125	75
Incremental Distance r m h d=1	0.75	229	301	356	411	136	84
Incremental Distance r m h d=10	0.72	204	277	337	400	148	93
Incremental Activity r d=0.5	**0.77**	**219**	**293**	**351**	**427**	**111**	**72**
Incremental Activity r d=1	0.77	220	319	358	426	121	75
Incremental Activity r d=10	**0.73**	**206**	**296**	**344**	**401**	**142**	**90**
Top-down d = 0.5	**0.76**	**227**	**301**	**350**	**415**	**130**	**63**
Top-down d = 1	0.75	232	299	359	413	134	63
Top-down d = 10	0.69	222	265	294	359	187	81
Top-down d = 100	0.66	206	256	283	337	214	90
Top-down d = 1000	**0.69**	**181**	**263**	**298**	**348**	**172**	**102**
Full Bottom Layers	**0.58**	**129**	**194**	**233**	**298**	**232**	**156**
Pairs	**0.69**	**161**	**288**	**321**	**362**	**165**	**111**

A.6. Normalized solve time PB10

Solver	**Average score**	**Best**	**At least 0.95**	**At least 0.9**	**At least 0.8**	**Less than 0.5**	**No solution**
Encoder	**0.12**	**0**	**0**	**0**	**0**	**137**	**27**
Propagator	**0.71**	**0**	**24**	**46**	**98**	**28**	**22**
Dynamic d=0.5	0.68	1	25	43	94	36	26
Dynamic d=1	0.67	1	23	41	93	36	26
Dynamic d=10	0.69	0	23	40	94	32	25
Dynamic m h d=0.5	**0.67**	**0**	**22**	**37**	**93**	**34**	**25**
Dynamic m h d=1	0.69	1	22	38	96	33	25
Dynamic m h d=10	**0.69**	**0**	**23**	**38**	**95**	**32**	**25**
Dynamic m d=0.5	0.67	1	22	34	94	36	26
Dynamic m d=1	0.67	0	23	38	95	36	26
Dynamic m d=10	0.68	0	23	38	94	34	26
Incremental Distance d=0.5	0.66	0	14	37	91	36	25
Incremental Distance d=1	0.68	0	17	38	93	34	23
Incremental Distance d=10	0.68	0	19	38	93	34	25
Incremental Distance m h d=0.5	0.66	0	14	35	89	36	24
Incremental Distance m h d=1	0.68	1	15	39	94	34	24
Incremental Distance m h d=10	0.68	0	15	37	91	36	24
Incremental Distance m d=0.5	0.66	0	15	33	91	36	25
Incremental Distance m d=1	0.67	0	15	34	91	34	23
Incremental Distance m d=10	0.67	0	16	34	92	34	25
Incremental Activity m h d=0.5	0.67	0	17	40	92	35	24
Incremental Activity m h d=1	0.68	1	17	40	92	33	24
Incremental Activity m h d=10	0.69	1	20	41	94	33	24
Incremental Distance r m h d=0.5	0.67	1	16	36	94	37	25
Incremental Distance r m h d=1	0.67	0	14	33	92	36	24
Incremental Distance r m h d=10	0.72	7	20	46	103	30	22
Incremental Activity r d=0.5	**0.69**	**0**	**17**	**44**	**97**	**33**	**24**
Incremental Activity r d=1	0.69	1	17	45	97	33	25
Incremental Activity r d=10	**0.71**	**5**	**23**	**49**	**101**	**31**	**23**
Top-down d = 0.5	**0.69**	**11**	**28**	**88**	**92**	**37**	**26**
Top-down d = 1	0.70	16	31	89	94	33	26
Top-down d = 10	0.70	12	31	87	95	33	26
Top-down d = 100	0.71	7	30	88	94	31	24
Top-down d = 1000	**0.73**	**11**	**30**	**87**	**100**	**28**	**23**
Full Bottom Layers	**0.40**	**0**	**0**	**0**	**0**	**64**	**33**
Pairs	**0.69**	**16**	**74**	**78**	**85**	**39**	**24**

A.7. Normalized solve time PB15

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.41**	**19**	**20**	**25**	**45**	**218**	**74**
Propagator	**0.44**	**40**	**61**	**68**	**83**	**207**	**78**
Dynamic Distance d=0.5	0.41	7	12	14	30	221	71
Dynamic Distance d=1	0.40	5	11	14	29	202	66
Dynamic Distance d=10	0.41	4	12	18	47	207	73
Dynamic Distance m h d=0.5	**0.40**	**6**	**10**	**13**	**30**	**215**	**75**
Dynamic Distance m h d=1	0.39	0	3	5	23	214	72
Dynamic Distance m h d=10	**0.41**	**9**	**15**	**22**	**44**	**215**	**73**
Dynamic Distance m d=0.5	0.40	2	10	12	28	223	75
Dynamic Distance m d=1	0.40	3	9	11	27	202	70
Dynamic Distance m d=10	0.40	4	9	16	45	209	77
Incremental Distance d=0.5	0.39	4	8	13	27	223	68
Incremental Distance d=1	0.39	6	11	16	27	228	69
Incremental Distance d=10	0.41	3	5	10	29	201	69
Incremental Distance m h d=0.5	0.39	5	11	13	31	223	67
Incremental Distance m h d=1	0.40	4	10	12	27	212	62
Incremental Distance m h d=10	0.40	5	6	12	29	214	73
Incremental Distance m d=0.5	0.39	3	8	13	26	221	68
Incremental Distance m d=1	0.38	2	11	16	26	228	69
Incremental Distance m d=10	0.41	2	5	11	31	203	69
Incremental Activity m h d=0.5	0.38	6	12	16	27	223	75
Incremental Activity m h d=1	0.38	6	10	13	23	228	73
Incremental Activity m h d=10	0.37	6	8	12	26	234	71
Incremental Distance r m h d=0.5	0.42	10	15	21	37	204	61
Incremental Distance r m h d=1	0.42	10	12	17	44	194	71
Incremental Distance r m h d=10	0.43	13	17	21	46	191	65
Incremental activity r d=0.5	**0.39**	**12**	**15**	**16**	**34**	**216**	**76**
Incremental activity r d=1	0.41	10	21	27	42	211	74
Incremental activity r d=10	**0.43**	**10**	**15**	**27**	**45**	**199**	**67**
Top-down d = 0.5	**0.36**	**5**	**9**	**12**	**22**	**229**	**84**
Top-down d = 1	0.36	4	7	10	25	227	77
Top-down d = 10	0.38	3	12	17	27	229	76
Top-down d = 100	0.45	40	57	64	80	201	77
Top-down d = 1000	**0.45**	**35**	**48**	**60**	**79**	**203**	**77**
Full Bottom Layers	**0.32**	**3**	**3**	**5**	**10**	**274**	**72**
Pairs	**0.45**	**23**	**28**	**37**	**58**	**186**	**71**

A.8. Normalized solve time d_n_k

Solver	**Average score**	**Best**	**At least 0.95**	**At least 0.9**	**At least 0.8**	**Less than 0.5**	**No solution**
Encoder	**0.37**	**4**	**4**	**4**	**9**	**197**	**26**
Propagator	**0.63**	**48**	**67**	**75**	**102**	**83**	**34**
Dynamic Distance d=0.5	0.62	35	53	64	80	78	27
Dynamic Distance d=1	0.64	38	55	64	91	74	26
Dynamic Distance d=10	0.65	34	49	57	94	72	26
Dynamic Distance m h d=0.5	**0.65**	**47**	**62**	**69**	**90**	**78**	**24**
Dynamic Distance m h d=1	0.64	42	52	65	92	76	25
Dynamic Distance m h d=10	**0.69**	**49**	**66**	**80**	**108**	**66**	**22**
Dynamic Distance m d=0.5	0.63	38	55	66	86	73	27
Dynamic Distance m d=1	0.63	37	47	59	89	80	26
Dynamic Distance m d=10	0.67	44	58	72	106	69	24
Incremental Distance d=0.5	0.56	33	34	50	79	111	31
Incremental Distance d=1	0.55	27	27	48	78	108	33
Incremental Distance d=10	0.58	33	35	54	83	98	33
Incremental Distance m h d=0.5	0.55	30	30	47	79	110	31
Incremental Distance m h d=1	0.56	35	36	53	79	107	32
Incremental Distance m h d=10	0.59	32	35	54	84	95	31
Incremental Distance m d=0.5	0.56	34	34	50	83	105	31
Incremental Distance m d=1	0.54	25	26	45	72	115	32
Incremental Distance m d=10	0.58	29	30	52	85	94	33
Incremental Activity m h d=0.5	0.56	30	30	45	76	103	33
Incremental Activity m h d=1	0.57	36	38	53	84	102	32
Incremental Activity m h d=10	0.58	30	32	52	78	91	33
Incremental Distance r m h d=0.5	0.59	32	34	43	85	93	30
Incremental Distance r m h d=1	0.61	35	39	51	91	90	29
Incremental Distance r m h d=10	0.66	47	53	71	108	75	27
Incremental activity r d=0.5	**0.59**	**34**	**36**	**46**	**86**	**90**	**30**
Incremental activity r d=1	0.60	27	28	40	85	89	29
Incremental activity r d=10	**0.64**	**40**	**43**	**59**	**102**	**72**	**28**
Top-down d = 0.5	**0.59**	**58**	**66**	**69**	**89**	**98**	**33**
Top-down d = 1	0.59	65	71	76	88	100	34
Top-down d = 10	0.61	59	64	71	92	89	34
Top-down d = 100	0.64	65	79	87	110	85	32
Top-down d = 1000	**0.66**	**73**	**88**	**92**	**119**	**79**	**32**
Full Bottom Layers	**0.27**	**6**	**6**	**6**	**7**	**196**	**69**
Pairs	**0.59**	**71**	**73**	**87**	**106**	**102**	**38**

A.9. Normalized solve time Elffers

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.38**	**45**	**45**	**47**	**56**	**179**	**102**
Propagator	**0.24**	**19**	**24**	**33**	**46**	**222**	**159**
Dynamic d=0.5	0.27	19	20	21	33	221	100
Dynamic d=1	0.26	20	20	23	34	222	98
Dynamic d=10	0.24	20	22	26	41	223	100
Dynamic m h d=0.5	**0.30**	**18**	**21**	**22**	**39**	**207**	**101**
Dynamic m h d=1	0.30	14	15	19	35	205	98
Dynamic m h d=10	**0.32**	**28**	**31**	**37**	**55**	**203**	**100**
Dynamic m d=0.5	0.30	16	19	21	40	207	100
Dynamic m d=1	0.31	16	17	21	38	206	99
Dynamic m d=10	0.32	20	23	27	46	206	99
Incremental Distance d=0.5	0.20	15	15	15	21	231	131
Incremental Distance d=1	0.20	18	18	18	22	228	133
Incremental Distance d=10	0.22	16	17	19	30	226	130
Incremental Distance m h d=0.5	0.20	12	13	13	20	233	132
Incremental Distance m h d=1	0.20	15	15	15	23	232	132
Incremental Distance m h d=10	0.22	18	18	21	36	229	141
Incremental Distance m d=0.5	0.20	15	16	16	21	233	131
Incremental Distance m d=1	0.19	9	9	9	18	234	133
Incremental Distance m d=10	0.22	15	16	18	30	224	130
Incremental Activity m h d=0.5	0.20	14	14	14	19	236	130
Incremental Activity m h d=1	0.20	11	11	11	19	232	131
Incremental Activity m h d=10	0.23	19	19	22	34	224	139
Incremental Distance r m h d=0.5	0.27	20	20	23	33	209	125
Incremental Distance r m h d=1	0.28	29	31	35	47	204	127
Incremental Distance r m h d=10	0.25	21	22	24	34	212	131
Incremental Activity r d=0.5	**0.28**	**31**	**32**	**35**	**44**	**205**	**124**
Incremental Activity r d=1	0.28	26	28	32	43	209	126
Incremental Activity r d=10	**0.26**	**23**	**27**	**29**	**39**	**215**	**127**
Top-down d = 0.5	**0.19**	**15**	**16**	**16**	**22**	**241**	**130**
Top-down d = 1	0.21	20	20	20	26	229	131
Top-down d = 10	0.21	21	21	21	29	231	128
Top-down d = 100	0.24	19	26	31	42	224	120
Top-down d = 1000	**0.25**	**21**	**28**	**37**	**51**	**220**	**121**
Full Bottom Layers	**0.34**	**52**	**52**	**52**	**57**	**197**	**119**
Pairs	**0.20**	**9**	**10**	**11**	**21**	**229**	**167**

A.10. Normalized solve time Elffers$_1$

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.17**	**3**	**3**	**4**	**6**	**150**	**102**
Propagator	**0.26**	**8**	**13**	**22**	**32**	**138**	**98**
Dynamic Distance d=0.5	0.21	2	3	4	14	138	100
Dynamic Distance d=1	0.22	2	2	5	14	139	98
Dynamic Distance d=10	0.23	3	5	9	21	140	100
Dynamic Distance m h d=0.5	**0.21**	**3**	**5**	**5**	**12**	**141**	**101**
Dynamic Distance m h d=1	0.23	3	4	6	14	139	98
Dynamic Distance m h d=10	**0.24**	**10**	**12**	**17**	**26**	**140**	**100**
Dynamic Distance m d=0.5	0.21	2	4	5	14	138	100
Dynamic Distance m d=1	0.22	2	2	5	14	140	99
Dynamic Distance m d=10	0.23	3	6	9	19	140	99
Incremental Distance d=0.5	0.19	1	1	1	3	148	102
Incremental Distance d=1	0.18	0	0	0	4	146	104
Incremental Distance d=10	0.22	1	2	4	13	141	102
Incremental Distance m h d=0.5	0.19	1	2	2	5	147	102
Incremental Distance m h d=1	0.19	0	0	0	4	147	102
Incremental Distance m h d=10	0.21	1	1	4	16	147	104
Incremental Distance m d=0.5	0.19	2	3	3	4	148	102
Incremental Distance m d=1	0.18	0	0	0	4	146	104
Incremental Distance m d=10	0.22	1	2	4	14	140	102
Incremental Activity m h d=0.5	0.18	3	3	3	4	151	101
Incremental Activity m h d=1	0.19	0	0	0	4	146	101
Incremental Activity m h d=10	0.23	3	3	6	16	139	100
Incremental Distance r m h d=0.5	0.29	7	7	10	17	125	95
Incremental Distance r m h d=1	0.31	14	16	20	28	121	95
Incremental Distance r m h d=10	0.27	8	9	11	19	128	94
Incremental activity r d=0.5	**0.30**	**13**	**14**	**17**	**24**	**122**	**96**
Incremental activity r d=1	0.31	11	13	17	26	124	96
Incremental activity r d=10	**0.28**	**8**	**12**	**14**	**21**	**130**	**94**
Top-down d = 0.5	**0.17**	**1**	**2**	**2**	**4**	**157**	**103**
Top-down d = 1	0.19	3	3	3	6	149	104
Top-down d = 10	0.20	3	3	3	10	148	104
Top-down d = 100	0.24	5	12	17	24	139	99
Top-down d = 1000	**0.27**	**6**	**13**	**22**	**32**	**136**	**95**
Full Bottom Layers	**0.13**	**1**	**1**	**1**	**1**	**168**	**119**
Pairs	**0.21**	**3**	**4**	**5**	**7**	**142**	**100**

A.11. Normalized solve time Elffers$_2$

Solver	Average score	Best	At least 0.95	At least 0.9	At least 0.8	Less than 0.5	No solution
Encoder	**0.74**	**42**	**42**	**43**	**50**	**29**	**0**
Propagator	**0.21**	**11**	**11**	**11**	**14**	**84**	**61**
Dynamic d=0.5	0.37	17	17	17	19	83	0
Dynamic d=1	0.32	18	18	18	20	83	0
Dynamic d=10	0.24	17	17	17	20	83	0
Dynamic m h d=0.5	**0.45**	**15**	**16**	**17**	**27**	**66**	**0**
Dynamic m h d=1	0.42	11	11	13	21	66	0
Dynamic m h d=10	**0.46**	**18**	**19**	**20**	**29**	**63**	**0**
Dynamic m d=0.5	0.46	14	15	16	26	69	0
Dynamic m d=1	0.45	14	15	16	24	66	0
Dynamic m d=10	0.46	17	17	18	27	66	0
Incremental Distance d=0.5	0.23	14	14	14	18	83	29
Incremental Distance d=1	0.24	18	18	18	18	82	29
Incremental Distance d=10	0.22	15	15	15	17	85	28
Incremental Distance m h d=0.5	0.22	11	11	11	15	86	30
Incremental Distance m h d=1	0.23	15	15	15	19	85	30
Incremental Distance m h d=10	0.23	17	17	17	20	82	37
Incremental Distance m d=0.5	0.22	13	13	13	17	85	29
Incremental Distance m d=1	0.21	9	9	9	14	88	29
Incremental Distance m d=10	0.23	14	14	14	16	84	28
Incremental Activity m h d=0.5	0.22	11	11	11	15	85	29
Incremental Activity m h d=1	0.22	11	11	11	15	86	30
Incremental Activity m h d=10	0.22	16	16	16	18	85	39
Incremental Distance r m h d=0.5	0.22	13	13	13	16	84	30
Incremental Distance r m h d=1	0.23	15	15	15	19	83	32
Incremental Distance r m h d=10	0.22	13	13	13	15	84	37
Incremental Activity r d=0.5	**0.24**	**18**	**18**	**18**	**20**	**83**	**28**
Incremental Activity r d=1	0.23	15	15	15	17	85	30
Incremental Activity r d=10	**0.23**	**15**	**15**	**15**	**18**	**85**	**33**
Top-down d = 0.5	**0.23**	**14**	**14**	**14**	**18**	**84**	**27**
Top-down d = 1	0.24	17	17	17	20	80	27
Top-down d = 10	0.23	18	18	18	19	83	24
Top-down d = 100	0.22	14	14	14	18	85	21
Top-down d = 1000	**0.23**	**15**	**15**	**15**	**19**	**84**	**26**
Full Bottom Layers	**0.70**	**51**	**51**	**51**	**56**	**29**	**0**
Pairs	**0.20**	**6**	**6**	**6**	**14**	**87**	**67**