UNIVERSITAT POLITÈCNICA DE CATALUNYA

SOFTWARE DEPARTMENT

PhD in Computing

Solving hard industrial combinatorial problems with SAT

Ignasi Abío Roig

Advisors:

Robert Nieuwenhuis

and

Albert Oliveras

and

Enric Rodríguez Carbonell

Barcelona, March 2013

Abstract

The topic of this thesis is the development of SAT-based techniques and tools for solving industrial combinatorial problems. First, it describes the architecture of state-of-the-art SAT and SMT Solvers based on the classical DPLL procedure. These systems can be used as black boxes for solving combinatorial problems. However, sometimes we can increase their efficiency with slight modifications of the basic algorithm. Therefore, the study and development of techniques for adjusting SAT Solvers to specific combinatorial problems is the first goal of this thesis.

Namely, SAT Solvers can only deal with propositional logic. For solving general combinatorial problems, two different approaches are possible:

- Reducing the complex constraints into propositional clauses.
- Enriching the SAT Solver language.

The first approach corresponds to encoding the constraint into SAT. The second one corresponds to using propagators, the basis for SMT Solvers. Regarding the first approach, in this document we improve the encoding of two of the most important combinatorial constraints: cardinality constraints and pseudo-Boolean constraints. After that, we present a new mixed approach, called Lazy Decomposition, which combines the advantages of encodings and propagators.

The other part of the thesis consists in uses these theoretical improvements in industrial combinatorial problems. We give a method for efficiently scheduling some professional sport leagues with SAT. The results are promising and show that a SAT approach is valid for these problems.

However, the chaotical behavior of CDCL-based SAT Solvers due to VSIDS heuristics makes it difficult to obtain a similar solution for two similar problems. This may be inconvenient in real-world problems, since a user expects similar solutions when it makes slight modifications to the problem specification. In order to overcome this limitation, we have studied and solved the close solution problem, i.e., the problem of quickly finding a close solution when a similar problem is considered.

Contents

1 Introduction

1.1 Propositional Satisfiability

The propositional satisfiability problem (SAT) consists in finding a model of a CNF propositional formula, this is, a set of Boolean clauses. The first steps for solving SAT were made in the 60's in the context of automated deduction for proving theorems of first-order logic [Gil60]. These first systems were hugely improved by, first, Davis and Putnam [DP60] and, secondly, Loveland and Logemann [DLL62]. The resulting algorithm is known as DPLL procedure. However, it was only useful for small problems. Real-world industrial instances, with millions of variables and clauses, were still far from being solved.

Some years later, SAT was found to be the first NP-complete problem [Coo71]. However, the DPLL method did not significantly improve until the 90's. The discovery of *VSIDS heuristics* [MSS99a, MMZ$^+$01], *Restarts* [MMZ$^+$01], *conflict analysis* [MSS99a] and more efficient structures such as the *two-watched literal lists* [MMZ$^+$01] constituted the beginning of modern SAT Solvers. The resulting algorithm is called *Conflict Driven Clause Learning* (CDCL) procedure.

The basic CDCL algorithm has been deeply improved over the last few years. For instance, preprocessing [EB05] and inprocessing [JHB12] methods make the current SAT Solvers some orders of magnitude faster than ten years ago: problems with hundreds of thousands of variables and millions of clauses are now routinely solved in few seconds.

The flowering of SAT Solver procedures has opened some new research fields: SMT Solvers [NOT06], Lazy Clause Generation [OSC09], MaxSAT [AM06], etc.

1.2 Applications of SAT

The advances of SAT Solving in the last 15 years are closely related with the successful application of propositional satisfiability in industrial problems of Electronic Design Automation (EDA). SAT Solving has efficiently been applied to many EDA

areas, some of them explained in the next paragraphs.

One of the most important SAT applications in EDA is the *Equivalence Checking* problem [KPKG02]. This problem consists in proving that two circuits are equivalent, this is, they generate the same output for any given input. Bjesse and Claessen [BC00] improved van Eijk's algorithm for Sequential Equivalence Checking [vE98] and encoded the problem into SAT.

Another important application of SAT to EDA is the *Model Checking* problem [CE82], i.e., testing if a model satisfies the given specification. As an example, Biere et Al. [BCC$^+$99] described an algorithm for bounded model checking based on SAT instead of classical BDD-based algorithms [McM92].

Another EDA field where SAT has been applied is *Automatic Test Pattern Generation* (ATPG) [JG03]. ATPG problem consists in, given a circuit and an incorrect model of it, searching for an input (*pattern*) on which the incorrect model and the circuit show a different output. Larrabe [Lar92] presented a method for solving this problem. A related problem is *Delay Fault Testing* [CG96], which searches for patterns when the faults of the circuits are caused due to the delays of the logical gates.

Other important applications in EDA are *Logic Synthesis* [EC93], this is, the field that designs a logical circuit that satisfies the given specifications; *FPGA routing* (Field-Programmable Gate Arrays Routing) [NSR99], i.e., the problem of finding a routing of wires in a FPGA; *Redundancy Identification* [KS$^+$97a], this is, to identify some redundant parts of a circuit; etc.

However, not only EDA industry has taken advantage of the SAT technology: it has also been applied to many other areas. An important one is Software Verification, i.e., to check whether a computer program is correct [DKW08]. Other examples are Bioinformatics [BJMA06], Statistical Physics [MSL92], etc.

1.3 Solving Hard Industrial Combinatorial Problems with SAT

Recently, SAT technology has been applied to solve industrial combinatorial problems. *Combinatorial Problems* consists in, given a set of variables with finite domains and a set of constraints over them, finding an assignment that satisfies all constraints. The focus here will be on *industrial* problems, i.e., coming from real-world companies, as opposed to random problems, which are randomly generated, or crafted problems, which are manually cooked.

Examples of hard industrial combinatorial problems include *Timetabling Problems* [AAN12], *Cumulative Scheduling* [SFSW09], *Sport Scheduling* [RT08], etc.

This thesis deals with the development of SAT-based techniques and tools for solving industrial combinatorial problems. It includes the first (to our knowledge) SAT-based method for solving professional sportive scheduling problems.

We start by describing the architecture of any state-of-the-art SAT and SMT Solver based on the classic DPLL procedure. SAT and SMT Solvers can be used

as *black boxes* for solving combinatorial problems. However, SAT Solvers need to be modified for increasing their efficiency in some kind of problems. Therefore, the study and development of techniques for adjusting SAT Solver to combinatorial problems is the first goal of this thesis.

Namely, SAT Solvers can only deal with propositional logic. For solving combinatorial problems, two different approaches are possible:

- Reduce the complex constraints into propositional clauses.
- Enrich the SAT Solver language.

The first approach corresponds to *encoding* the constraint into SAT. The second one corresponds to using *propagators*, the basis for SMT Solvers.

In this document we improve the encoding of two of the most important combinatorial constraints: cardinality constraints and Pseudo-Boolean constraints. After that, we present a new mixed approach, called *Lazy Decomposition*, which combines the advantages of encodings and propagators.

The other part of the thesis consists in using these theoretical improvements in industrial combinatorial problems. We have given a method for efficiently scheduling some professional sport leagues with SAT. The results are promising and show that a SAT approach is valid for these problems.

However, the chaotical behavior of CDCL-based SAT Solvers due to VSIDS heuristics make it difficult to obtain a similar solution for two similar problems. This may be inconvenient in real-world problems, since a company expects similar solutions when it makes slight modifications to the problem specification. In order to overcome this limitation, we have studied and resolved the *close solution* problem, i.e., the problem of quickly finding a similar solution when a similar problem is considered.

1.4 Contributions of this Document

Contributions are present both in the theoretical and in the applied part. From the theoretical point of view, the main contributions of this thesis are:

- A new encoding for cardinality constraints that requires approximately half of the auxiliary variables than the best method know so far. Experimentally, problems encoded with this method can be solved faster than with the state-of-the-art encodings.
- A new encoding for Pseudo-Boolean constraints using BDDs. Experimentally, it has been shown to be the best encoding for these constraints. Moreover, we have found a new and much simpler proof that some Pseudo-Boolean constraints do not admit polynomial BDDs in any order.
- An arc-consistent SAT encoding of BDDs for monotonic functions that uses one binary and one ternary clauses per node (the standard if-then-else encoding

for BDDs requires six ternary clauses per node). Moreover, this translation works for any BDD variable ordering.

- A general method for dealing with complex constraints within SAT called *lazy decomposition*, which experimentally is nearly as good as the best of the possible approaches (eager encoding and global propagation approaches), and often better.

In the applied part, the main contributions are:

- A simple and very efficient approach for solving the close-solution problem: our method frequently finds the optimal solution in 25% of the time the SAT solver took in solving the original problem.
- The first (to our knowledge) encoding of the professional sport league scheduling problem into SAT.
- A comparison of the different encodings into SAT for *Symmetric-all-different* and *All-different* constraints with industrial benchmarks.

1.5 Outline of this Document

The thesis is divided into three different parts: Chapter 2 contains a background section about SAT and SMT Solving. Chapters 3 to 5 conform the mainly-theoretical part and deal with cardinality and PB constraints. Finally, it contains the applied part (Chapters 6 and 7) that deals with sport league scheduling and close-solution problems.

More specifically, this document will be structured as follows:

- In Chapter 2 we describe the classical and modern SAT Solvers designs.
- Chapter 3 contains in first place a description of the best encoding for cardinality constraints, this is, the encoding of cardinality constraints using cardinality networks. This encoding is slightly improved by allowing to encode constraints of any size instead of power-of-two-sized constraints. Next, we show a direct method for building cardinality networks without auxiliary variables but, usually, with a huge amount of clauses. Then, a mixed method is presented, combining both approaches. This method uses many fewer variables than the original method and fewer clauses. Finally, an experimental section compares this approach with the previous encoding using cardinality networks.
- In Chapter 4 we study the encoding of Pseudo-Boolean constraints through Reduced Ordered Binary Decision Diagrams (ROBDDs). This chapter shows an example of a family of PB constraints whose ROBDD are exponential in any ordering. Next, it contains a polynomial consistent and a polynomial arc-consistent BDD-based encodings, using fewer variables and clauses that the

state-of-the-art encodings. The chapter explains a better encoding for BDDs into SAT and a polynomial algorithm for representing a PB constraint into a BDD. Finally, it contains an experimental section comparing the proposed encodings with the state-of-the-art ones.

- Chapter 5 contains a new approach for dealing with complex constraints within SAT. It is similar to an SMT approach, but instead of generating (lazily) a naive encoding of the constraints, our method can lazily generate an encoding with auxiliary variables. In this way, our method decomposes only some part of the most useful constraints, achieving the advantages of the eager encoding (auxiliary constraints and better reasons) and SMT (fast handling of unused constraints) approaches.

- Chapter 6 deals with the close-solution problem, this is, given a model of a Boolean formula, to search for a similar model when few clauses are added to the formula. First we show that trivial solutions like relaunch the SAT Solver do not produce satisfactory results. Next, we present our method, and we experimentally prove that it behaves very well in practice.

- In Chapter 7, we present the problem of sportive league scheduling. We introduce an encoding into SAT for it and some modifications of the SAT Solver for dealing with it. We show that our method can solve real-world problems and that the proposed modifications improve the performance.

- Finally, we conclude in Chapter 8.

2
SAT Solving

This chapter summarizes the state-of-the-art methods for solving propositional problems, this is, SAT solving. Any expert in the area may skip it.

Section 2.1 contains the definitions and nomenclature used throughout all the document together with some concepts of transition systems, needed to present the SAT Solver algorithms. Section 2.2 contains the classical DPLL method for solving propositional problems. This method was largely improved by the CDCL algorithm, presented in Section 2.3. Section 2.4 outlines the implementation of the state-of-the-art SAT Solvers. Finally, Section 2.5 presents the structure of a standard SMT Solver.

2.1 Preliminaries

2.1.1 Basic Notions

Let $\mathcal{X} = \{x_1, x_2, \ldots\}$ be a fixed finite set of propositional *variables*. If $x \in \mathcal{X}$ then x and $\overline{x}$ are *positive and negative literals*, respectively. The *negation* of a literal l, written $\bar{l}$, denotes $\overline{x}$ if l is x, and x if l is $\overline{x}$. A *clause* is a disjunction of literals $\overline{x}_1 \vee \ldots \vee \overline{x}_p \vee x_{p+1} \vee \ldots \vee x_n$, sometimes written as $x_1 \wedge \ldots x_p \rightarrow x_{p+1} \vee \ldots x_n$. A *unit clause* is a clause consisting of a single literal. The *empty clause* is the clause that has no literals and will be denoted by $\Box$. A (CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \ldots \wedge C_n$, which sometimes will be written in set notation $\{C_1, \ldots, C_n\}$, or simply $C_1, \ldots, C_n$.

A (partial) *assignment* A is a set of literals such that $\{l, \bar{l}\} \subseteq A$ for no l, i.e., no contradictory literals appear. A literal l is *true* in A if $l \in A$, is *false* in A if $\bar{l} \in A$, and is *undefined* in A otherwise. Assignments are sometimes written as a set of pairs $x = v$, where v is 1 if x is true in A and 0 if x is false in A. A clause C is true in A if at least one of its literals is true in A, is false in A if all its literals are false in A, and is undefined in A otherwise. A formula F is true in A if all its clauses are true in A. In that case, A is a *model* of F. A formula F' is *entailed* by F if all the models of F are models of F'.

Given two clauses (called premises) of the form $p \vee C$ and $\neg p \vee D$ the *resolution* inference rule allows us to infer the clause $C \vee D$ (called conclusion):

$$\frac{p \vee C \qquad \neg p \vee D}{C \vee D}$$

The problem we are interested in is the *SAT problem*: given a formula F, decide whether there exists a model that satisfies F. Since a polynomial transformation for any arbitrary formula to an equisatisfiable CNF one exists (see [Tse68]), we will assume w.l.o.g. that F is in CNF. Systems that solve these problems are called *SAT-solvers*, and the main inference rule they implement is *unit propagation*: given a CNF formula F and an assignment A, find a clause in F such that all its literals are false in A except one, say l, which is undefined, add l to A and repeat the process until reaching a fixpoint.

In what follows, (possibly subscripted or primed) lowercase l *always* denotes literals. Similarly C and D always denote clauses, F and G denote formulas, and A and B denote assignments.

2.1.2 Transition Systems

In what follows, in order to describe a solving procedure for SAT, we will assume that a state of such procedure will be either the distinguished state *FailState* or a pair of the form $A \parallel F$, where F is a CNF formula and A is, essentially, an assignment. More precisely, A is a *sequence* of sequences of literals $A_0, A_1, A_2, \ldots, A_r$, such that

- $A_i \neq \emptyset$ for all $1 \leqslant i \leqslant r$ (however, A_0 may be empty),
- $A_i \cap A_j = \emptyset$ for all $0 \leqslant i < j \leqslant r$; and
- $A_0 \cup A_1 \cup \ldots \cup A_r$ does not contain a literal and its negation.

If $A = (A_0, A_1, \ldots, A_r)$, the state is said to be at *decision level* r. Similarly, a literal $l \in A_k$ (with $0 \leqslant k \leqslant r$) is said to be at *decision level* k. The first literals of every sequence A_i with $i > 0$ are called *decision literals*.

Assume

$$A = (A_0, A_1, \ldots, A_r), \text{ where } \begin{cases} A_0 = (l_1, l_2, \ldots, l_{n_0}), \\ A_1 = (l_{n_0+1}, l_{n_0+2}, \ldots, l_{n_1}), \\ \ldots, \\ A_r = (l_{n_{r-1}+1}, l_{n_{r-1}+2}, \ldots, l_{n_r}). \end{cases}$$

Then A can be represented by

$$A = l_1, l_2, \ldots, l_{n_0}, l^{\mathsf{d}}_{n_0+1}, l_{n_0+2}, \ldots, l_{n_r},$$

where l^{d} means that l is a decision literal. Notice that since $A_1, A_2, \ldots, A_r$ are not empty, there are exactly r decision literals, so this notation is not ambiguous. We

will frequently consider A just as a partial assignment, or as a set or conjunction of literals (and hence as a formula).

The *concatenation* of two sequences $A = l_1, l_2, \ldots, l_k$ and $B = l'_1, l'_2, \ldots, l'_{k'}$ will be denoted by simple juxtaposition AB, and is defined by

$$AB = l_1, l_2, \ldots, l_k, l'_1, l'_2, \ldots, l'_{k'}.$$

We will denote the empty sequence of literals (or the empty assignment) by $\emptyset$. We say that a clause C is *conflicting* in a state $A \parallel F$ if A entails $\neg C$, i.e., if C is false in the assignment A.

DPLL-based procedures are modeled by means of a set of states together with a binary relation $\Longrightarrow$ over these states, called the *transition relation*. As usual, the infix notation is used, writing $S \Longrightarrow S'$ instead of $(S, S') \in \Longrightarrow$. If $S \Longrightarrow S'$ we say that there is a *transition* from S to S'. Any sequence of transitions of the form $S_0 \Longrightarrow S_1,\ \ S_1 \Longrightarrow S_2,\ \ \ldots\ \ , S_{n-1} \Longrightarrow S_n$ is denoted by

$$S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \ldots\ \ \Longrightarrow S_n$$

and is called a *derivation*.

In what follows, transition relations will be defined by means of conditional *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. Such a transition is called an *application step* of the rule.

A *transition system* is a set of transition rules defined over some given set of states. Given a transition system R, the transition relation defined by R will be denoted by $\Longrightarrow_R$. If there is no transition from S by $\Longrightarrow_R$, we will say that S is *final* with respect to R. Examples of a transition system and a final state with respect to it can be found in Definition 2.1 and Example 2.3, respectively.

2.2 Classical SAT Solvers: DPLL Procedure

A very simple transition system for SAT solving is the classical DPLL algorithm [DLL62]. We give it here mainly for explanatory and historical reasons. The system can be defined as follows

Definition 2.1. *The* Classical DPLL system *is the transition system* `DPLL` *consisting of the following five transition rules.*

UnitPropagate :

$$A \parallel F, C \vee l \quad \Longrightarrow \quad A\, l \parallel F, C \vee l \quad \textbf{if} \begin{cases} C \text{ is false in } A, \\ l \text{ is undefined in } A. \end{cases}$$

PureLiteral :

$$A \parallel F \quad \Longrightarrow \quad A\, l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F, \\ \bar{l} \text{ occurs in no clause of } F, \\ l \text{ is undefined in } A. \end{cases}$$

Decide :

$$A \parallel F \quad \Longrightarrow \quad A\, l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \bar{l} \text{ occurs in a clause of } F, \\ l \text{ is undefined in } A. \end{cases}$$

Fail :

$$A \parallel F, C \quad \Longrightarrow \quad \mathit{FailState} \quad \textbf{if} \begin{cases} C \text{ is false in } A, \\ A \text{ contains no decision literals.} \end{cases}$$

Backtrack :

$$A\, l^{\mathsf{d}}\, B \parallel F, C \quad \Longrightarrow \quad A\, \bar{l} \parallel F, C \quad \textbf{if} \begin{cases} C \text{ is false in } A\, l^{\mathsf{d}}\, B, \\ B \text{ contains no decision literals.} \end{cases}$$

In what follows, we have included a more detailed explanation of each of the rules:

- UnitPropagate: to satisfy a CNF formula, all its clauses must be true. Hence, if there is a clause in F consisting of an undefined literal l and false literals (with respect the assignment A), then A must be extended with l to make that clause true.
- PureLiteral: a literal l is *pure* in F if it occurs in F while its negation does not. In this case, given a model of F either contains l or switching $\bar{l}$ for l is still a model of F. Hence, if A does not define l it can be extended to make l true.
- Decide: this rule represents a case split. An undefined literal l is chosen from F and added to A as a *decision literal*. This means that if the assignment $A\, l$ cannot be extended to a model of F, then the alternative extension $A\, \bar{l}$ must still be considered. This is done by means of the Backtrack rule.
- Fail: this rule detects a *conflicting clause* C and produces the *FailState* state whenever A contains no decision literals.
- Backtrack: this rule detects a conflicting clause C and the most recent decision literal l^{d}, and backtracks one *decision level*: l^{d} is replaced by $\bar{l}$ and any subsequent literals in the current assignment are removed. Note that $\bar{l}$ is annotated as a non-decision literal, since the other possibility l has already been explored.

The transition system DPLL can be used for deciding the satisfiability of an input formula F, as shown in this result from [NOT06]:

Theorem 2.2. *Let F be a CNF formula. Then, any derivation from the initial state $\emptyset \parallel F$ is finite, i.e., there exists n such that $\emptyset \parallel F \Longrightarrow_{\mathtt{DPLL}} S_1 \Longrightarrow_{\mathtt{DPLL}} S_2 \Longrightarrow_{\mathtt{DPLL}} \ldots \Longrightarrow_{\mathtt{DPLL}} S_n$; and S_n is a final state with respect to* DPLL. *Moreover, if F is unsatisfiable, the final state S_n is FailState, whereas if F is satisfiable the last state is of the form $S_n = A \parallel F$, where A is a model of F.*

Notice that in most of the situations more than one rule can be applied and, in virtue of the previous theorem, the system is correct no matter which rules it applies. However, in terms of efficiency, it is convenient to apply Backtrack and Fail when possible, and Decide only when no other rule can be applied.

Example 2.3. *The following is a derivation in the Classical DPLL system, with each transition annotated by the rule that makes it possible. To improve readability propositional variables are denoted by natural numbers.*

$$
\begin{array}{rcl@{\quad}l}
\emptyset & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(Decide)} \\
1^{\mathsf{d}} & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ \overline{2} & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ \overline{2}\ 3 & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ \overline{2}\ 3\ 4 & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(Backtrack)} \\
\overline{1} & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
\overline{1}\ 4 & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(Decide)} \\
\overline{1}\ 4\ \overline{3}^{\,\mathsf{d}} & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
\overline{1}\ 4\ \overline{3}^{\,\mathsf{d}}\ 2 & \| & \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{1}\vee\overline{3}\vee4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee4 &
\end{array}
$$

The last state of this derivation is final. The (total) assignment in it is a model of the formula. □

2.3 The CDCL Procedure

Classical DPLL system can be improved by modifying some of the rules and adding some more. For example, due to efficiency reasons the pure literal rule is normally only used as a preprocessing step; hence, this rule will not be considered in the following. Moreover, DPLL system can be largely improved by replacing backtracking with *backjumping*, a more general and more powerful non-chronological backtracking mechanism. Together with some further changes, the resulting system is called Conflict-Driven-Clause-Learning (CDCL) system.

The usefulness of this more sophisticated backtracking mechanism for CDCL solvers is perhaps best illustrated with another example of derivation in the Classical DPLL system.

Example 2.4.

$$
\begin{array}{rcl@{\quad}l}
\emptyset & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(Decide)} \\
1^{\mathsf{d}} & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(Decide)} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4 & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(Decide)} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}} & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \text{(Backtrack)} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5} & \| & \overline{1}\vee2,\ \overline{3}\vee4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathtt{DPLL}} \ldots
\end{array}
$$

Before the Backtrack *step, the clause* $6 \vee \overline{5} \vee \overline{2}$ *is conflicting: it is false in the assignment* $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6}$. Backtrack *will produce the assignment* $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5}$*, this is, the decisions* 1^{d} *and* 3^{d} *imply that* 5 *must be false. However, a detailed analysis of the conflict could reveal a more general relation.*

The conflict is a consequence of the unit propagation 2 *of the decision* 1^{d}*, together with the decision* 5^{d} *and its unit propagation* $\overline{6}$*. Therefore, one can infer that* 2 *is incompatible with the decision* 5^{d}*, i.e., that the given clause set entails* $\overline{2} \vee \overline{5}$*. Such entailed clause is called a* backjump clause *if its presence would have allowed a unit propagation at an earlier decision level.*

This is precisely what backjumping *does: given a backjump clause, it goes back to that level and adds the unit propagated literal. In our case, using* $\overline{2} \vee \overline{5}$ *as a backjump clause,* Backjump *goes to a state with first component* $1^{\mathsf{d}}\ 2\ \overline{5}$*.*

Notice that this state is better *than the state produced by* Backtrack*: in the latter one, we obtained that* $\overline{5}$ *is consequence of* 1^{d} *and* 3^{d}*, whereas in the former one we obtained that* $\overline{5}$ *is only consequence of* 1^{d}*.*

Backjumping is modeled with the following Backjump *rule, of which* Backtrack *is a particular case. In this rule, the clause* $C' \vee l'$ *is the backjump clause, where* l' *is the literal that can be unit propagated (in our example,* C' *is* $\overline{2}$ *and* l' *is* $\overline{5}$*).* □

Backjump :

$$A\ l^{\mathsf{d}}\ B \parallel F, C \implies A\ l' \parallel F, C \quad \textbf{if} \begin{cases} C \text{ is false in } A\ l^{\mathsf{d}}\ B, \text{and} \\ \text{there exists a clause } C' \vee l' \\ \text{entailed by } F \wedge C \text{ such that} \\ C' \text{ is false in } A, \\ l' \text{ is undefined in } A \text{ and} \\ l' \text{ or } \overline{l'} \text{ occurs in } F \vee A\ l^{\mathsf{d}}\ B. \end{cases}$$

At this point, let us explain the method for obtaining a backjump clause. We will assume that the system holds two properties: first, for every non-decision literal in the assignment, the system saves the reason why that literal was added to the assignment (this is, the clause that it propagated if it was added in a UnitPropagate step or the backjump clause if it was added in a Backjump step). Secondly, we assume the system does not apply Decide rule if either UnitPropagate, Backjump or Fail can be applied, which on the other hand is convenient for efficiency reasons.

The backjump clause search process is called *conflict analysis*. In order to illustrate how it works, let us consider this example:

Example 2.5. *Let us consider a formula* F *with, among other, these clauses:*

$$\overline{1} \vee \overline{3} \vee 4, \quad 2 \vee \overline{3} \vee \overline{5}, \quad 2 \vee 5 \vee 6, \quad \overline{3} \vee 5 \vee \overline{7}, \quad \overline{1} \vee \overline{6} \vee 8, \quad 5 \vee \overline{6} \vee \overline{8} \vee 9, \quad \overline{1} \vee 5 \vee \overline{9},$$

and assume the system is in the state

$$\ldots, 1, \ldots, \overline{2}, \ldots, 3^{\mathsf{d}}, 4, \overline{5}, 6, \overline{7}, 8, 9 \parallel F.$$

The system, as said, has the reasons why every non-decision literal is in the assignment. For instance,

$$\begin{array}{llllll} 4 & \rightsquigarrow & \overline{1} \vee \overline{3} \vee 4, & \overline{5} & \rightsquigarrow & 2 \vee \overline{3} \vee \overline{5}, \\ \overline{7} & \rightsquigarrow & \overline{3} \vee 5 \vee \overline{7}, & 8 & \rightsquigarrow & \overline{1} \vee \overline{6} \vee 8, \end{array} \qquad \begin{array}{lll} 6 & \rightsquigarrow & 2 \vee 5 \vee 6, \\ 9 & \rightsquigarrow & 5 \vee \overline{6} \vee \overline{8} \vee 9. \end{array}$$

The system has found the conflicting clause $\overline{1} \vee 5 \vee \overline{9}$. *The conflict analysis consists of successively applying resolution steps on these reasons until a clause with only one literal of the current decision level is obtained. In our example, we bold the literals of the current decision level. We start with the conflicting clause*

$$\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{9}}.$$

Since there are two literals from the current decision level (i.e., two bold literals), a resolution step is needed. The last literal from the assignment is 9*, and its reason is* $5 \vee \overline{6} \vee \overline{8} \vee 9$. *Therefore:*

$$\frac{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{9}}. \quad \mathbf{5} \vee \mathbf{\overline{6}} \vee \mathbf{\overline{8}} \vee \mathbf{9}}{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{6}} \vee \mathbf{\overline{8}}}$$

Now there are 3 bold literals, so another resolution step is needed. 8 is the latest bold literal in the assignment, so we use its reason in the next resolution step:

$$\frac{\dfrac{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{9}}. \quad \mathbf{5} \vee \mathbf{\overline{6}} \vee \mathbf{\overline{8}} \vee \mathbf{9}}{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{6}} \vee \mathbf{\overline{8}}} \qquad \overline{1} \vee \mathbf{\overline{6}} \vee \mathbf{8}}{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{6}}}$$

While more than one bold literal appears, the process continue:

$$\frac{\dfrac{\dfrac{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{9}}. \quad \mathbf{5} \vee \mathbf{\overline{6}} \vee \mathbf{\overline{8}} \vee \mathbf{9}}{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{6}} \vee \mathbf{\overline{8}}} \qquad \overline{1} \vee \mathbf{\overline{6}} \vee \mathbf{8}}{\overline{1} \vee \mathbf{5} \vee \mathbf{\overline{6}}} \qquad 2 \vee \mathbf{5} \vee \mathbf{6}}{\overline{1} \vee 2 \vee \mathbf{5}}$$

Finally, the obtained clause is a backjump clause: $\overline{1} \vee 2 \vee 5$.

Modern CDCL implementations add the backjump clauses to the formula [MSS99b]. These new clauses are called *learned clauses* or *lemmas*. This technique is usually called *conflict-driven clause learning*, and prevents reaching similar conflicts in the future. In Example 2.5, learning the clause $\overline{1} \vee 2 \vee 5$ will allow the application of UnitPropagate rule to any state whose assignment contains either 1 and $\overline{2}$, 1 and $\overline{5}$ or $\overline{2}$ and $\overline{5}$, so it will prevent any conflict caused by having 1, $\overline{2}$ and $\overline{5}$ in the assignment. Reaching such similar conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas has been shown to be very effective in improving performance.

Since a lemma is aimed at preventing future similar conflicts, when these conflicts are not very likely to be found again the lemma can be removed. In practice, a lemma is removed when its *relevance* (see, e.g., [BS97]) or its *activity level* drops below a certain threshold; the activity can be, e.g., the number of times it becomes a unit or a conflicting clause [GN02]. Therefore, CDCL systems have two additional rules:

Learn and Forget.

$$\begin{array}{l}\textsf{Learn}: \\ \quad A \parallel F \implies A \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \vee A, \\ C \text{ is entailed by } F. \end{cases} \\ \textsf{Forget}: \\ \quad A \parallel F, C \implies A \parallel F \quad \textbf{if} \quad C \text{ is entailed by } F. \end{array}$$

The last rule of the CDCL transition system is Restart [GSK98].

$$\begin{array}{l}\textsf{Restart}: \\ \quad A \parallel F \implies \emptyset \parallel F. \end{array}$$

It consists in removing the current assignment when the search is not making enough progress according to some measure. In these cases, the additional clauses added to the formula (together with the different literals picked by Decide due to the heuristics, see Section 2.4.1) allow the system to explore the search space in a different way. In practice, the combination of Learn and Restart has been shown to be a powerful tool (see [KS97b, MSG99, GSK98, BS00, KSMS11]). Moreover, from a theoretical point of view, restarts are also important: any Basic DPLL derivation to Fail is equivalent to a tree-like refutation by resolution; so, for some classes of tree-like problems, proofs are always exponentially larger than the smallest general, i.e., DAG-like, resolution ones [BEGJ00]. However, DPLL with learning and restarts becomes again equivalent to general resolution with respect to such notions of proof complexity [BKS03].

Finally, we can define a modern CDCL system as follows:

Definition 2.6. *The* Modern CDCL system *is the transition system* `CDCL` *consisting of the following seven transition rules.*

$$\begin{array}{l}\textsf{UnitPropagate}: \\ \quad A \parallel F, C \vee l \implies A\, l \parallel F, C \vee l \quad \textbf{if} \begin{cases} C \textit{ is false in } A, \\ l \textit{ is undefined in } A. \end{cases} \\ \textsf{Decide}: \\ \quad A \parallel F \implies A\, l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \textit{ or } \bar{l} \textit{ occurs in a clause of } F, \\ l \textit{ is undefined in } A. \end{cases} \\ \textsf{Fail}: \\ \quad A \parallel F, C \implies \mathit{FailState} \quad \textbf{if} \begin{cases} C \textit{ is false in } A, \\ A \textit{ contains no decision literals.} \end{cases} \end{array}$$

Backjump :

$$A\ l^{\mathsf{d}}\ B \parallel F, C \Longrightarrow A\ l' \parallel F, C \quad \textbf{if} \quad \begin{cases} C \text{ is false in } A\ l^{\mathsf{d}}\ B, \text{and} \\ \text{there exists a clause } C' \vee l' \\ \text{entailed by } F \wedge C \text{ such that} \\ C' \text{ is false in } A, \\ l' \text{ is undefined in } A \text{ and} \\ l' \text{ or } \overline{l'} \text{ occurs in } F \vee A\ l^{\mathsf{d}}\ B. \end{cases}$$

Learn :

$$A \parallel F \Longrightarrow A \parallel F, C \quad \textbf{if} \quad \begin{cases} \text{each atom of } C \text{ occurs in } F \vee A, \\ C \text{ is entailed by } F. \end{cases}$$

Forget :

$$A \parallel F, C \Longrightarrow A \parallel F \quad \textbf{if} \quad C \text{ is entailed by } F.$$

Restart :

$$A \parallel F \Longrightarrow \emptyset \parallel F.$$

CDCL Systems apply Learn after every Backjump in order to learn the backjump clause. However, notice that Learn can be applied in other situations. For instance, Siege SAT Solver also learns some of the intermediate clauses in the resolution derivations of the conflict analysis [Rya04]; Precosat and Lingeling, among others, periodically use some sophisticate techniques in order to simplify some clauses of the formula (which corresponds to apply Learn and Forget), etc.

Example 2.7. *The same set of clauses of Example 2.4 can now be processed with the* CDCL *transition system:*

$$\begin{array}{rcl}
\emptyset & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2 & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4 & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2\ \overline{5} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} \\
1^{\mathsf{d}}\ 2\ \overline{5} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2},\ \overline{2}\vee\overline{5} \\
1^{\mathsf{d}}\ 2\ \overline{5}\ 3^{\mathsf{d}} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2},\ \overline{2}\vee\overline{5} \\
1^{\mathsf{d}}\ 2\ \overline{5}\ 3^{\mathsf{d}}\ 4 & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2},\ \overline{2}\vee\overline{5} \\
1^{\mathsf{d}}\ 2\ \overline{5}\ 3^{\mathsf{d}}\ 4\ 6^{\mathsf{d}} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2},\ \overline{2}\vee\overline{5}
\end{array}
\begin{array}{ll}
\Longrightarrow_{\texttt{CDCL}} & \text{(Decide)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(UnitPropagate)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(Decide)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(UnitPropagate)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(Decide)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(UnitPropagate)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(Backjump)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(Learn)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(Decide)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(UnitPropagate)} \\
\Longrightarrow_{\texttt{CDCL}} & \text{(Decide)}
\end{array}$$

Note that in the backjump step we have used the backjump clause $\overline{2} \vee \overline{5}$. *Note also that, before the last* Decide *step, the partial interpretation was already a model. However, CDCL SAT-solvers do not check that, since completing a partial model only requires a polynomial amount of work.* □

Notice that derivations from CDCL may be infinite because of the restarts and subderivations of learns and forgets. In order to avoid these infinite derivations, we need to impose two conditions: firstly, infinite subderivations with only Learn and Forget steps are not allowed. Secondly, let a_k be the number of Backjump rules between the k-th ant the $k+1$-th restart. Then, the sequence $(a_k)_k$ must be unbounded. The transition system CDCL (with these two additional conditions) can be used for deciding the satisfiability of an input formula F, as shown in this result from [NOT06]:

Theorem 2.8. *Let F be a CNF formula. Then, any derivation from the initial state $\emptyset \parallel F$ is finite, i.e., there exists n such that $\emptyset \parallel F \Longrightarrow_{\texttt{CDCL}} S_1 \Longrightarrow_{\texttt{CDCL}} S_2 \Longrightarrow_{\texttt{CDCL}} \ldots \Longrightarrow_{\texttt{CDCL}} S_n$; and S_n is a final state with respect to the classic* DPLL *system*[1]*. Moreover, if F is unsatisfiable, the final state S_n is FailState, whereas if F is satisfiable the last state is of the form $S_n = A \parallel G$, where A is a model of F, and G is formula with the same models than F.*

2.4 Modern SAT Solvers

In this section we sketch the architecture of a typical modern SAT Solvers. Some of the features are common in practically all the state-of-the-art SAT Solvers, whereas others are only present in some of the most recent programs.

The core of a modern SAT Solver is the CDCL transition system explained in the previous section. Fail and Backjump have the maximum priority among the rules. Learn is always applied after a backjump. UnitPropagate is applied when no conflict clause has been found. Restart is periodically applied, depending on the heuristics (see Section 2.4.2). Also periodically, solvers *clean up* the formula, applying Forget to every useless learned clause. When no other rule applies, Decide is done.

2.4.1 Decision Heuristics

When Decide takes place, choosing which literal will be added to the current assignment has a huge impact on the solver performance. Naive strategies like choosing a random literal or the variable with most occurrences in the formula (or variations of this ideas like BOHM Heuristic [BB92], MOM Heuristics [JT96], etc.) are not competitive in industrial problems, as was experimentally proven at [Sil99].

Modern SAT Solvers use different variants of the *Variable State Independent Decaying Sum* (*VSIDS* for shortening) heuristic [MMZ+01]. It consists in assigning a value to every variable and picking the variable with highest value. The value of a variable is incremented every time it appears in a conflict analysis. In order to prioritize the variables appearing in the most recent conflicts, this increment increases geometrically during the search.

[1] It is not a final state with respect CDCL since, due to Restart, there are not final states in CDCL.

VSIDS heuristic is one of the major reasons of the impressive improvement on performance of the SAT Solvers in the last decade (see [MMZ+01, KSMS11]). However, Decide does not need to pick a variable but a literal. SAT Solvers usually choose the polarity of the literal with the *last phase* technique [PD07], that is, picking the last value that was assigned to the variable.

2.4.2 Restart Policies

Another important point consists in deciding when the solver must restart. Modern SAT Solvers use variations of one of these two policies: *geometrical restarts* and *Luby restarts* [Hua07]. In geometrical restarts, the k-th restart takes place when the solver has made Nr^k backjumps since the last restart, where N and r are fixed parameters. In the Luby restarts, the k-th restart takes place when the solver has made NL_k backjumps since the last restart, where N is a fixed parameter and L_k is the k-th Luby sequence number [LSZ93].

Recently, other policies have been proposed. In [Bie08] a dynamical adaptive strategy is explained. In [SI09] and [NMJ09] the authors propose the use of different restart strategies depending on the problem structure.

2.4.3 Cleanup Policies

A similar important issue consists in deciding the moment when Forget will be applied and which clauses will be removed. Cleanups take place much less often than restarts. Distance between cleanups may be geometrical (this is, after Nr^k conflicts, where k is the number of cleanups previously done), linear (i.e., after $N + dk$ conflicts, where k is again the number of previous cleanups) or follow some other sequence.

The classical method for deciding which clauses should be removed was similar to the VSIDS heuristic: each lemma is given a value, which is incremented each time the clause appears as a reason in a conflict analysis. During the cleanup, clauses with null value are removed and the other clauses decrement (for instance, dividing by 2) their value.

Recently, a new method has been presented [AS09]. In this method, lemmas are given a value corresponding to the number of different decision levels of their literals. This value may be static (computed when the lemma is created in the conflict analysis) or dynamic (recomputed any time the clause propagates some literal). In the cleanup, lemmas with highest values are removed. This new methodology has widely replaced the previous one in the last years.

Other techniques are possible. For instance, in [ALMS11] another way of computing lemmas' values is proposed. Moreover, the authors suggest to *freeze* the lemmas instead of removing, i.e., when the solver detects some useless lemmas, instead of removing them, they are isolated and not used in the propagation process until the solver detects they could be useful.

2.4.4 Preprocessing Techniques

Before starting the CDCL search for a model, modern SAT Solvers try to simplify the formula. This step is called *preproccess* and usually improves the performance of the solver. A lot of different techniques are possible during the preprocess. See, for instance, [Bra04, Nov03, EB05, DLF+02, Sil00, BLS11, HJB11].

Recently, some solvers [Bie10a, Bie10b, Soo10] spend periodically some time in simplifying the formula. In this case, the solver has an *inprocess* simplification.

2.4.5 Efficient Implementation Structures

State-of-the-art SAT Solvers can solve problems with millions of clauses and variables. In these problems, it is crucial to efficiently detect if a propagation is possible, or if a conflict has occurred. Hence, some research has been devoted to study which data structures can be used in order to make as efficient as possible the processes of the solver. The most important achievement in this field is the *two-watched-literal* structure [MMZ+01], a data structure for the formula which allows a very efficient unit propagation and conflicting clause detection. Another improvement is the use of specific structure for representing the clauses of two literals, currently implemented in most SAT Solvers.

2.4.6 Lemma Shortening

As an improvement of the conflict analysis process explained in Example 2.5, modern SAT Solvers try to simplify the obtained lemma before learning it. This can be done in different ways. In [SB09, Ass10] some different methods are experimentally compared. Recently, Van Gelder presented a new method [VG11].

2.5 SMT Solvers

SAT Solver transition system can only deal with propositional logical problems. Every NP problem can be reduced to such a problem in polynomial time. However, pure SAT approaches has been shown to be inefficient for some kind of problems: SMT, an extension of SAT for first-order logical problems, may be useful in these cases.

In this section we define a transition system for solving a formula whose clauses contain not only Boolean literals, but also atoms coming from some *Theories*. A *theory* is a set of closed first-order formulas. A formula F is T-satisfiable if $F \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called T-unsatisfiable.

As before, a partial assignment A will sometimes also be seen as a conjunction of literals and hence as a formula. If A is a T-consistent partial assignment and it is the (propositional) model of a formula F, then we say that A is a T-model of F. If F and G are formulas, then F entails G in T, if $F \wedge \overline{G}$ is T-inconsistent. If F entails

G in T and vice versa, we say that F and G are T-equivalent. A *theory lemma* is a clause entailed in T by the empty formula.

The problem consisting of finding a T-model for a formula F is called *Satisifability Modulo Theories* or simply *SMT*. Programs for solving these problems are called *SMT Solvers*.

2.5.1 SMT Solvers as Transition Systems

In this section we describe SMT solvers as a transition system; some of the rules are equal to the CDCL-solvers case, other rules are slightly modified (T-Learn, T-Forget and T-Backjump) and a new rule appears: T-Propagate. Let us define the transition system's rules:

Definition 2.9. *The* SMT system *is the transition system* `SMT` *consisting of the following eight transition rules.*

UnitPropagate :

$$A \parallel F, C \vee l \implies A\ l \parallel F, C \vee l \quad \textbf{if} \quad \begin{cases} C \text{ is false in } A, \\ l \text{ is undefined in } A. \end{cases}$$

T-Propagate :

$$A \parallel F \implies A\ l \parallel F \quad \textbf{if} \quad \begin{cases} l \text{ is entailed in } T \text{ by } A, \\ l \text{ or } \bar{l} \text{ occurs in } F. \\ l \text{ is undefined in } A. \end{cases}$$

Decide :

$$A \parallel F \implies A\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \quad \begin{cases} l \text{ or } \bar{l} \text{ occurs in a clause of } F, \\ l \text{ is undefined in } A. \end{cases}$$

Fail :

$$A \parallel F, C \implies \mathit{FailState} \quad \textbf{if} \quad \begin{cases} C \text{ is false in } A, \\ A \text{ contains no decision literals.} \end{cases}$$

T-Backjump :

$$A\ l^{\mathsf{d}}\ B \parallel F, C \implies A\ l' \parallel F, C \quad \textbf{if} \quad \begin{cases} C \text{ is false in } A\ l^{\mathsf{d}}\ B, \text{and} \\ \text{there exists a clause } C' \vee l' \\ \text{entailed in } T \text{ by } F \wedge C \text{ such that} \\ C' \text{ is false in } A, \\ l' \text{ is undefined in } A \text{ and} \\ l' \text{ or } \overline{l'} \text{ occurs in } F \vee A\ l^{\mathsf{d}}\ B. \end{cases}$$

T-Learn :

$$A \parallel F \implies A \parallel F, C \quad \textbf{if} \quad \begin{cases} \text{each atom of } C \text{ occurs in } F \vee A, \\ C \text{ is entailed in } T \text{ by } F. \end{cases}$$

T-Forget :

$$A \parallel F, C \implies A \parallel F \qquad \textbf{if} \quad C \textit{ is entailed in } T \textit{ by } F.$$

Restart :

$$A \parallel F \qquad \implies \emptyset \parallel F.$$

T-Learn and T-Forget behaves similarly to the CDCL case: however, the clause added or removed is now entailed by F in the theory T whereas on CDCL it was entailed in the propositional sense. Similarly, in T-Backjump, the backjump clause is entailed in T instead of in the propositional sense.

The new rule, T-Propagate, is similar to the unit propagation in the propositional context: the solver adds to the assignment the literals that must be true (due to the theory) in the current assignment.

2.5.2 SMT Solvers as SAT Solvers + Propagators

SMT Solvers considered in this document are composed of a SAT Solver engine that deals with the propositional part of the problem and some propagators that deals with the non-propositional part. The SAT Solver engine is very similar to modern CDCL Solvers explained at Section 2.4. The three differences are listed here:

- The SAT Solver engine sends *periodically* the current assignment to the propagators, and collect their propagations. If the assignment A is not T-consistent for some theory, its propagator propagates $\bar{l}$ for some $l \in A$.
- In the conflict analysis, if some literal was propagated for a propagator, the solver demands to it an *explaination* or *reason*, that is, a theory lemma such that, if added, the literal would be entailed by unit propagation. This clause is used in the usual conflict analysis step of the SAT Solver.
- When the SAT Solver backjumps or restarts, the propagator must return to the previous state.

Therefore, the propagators must perform these actions: *T-propagate* an assignment and *give explainations* to the literals previously propagated. T-propagation must be incrementally done: this means that the propagator must save the current state, and update it when the SAT solver sends the new assignment due to new decisions, unit propagations, backjumps or restarts.

There are some theories where T-Propagate is very costly. For efficiency reasons, the frequency which T-propagations are demanded depends of the cost of this operation: in the easy cases, the SAT Solver will frequently sends the assignment to the theory, whereas on more complicated (and costly) theories, it will happens with less frequency. Section 5.2 contains two detailed examples of propagators for SMT Solvers.

3

Encoding Cardinality Constraints into SAT

3.1 Introduction

This section[1] presents a new encoding into SAT of *cardinality constraints*, that is, constraints of the form $x_1 + \cdots + x_n \ \# \ k$, where k is a natural number, the x_i are Boolean (0/1) variables, and the relation operator $\#$ belongs to $\{<, >, \leqslant, \geqslant, =\}$.

Cardinality constraints are present in many practical SAT applications, such as cumulative scheduling [SFSW09] or timetabling [AAN12]; or are part of some SAT technique, such as some encodings of *All-Different constraints* (see Section 7.5.2). They are also present in MaxSAT problems [AM06], since these problems can be solved adding a cardinality constraint (see Section 5.4.1); or are part of some MaxSAT techniques, as in *Fu & Malik algorithm* [FM06] (and some other algorithms based on it).

Here we are interested in encoding a cardinality constraint C with a clause set S (possibly with auxiliary variables) that is not only equisatisfiable, but also *arc-consistent*: given a partial assignment A, if x_i is true (false) in every extension of A satisfying C, then unit propagating A on S sets x_i to true (false)[2]. Enforcing arc-consistency by unit propagation in this way has of course an important positive impact on the practical efficiency of SAT solvers.

A straightforward encoding of a cardinality constraint $x_1 + \cdots + x_n \leqslant k$ is to state, for each subset Y of $\{x_1, \ldots, x_n\}$ with $|Y| = k + 1$, that at least one variable of Y must be false. This can be done by asserting $\binom{n}{k+1}$ clauses of the form $\overline{y_1} \vee \ldots \vee \overline{y_{k+1}}$. This kind of construction frequently works well, although it is of course not reasonable for large n and k. Therefore, successively more sophisticated

[1] A paper based on this chapter, named "A Smaller and Better Encoding for Cardinality Constraints", has been submited in the SAT Conference 2013

[2] Sometimes this notion is called *generalized arc-consistency*.

encodings using auxiliary variables and requiring fewer clauses have been defined. But still, for small n and k the straightforward encodings may behave better in practice. An additional issue is that, for the efficiency of the SAT solver, the choice of the right trade-off between minimizing either the number of auxiliary variables or the number of clauses is highly application-dependent.

Here we build upon previous work on Cardinality Networks of [ANORC09, ANORC11b], although the concepts are applicable as well to, for example, the Pairwise Cardinality Networks of [CZI10]. The idea is to get the best of several worlds: we develop an arc-consistent encoding that, by recursively decomposing the constraint into smaller ones, allows us to decide which encoding to apply to each sub-constraint. This process minimizes a function $\lambda \cdot num\_vars + num\_clauses$, where the parameter λ is user-defined. Our experimental evaluation shows that (e.g., for $\lambda = 5$) this new technique produces much smaller encodings in variables *and* clauses, and also strongly improves the performance of SAT solvers.

The starting point is the encoding based on *sorting networks* [Bat68] for input variables $(x_1 \dots x_n)$ and output $(y_1 \dots y_n)$. It consists of a set of clauses (and auxiliary variables) such that the output variables become ordered decreasingly, i.e., an output variable y_k will become true by unit propagation using the clauses if and only if there are at least k true input variables, and false iff there are at least $n - k + 1$ false ones. Sorting networks are used in, e.g., MiniSAT+ [ES06] to encode cardinality constraints: to express $x_1 + \dots + x_n \geqslant k$, it clearly suffices to add a unit clause y_k; similarly, for $x_1 + \dots + x_n \leqslant k$ one adds $\overline{y_{k+1}}$, and both unit clauses are added if the relation is $=$. Sorting networks require $O(n \log^2 n)$ clauses and auxiliary variables. The Cardinality Networks of [ANORC09, CZI10, ANORC11b] reduce this to $O(n \log^2 k)$, which is important since usually $n \gg k$, and moreover, for the relations $\leqslant$ and $\geqslant$, the number of clauses is halved (see Section 3.2 below).

All these networks can be built by recursive combinations of sorting networks and *merging networks*, the latter ones performing a sorted merge of two already sorted lists of variables. However, these constructions assume that the number of outputs m is a power of two and the number of inputs is a multiple of m, which requires one to add useless dummy variables and outputs. Here, as a first step, in Section 3.3, we improve on this by building these components for arbitrary-sized inputs and outputs. After that, in Section 3.4 different *direct encodings* for both cardinality networks and for merging networks are given. Then, in Section 3.5, we show that both types of encodings can be combined in a flexible way (somewhat similar to the enhancement of quicksort [Sed78] using insertion sort for short arrays). That combination is done minimizing a function $\lambda \cdot num\_vars + num\_clauses$ for any value of the user-defined parameter λ. Finally, extensive experimental evidence for the practical relevance of our techniques is given in Section 3.6. We conclude in Section 3.7.

3.2 Preliminaries

In this chapter we describe a method for producing cardinality networks that generalizes the construction of [ANORC11b]. The core idea of these approaches, which dates back to [Bat68], consists in encoding a circuit that implements mergesort by means of a set of clauses. The most basic components of these circuits are 2-comparators.

A *2-comparator* is a sorting network of size 2, i.e., it has 2 input variables (x_1 and x_2) and 2 output variables (y_1 and y_2) such that y_1 is true if and only if at least one of the input variables is true (i.e., it is the maximum or their disjunction), and y_2 is true if and only if both two input variables are true (i.e., it is the minimum or their conjunction):

$$\begin{array}{lllllllll} x_1 & \rightarrow & y_1, & x_2 & \rightarrow & y_1, & x_1 \wedge x_2 & \rightarrow & y_2, \\ \overline{x_1} & \rightarrow & \overline{y_2}, & \overline{x_2} & \rightarrow & \overline{y_2}, & \overline{x_1} \wedge \overline{x_2} & \rightarrow & \overline{y_1}. \end{array}$$

As pointed out in [ANORC11b], for encoding $\leqslant$-constraints, only the three clauses on the first row are needed to guarantee arc-consistency. The three clauses on the second row suffice for $\geqslant$-constraints and all six must be present when encoding $=$-constraints.

In the following, 2-comparators are denoted by $(y_1, y_2) = 2\text{-Comp}(x_1, x_2)$. An alternative common graphical representation of 2-comparators, from [CSRL01], is shown in Figure 3.1.

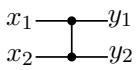

Figure 3.1: A 2-comparator.

3.3 Cardinality Networks of Arbitrary Size

In this section we improve the recursive construction of cardinality networks given in [ANORC11b] by allowing inputs and outputs of any size, not necessarily a power of two. Not only does this avoid adding dummy variables that are not actually needed, but also becomes useful when combining with the direct (non-recursive) constructions of Section 3.4.

In what follows, we denote by $\lfloor r \rfloor$ and $\lceil r \rceil$ the floor and ceiling functions respectively. Moreover, for simplicity, we will assume that the constraint to be encoded is a $\leqslant$-constraint. However, similar constructions for the other constraints can be devised.

3.3.1 Merge Networks

A *merge network* takes as input two (decreasingly) ordered sets of sizes a and b and produces a (decreasingly) ordered set of size $a + b$. We can build a merge network with inputs $(x_1, \ldots, x_a)$ and $(x'_1, \ldots, x'_b)$ in a recursive way as follows[3]:

- If $a = b = 1$, a merge network is a 2-comparator:

$$\text{Merge}(x_1; x'_1) := \text{2-Comp}(x_1, x'_1).$$

- If $a = 0$, a merge network returns the second input:

$$\text{Merge}(; x'_1, x'_2, \ldots, x'_b) := (x'_1, x'_2, \ldots, x'_b).$$

- If a and b are even, $a > 0$, $b > 0$ and either $a > 1$ or $b > 1$, let us define

$$\begin{aligned}
(z_1, z_3, \ldots, z_{a-3}, z_{a-1}, & \\
z_{a+1}, z_{a+3}, \ldots, z_{a+b-1}) &= \text{Merge}(x_1, x_3, \ldots, x_{a-1}; \\
& \quad x'_1, x'_3, \ldots, x'_{b-1}), \\
(z_2, z_4, \ldots, z_{a-2}, z_a, & \\
z_{a+2}, z_{a+4}, \ldots, z_{a+b}) &= \text{Merge}(x_2, x_4, \ldots, x_a; \\
& \quad x'_2, x'_4, \ldots, x'_b), \\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3), \\
(y_4, y_5) &= \text{2-Comp}(z_4, z_5), \\
& \ldots \\
(y_{a+b-2}, y_{a+b-1}) &= \text{2-Comp}(z_{a+b-2}, z_{a+b-1}).
\end{aligned}$$

  Then,

$$\text{Merge}(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_{a+b-1}, z_{a+b}).$$

- If a is even, b is odd, $a > 0$, $b > 0$ and either $a > 1$ or $b > 1$, let us define

$$\begin{aligned}
(z_1, z_3, \ldots, z_{a-1}, & \\
z_{a+1}, z_{a+3}, \ldots, z_{a+b}) &= \text{Merge}(x_1, x_3, \ldots, x_{a-1}; \\
& \quad x'_1, x'_3, \ldots, x'_b), \\
(z_2, z_4, \ldots, z_a, z_{a+2}, & \\
z_{a+4}, \ldots, z_{a+b-1}) &= \text{Merge}(x_2, x_4, \ldots, x_a; \\
& \quad x'_2, x'_4, \ldots, x'_{b-1}), \\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3), \\
(y_4, y_5) &= \text{2-Comp}(z_4, z_5), \\
& \ldots \\
(y_{a+b-1}, y_{a+b}) &= \text{2-Comp}(z_{a+b-1}, z_{a+b}).
\end{aligned}$$

  Then,

$$\text{Merge}(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_{a+b-1}, y_{a+b}).$$

[3] Notice we use the notation $\text{Merge}(X; X')$ instead of $\text{Merge}((X), (X'))$ for simplicity.

- If a and b are odd, $a > 0$, $b > 0$ and either $a > 1$ or $b > 1$, let us define

$$\begin{aligned}
(z_1, z_3, \ldots, z_{a-2}, z_a, z_{a+1}, z_{a+3}, \ldots, z_{a+b}) &= \text{Merge}(x_1, x_3, \ldots, x_a; x'_1, x'_3, \ldots, x'_b),\\
(z_2, z_4, \ldots, z_{a-3}, z_{a-1}, z_{a+2}, z_{a+4}, \ldots, z_{a+b-1}) &= \text{Merge}(x_2, x_4, \ldots, x_{a-3}, x_{a-1}; x'_2, x'_4, \ldots, x'_{b-1}),\\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\
(y_4, y_5) &= \text{2-Comp}(z_4, z_5),\\
&\ldots\\
(y_{a+b-2}, y_{a+b-1}) &= \text{2-Comp}(z_{a+b-2}, z_{a+b-1}).
\end{aligned}$$

  Then,

$$\text{Merge}(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_{a+b-1}, z_{a+b}).$$

- The remaining cases are defined thanks to the symmetry of the merge function, i.e., due to $\text{Merge}(X; X') = \text{Merge}(X'; X)$.

The base cases do not require any explanation. Regarding the recursive ones, first notice that the set of values $x_1, x_2, \ldots, x_a, x'_1, x'_2, \ldots, x'_b$ is always preserved. Further, the output bits are sorted, as $z_{2i} \geq z_{2(i+1)}$, $z_{2i} \geq z_{2(i+1)+1}$, $z_{2i+1} \geq z_{2(i+1)}$ and $z_{2i+1} \geq z_{2(i+1)+1}$ imply that $\min(z_{2i}, z_{2i+1}) \geq \max(z_{2(i+1)}, z_{2(i+1)+1})$.

Figures 3.2, 3.3 and 3.4 show examples of some of these recursive cases. The left side of these figures contain the recursive definition of the network. On the right, the recursive networks are substituted for all the 2-comparators. The same pattern will be followed in the rest of the section.

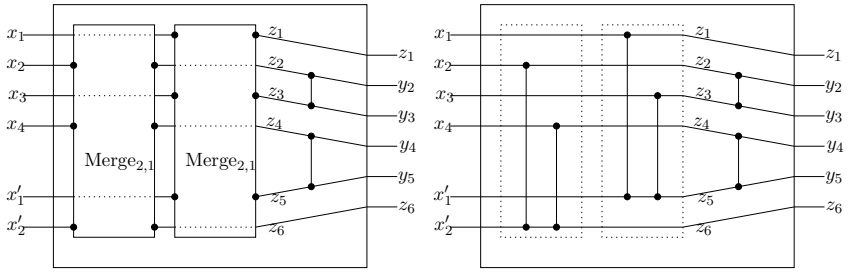

Figure 3.2: A (2.4)-merge network.

The number of auxiliary variables and clauses of a merge network defined in this way can be recursively computed. A merge network with inputs of size $(1, 1)$ needs 2 variables and 3 clauses. A merge network with inputs of size $(0, b)$ needs no variables and clauses. A merge network with inputs of size (a, b) with $a > 1$ or $b > 1$ needs

$$V_1 + V_2 + 2\left\lfloor \frac{a+b-1}{2} \right\rfloor$$

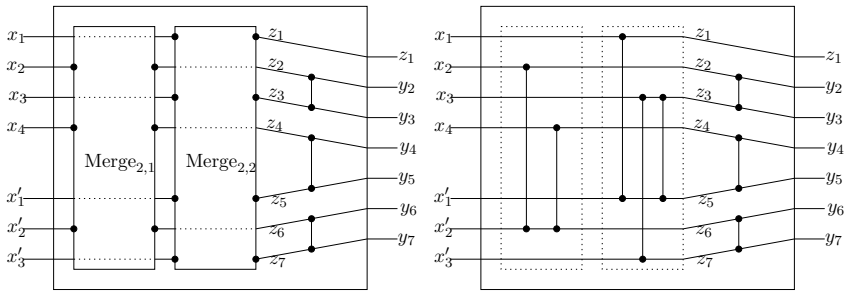

Figure 3.3: A (3,4)-merge network.

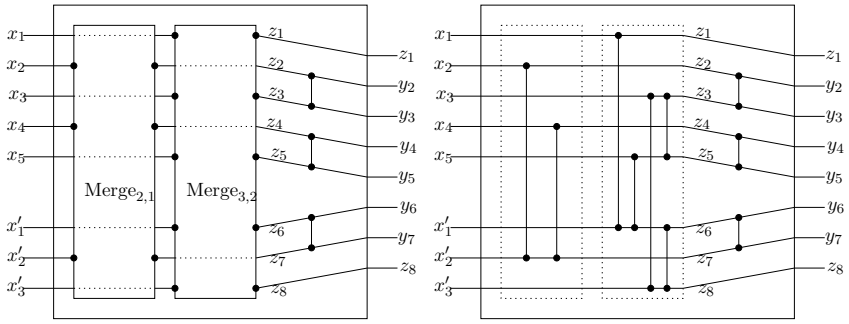

Figure 3.4: A (3,5)-merge network.

variables, and

$$C_1 + C_2 + 3\left\lfloor \frac{a+b-1}{2} \right\rfloor$$

clauses, where V_1 and C_1 are the number of variables and clauses needed in a merge network with inputs of size $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil\right)$, and V_2, C_2 are the number of variables and clauses needed in a merge network with inputs of size $\left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor\right)$.

3.3.2 Sorting Networks

A *sorting network* takes an input of size n and sorts it. It can be built in a recursive way as follows, using the same strategy as in mergesort:

- If $n = 1$, the output of the sorting network is its input:

$$\text{Sorting}(x_1) := x_1$$

- If $n = 2$, a sorting network is a single merge (i.e., a 2-comparator):

$$\text{Sorting}(x_1, x_2) := \text{Merge}(x_1; x_2).$$

- For $n > 2$, take l with $1 \leqslant l < n$: Let us define

$$\begin{aligned}(z_1, z_2, \ldots, z_l) &= \text{Sorting}(x_1, x_2, \ldots, x_l),\\ (z_{l+1}, z_{l+2}, \ldots, z_n) &= \text{Sorting}(x_{l+1}, x_{l+2}, \ldots, x_n),\\ (y_1, y_2, \ldots, y_n) &= \text{Merge}(z_1, z_2, z_l; z_{l+1}, \ldots, z_n).\end{aligned}$$

  Then,

$$\text{Sorting}(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_n).$$

In this way, we can build sorting networks of any size in a recursive way. Moreover, the two recursive sorting networks can be of any size. Note that in [ANORC11b], l was always chosen to be $n/2$. Figure 3.5 shows an example of these sorting networks.

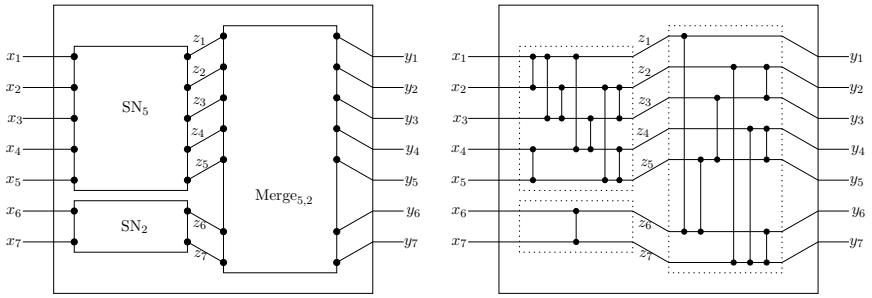

Figure 3.5: A sorting network of size 7 composed by sorting networks of size 2 and 5.

As in the previous section, the number of auxiliary variables and clauses needed in this networks can be recursively computed. A sorting network of size 1 needs no variables and clauses. A sorting network of size 2 needs 2 variables and 3 clauses. A sorting network of size n composed by a sorting network of size l and a sorting network of size $n - l$ needs $V_1 + V_2 + V_3$ variables and $C_1 + C_2 + C_3$ clauses, where $(V_1, C_1), (V_2, C_2)$ are the number of variables and clauses used in the sorting networks of size l and $n - l$, and (V_3, C_3) are the number of variables and clauses needed in the merge network with input of sizes $(l, n - l)$.

3.3.3 Simplified Merges

A *simplified merge* is a reduced version of a merge, used when we are only interested in some of the outputs, but not all. Recall that we want to encode a constraint of the form $x_1 + \ldots + x_n \leqslant k$, and hence we are only interested in the first $k+1$ bits of the sorted output. Thus, in a c-simplified merge network, the inputs are two sorted sequences of variables $(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b)$, and the network produces a sorted output of the desired size, c, $(y_1, y_2, \ldots, y_c)$. The network satisfies that y_r is

true if there are at least r true inputs[4]. We can build a recursive simplified merge as follows:

- If $a = b = c = 1$, a simplified merge network can be defined as follows:

$$\text{SMerge}_1(x_1; x'_1) := y,$$

  adding the clauses $x_1 \to y,\ x'_1 \to y$.

- If $a > c$, we can ignore the last $a - c$ bits of the first input (the same happens if $b > c$):

$$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, \ldots, x'_b) = \text{SMerge}_c(x_1, x_2, \ldots, x_c; x'_1, \ldots, x'_b).$$

- If $a + b \leqslant c$, the simplified merge is a merge:

$$\text{SMerge}_c(x_1, \ldots, x_a; x'_1, \ldots, x'_b) = \text{Merge}(x_1, \ldots, x_a; x'_1, \ldots, x'_b).$$

- If $a, b \leqslant c$, $a + b > c$ and c is even: Let us define

$$\begin{array}{rcl}
(z_1, z_3, \ldots, z_{c+1}) & = & \text{SMerge}_{c/2+1}(x_1, x_3, \ldots; x'_1, x'_3, \ldots), \\
(z_2, z_4, \ldots, z_c) & = & \text{SMerge}_{c/2}(x_2, x_4, \ldots; x'_2, x'_4, \ldots), \\
(y_2, y_3) & = & \text{2-Comp}(z_2, z_3), \\
(y_4, y_5) & = & \text{2-Comp}(z_4, z_5), \\
 & \ldots & \\
(y_{c-2}, y_{c-1}) & = & \text{2-Comp}(z_{c-2}, z_{c-1}).
\end{array}$$

  Then,

$$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_c),$$

  adding the clauses $z_c \to y_c,\ z_{c+1} \to y_c$.

- If $a, b \leqslant c$, $a + b > c$ and $c > 1$ is odd: Let us define

$$\begin{array}{rcl}
(z_1, z_3, \ldots, z_c) & = & \text{SMerge}_{\frac{c+1}{2}}(x_1, x_3, \ldots; x'_1, x'_3, \ldots), \\
(z_2, z_4, \ldots, z_{c-1}) & = & \text{SMerge}_{\frac{c-1}{2}}(x_2, x_4, \ldots; x'_2, x'_4, \ldots), \\
(y_2, y_3) & = & \text{2-Comp}(z_2, z_3), \\
(y_4, y_5) & = & \text{2-Comp}(z_4, z_5), \\
 & \ldots & \\
(y_{c-1}, y_c) & = & \text{2-Comp}(z_{c-1}, z_c).
\end{array}$$

  Then,

$$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_c).$$

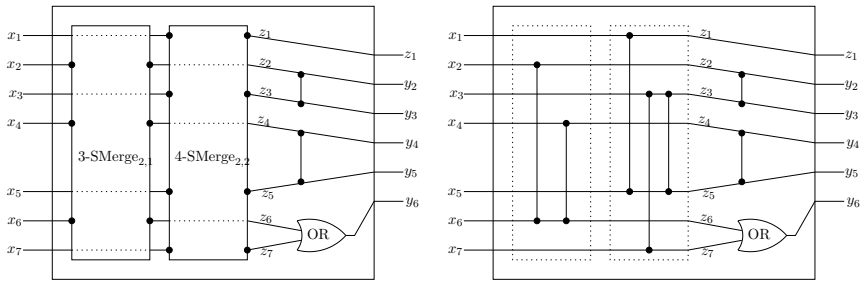

Figure 3.6: A 6-(4,3) simplified merge network.

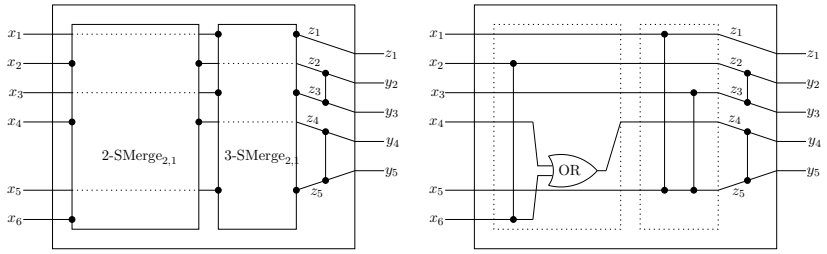

Figure 3.7: A 5-(4,2) simplified merge network.

Figures 3.6 and 3.7 show two examples of simplified merges: The first one shows a 6-simplified merge with inputs of sizes 3 and 4. The second one corresponds to a 5-simplified merge with inputs of sizes 2 and 4.

We can recursively compute the auxiliary variables and clauses needed in these simplified merge networks. In the recursive case, we need V_1+V_2+c-1 variables and $C_1+C_2+C_3$ clauses, where $(V_1, C_1), (V_2, C_2)$ are the number of clauses and variables needed in simplified merge networks of sizes $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil, \left\lfloor \frac{c}{2} \right\rfloor + 1\right)$, $\left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor, \left\lfloor \frac{c}{2} \right\rfloor\right)$, and

$$C_3 = \begin{cases} \frac{3c-3}{2} & \text{if } c \text{ is odd,} \\ \frac{3c-2}{2} + 2 & \text{if } c \text{ is even.} \end{cases}$$

3.3.4 m-Cardinality Networks

An *m-cardinality network* takes an input of size n and outputs the first m sorted bits. In a recursive way, an m-cardinality network with input $x_1, x_2, \ldots, x_n$ can be defined as follows:

- If $n \leqslant m$, a cardinality network is a sorting network:

$$\text{Card}_m(x_1, x_2, \ldots, x_n) := \text{Sorting}(x_1, x_2, \ldots, x_n).$$

[4]Notice that simplified merges for $\geqslant$ constraints satisfy that y_r is false if there are at least $a + b + 1 - r$ false inputs. The other networks behave similarly.

- If $n > m$, take l with $1 \leqslant l < n$. Let us define

$$\begin{array}{rcl}(z_1, z_2, \ldots, z_A) & = & \mathrm{Card}_m(x_1, x_2, \ldots, x_l), \\ (z'_1, z'_2, \ldots, z'_B) & = & \mathrm{Card}_m(x_{l+1}, x_{l+2}, \ldots, x_n), \\ (y_1, y_2, \ldots, y_m) & = & \mathrm{SMerge}_m(z_1, z_2, \ldots, z_A; z'_1, z'_2, \ldots, z'_B),\end{array}$$

  where $A = \min\{l, m\}$ and $B = \min\{n - l, m\}$. Then,

$$\mathrm{Card}_m(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_m).$$

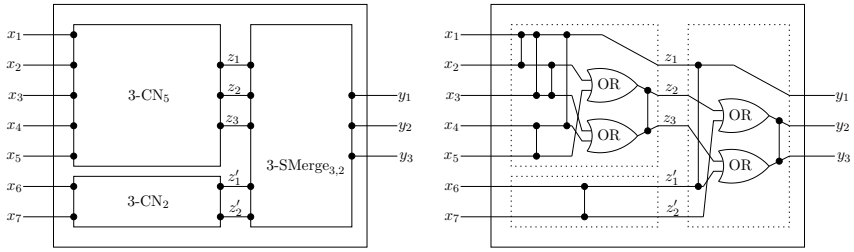

Figure 3.8: A 3-cardinality network of size $n = 7$.

Figure 3.8 shows a 3-cardinality network of input size 7, composed by a 3-cardinality network of input size 5 and a 3-cardinality network of input size 2 (this is, a sorting network) and a 3-simplified merge.

Again, the number of auxiliary variables and clauses needed in these networks can be recursively computed. An m-cardinality network of size n composed by an m-cardinality network of size l and an m-cardinality network of size $n - l$ needs $V_1 + V_2 + V_3$ variables and $C_1 + C_2 + C_3$ clauses, where $(V_1, C_1), (V_2, C_2)$ are the number of variables and clauses used in the m-cardinality networks of sizes l and $n-l$, and (V_3, C_3) are the number of variables and clauses needed in the m-simplified merge network with inputs of sizes $(\min\{l, m\}, \min\{n - l, m\})$.

With the same techniques used in [ANORC11b], one could easily prove the arc-consistency of the encoding.

Theorem 3.1. *The Recursive Cardinality Network encoding is arc-consistent: consider a cardinality constraint $x_1 + \ldots + x_n \leqslant k$, its corresponding cardinality network $(y_1, y_2, \cdots, y_{k+1}) = \mathrm{Card}_{k+1}(x_1, x_2, \ldots, x_n)$, and the unit clause $\overline{y_{k+1}}$. If we now set to true k input variables, then unit propagation sets to false the remaining $n - k$ input variables.*

3.4 Direct Cardinality Networks

In this section we introduce an alternative technique for building cardinality networks which we call *direct*, as it is non-recursive. This method uses many fewer

auxiliary variables than the recursive approach explained in Section 3.3. On the other hand, the number of clauses of this construction makes it competitive only for small sizes. However, this is not a problem as we will see in Section 3.5, since a combination of the two techniques is possible.

As in the recursive construction described in Section 3.3, the building blocks of direct cardinality networks are merge, sorting and simplified merge networks:

- **Merge Networks**. If a or b are 0, they are defined as in Section 3.3.1. If not, they are defined as follows:

  $$\text{Merge}(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (y_1, y_2, y_3, \ldots, y_{a+b-1}, y_{a+b}),$$

  with clauses

  $$\{x_i \to y_i,\ x'_j \to y_j,\ x_i \wedge x'_j \to y_{i+j}\ :\ 1 \leqslant i \leqslant a, 1 \leqslant j \leqslant b\}.$$

  Notice we need $a + b$ variables and $ab + a + b$ clauses.

- **Sorting Networks**. The case $n = 1$ is defined as in Section 3.3.2. The case $n > 1$ can be built as follows:

  $$\text{Sorting}(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_n),$$

  with clauses

  $$\{x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_k} \to y_k\ :\ 1 \leqslant k \leqslant n, 1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n\}.$$

  Therefore, we need n auxiliary variables and $2^n - 1$ clauses.

- **Simplified Merge Networks**. The definition of c-simplified merge is the same as in Section 3.3.3, except for the cases in which $a, b \leqslant c$ and $a + b > c$, where:

  $$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (y_1, y_2, \ldots, y_c),$$

  with clauses

  $$\{x_i \to y_i,\ x'_j \to y_j,\ x_i \wedge x'_j \to y_{i+j}\ :\ 1 \leqslant i \leqslant a, 1 \leqslant j \leqslant b, i + j \leqslant c\}.$$

  This approach needs c variables and $(a+b)c - \frac{c(c-1)}{2} - \frac{a(a-1)}{2} - \frac{b(b-1)}{2}$ clauses.

- m**-Cardinality Networks**. The definition is the same as in Section 3.3.4, except for the case $n > m$, where:

  $$\mathrm{Card}_m(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_m)$$

  with clauses

  $$\{x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_k} \rightarrow y_k \ : \ 1 \leqslant k \leqslant m, 1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n\}.$$

  This approach needs m variables and $\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{m}$ clauses.

As regards the arc-consistency of the encoding, the following result can be easily proved:

Theorem 3.2. *The Direct Cardinality Network encoding is arc-consistent.*

3.5 Combining Recursive and Direct Networks

The recursive approach produces shorter networks than the direct approach when the input is middle-sized. However, the recursive method for building a network needs to inductively produce networks for smaller and smaller input sizes. At some point, the networks we need have a sufficiently small number of inputs such that the direct method can build them using fewer clauses and variables than the recursive approach. In this section a *mixed encoding* is presented: large cardinality networks are build with the recursive approach but their components are produced with the direct approach when their size is small enough.

In more detail, assume a merge of input sizes a and b is needed. We can use the direct approach, which needs $V_D = a + b$ auxiliary variables and $C_D = ab + a + b$ clauses; or we could use the recursive approach. With the recursive approach, we have to built two merge networks of sizes $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil\right)$ and $\left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor\right)$. These networks are also built with this mixed approach. Then, we compute the clauses and variables needed in the recursive approach, V_R and C_R, with the formula of Section 3.3.1:

$$V_R = V_1 + V_2 + 2\left\lfloor \frac{a+b-1}{2} \right\rfloor,$$

$$C_R = C_1 + C_2 + 3\left\lfloor \frac{a+b-1}{2} \right\rfloor,$$

where (V_1, C_1) and (V_2, C_2) are, respectively, the number of variables and clauses needed in the recursive merge networks.

Finally, we compare the values of V_R, V_D, C_R and C_D, and decide which method is better for building the merge network. Notice that we cannot minimize both the number of variables and clauses; therefore, here we try to minimize the function $\lambda \cdot V + C$, for some fixed value $\lambda > 0$. The parameter λ allows us to adjust the relative importance of the number of variables with respect to the number of

clauses of the encoding. Notice that this algorithm for building merge networks (and similarly, sorting, simplified merge and cardinality networks) can easily be implemented with dynamic programming. See Section 3.6 for an experimental evaluation of the numbers of variables and clauses in cardinality networks built with this mixed approach.

3.6 Experimental Evaluation

In previous work [ANORC11b], it was shown that power-of-two (Recursive) Cardinality Networks were superior to other well-known methods such as Sorting Networks [ES06], Adders [ES06] and the BDD-based encoding of [BBR06b]. In what follows we will show that the generalization of Cardinality Networks to arbitrary size and their combination with Direct Encodings, yielding what we have called here the **Mixed** approach, makes them significantly better.

We start the evaluation focusing on the size of the resulting encoding. In Figure 3.9 we show the size, in terms of variables and clauses, of the encoding of a cardinality network with input size 100 and varying output size m.

It can be seen that, since we minimize the function $\lambda \cdot V + C$, where V is the number of variables and C the number of clauses, the bigger λ is, the fewer variables we obtain, at the expense of a slight increase in the number of clauses. Also, it can be seen that using power-of-two Cardinality Networks, as it was done in [ANORC11b] is particularly harmful when m is slightly larger than a power of two.

Although having a smaller encoding is beneficial, this should be accompanied with a reduction in SAT solver runtime. Hence, let us now move to assess how our new encoding affects the performance of SAT solvers. Since, as we showed, power-of-two Recursive Cardinality Networks were shown to be superior to other methods we will only compare **Mixed** with the former. However, we want to point out that the ideas underlying our novel encoding could also be applied to the Pairwise Cardinality Networks of [CZI10], which introduce another variant of Cardinality Networks.

The SAT solver we have used is Lingeling version *ala*, a state-of-the-art CDCL (Conflict-Driven Clause Learning) SAT solver that implements several inprocessing and preprocessing techniques. All experiments were conducted on a 2Ghz Linux Quad-Core AMD with the three following sets of benchmarks:

1.-MSU4 suite. These benchmarks are intermediate problems generated by an implementation of the *msu4* algorithm [MSP08], which reduces a Max-SAT problem to a series of SAT problems with cardinality constraints. The *msu4* implementation was run of a variety of problems (filter design, logic synthesis, minimum-size test pattern generation, haplotype inference and maximum-quartet consistency) from the Partial Max-SAT division of the Third Max-SAT evaluation[5]. The suite consists of about 14000 benchmarks, each of which contains multiple $\leqslant$-cardinality constraints.

[5]See `http://www.maxsat.udl.cat/08/index.php?disp=submitted-benchmarks`.

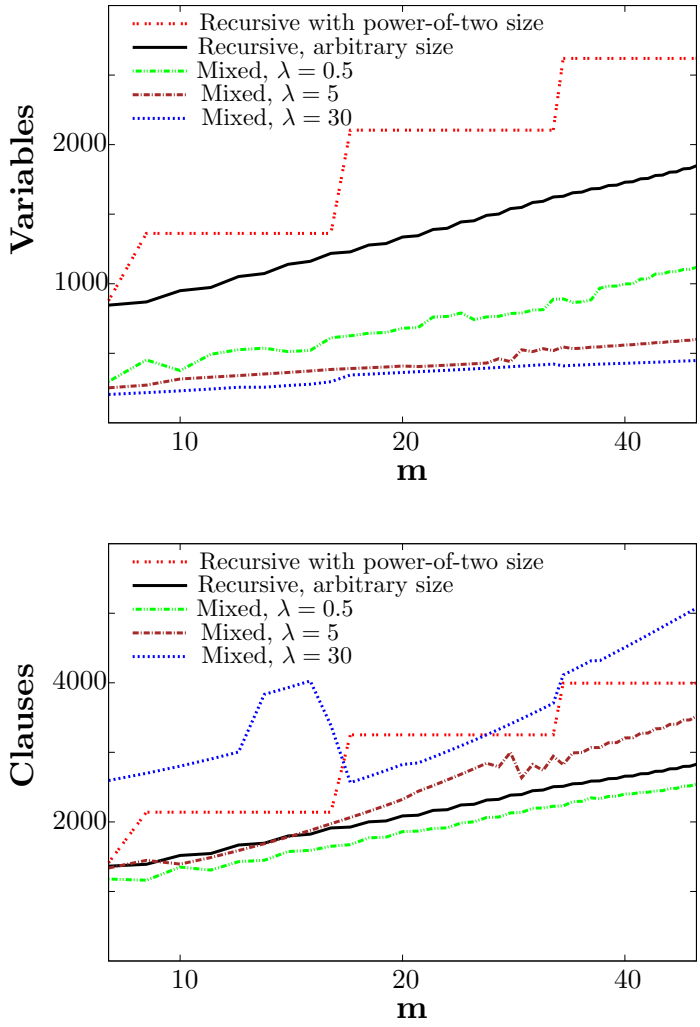

Figure 3.9: Variables and clauses generated by **Mixed** and the Recursive Cardinality Networks approaches for encoding cardinality networks of input size 100 and different output sizes m.

	Speed-up factor of Mixed					Slow-down factor of Mixed				
Suite	**TO**	**4**	**2**	**1.5**	**TOT.**	**1.5**	**2**	**4**	**TO**	**TOT.**
MSU4	43	732	2957	1278	*5010*	1	23	13	11	*48*
DES	12	21	265	638	*936*	6	12	7	46	*71*
Tomography	121	387	407	174	*1089*	64	82	159	121	*426*

Table 3.1: Comparison in terms of SAT solver runtime. Numbers indicate number of benchmarks in which **Mixed** showed the corresponding speed-up or slow-down factor w.r.t. power-of-two Recursive Cardinality Networks.

2.-Discrete-event system diagnosis suite. The second set of benchmarks we have used is the one introduced in [AG09]. These problems come from discrete-event system (DES) diagnosis. As it happened with the Max-SAT problems, a single DES problem produced a family of "SAT + cardinality constraints" problems. This way, out of the roughly 600 DES problems, we obtained a set of 6000 benchmarks, each of which contained a single very large $\leqslant$-cardinality constraint.

3.-Tomography suite. The last set of benchmarks we have used is the one introduced in [BB03a]. The idea is to first generate an $N \times N$ grid in which some cells are filled and some others are not. The problem consists in finding out which are the filled cells using only the information of how many filled cells there are in each row, column and diagonal. For that purpose, variables x_{ij} are used to indicate whether cell (i, j) is filled and several $=$-cardinality constraints are used to impose how many filled cells there are in each row, column and diagonal. We generated 2600 benchmarks (100 instances for each grid size $N = 15 \dots 40$).

Results are summarized[6] in Table 3.1, which presents a comparison of the **Mixed** (with $\lambda = 5$) encoding with respect to the power-of-two Recursive Cardinality Networks of [ANORC11b]. The time limit was set to 600 seconds per benchmark and we only considered benchmarks for which at least one of the methods took more than 5 seconds. Columns indicate in how many benchmarks the **Mixed** encoding exhibits the corresponding speed-up or slow-down factor. For example, the **TO** column for the **MSU4** suite indicates that in 43 benchmarks, Recursive Cardinality Networks timed out whereas our new encoding did not. The columns next to it indicate that in 732 benchmarks the novel encoding was at least 4 times faster, in 2957 between 2 and 4 times faster, etc.

We can see from the table that in all three suites the new encoding clearly outperforms Recursive Cardinality Networks. The difference is larger in the **MSU4** and the **Tomography** suites, which contain benchmarks coming from real-world application. In the **Tomography** suite, where benchmarks are more hand-crafted, the difference is still significant.

[6] See `http://www.lsi.upc.edu/~oliveras/espai/detailed-SAT13.ods` for detailed results.

3.7 Conclusions and Future Work

We have introduced a new method for encoding cardinality constraints. Experimental results show that the method needs many fewer variables than the state-of-the-art method for encoding these constraints, while the number of clauses may increase in a controlled way. This reduction also yields significant speedups in the performance of SAT solvers.

As regards future work, we plan to combine non-recursive cardinality networks with other recursive approaches, for example the introduced in [CZI10], and compare the resulting mixed methods with the one described here.

Another line of research is to develop encoding techniques for cardinality constraints that do not process constraints one-at-a-time but simultaneously, in order to exploit their similarities. We foresee that the flexibility of the recursive cardinality networks presented here with respect to the original construction in [ANORC11b], will open the door to sharing the internal networks among the different cardinality constraints present in a SAT problem.

4

Encoding Pseudo-Boolean Constraints into SAT

4.1 Introduction

This section[1] deals with Pseudo-Boolean constraints (PB constraints for short), that is, constraints of the form $a_1x_1 + \cdots + a_nx_n \ \# \ K$, where the a_i and K are integer coefficients, the x_i are Boolean (0/1) variables, and the relation operator $\#$ belongs to $\{<, >, \leqslant, \geqslant, =\}$. We will assume that $\#$ is $\leqslant$ and the a_i and K are positive since other cases can be easily reduced to this one (see [ES06]).

Such a constraint ($\leqslant$ with positive coefficients) is a Boolean function

$$C\colon \{0,1\}^n \to \{0,1\}$$

that is monotonic decreasing in the sense that any solution for C remains a solution after flipping inputs from 1 to 0. Therefore these constraints can be expressed by a set of clauses with only negative literals. For example, each clause could simply define a (minimal) subset of variables that cannot be simultaneously true. Note however that not every such a monotonic function is a PB constraint. For example, the function expressed by the two clauses $\overline{x}_1 \vee \overline{x}_2$ and $\overline{x}_3 \vee \overline{x}_4$ has no (single) equivalent PB constraint $a_1x_1 + \cdots + a_4x_4 \leqslant K$ (since without loss of generality $a_1 \geqslant a_2$ and $a_3 \geqslant a_4$, and then also $\overline{x}_1 \vee \overline{x}_3$ is needed). Hence, even among the monotonic Boolean functions, PB constraints are a rather restricted class [Sma07].

PB constraints are omnipresent in practical SAT applications, not just in typical 0-1 linear integer problems, but also as an ingredient in new SAT approaches to, e.g., cumulative scheduling [SFSW09], logic synthesis [ARMS02] or verification [BLS02],

[1] Based in the paper "BDDs for Pseudo-Boolean Constraints - Revisited" [ANORC11a] from the SAT 2011 conference and the article "A New Look at BDDs for Pseudo-Boolean Constraints" [ANO$^+$] from the JAIR.

so it is not surprising that a significant number of SAT encodings for these constraints have been proposed in the literature. Here we are interested in encoding a PB constraint C by a clause set S (possibly with auxiliary variables) that is not only equisatisfiable, but also *arc-consistent*.

To our knowledge, the only polynomial arc-consistent encoding so far was given at [BBR09]. Some other existing encodings are based on building (forms of) Binary Decision Diagrams (BDDs) and translating these into CNF. Although the approach [BBR09] is not BDD-based, our main motivation to revisit BDD-based encodings is the following:

Example 4.1. *Let us consider two Pseudo-Boolean constraints:* $3x_1+2x_2+4x_3 \leqslant 5$ *and* $30001x_1+19999x_2+39998x_3 \leqslant 50007$. *Both are clearly equivalent: the Boolean function they represent can be expressed, for instance, by the clauses* $\overline{x}_1 \vee \overline{x}_3$ *and* $\overline{x}_2 \vee \overline{x}_3$. *However, encodings like the one at [BBR09] heavily depend on the concrete coefficients of each constraint, and generate a significantly larger SAT encoding for the second one. Since, given a variable ordering, ROBDDs are a canonical representation for Boolean functions [Bry86], i.e., each Boolean function has a unique ROBDD, a ROBDD-based encoding will treat both constraints equivalently.*

Another reason for revisiting BDDs is that in practical problems numerous PB constraints exist that share variables among each other. Representing them all as a single ROBDD has the potential of generating a much more compact SAT encoding that is moreover likely to have better propagation properties.

As we have mentioned, BDD-based approaches have already been studied in the literature. A good example is the work of MiniSAT+ [ES06], where an arc-consistent encoding using six three-literals clauses per BDD node is given. However, when it comes to study the BDD size, they cite at [BBR06a] to say *"It is proven that in general a PB-constraint can generate an exponentially sized BDD [BBR06a]"*. In Section 4.7 we explain why the approach of [BBR06a] does not use ROBDDs, and prove that the example they use to show the exponentiality of their method turns out to have polynomial ROBDDs. Somewhat surprisingly, probably due to the different names that PB constraints receive (0-1 integer linear constraints, linear threshold functions, weight constraints, knapsack constraints), the work explained at [HTY94] has remained unknown to our research community. In that paper, it is proved that there are PB constraints for which no polynomial-sized ROBDDs exist. For self-containedness, and to bring this interesting result to the knowledge of our research community, we include this family of PB constraints and prove that, regardless of the variable ordering, the corresponding ROBDD will always have exponential size.

This chapter is organized as follows: Section 4.2 contains some basic notions about Pseudo-Boolean constraints and BDDs. In Section 4.3 we introduce the notion of *interval* of a Pseudo-Boolean constraint and construct the ROBDDs for some families of constraints. In particular, an exponential example is presented, together with some intuitive relation of the BDD size and the subset sum problem. Section 4.4 contains a method for avoiding the exponentiality of the BDDs. In Section 4.5

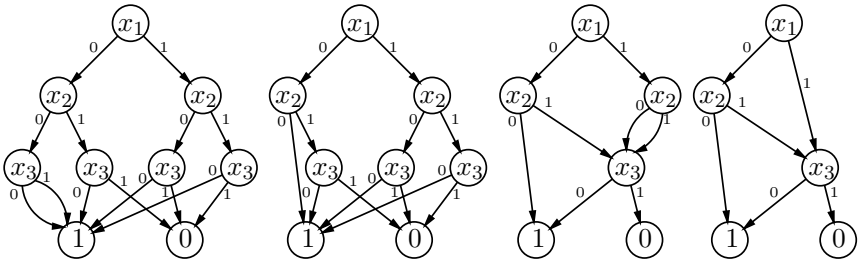

Figure 4.1: Construction of a ROBDD for $2x_1 + 3x_2 + 5x_3 \leqslant 6$

a polynomial algorithm for constructing a ROBDD of a Pseudo-Boolean constraint. In Section 4.6 a new method for encoding these ROBDDs into SAT is explained. This method uses only two clauses per node and is valid for any monotonic function. Section 4.7 contains the related work and Section 4.8 evaluates experimentally our methods with some other encoding methods and other Pseudo-Boolean Solvers. Finally, we conclude in Section 4.9.

4.2 Preliminaries

PB constraints are constraints of the form $a_1x_1 + \cdots + a_nx_n \leqslant K$, where the a_i and K are integer coefficients and the x_i are propositional variables. A particular case of Pseudo-Boolean constraints is the one of *cardinality constraints*, in which all coefficients a_i are equal to 1.

Our main goal is to find CNF encodings for PB constraints. That is, given a PB-constraint C, construct an equisatisfiable clause set (a CNF) S such that any model for S restricted to the variables of C is a model of C and viceversa. Two extra properties are sought: (i) *consistency checking by unit propagation* or simply *consistency*: whenever a partial assignment A cannot be extended to a model for C, unit propagation on S and A produces a contradiction (a literal l and its negation $\bar{l}$); and (ii) *arc-consistency* (again by unit propagation): given an assignment A that can be extended to a model of C, but such that $A \cup \{x\}$ cannot, unit propagation on S and A produces $\overline{x}$. More concretely, we will use ROBDDs for finding such encodings. ROBDDs are introduced by means of the following example.

Example 4.2. *Figure 4.1 explains (one method for) the construction of a ROBDD for the PB constraint* $2x_1+3x_2+5x_3 \leqslant 6$ *and the ordering* $[x_1, x_2, x_3]$. *The root node has as* selector variable x_1. *Its* false child *represents the PB constraint assuming* $x_1 = 0$ *(i.e.,* $3x_2 + 5x_3 \leqslant 6$*) and its* true child *represents* $2 + 3x_2 + 5x_3 \leqslant 6$, *that is,* $3x_2 + 5x_3 \leqslant 4$. *The two children have the next variable in the ordering* (x_2) *as selector, and the process is repeated until we reach the last variable in the sequence. Then, a constraint of the form* $0 \leqslant K$ *is the* True *node (1 in the figure) if* $K \geqslant 0$ *is positive, and the* False *node (0) if* $K < 0$. *This construction (leftmost*

in the figure), is known as an Ordered BDD. For obtaining a Reduced Ordered BDD (ROBDD for short in the rest of the document), two reductions are applied until fixpoint: removing nodes with identical children (as done with the leftmost x_3 node in the second BDD of the figure), and merging isomorphic subtrees, as done for x_3 in the third BDD. The fourth final BDD is a fixpoint. A BDD where only the second reduction is done is called quasi-reduced BDD*. For a given ordering, ROBDDs are a canonical representation of Boolean functions: each Boolean function has a unique ROBDD. BDDs can be encoded into CNF by introducing an auxiliary variable a for every node. If the selector variable of the node is x and the auxiliary variables for the false and true child are f and t, respectively, add the if-then-else clauses:*

$$\overline{x} \wedge \overline{f} \rightarrow \overline{a} \qquad x \wedge \overline{t} \rightarrow \overline{a} \qquad \overline{f} \wedge \overline{t} \rightarrow \overline{a}$$
$$\overline{x} \wedge f \rightarrow a \qquad x \wedge t \rightarrow a \qquad f \wedge t \rightarrow a$$

In what follows, the *size* of a BDD is its number of nodes. We will say that a BDD *represents* a PB constraint if they represent the same Boolean function. Given an assignment A over the variables of a BDD, we define the *path induced by* A as the path that starts at the root of the BDD and at each step, moves to the false (true) child of a node if and only if its selector variable is false (true) in A.

4.3 Exponential ROBDDs for PB Constraints

In this section we study the size of ROBDDs for PB constraints. We start by defining the notion of the *interval* of a PB constraint. Then, in Section 4.3.2 we consider two families of PB constraints and study their ROBDD size: we first prove that the example given at [BBR06a] has polynomial ROBDDs, and then we reproduce the example of [HTY94] that has exponential ROBDDs regardless of the variable ordering. Finally, we relate the ROBDD size of a PB constraint with the well-known subset sum problem.

4.3.1 Intervals

Before formally defining the notion of interval of a PB constraint, let us first give some intuitive explanation.

Example 4.3. *Consider the constraint $2x_1 + 3x_2 + 5x_3 \leqslant 6$. Since no combination of its coefficients adds to 6, the constraint is equivalent to $2x_1 + 3x_2 + 5x_3 < 6$, and hence to $2x_1 + 3x_2 + 5x_3 \leqslant 5$. This process cannot be repeated again since 5 can be obtained with the existing coefficients.*

Similarly, we could try to increase the right-hand side of the constraint. However, there is a combination of the coefficients that adds to 7, which implies that the constraint is not equivalent to $2x_1 + 3x_2 + 5x_3 \leqslant 7$. All in all, we can state that the constraint is equivalent to $2x_1 + 3x_2 + 5x_3 \leqslant K$ for any $K \in [5, 6]$. It is trivial to see that the set of valid K's is always an interval.

Definition 4.4. *Let C be a constraint of the form $a_1x_1 + \cdots + a_nx_n \leqslant K$. The* interval *of C consists of all integers M such that $a_1x_1 + \cdots + a_nx_n \leqslant M$, seen as a Boolean function, is equivalent to C.*

Similarly, given a ROBDD representing a PB constraint and a node ν with selector variable x_i,we will refer to the interval of *ν as all the integers M such that the constraint $a_ix_i + \cdots a_nx_n \leqslant M$ is represented (as a Boolean function) by the ROBDD rooted at ν.*

In the following, unless stated otherwise, the ordering used in the ROBDD will be $[x_1, x_2, \ldots, x_n]$.

Proposition 4.5. *If $[\beta, \gamma]$ is the interval of a node ν with selector variable x_i then:*

1. *There is an assignment $\{x_j = v_j\}_{j=i}^{n}$ such that $a_iv_i + \cdots + a_nv_n = \beta$.*
2. *There is an assignment $\{x_j = v_j\}_{j=i}^{n}$ such that $a_iv_i + \cdots + a_nv_n = \gamma + 1$.*
3. *There is an assignment $\{x_j = v_j\}_{j=1}^{i-1}$ such that $K - a_1v_1 - a_2v_2 - \cdots - a_{i-1}v_{i-1} \in [\beta, \gamma]$*
4. *Take $h < \beta$. There exists an assignment $\{x_j = v_j\}_{j=i}^{n}$ such that $a_iv_i + \cdots + a_nv_n > h$ and its path goes from ν to True.*
5. *Take $h > \gamma$. There exists an assignment $\{x_j = v_j\}_{j=i}^{n}$ such that $a_iv_i + \cdots + a_nv_n \leqslant h$ and its path goes from ν to False.*
6. *The interval of the True node is $[0, \infty)$.*
7. *The interval of the False node is $(-\infty, -1]$. Moreover, it is the only interval with negative values.*

Proof.

1. Since $\beta - 1$ does not belong to the interval of ν, the constraints

$$\begin{aligned} a_ix_i + a_{i+1}x_{i+1} + \cdots + a_nx_n &\leqslant \beta - 1 \\ a_ix_i + a_{i+1}x_{i+1} + \cdots + a_nx_n &\leqslant \beta \end{aligned}$$

   are different. This means that there is a partial assignment satisfying the second one but not the first one.

2. The proof is analogous to the previous one.

3. Take a partial assignment $\{x_1 = v_1, \ldots, x_{i-1} = v_{i-1}\}$ whose path goes from the root to ν. Therefore, by definition of the ROBDD, ν is the ROBDD of the constraint

$$a_ix_i + a_{i+1}x_{i+1} + \cdots + a_nx_n \leqslant K - a_1v_1 - \cdots - a_{i-1}v_{i-1}.$$

   Therefore, by definition of the interval of ν,

$$K - a_1v_1 - a_2v_2 - \cdots - a_{i-1}v_{i-1} \in [\beta, \gamma].$$

4. Intuitively, this property states that, if h is not in the interval of ν, there is an assignment that satisfies the ROBDD rooted at ν but not the constraint $a_i x_i + \cdots + a_n x_n \leqslant h$.

   Since h does not belong to the interval of ν, the ROBDD of

   $$C' : a_i x_i + \cdots + a_n x_n \leqslant h$$

   is not ν. Therefore, there exists an assignment that either

   (i) goes from ν to False but satisfies C'; or

   (ii) goes to True but does not satisfy C'.

   We want to prove that the assignment satisfies (ii). Assume that it satisfies (i). Since it goes from ν to False and β belongs to the interval of ν, it holds

   $$a_i v_i + \cdots + a_n v_n > \beta.$$

   Since $\beta > h$, the assignment does not satisfy C', which is a contradiction. Therefore, the assignment satisfies (ii).

5. Take the assignment of the second point of this proposition. Since $\gamma + 1$ does not belong to the interval, the path of the assignment goes from ν to False. Moreover, $a_i v_i + \cdots + a_n v_n = \gamma + 1 \leqslant h$.

6. The True node is the ROBDD of the tautology. Therefore, it represents the PB constraint $0 \leqslant h$ for $h \in [0, \infty)$.

7. The False node is the ROBDD of the contradiction. Therefore, represents the PB constraint $0 \leqslant h$ for $h \in (-\infty, -1]$. Moreover, $a_i x_i + \cdots + a_n x_n < 0$ is also a contradiction, hence that constraint is also represented by the False node. Therefore, there is no other node with an interval with negative values.

□

We now prove that, given a ROBDD for a PB constraint, one can easily compute the intervals for every node bottom-up. We first start with a motivating example.

Example 4.6. *Let us consider again the constraint* $2x_1+3x_2+5x_3 \leqslant 6$. *Assume that all variables appear in every path from the root to the leaves (otherwise, add extra nodes as in the rightmost BDD of Figure 4.2). Assume now that we have computed the intervals for the two children of the root (rightmost BDD in Figure 4.2). This means that the false child of the root is the BDD for* $3x_2 + 5x_3 \leqslant [5, 7]$ *and the true child the BDD for* $3x_2 + 5x_3 \leqslant [3, 4]$. *Assuming* x_1 *to be false, the false child would also represent the constraint* $2x_1 + 3x_2 + 5x_3 \leqslant [5, 7]$, *and assuming* x_1 *to be true, the true child would represent the constraint* $2x_1 + 3x_2 + 5x_3 \leqslant [5, 6]$. *Taking the intersection of the two intervals, we can infer that the root node represents* $2x_1 + 3x_2 + 5x_3 \leqslant [5, 6]$.

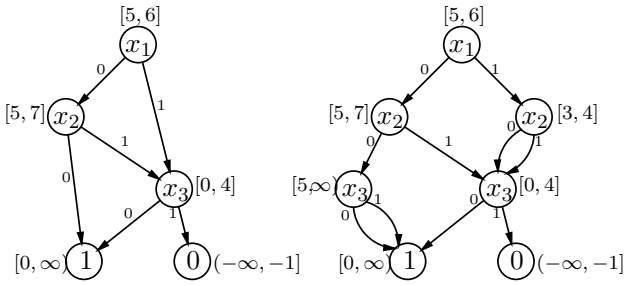

Figure 4.2: Intervals of the ROBDD for $2x_1 + 3x_2 + 5x_3 \leqslant 6$

More formally, the interval of every node can be computed as follows:

Proposition 4.7. *Let $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leqslant K$ be a constraint, and let $\mathcal{B}$ be its ROBDD with the order $[x_1, x_2, \ldots, x_n]$. Consider a node ν with selector variable x_i, false child ν_f (with selector variable x_f and interval $[\beta_f, \gamma_f]$) and true child ν_t (with selector variable x_t and interval $[\beta_t, \gamma_t]$), as shown in Figure 4.3. The interval of ν is $[\beta, \gamma]$, with:*

$$\beta = \max\{\beta_f + a_{i+1} + \cdots + a_{f-1},\ \ \beta_t + a_i + a_{i+1} + \cdots + a_{t-1}\},$$
$$\gamma = \min\{\gamma_f,\ \ \gamma_t + a_i\}.$$

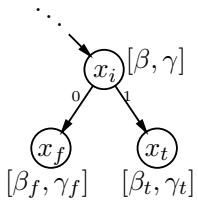

Figure 4.3: The interval of a node can be computed from its children's intervals.

Before moving to the proof, we want to note that if in every path from the root to the leaves of the ROBDD all variables were present, the definition of β would be much simpler ($\beta = \max\{\beta_f, \beta_t + a_i\}$). The other coefficients are necessary to account for the variables that have been removed due to the ROBDD reduction process.

Proof. Let us assume that $[\beta, \gamma]$ is not the interval of ν. One of the following statements should hold:

1. There exists $h \in [\beta, \gamma]$ that does not belong to the interval of ν.
2. There exists $h < \beta$ belonging to the interval of ν.
3. There exists $h > \gamma$ belonging to the interval of ν.

We will now prove that none of these cases can hold.

1. Let us define
$$C' : a_i x_i + \cdots + a_n x_n \leqslant h.$$
If h does not belong to the interval, there exists an assignment $\{x_j = v_j\}_{j=i}^{n}$ that either satisfies C' and its path goes from ν to False or it does not satisfy C' and its path goes to True. Assume that the assignment satisfies C' and its path goes from ν to False (the other case is similar). There are two possibilities:

   - The assignment satisfies $v_i = 0$. Since $h \geqslant \beta$, it holds
$$\begin{array}{rcl} h - a_{i+1}v_{i+1} - \cdots - a_{f-1}v_{f-1} & \geqslant & \beta - a_{i+1}v_{i+1} - \cdots - a_{f-1}v_{f-1} \\ & \geqslant & \beta - a_{i+1} - \cdots - a_{f-1} \geqslant \beta_f. \end{array}$$
     On the other hand, since $h \leqslant \gamma$,
$$h - a_{i+1}v_{i+1} - \cdots - a_{f-1}v_{f-1} \leqslant h \leqslant \gamma \leqslant \gamma_f.$$
     Therefore, $h - a_{i+1}v_{i+1} - \cdots - a_{f-1}v_{f-1}$ belongs to the interval of ν_f. Since the assignment $\{x_f = v_f, \ldots, x_n = v_n\}$ goes from ν_f to False, we have:
$$a_f v_f + \cdots + a_n v_n > h - a_{i+1}v_{i+1} - \cdots - a_{f-1}v_{f-1}$$
$$a_{i+1}v_{i+1} + \cdots + a_f v_f + \cdots a_n v_n > h$$
     Hence, adding $a_i v_i$ to the sum one can see that the assignment does not satisfy C', which is a contradiction.
   - The case $v_i = 1$ gives a similar contradiction.

2. By definition of β, either $h < \beta_f + a_{i+1} + \cdots + a_{f-1}$ or $h < \beta_t + a_i + a_{i+1} + \cdots + a_{t-1}$. We will only consider the first case, since the other one is similar. Therefore, $h - a_{i+1} - \cdots - a_{f-1} < \beta_f$. Due to point 4 of Proposition 4.5, there exists an assignment $\{x_f = v_f, \ldots x_n = v_n\}$ such that
$$a_f v_f + \cdots a_n v_n > h - a_{i+1} - \cdots - a_{f-1}$$
   and its path goes from ν_f to True. Hence, the assignment
$$\{x_i = 0, x_{i+1} = 1, \ldots, x_{f-1} = 1, x_f = v_f, \ldots, x_n = v_n\}$$
   does not satisfy the constraint $a_i x_i + \cdots + a_n x_n \leqslant h$ and its path goes from ν to True. By definition of interval, h cannot belong to the interval of ν.

3. This case is very similar to the previous one.

□

This proposition gives a natural way of computing all intervals of a ROBDD in a bottom-up fashion. The procedure is initialized by computing the intervals of the terminal nodes as detailed in Proposition 4.5, points 6 and 7.

Example 4.8. *Let us consider again the constraint* $2x_1+3x_2+5x_3 \leqslant 6$. *Its ROBDD is shown in the left-hand side of Figure 4.2, together with its intervals. For their computation, we first compute the intervals of the True and False nodes, which are* $[0,\infty)$ *and* $(-\infty,-1]$ *in virtue of Proposition 4.5 (points 6 and 7). Then, we can compute the interval of the node having* x_3 *as selector variable with the previous proposition's formula:* $\beta_3 = \max\{0,-\infty+5\} = 0$, $\gamma_3 = \min\{\infty,-1+5\} = 4$. *Therefore, its interval is* $[0,4]$.

In the next step, we compute the interval for the node with selector variable x_2*:* $\beta_2 = \max\{0+5,0+3\} = 5$, $\gamma_2 = \min\{\infty,4+3\} = 7$. *Thus, it its interval is* $[5,7]$. *Finally, we can compute the root's interval:* $\beta_1 = \max\{5,0+2+3\} = 5$, $\gamma_1 = \min\{7,4+2\} = 6$, *that is,* $[5,6]$.

4.3.2 Some Families of PB Constraints and their ROBDD Size

We start by revisiting the family of PB constraints given at [BBR06a], where it is proved that, for their concrete variable ordering, their non-reduced BDDs grow exponentially for this family. Here we prove that ROBDDs are polynomial for this family, and that this is even independent of the variable ordering. The family is defined by considering a, b and n positive integers such that $\sum_{i=1}^{n} b^i < a$. The coefficients are $\omega_i = a + b^i$ and the right-hand side of the constraint is $K = a \cdot n/2$. We will first prove that the constraint $C : \omega_1 x_1 + \cdots + \omega_n x_n \leqslant K$ is equivalent to the cardinality constraint $C' : x_1 + \cdots + x_n \leqslant n/2 - 1$. For simplicity, we assume that n is even.

- Take an assignment satisfying C'. In this case, there are at most $n/2-1$ variables x_i assigned to true, and the assignment also satisfies C since:

$$\omega_1 x_1 + \cdots + \omega_n x_n \leqslant \sum_{i=n/2+2}^{n} \omega_i = (n/2-1)a + \sum_{i=n/2+2}^{n} b^i < K - a + \sum_{i=1}^{n} b^i < K.$$

- Consider now an assignment not satisfying C'. In this case, there are at least $n/2$ true variables in the assignment and it does not satisfy C either:

$$\omega_1 x_1 + \cdots + \omega_n x_n \geqslant \sum_{i=1}^{n/2} \omega_i = (n/2) \cdot a + \sum_{i=1}^{n/2} b^i > (n/2) \cdot a = K.$$

Since the two constraints are equivalent and ROBDDs are canonical, the ROBDD representation of C and C' are the same. But the ROBDD of C' is known to be of quadratic size because it is a cardinality constraint (see, for instance, [BBR06a]).

In the following, we present a family of PB constraints that only admit exponential ROBDDs. This example was first given in [HTY94], but a clearer alternative proof is given next. First of all, we prove a lemma that, under certain technical conditions, gives a lower bound on the number of nodes of the ROBDD for a PB constraint.

Lemma 4.9. *Let $a_1x_1 + \cdots + a_nx_n \leqslant K$ be a PB constraint, and let i be an integer with $1 \leqslant i \leqslant n$. Assume that every assignment $\{x_1 = v_1, x_2 = v_2, \ldots, x_i = v_i\}$ admits an extension $\{x_1 = v_1, \ldots, x_n = v_n\}$ such that $a_1v_1 + \cdots + a_nv_n = K$. Let M be the number of different results we can obtain adding some subset of the coefficients $a_1, a_2, \ldots, a_i$, i.e., $M = |\{\sum_{j=1}^{i} a_jb_j \ : \ b_j \in \{0,1\}\}|$. Then, the ROBDD size with ordering $[x_1, x_2, \ldots, x_n]$ is at least M.*

Proof. Let us consider a PB constraint that satisfies the conditions of the lemma. We will prove that its ROBDD has at least M distinct nodes by showing that any two assignments of the form $\{x_1 = v_1, \ldots, x_i = v_i\}$ and $\{x_1 = v'_1, \ldots, x_i = v'_i\}$ with $a_1v_1 + \cdots + a_iv_i \neq a_1v'_1 + \cdots + a_iv'_i$ lead to different nodes in the ROBDD.

Assume that it is not true: there are two assignments $\{x_1 = v_1, \ldots, x_i = v_i\}$ and $\{x_1 = v'_1, \ldots, x_i = v'_i\}$ with $a_1v_1 + \cdots + a_iv_i < a_1v'_1 + \cdots + a_iv'_i$ such that their paths end at the same node. Take the extended assignment $A = \{x_1 = v_1, \ldots, x_n = v_n\}$ such that $a_1v_1 + \cdots a_nv_n = K$. Since A satisfies the PB constraint, the path it defines ends at the true node. However, the assignment $A' = \{x_1 = v'_1, \ldots, x_i = v'_i, x_{i+1} = v_{i+1}, \ldots, x_n = v_n\}$ does not satisfy the constraint, since

$$a_1v'_1 + \cdots a_iv'_i + a_{i+1}v_{i+1} + \cdots a_nv_n > a_1v_1 + \cdots + a_nv_n = K.$$

However, the nodes defined by $\{x_1 = v_1, \ldots, x_i = v_i\}$ and $\{x_1 = v'_1, \ldots, x_i = v'_i\}$ were the same, so the path defined by A' must also end at the true node, which is a contradiction. □

We can now show a family of PB constraints that only admits exponential ROBDDs.

Theorem 4.10. *Let n be a positive integer, and let us define $a_{i,j} = 2^{j-1} + 2^{2n+i-1}$ for all $1 \leqslant i, j \leqslant 2n$; and $K = (2^{4n} - 1)n$. Then, the PB constraint*

$$\sum_{i=1}^{2n} \sum_{j=1}^{2n} a_{i,j}x_{i,j} \leqslant K$$

has at least 2^n nodes in any variable ordering.

Proof. It is convenient to describe the coefficients in binary notation:

$$
\begin{array}{rcc|cccc|cccc}
 & & \overbrace{}^{2n} & & & & & \overbrace{}^{2n} & & & \\
a_{1,1} & = & 0 & 0 & \cdots & 0 & \mathbf{1} & 0 & 0 & \cdots & 0 & \mathbf{1} \\
a_{1,2} & = & 0 & 0 & \cdots & 0 & \mathbf{1} & 0 & 0 & \cdots & \mathbf{1} & 0 \\
\cdots & & & & & & & & & \iddots & & \\
a_{1,2n-1} & = & 0 & 0 & \cdots & 0 & \mathbf{1} & 0 & \mathbf{1} & \cdots & 0 & 0 \\
a_{1,2n} & = & 0 & 0 & \cdots & 0 & \mathbf{1} & \mathbf{1} & 0 & \cdots & 0 & 0 \\
 & & & & & & & & & & & \\
a_{2,1} & = & 0 & 0 & \cdots & \mathbf{1} & 0 & 0 & 0 & \cdots & 0 & \mathbf{1} \\
a_{2,2} & = & 0 & 0 & \cdots & \mathbf{1} & 0 & 0 & 0 & \cdots & \mathbf{1} & 0 \\
\cdots & & & & & & & & & \iddots & & \\
a_{2,2n-1} & = & 0 & 0 & \cdots & \mathbf{1} & 0 & 0 & \mathbf{1} & \cdots & 0 & 0 \\
a_{2,2n} & = & 0 & 0 & \cdots & \mathbf{1} & 0 & \mathbf{1} & 0 & \cdots & 0 & 0 \\
\cdots & & & & \iddots & & & & & & & \\
 & & & & & & & & & & & \\
a_{2n,2n} & = & \mathbf{1} & 0 & \cdots & 0 & 0 & \mathbf{1} & 0 & \cdots & 0 & 0 \\
 & & & & & & & & & & & \\
K/n & = & \mathbf{1} & \mathbf{1} & \cdots & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \cdots & \mathbf{1} & \mathbf{1}
\end{array}
$$

First of all, one can see that the sum of all the a's is $2K$.

Let us take an arbitrary bijection

$$F = (F_1, F_2) : \{1, 2, \ldots, 4n^2\} \to \{1, 2, \ldots, 2n\} \times \{1, 2, \ldots, 2n\},$$

and consider the ordering defined by it: $[x_{F(1)}, x_{F(2)}, \ldots, x_{F(4n^2)}]$, where $x_{F(k)} = x_{F_1(k),F_2(k)}$ for every k. We want to prove that the ROBDD of the PB constraint with this ordering has at least 2^n nodes.

The proof will consist in showing that the hypotheses of Lemma 4.9 hold. That is, first we show that for this arbitrary ordering, we can find an integer s such that any assignment to the first s variables can be extended to a full assignment that adds K. Then, we prove that there are at least 2^n different values we can add with the first s coefficients, as required by Lemma 4.9.

Let us define b_k with $1 \leqslant k \leqslant 2n$ as the position of the k-th different value of the tuple $(F_1(1), F_1(2), \ldots, F_1(4n^2))$. More formally,

$$b_k = \begin{cases} 1 & \text{if } k = 1, \\ \min\left\{r \ : \ F_1(r) \notin \{F_1(b_1), F_1(b_2), \ldots, F_1(b_{k-1})\}\right\} & \text{if } k > 1. \end{cases}$$

Analogously, let us define $c_1, \ldots, c_{2n}$ as

$$c_k = \begin{cases} 1 & \text{if } k = 1, \\ \min\left\{s \ : \ F_2(s) \notin \{F_2(c_1), F_2(c_2), \ldots, F_2(c_{k-1})\}\right\} & \text{if } k > 1. \end{cases}$$

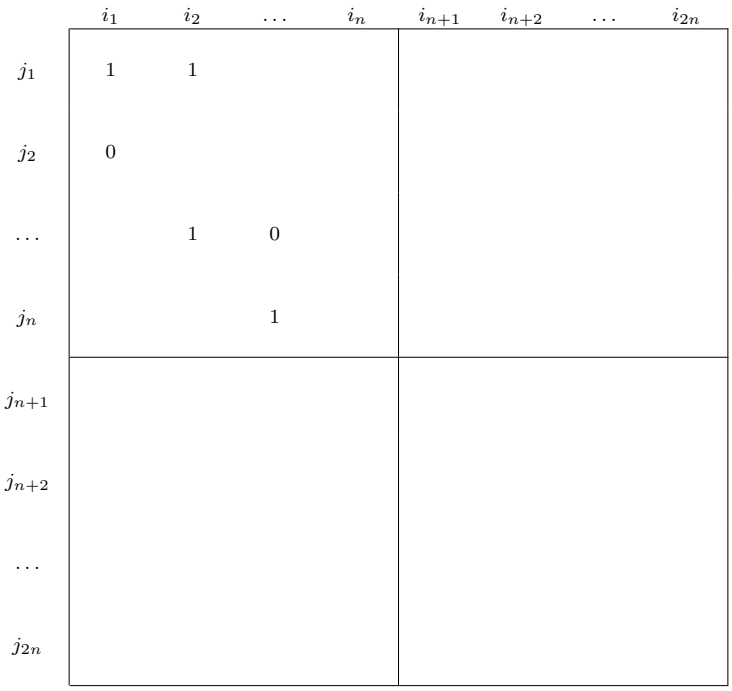

Figure 4.4: An arbitrary assignment. There is a 0, 1 or nothing at position (i_r, j_s) depending on whether x_{i_r,j_s} is false, true or unassigned.

Let us denote by $i_r = F_1(b_r)$ and $j_s = F_2(c_s)$ for all $1 \leqslant r, s \leqslant 2n$. Notice that $\{i_1, i_2, \ldots, i_{2n}\}$ and $\{j_1, j_2, \ldots, j_{2n}\}$ are permutations of $\{1, 2, \ldots, 2n\}$. Assume that $b_n \geqslant c_n$ (the other case is analogous), and take an arbitrary assignment $\{x_{F(1)} = v_{F(1)}, x_{F(2)} = v_{F(2)}, \ldots, x_{F(c_n)} = v_{F(c_n)}\}$. We want to extend it to a complete assignment such that

$$\sum_{k=1}^{4n^2} a_{F(k)} v_{F(k)} = K.$$

Figure 4.4 represents the initial assignment. All the values are in the top-left square since the assignment is undefined for all x_{i_r,j_s} with $r > n$ or $s > n$. Extending the assignment so that the sum is K amounts to completing the table in such a way that there are exactly n ones in every column and row.

The assignment can be completed in the following way: first, complete the top left square in any way, for instance, adding zeros to every non-defined cell. Then, copy that square to the bottom-right square and, finally, add the complementary square to the other two squares (i.e., write 0 instead of 1 and 1 instead of 0). Figure 4.5 shows the extended assignment for that example.

	i_1	i_2	...	i_n	i_{n+1}	i_{n+2}	...	i_{2n}
j_1	**1**	**1**	0	0	0	0	1	1
j_2	**0**	0	0	0	1	1	1	1
...	0	**1**	**0**	0	1	0	1	1
j_n	0	0	**1**	0	1	1	0	1
j_{n+1}	0	0	1	1	1	1	0	0
j_{n+2}	1	1	1	1	0	0	0	0
...	1	0	1	1	0	1	0	0
j_{2n}	1	1	0	1	0	0	1	0

Figure 4.5: Extended assignment. There are exactly n ones in every column and row.

More formally, the assignment is completed as follows:

$$v_{i_r,j_s} = \begin{cases} 0 & \text{if } r,s \leqslant n \text{ and } v_{i_r,j_s} \text{ was undefined,} \\ \neg v_{i_{r-n},j_s} & \text{if } r > n \text{ and } s \leqslant n, \\ \neg v_{i_r,j_{s-n}} & \text{if } s > n \text{ and } r \leqslant n, \\ v_{i_{r-n},j_{s-n}} & \text{if } r,s > n, \end{cases}$$

where $\neg 0 = 1$ and $\neg 1 = 0$.

Now, let us prove that it satisfies the requirements, i.e., the coefficients corresponding to true variables in the assignment add exactly K. Let us fix $r,s \leqslant n$. Denote by $i = i_r$, $j = j_s$, $i' = i_{r+n}$ and $j' = j_{s+n}$.

- If $v_{i,j} = 0$, by definition $v_{i',j} = v_{i,j'} = 1$ and $v_{i',j'} = 0$. Therefore,

$$a_{i,j}v_{i,j} + a_{i',j}v_{i',j} + a_{i,j'}v_{i,j'} + a_{i',j'}v_{i',j'} = a_{i',j} + a_{i,j'} = \\ 2^{2n+i'-1} + 2^{j-1} + 2^{2n+i-1} + 2^{j'-1} = (a_{i,j} + a_{i',j} + a_{i,j'} + a_{i',j'})/2.$$

- Analogously, if $v_{i,j} = 1$,

$$a_{i,j}v_{i,j} + a_{i',j}v_{i',j} + a_{i,j'}v_{i,j'} + a_{i',j'}v_{i',j'} = \frac{a_{i,j} + a_{i',j} + a_{i,j'} + a_{i',j'}}{2}.$$

Therefore,

$$\sum_{k=1}^{4n^2} a_{F(k)}v_{F(k)} = \frac{1}{2}\sum_{k=1}^{4n^2} a_{F(k)} = K.$$

By Lemma 4.9, the number of nodes of the ROBDD is at least the number of different results we can obtain by adding some subset of the coefficients $a_{F(1)}, a_{F(2)}, \ldots, a_{F(c_n)}$. Consider the set $a_{F(c_1)}, a_{F(c_2)}, \ldots, a_{F(c_n)}$. We will now see that all its different subsets add different values, and hence the ROBDD size is at least 2^n.

The sum of a subset of $\{a_{F(c_1)}, a_{F(c_2)}, \ldots, a_{F(c_n)}\}$ is

$$S = a_{F(c_1)}v_1 + a_{F(c_2)}v_2 + \cdots + a_{F(c_n)}v_n, \quad v_r \in \{0,1\}.$$

Let us look at the $2n$ last bits of S in binary notation: all the digits are 0 except for the positions $F_2(c_1), F_2(c_2), \ldots, F_2(c_n)$, which are $v_1, v_2, \ldots, v_n$. Therefore, if two subsets add the same, the $2n$ last digits of the sum are the same. This means that the values of $(v_1, \ldots, v_n)$ are the same, and thus they are the same subset. □

4.3.3 Relation between the Subset Sum Problem and the ROBDD size

In this section, we study the relationship between the ROBDD size for a PB constraint and the subset sum problem. This allows us to, assuming that NP and co-NP are different, give a much simpler proof that there exist PB constraints that do not admit polynomial ROBDDs.

Lemma 4.9 and the exponential ROBDD example of Theorem 4.10 suggest that there is a relationship between the size of ROBDDs and the number of ways we can obtain K by adding some of the coefficients of the PB. It seems that if K can be obtained in *a lot* of different ways, the ROBDD will be *large*.

In this section we explore another relationship between the problem of adding K with a subset of the coefficients and the size of the ROBDDs. In a sense, we give a proof that the converse of the last paragraph is not true: if NP and co-NP are different, there are exponentially-sized ROBDDs of PB constraints with no subsets of their coefficients adding K. Let us start by defining one version of the well-known *subset sum* problem.

Definition 4.11. *Given a set of positive integers $S = \{a_1, \ldots, a_n\}$ and an integer K, the* subset sum problem *of (K, S) consists in determining whether there exists a subset of $\{a_1, \ldots, a_n\}$ that sums to exactly K.*

It is well-known that the subset sum problem is NP-complete when $K \sim 2^n$, but there are polynomial algorithms in n when K is also a polynomial in n. For a given subset sum problem (K, S) with $S = \{a_1, \ldots, a_n\}$, we can construct its *associated PB constraint* $a_1x_1 + \cdots + a_nx_n \leqslant K$. In the previous section we have seen one PB constraint family whose coefficients can add K in an exponential number of ways, thus generating an exponential ROBDD. Now, assuming that NP and co-NP are different, we will see that there exists a PB constraint family with exponential ROBDDs in any ordering such that their coefficients cannot add K. First, we show how ROBDDs can act as unsatisfiability certificates for the subset sum problem.

Theorem 4.12. *Let (K, S) be an UNSAT subset sum problem. Then, if a ROBDD for its associated PB constraint has polynomial size, it can act as a polynomial unsatisfiability certificate for (K, S).*

Proof. We only need to show how, in polynomial time, we can check whether the ROBDD is an unsatisfiability certificate for (K, S). For that, we note that the subset sum problem is UNSAT if and only if the PB constraints

$$a_1x_1 + \cdots + a_nx_n \leqslant K, \quad a_1x_1 + \cdots + a_nx_n \leqslant K - 1$$

are equivalent, and this happens if and only if their ROBDDs are the same. Therefore, we have to show, in polynomial time, that the given ROBDD represents both constraints. It can be done, for instance, by building the ROBDD (using Algorithm 4.1 of Section 4.5) and comparing the ROBDDs. □

The key point now is that, if we assume NP to be different from co-NP, there exists a family of UNSAT subset sum problems with no polynomial-sized unsatisfiability certificate. Hence, the family consisting of the associated PB constraints does not admit polynomial ROBDDs.

Hence, PB constraints associated with difficult-to-certify UNSAT subset sum problems will produce exponential ROBDDs. However, subset sum is NP-complete if $K \sim 2^n$. In PB constraints from industrial problems usually $K \sim n^r$ for some r, so we could expect non-exponential ROBDDs for these constraints.

4.4 Avoiding Exponential ROBDDs

In this section we introduce our positive results. We restrict ourselves to a particular class of PB constraints, where all coefficients are powers of two. As we will show below, these constraints admit polynomial ROBDDs. Moreover, any PB constraint can be reduced to this class by means of *coefficient decomposition*.

Example 4.13. *Let us take the PB constraint* $9x_1 + 8x_2 + 3x_3 \leqslant 10$. *Considering the binary representation of the coefficients, this constraint can be rewritten into* $(2^3x_{3,1} + 2^0x_{0,1}) + (2^3x_{3,2}) + (2^1x_{1,3} + 2^0x_{0,3}) \leqslant 10$ *if we add the binary clauses expressing that* $x_{i,r} = x_r$ *for appropriate* i *and* r.

4.4.1 Power-of-two PB Constraints Do Have Polynomial-size ROBDDs

Let us consider a PB constraint of the form:

$$\begin{array}{rccccccccccc} C: & 2^0\cdot\delta_{0,1}\cdot x_{0,1} & + & 2^0\cdot\delta_{0,2}\cdot x_{0,2} & + & \cdots & + & 2^0\cdot\delta_{0,n}\cdot x_{0,n} & + \\ & 2^1\cdot\delta_{1,1}\cdot x_{1,1} & + & 2^1\cdot\delta_{1,2}\cdot x_{1,2} & + & \cdots & + & 2^1\cdot\delta_{1,n}\cdot x_{1,n} & + \\ & & & & & \cdots & & & + \\ & 2^m\cdot\delta_{m,1}\cdot x_{m,1} & + & 2^m\cdot\delta_{m,2}\cdot x_{m,2} & + & \cdots & + & 2^m\cdot\delta_{m,n}\cdot x_{m,n} & \leqslant & K, \end{array}$$

where $\delta_{i,r} \in \{0,1\}$ for all i and r. Notice that every PB constraint whose coefficients are powers of 2 can be expressed in this way. Let us consider its ROBDD representation with the ordering $[x_{0,1}, x_{0,2}, \ldots, x_{0,n}, x_{1,1}, \ldots, x_{m,n}]$.

Lemma 4.14. *Let* $[\beta, \gamma]$ *be the interval of a node with selector variable* $x_{i,r}$. *Then* 2^i *divides* β *and* $0 \leqslant \beta < (n + r - 1)\cdot 2^i$.

Proof. By Proposition 4.5.1, β can be expressed as a sum of coefficients all of which are multiples of 2^i, and hence β itself is a multiple of 2^i. By Proposition 4.5.7, the only node whose interval contains negative values is the False node, and hence $\beta \geqslant 0$. Now, using Proposition 4.5.3, there must be an assignment to the variables $\{x_{0,1}, \ldots, x_{i,r-1}\}$ such that $2^0\delta_{0,1}x_{0,1} + \cdots + 2^i\delta_{i,r-1}x_{i,r-1}$ belongs to the interval. Therefore:

$$\begin{aligned} \beta &\leqslant 2^0\delta_{0,1}x_{0,1} + \cdots + 2^i\delta_{i,r-1}x_{i,r-1} \leqslant 2^0 + 2^0 + \cdots + 2^i \\ &= n2^0 + n2^1 + \cdots + n2^{i-1} + (r-1)\cdot 2^i = n(2^i - 1) + 2^i(r-1) \\ &< 2^i(n + r - 1) \end{aligned}$$

□

Corollary 4.15. *The number of nodes with selector variable* $x_{i,r}$ *is bounded by* $n + r - 1$. *In particular, the size of the ROBDD belongs to* $\mathcal{O}(n^2m)$.

Proof. Let $\nu_1, \nu_2, \ldots, \nu_t$ be all the nodes with selector variable $x_{i,r}$. Let $[\beta_j, \gamma_j]$ the interval of ν_j. Note that such intervals are pair-wise disjoint since a non-empty intersection would imply that there exists a constraint represented by two different ROBDDs. Hence we can assume, without loss of generality, that $\beta_1 < \beta_2 < \cdots < \beta_t$. Due to Lemma 4.14, we know that $\beta_j - \beta_{j-1} \geqslant 2^i$. Hence $2^i(n + r - 1) > \beta_t \geqslant \beta_{t-1} + 2^i \geqslant \cdots \geqslant \beta_1 + 2^i(t-1) \geqslant 2^i(t-1)$ and we can conclude that $t < n + r$. □

4.4.2 A Consistent Encoding for PB Constraints

Let us now take an arbitrary PB constraint $C : a_1x_1 + \cdots a_nx_n \leqslant K$ and assume that a_M is the largest coefficient. For $m = \log a_M$, we can rewrite C splitting the coefficients into powers of two as shown in Example 4.13:

$$\begin{array}{rccccccccc}
\tilde{C}: & 2^0 \cdot \delta_{0,1} \cdot x_{0,1} & + & 2^0 \cdot \delta_{0,2} \cdot x_{0,2} & + \cdots + & 2^0 \cdot \delta_{0,n} \cdot x_{0,n} & + \\
& 2^1 \cdot \delta_{1,1} \cdot x_{1,1} & + & 2^1 \cdot \delta_{1,2} \cdot x_{1,2} & + \cdots + & 2^1 \cdot \delta_{1,n} \cdot x_{1,n} & + \\
& & & \cdots & & & + \\
& 2^m \cdot \delta_{m,1} \cdot x_{m,1} & + & 2^m \cdot \delta_{m,2} \cdot x_{m,2} & + \cdots + & 2^m \cdot \delta_{m,n} \cdot x_{m,n} & \leqslant & K,
\end{array}$$

where $\delta_{m,r}\ \delta_{m-1,r}\ \cdots\ \delta_{0,r}$ is the binary representation of a_r. Notice that C and $\tilde{C}$ represent the same constraint if we add clauses expressing that $x_{i,r} = x_r$ for appropriate i and r. This process is called *coefficient decomposition* of the PB constraint. A similar idea was given at [BB03b].

The important remark is that, using a consistent SAT encoding of the ROBDD for $\tilde{C}$ (e.g. the one given at [ES06] or the one presented in Section 4.6) and adding clauses expressing that $x_{i,r} = x_r$ for appropriate i and r, we obtain a consistent encoding for the original constraint C using $\mathcal{O}(n^2 \log a_M)$ auxiliary variables and clauses.

This is not difficult to see. Take an assignment A over the variables of C which cannot be extended to a model of C. This is because the coefficients corresponding to the variables true in A add more than K. Using the clauses for $x_{i,r} = x_r$, unit propagation will produce an assignment to the $x_{i,r}$'s that cannot be extended to a model of $\tilde{C}$. Since the encoding for $\tilde{C}$ is consistent, a false clause will be found. Conversely, if we consider an assignment A over the variables of C that can be extended to a model of C, this assignment can clearly be extended to a model for $\tilde{C}$ and the clauses expressing $x_{i,r} = x_r$. Hence, unit propagation on those clauses and the encoding of $\tilde{C}$ will not detect a false clause.

Example 4.16. *Consider the PB constraint $C : 2x_1 + 3x_2 + 5x_3 \leqslant 6$. For obtaining the consistent encoding we have presented, we first rewrite C by splitting the coefficients into powers of two:*

$$\tilde{C} : 1x_{0,2} + 1x_{0,3} + 2x_{1,1} + 2x_{1,2} + 4x_{2,3} \leqslant 6.$$

Next, we encode $\tilde{C}$ into a ROBDD and finally encode the ROBDD into SAT and add clauses for enforcing the relations $x_{i,j} = x_j$. Or, instead of that, we can replace $x_{i,j}$ by x_j into the ROBDD, and encode the result into SAT. Figure 4.6 shows the decision diagram after the replacement.

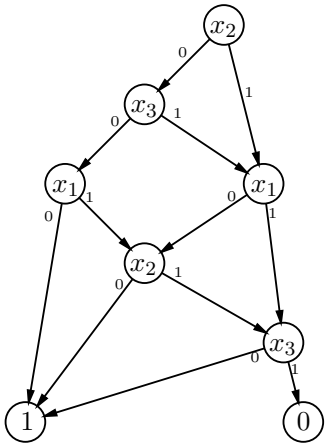

Figure 4.6: Decision Diagram of $2x_1 + 3x_2 + 5x_3 \leqslant 6$ after decomposing the coefficients into powers of two.

4.4.3 An Arc-consistent Encoding for PB Constraints

Unfortunately, the previous approach does not result in an arc-consistent encoding. The intuitive idea can be seen in the following example:

Example 4.17. *Let us consider the constraint* $3x_1 + 4x_2 \leqslant 6$. *After splitting the coefficients into powers of two, we obtain* $\tilde{C} : x_{0,1} + 2x_{1,1} + 4x_{2,2} \leqslant 6$. *If we set* $x_{2,2}$ *to true,* $\tilde{C}$ *implies that either* $x_{0,1}$ *or* $x_{1,1}$ *have to be false, but the encoding cannot exploit the fact that both variables will receive the same truth value and hence both should be propagated. Adding clauses stating that* $x_{0,1} = x_{1,1}$ *does not help in this sense.*

In order to overcome this limitation, we follow the method presented at the papers [BKNW09, BBR09]. Let $C : a_1x_1 + \cdots + a_nx_n \leqslant K$ be an arbitrary PB constraint. We denote as C_i the constraint $a_1x_1 + \cdots + a_i \cdot 1 + \cdots + a_nx_n \leqslant K$, i.e., the constraint assuming x_i to be true. For every i with $1 \leqslant i \leqslant n$, we encode C_i as in Section 4.4.2 and, in addition, we add the binary clause $r_i \vee \overline{x}_i$, where r_i is the root of the ROBDD for C_i. This clause helps us to preserve arc-consistency: given an assignment A such that $A \cup \{x_i\}$ cannot be extended to a model of C, literal $\overline{r}_i$ will be propagated using A (because the encoding for C_i is consistent). Hence the added clause will allow us to propagate $\overline{x}_i$.

Example 4.18. *Consider again the PB constraint* $C : 2x_1 + 3x_2 + 5x_3 \leqslant 6$. *Let us define the constraints* $C_1 : 3x_2 + 5x_3 \leqslant 4$, $C_2 : 2x_1 + 5x_3 \leqslant 3$ *and* $C_3 : 2x_1 + 3x_2 \leqslant 1$. *Now, we encode these constraints into ROBDDs as in the previous section, with coefficient decomposition. Figure 4.7 shows the resulting ROBDDs. Finally, we need to encode them into SAT consistently, and then add the clauses* $r_i \vee \overline{x}_i$, *assuming that the variable associated with the root of the ROBDD for* C_i *is* r_i.

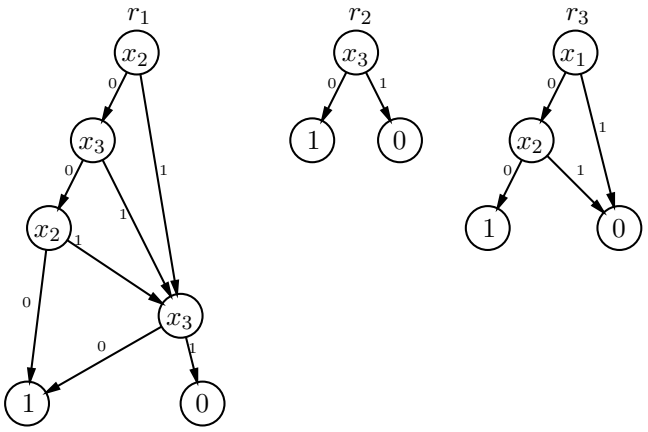

Figure 4.7: ROBDDs[2] of C_1, C_2 and C_3 with coefficient decomposition.

This encoding is arc-consistent: take for instance the assignment $A = \{x_1 = 1\}$. Constraint C_3 is not satisfied. Hence, by consistency, $\overline{r}_3$ is propagated. Therefore, clause $r_3 \vee \overline{x}_3$ propagates $\overline{x}_3$, as wanted. The propagation with other assignments is similar.

All in all, the suggested encoding is arc-consistent and uses $\mathcal{O}(n^3 \log(a_M))$ clauses and auxiliary variables, where a_M is the largest coefficient.

4.5 An Algorithm for Constructing ROBDDs for Pseudo-Boolean Constraints

Let us fix a Pseudo-Boolean constraint $a_1x_1 + \cdots + a_nx_n \leqslant K$ and a variable ordering $[x_1, x_2, \ldots, x_n]$. The goal of this section is to prove that one can construct the ROBDD of this constraint using this ordering in polynomial time with respect to the ROBDD size and n.

This algorithm builds standard ROBDDs, but it can be used to build ROBDDs with coefficient decomposition: we just need to first split the coefficients and, secondly, apply the algorithm. Forthcoming Example 4.21 shows in detail the overall process.

The key point of the algorithm will be to label each node of the ROBDD with its interval. In the following, for every $i \in \{1, 2, \ldots, n+1\}$, we will use a set L_i consisting of pairs $([\beta, \gamma], \mathcal{B})$, where $\mathcal{B}$ is the ROBDD of the constraint $a_ix_i + \cdots + a_nx_n \leqslant K'$ for every $K' \in [\beta, \gamma]$ (i.e., $[\beta, \gamma]$ is the interval of $\mathcal{B}$). All these sets will be kept in a tuple $\mathcal{L} = (L_1, L_2, \ldots, L_{n+1})$.

[2]Actually, the diagram after replacing the variables $x_{i,j}$ by x_j is not a ROBDD. However, we will denote them as ROBDDs for simplicity.

Note that by definition of the ROBDD's intervals, if L_i contains $([\beta_1, \gamma_1], \mathcal{B}_1)$ and $([\beta_2, \gamma_2], \mathcal{B}_2)$, then either $[\beta_1, \gamma_1] = [\beta_2, \gamma_2]$ or $[\beta_1, \gamma_1] \cap [\beta_2, \gamma_2] = \emptyset$. Moreover, the first case holds if and only if $\mathcal{B}_1 = \mathcal{B}_2$. Therefore, L_i can be represented with a *binary search tree-like* data structure, where insertions and searches can be done in logarithmic time. The function **search**(K, L_i) searches whether there exists a pair $([\beta, \gamma], \mathcal{B}) \in L_i$ with $K \in [\beta, \gamma]$. Such a tuple is returned if it exists, otherwise an empty interval is returned in the first component of the pair. Similarly, we will also use function **insert**$(([\beta, \gamma], \mathcal{B}), L_i)$ for insertions. The overall algorithm is detailed in Algorithm 4.1 and Algorithm 4.2:

Algorithm 4.1 Construction of ROBDD algorithm

Require: Constraint $C : a_1x_1 + \ldots + a_nx_n \leqslant K'$.
Ensure: returns $\mathcal{B}$ the ROBDD of C.
1: **for all** i such that $1 \leqslant i \leqslant n+1$ **do**
2: $\quad L_i \leftarrow \Big\{ \big((-\infty, -1], False\big),\ \big([a_i + a_{i+1} + \cdots + a_n, \infty), True\big) \Big\}$.
3: **end for**
4: $\mathcal{L} \leftarrow (L_1, \ldots, L_{n+1})$.
5: $([\beta, \gamma], \mathcal{B}) \leftarrow$ **BDDConstruction**$(1, a_1x_1 + \ldots + a_nx_n \leqslant K', \mathcal{L})$.
6: **return** $\mathcal{B}$.

Algorithm 4.2 Procedure BDDConstruction

Require: integer $i \in \{1, 2, \ldots, n+1\}$, constraint $C : a_ix_i + \ldots + a_nx_n \leqslant K'$ and set $\mathcal{L}$.
Ensure: returns $[\beta, \gamma]$ interval of C and $\mathcal{B}$ its ROBDD.
1: $([\beta, \gamma], \mathcal{B}) \leftarrow$ **search**(K', L_i).
2: **if** $[\beta, \gamma] \neq \emptyset$ **then**
3: $\quad$ **return** $([\beta, \gamma], \mathcal{B})$.
4: **else**
5: $\quad ([\beta_F, \gamma_F], \mathcal{B}_F) :=$ **BDDConstruction**$(i+1, a_{i+1}x_{i+1} + \ldots + a_nx_n \leqslant K', \mathcal{L})$.
6: $\quad ([\beta_T, \gamma_T], \mathcal{B}_T) :=$ **BDDConstruction**$(i+1, a_{i+1}x_{i+1} + \ldots + a_nx_n \leqslant K' - a_i, \mathcal{L})$.
7: $\quad$ **if** $[\beta_T, \gamma_T] = [\beta_F, \gamma_F]$ **then**
8: $\quad\quad \mathcal{B} \leftarrow \mathcal{B}_T$.
9: $\quad\quad [\beta, \gamma] \leftarrow [\beta_T + a_i, \gamma_T]$.
10: $\quad$ **else**
11: $\quad\quad \mathcal{B} \leftarrow$ **ite**$(x_i, \mathcal{B}_T, \mathcal{B}_F)$ // root x_i, true branch $\mathcal{B}_T$ and false branch $\mathcal{B}_F$.
12: $\quad\quad [\beta, \gamma] \leftarrow [\beta_F, \gamma_F] \cap [\beta_T + a_i, \gamma_T + a_i]$.
13: $\quad$ **end if**
14: $\quad$ **insert**$(([\beta, \gamma], \mathcal{B}), L_i)$.
15: $\quad$ **return** $([\beta, \gamma], \mathcal{B})$.
16: **end if**

Theorem 4.19. *Algorithm 4.1 runs in $\mathcal{O}(nm \log m)$ time (where m is the size of*

the ROBDD) and is correct in the following sense:

1. *K' belongs to the interval returned by* **BDDConstruction**$(a_i x_i + \cdots + a_n x_n \leqslant K', \mathcal{L})$.
2. *The tuple $([\beta, \gamma], \mathcal{B})$ returned by* **BDDConstruction** *consist of a BDD $\mathcal{B}$ and its interval $[\beta, \gamma]$.*
3. *If* **BDDConstruction** *returns $([\beta, \gamma], \mathcal{B})$, then the BDD $\mathcal{B}$ is reduced.*

Proof. Let us first start with the three correctness statements:

1. If K' is found in L_i at line 1 of Algorithm 4.2 the statement is obviously true. Otherwise let us reason by induction on i. The base case is when $i = n + 1$, and since L_{n+1} contains the intervals $(-\infty, -1]$ and $[0, \infty]$, the **search** call at line 1 will succeed and hence the result holds. For $i < n + 1$ we can assume, by induction hypothesis, that $K' \in [\beta_F, \gamma_F]$ and $K' - a_i \in [\beta_T, \gamma_T]$. If the two intervals coincide the result is obvious, otherwise it is also easy to see that $K' \in [\beta_F, \gamma_F] \cap [\beta_T + a_i, \gamma_T + a_i]$.

2. Let us prove that in every moment all the tuples of $\mathcal{L}$ are correct, i.e., they contain BDDs with their correct interval. Since the returned value is always an element of some L_i, this proves the statement.

   By Proposition 4.5.6 and 4.5.7 initial tuples of $\mathcal{L}$ are correct. We have to prove that if all the previously inserted intervals are correct, the current interval is also correct. It follows in virtue of Proposition 4.7.

3. Let us prove that all the tuples of $\mathcal{L}$ contain only reduced BDDs. As before, all the initial BDDs in $\mathcal{L}$ are reduced. Let $\mathcal{B}$ be a BDD computed by the algorithm, with children $\mathcal{B}_T$ and $\mathcal{B}_F$. By induction hypothesis, they are reduced, so $\mathcal{B}$ is reduced if and only if its two children are not equal. The algorithm creates a node only if its children's intervals are different. Therefore, $\mathcal{B}_T$ and $\mathcal{B}_F$ do not represent the same Boolean constraint, so they are different BDDs.

Regarding runtime, since the cost of search and insertion in L_i is logarithmic in its size, the cost of the algorithm is $\mathcal{O}(\log m)$ times the number of calls to **BDDConstruction**. Hence, it only remains to show that there are at most $\mathcal{O}(nm)$ calls.

Every call (but the first one) to **BDDConstruction** is done when we are exploring an edge of the ROBDD. Notice that no edge is explored twice, since the edges are only explored from the parent node and whenever we reach an explored node there are no recursive calls to **BDDConstruction**. On the other hand, for every edge of the ROBDD we make $2k - 1$ calls, where k is the *length* of the edge (if the nodes joined by the edge have variables x_i and x_j we say that its *length* is $|i - j|$). Since the ROBDD has $\mathcal{O}(m)$ edges and their length is $\mathcal{O}(n)$, the number of calls is $\mathcal{O}(nm)$. □

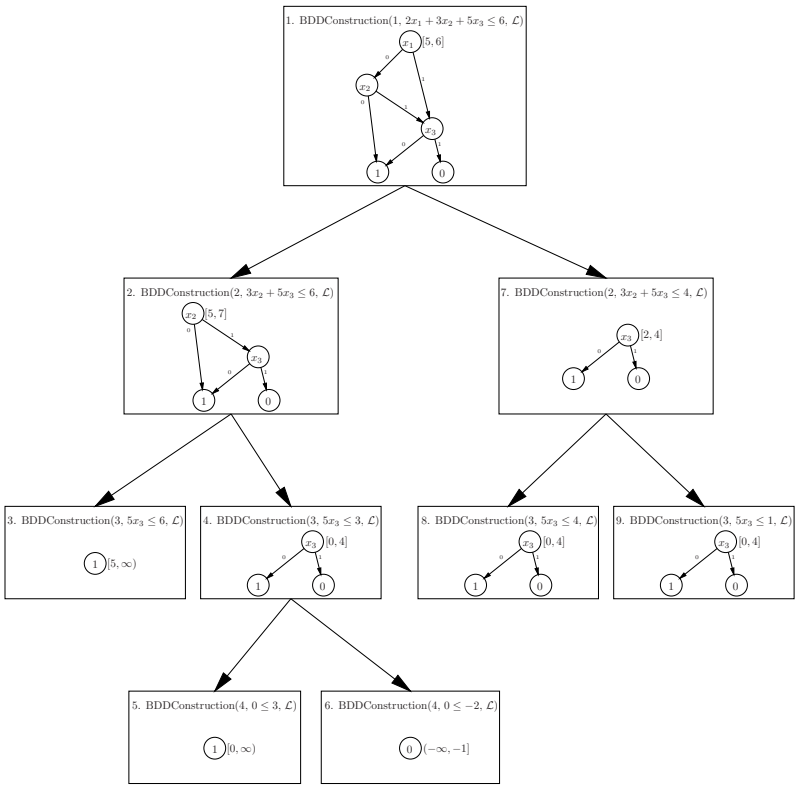

Figure 4.8: Recursive calls to **BDDConstruction**, with the returned values.

Let us illustrate the algorithm with an example:

Example 4.20. *Take the constraint* $C : 2x_1 + 3x_2 + 5x_3 \leqslant 6$*, and let us apply the algorithm to obtain the ROBDD in the ordering* $[x_1, x_2, x_3]$*. Figure 4.8 represents the recursive calls to BDDConstruction and the returned parameters (the ROBDD and the interval).*

- *In calls number 3, 5, 6, 8 and 9, the search function returns true and the interval and the ROBDD are returned without any other computation.*
- *In call number 7, the two recursive calls return the same interval (and, therefore, the same ROBDD). Hence, that ROBDD is returned.*
- *In call number 1 the two recursive calls return two different ROBDDs, so it adds a node to join the two ROBDDs into another one, which is returned. The same happens in calls number 2 and 4.*

The overall process with coefficient decomposition would work as in the following example:

Example 4.21. *Let us take the constraint* $C : 2x_1 + 3x_2 + 5x_3 \leqslant 6$. *If we want to build the ROBDD with coefficient decomposition using Algorithm 4.1, we proceed as follows:*

1. *Split the coefficients and obtain* $\tilde{C} : 1y_1 + 1y_2 + 2y_3 + 2y_4 + 4y_5 \leqslant 6$, *where* $x_1 = y_3$, $x_2 = y_1 = y_4$ *and* $x_3 = y_2 = y_5$.
2. *Apply the algorithm to* $\tilde{C}$ *and obtain a ROBDD* $\tilde{\mathcal{B}}$.
3. *Replace* y_1 *for* x_2, y_2 *for* x_3, *etc. in the nodes of* $\tilde{\mathcal{B}}$.

4.6 Encoding a BDDs for Monotonic Functions into SAT

In this section we consider a BDD representing a monotonic function F and we want to encode it into SAT. As expected, we want the encoding to be as small as possible and arc-consistent.

The encoding explained here is valid with any type of BDDs, so, in particular, it is valid with ROBDDs. The main differences with the Minisat+ encoding [ES06] is the number of clauses generated (6 ternary clauses per node versus one binary and one ternary clauses per node) and that our encoding is arc-consistent with any variable ordering.

As usual, the encoding introduces an auxiliary variable for every node. Let ν be a node with selector variable x and auxiliary variable n. Let f be the variable of its false child and t be the variable of its true child. Only two clauses per node are needed:

$$\overline{f} \to \overline{n} \qquad \overline{t} \wedge x \to \overline{n}.$$

Furthermore, we add a unit clause with the variable of the True node and another one with the negation of the variable of the False node.

Theorem 4.22. *The encoding is consistent in the following sense: a partial assignment* A *cannot be extended to a model of* F *if and only if* $\overline{r}$ *is propagated by unit propagation, where* r *is the root of the BDD.*

Proof. We prove the theorem by induction on the number of variables of the BDD. If the BDD has no variables, then the BDD is either the True node or the False node and the result is trivial.

Assume that the result is true for BDDs with less than k variables, and let F be a function whose BDD has k variables. Let r be the root node, x_1 its selector variable and f, t respectively its false and true children (note that we abuse the notation and identify nodes with their auxiliary variable). We denote by F_1 the function $F_{|x_1=1}$ (i.e., F after setting x_1 to true) and by F_0 the function $F_{|x_1=0}$.

- Let A be a partial assignment that cannot be extended to a model of F.

  - Assume $x_1 \in A$. Since A cannot be extended, the assignment $A \setminus \{x_1\}$ cannot be extended to a model of F_1. By definition of the BDD, the function F_1 has t as a BDD. By induction hypothesis, $\overline{t}$ is propagated, and since $x_1 \in A$, $\overline{r}$ is also propagated.
  - Assume $x_1 \notin A$. Then, the assignment $A \setminus \{\overline{x}_1\}$ cannot be extended to a model of F_0. Since F_0 has f as a BDD, by induction hypothesis $\overline{f}$ is propagated, and hence $\overline{r}$ also is.

- Let A be a partial assignment, and assume $\overline{r}$ has been propagated. Then, either $\overline{f}$ has also been propagated or $\overline{t}$ has been propagated and $x_1 \in A$ (note that x_1 has not been propagated because it only appears in one clause which is already true).
  - Assume that $\overline{f}$ has been propagated. Since f is the BDD of F_0, by induction hypothesis the assignment $A \setminus \{x_1, \overline{x}_1\}$ cannot be extended to a model of F_0. Since the function is monotonic, neither can $A \setminus \{x_1, \overline{x}_1\}$ be extended to a model of F. Therefore, A cannot be extended to a model of F.
  - Assume that $\overline{t}$ has been propagated and $x_1 \in A$. Since t is the BDD of F_1, by induction hypothesis $A \setminus \{x_1\}$ cannot be extended to a model of F_1, so neither can A be extended to a model of F.

□

For obtaining an arc-consistent encoding, we only have to add a unit clause.

Theorem 4.23. *If we add a unit clause forcing the variable of the root node to be true, the previous encoding becomes arc-consistent.*

Proof. We will prove it by induction on the variables of the BDD. The case with zero variables is trivial, so let us prove the induction case.

As before, let r be the root node, with x_1 its selector variable, and f, t its false and true children. We denote by F_0 and F_1 the functions $F_{|x_1=0}$ and $F_{|x_1=1}$.

Let A be a partial assignment that can be extended to a model of F. Assume that $A \cup \{x_i\}$ cannot be extended. We want to prove that $\overline{x}_i$ will be propagated.

- Let us assume that $x_1 \in A$. In this case, t is propagated due to the clause $\overline{t} \wedge x_1 \rightarrow \overline{r}$ and the unit clause r. Since $x_1 \in A$ and $A \cup \{x_i\}$ cannot be extended to a model of F, $A \setminus \{x_1\} \cup \{x_i\}$ neither can be extended to an assignment satisfying F_1. By induction hypothesis, since t is the BDD of the function F_1, $\overline{x}_i$ is propagated.

- Let us assume that $x_1 \notin A$ and $x_i \neq x_1$. Since F is monotonic, $A \cup \{x_i\}$ cannot be extended to a model of F if and only if it cannot be extended to a model of F_0. Notice that f is propagated thanks to the clause $\overline{f} \rightarrow \overline{r}$ and the unit clause r. By induction hypothesis, the method is arc-consistent for F_0, so $\overline{x}_i$ is propagated.

- Finally, assume that $x_1 \notin A$ and $x_i = x_1$. Since $A \cup \{x_1\}$ cannot be extended to a model of F, A cannot be extended to a model of F_1. By Theorem 4.22, $\overline{t}$ is propagated and, due to $\overline{t} \wedge x_1 \rightarrow \overline{r}$ and r, also is $\overline{x}_1$.

□

The next result gives an idea of the meaning of the encoding.

Theorem 4.24. *Let ν be a node of the BDD with selector variable x_i, and A be a partial assignment. If we add a unit clause forcing the variable of the root node to be true, then*

- *ν is propagated by unit propagation if and only if there exists a partial assignment $B = \{x_1 = v_1, x_2 = v_2, \ldots, x_{i-1} = v_{i-1}\}$ such that B defines a path from the root to the node ν and all $x_j = 1$ of B also belongs to A.*
- *$\overline{\nu}$ is propagated by unit propagation if and only if there is an assignment $B = \{x_i = v_i, x_{i+1} = v_{i+1}, \ldots, x_n = v_n\}$ such that B defines a path from ν to the false terminal node and all $x_j = 1$ of B also belongs to A.*

Proof. We prove the first statement of the theorem by induction on the level of the node; the second one can be similarly proven. If ν is the root node, ν has been propagated, and we can pick a partial assignment $B = \emptyset$ which defines a path to ν.

Assume now that ν is not the root of the BDD.

- Assume that ν has been propagated.
  - If it has been propagated by a clause $\overline{\nu} \rightarrow \overline{\nu'}$, then ν is the false child of ν', and ν' has been propagated to true. Let $x_{i'}$ be the selector variable of ν'. By induction hypothesis, there exists a partial assignment $B' = \{x_1 = v_1, x_2 = v_2, \ldots, x_{i'-1} = v_{i'-1}\}$ such that B' defines a path to ν' and $x_j = 1 \in A$ for all $x_j = 1 \in B'$. The partial assignment $B = B' \cup \{x_{i'} = 0, \ldots, x_{i-1} = 0\}$ satisfies the statement.
  - If ν has been propagated by a clause $\overline{\nu} \wedge x_{i'} \rightarrow \overline{\nu'}$, then ν is the true child of ν' with selector variable $x_{i'}$, ν' has been propagated and $x_{i'} = 1$ belongs to A. As before, by induction hypothesis there exists a partial assignment $B' = \{x_1 = v_1, x_2 = v_2, \ldots, x_{i'-1} = v_{i'-1}\}$ such that B' defines a path to n' and $x_j = 1 \in A$ for all $x_j = 1 \in B'$. The partial assignment $B = B' \cup \{x_{i'} = 1, x_{i'+1} = 0, \ldots, x_{i-1} = 0\}$ satisfies the statement.
- Assume there exists a partial assignment $B = \{x_1 = v_1, x_2 = v_2, \ldots, x_{i-1} = v_{i-1}\}$ defining a path to ν and such that all $x_j = 1$ of B also belongs to A. Let ν' be the last node of the path defined by B before ν (i.e., ν' is the parent of ν in the path defined by B). Let $x_{i'}$ be the selector variable of ν'. Notice that $B' = \{x_j = v_j \in B \ : \ j < i'\}$ is an assignment holding the property for ν', so, by induction hypothesis, ν' has been propagated. If ν is the true child of ν', then $x_{i'} = 1$ belongs to B, so it also belongs to A, so the clause $\overline{\nu} \wedge x_{i'} \rightarrow \overline{\nu'}$ propagates ν. If ν is the false child of ν', the clause $\overline{\nu} \rightarrow \overline{\nu'}$ propagates ν.

□

Corollary 4.25. *Assume the BDD represents a Pseudo-Boolean constraint* $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leqslant K$. *Let* A *be a partial assignment and* ν *a node with selector variable* x_i *and interval* $[\alpha, \beta]$. *Then,* ν *is propagated if and only if*

$$\sum_{1 \leqslant j < i,\ x_j \in A} a_j \geqslant K - \beta,$$

and $\overline{\nu}$ *is propagated if and only if*

$$\sum_{i \leqslant j \leqslant n,\ x_j \in A} a_j > \beta.$$

We finish this section with an example illustrating how the suggested encoding of BDDs into SAT can be used in the different PB encoding methods we have presented in this section.

Example 4.26. *Consider the constraint* $C : 2x_1 + 3x_2 + 5x_3 \leqslant 6$. *We will encode this constraint into SAT with three methods: with the usual ROBDD encoding; with the consistent approach of ROBDDs and splitting of the coefficients, explained in Section 4.4.2; and with the arc-consistent approach of ROBDDs and splitting of the coefficients explained in Section 4.4.3.*

1. *BDD-1: this method builds the ROBDD for* C *and then encodes it into SAT. Hence we start by building the ROBDD of* C, *which can be seen in the last picture of Figure 4.1. Now, we need to encode it into SAT. Let* y_1, y_2 *and* y_3 *be fresh variables corresponding to the nodes of the ROBDD of* C *having respectively* x_1, x_2 *and* x_3 *as selector variable.*

   For node y_1, *we have to add the clauses* $\overline{y_2} \rightarrow \overline{y_1}$ *and* $x_1 \wedge \overline{y_3} \rightarrow \overline{y_1}$.

   For y_2, *we have to add the clauses* $\overline{\top} \rightarrow \overline{y_2}$ *and* $x_2 \wedge \overline{y_3} \rightarrow \overline{y_2}$, *where* $\top$ *is the tautology symbol.*

   For y_3, *we have to add the clauses* $\overline{\top} \rightarrow \overline{y_3}$ *and* $x_3 \wedge \overline{\bot} \rightarrow \overline{y_3}$, *where* $\bot$ *is the contradiction symbol.*

   Moreover, we have to add the unit clauses $\top$, $\overline{\bot}$ *and* y_1. *All in all, after removing the units and tautologies, the clauses obtained are* y_1, y_2, $\overline{x_1} \vee y_3$, $\overline{x_2} \vee y_3$ *and* $\overline{x_3} \vee \overline{y_3}$.

2. *BDD-2: we build the ROBDD of* C *with coefficient decomposition as in Example 4.21. Figure 4.6 shows the resulting ROBDD. We introduce variables* $y_1, y_2, \ldots, y_6$ *for every node of the ROBDD. More precisely, the first* x_2 *node (starting top-down) receives variable* y_1, *the next* x_2 *node gets* y_5. *The first* x_3 *receives* y_2 *and the other one* y_6. *Finally the leftmost* x_1 *node gets variable* y_3 *and the other one* y_4. *We have to add the following clauses:* $\overline{y_2} \rightarrow \overline{y_1}$, $\overline{y_4} \wedge x_2 \rightarrow \overline{y_1}$, $\overline{y_3} \rightarrow \overline{y_2}$, $\overline{y_4} \wedge x_3 \rightarrow \overline{y_2}$, $\overline{\top} \rightarrow \overline{y_3}$, $\overline{y_5} \wedge x_1 \rightarrow \overline{y_3}$, $\overline{y_5} \rightarrow \overline{y_4}$,

$\overline{y_6} \wedge x_1 \rightarrow \overline{y_4}$, $\overline{\top} \rightarrow \overline{y_5}$, $\overline{y_6} \wedge x_2 \rightarrow \overline{y_5}$, $\overline{\top} \rightarrow \overline{y_6}$, $\overline{\bot} \wedge x_3 \rightarrow \overline{y_6}$, *and the unit clauses* $\top$, $\overline{\bot}$ *and* y_1.

After removing the units from the clauses and tautologies, we obtain y_1, y_2, y_3, $y_4 \vee \overline{x_2}$, $y_4 \vee \overline{x_3}$, $y_5 \vee \overline{x_1}$, $y_5 \vee \overline{y_4}$, $y_6 \vee \overline{x_1} \vee \overline{y_4}$, $y_6 \vee \overline{x_2} \vee \overline{y_5}$ *and* $\overline{x_3} \vee \overline{y_6}$.

Notice that this encoding is consistent: if we have the assignment $A = \{x_2, x_3\}$, *then* y_4 *is propagated by the clause* $y_4 \vee \overline{x_2}$, *which in turn propagates* y_5 *due to clause* $y_5 \vee \overline{y_4}$ *and finally* y_6 *is propagated by the clause* $y_6 \vee \overline{x_2} \vee \overline{y_5}$. *A contradiction is found with clause* $\overline{x_3} \vee \overline{y_6}$.

However, the encoding is not arc-consistent: the partial assignment $A = \{x_1\}$ *can only propagate* y_5. *However,* $\overline{x_3}$ *should also be propagated.*

3. *BDD-3: let* C_1, C_2 *and* C_3 *be the constraints setting respectively* x_1, x_2 *and* x_3 *to true. Figure 4.7 shows the ROBDDs of these constraints. We have to encode these ROBDDs as usual, as in BDD-2, but replacing the unit clause* r *of the root by* $\overline{r} \rightarrow \overline{x_i}$. *In this case the variables associated with the roots of* C_1, C_2 *and* C_3 *will be* y_1, z_1 *and* w_1 *respectively.*

   After removing the units and tautologies, clauses from C_1 *are* $y_1 \vee \overline{x_1}$, $y_2 \vee \overline{y_1}$, $y_4 \vee \overline{x_2} \vee \overline{y_1}$, $y_3 \vee \overline{y_2}$, $y_4 \vee \overline{x_3} \vee \overline{y_2}$, $y_4 \vee \overline{x_2} \vee \overline{y_3}$ *and* $\overline{x_3} \vee \overline{y_4}$.

   Clauses from C_2 *are* $z_1 \vee \overline{x_2}$ *and* $\overline{x_3} \vee \overline{z_1}$.

   Finally, clauses from C_3 *are* $w_1 \vee \overline{x_3}$, $w_2 \vee \overline{w_1}$, $\overline{x_1} \vee \overline{w_1}$ *and* $\overline{x_2} \vee \overline{w_2}$.

   This encoding is arc-consistent. Take, for instance, the assignment $A = \{x_1\}$. *In this case,* $\overline{w_1}$ *is propagated in virtue of* $\overline{x_1} \vee \overline{w_1}$ *and* $\overline{x_3}$ *is propagated by clause* $w_1 \vee \overline{x_3}$.

4.7 Related Work

Due to the ubiquity of Pseudo-Boolean constraints and the success of SAT solvers, the problem of encoding those constraints into SAT has been thoroughly studied in the literature. In the following we review the most important contributions, paying special attention to the basic idea on which they are based, the encoding size, and the propagation properties the encodings fulfill. To ease the presentation, in the remaining of this section we will always assume that the constraint we want to encode is $a_1x_1 + \ldots + a_nx_n \leqslant k$, with maximum coefficient a_{max}.

The first encoding to mention is the one at [War98]. In a nutshell, the encoding uses several adders for numbers in binary representation. First of all, the left hand side of the constraint is split into two halves, each of which is recursively treated to compute the corresponding partial sum. After that, the two partial sums are added and the final result is compared with k . The encoding uses $O(n \log(a_{max}))$ clauses and variables and is neither consistent nor arc-consistent. This is not surprising, since adders for numbers in binary make extensive use of xors, which do not have good propagation properties.

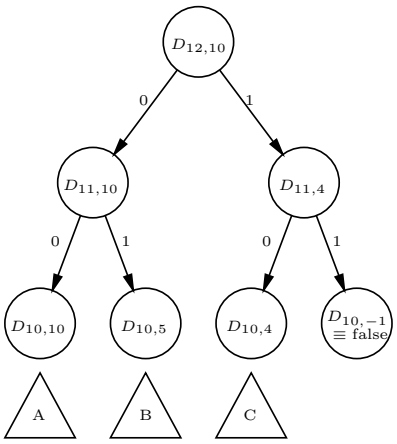

Figure 4.9: Tree-like construction of [BBR06a] for $2x_1 + \cdots + 2x_{10} + 5x_{11} + 6x_{12} \leqslant 10$

An encoding "very close to those using a BDD and translating it into clauses" was introduced at [BBR06a]. In order to understand the differences between their construction and BDDs let us introduce it in detail. First of all, the coefficients are ordered from small to large. Then, the root is labeled with variable $D_{n,k}$, expressing that the sum of the first n terms is no more than k. Its two children are $D_{n-1,k}$ and $D_{n-1,k-a_n}$, which correspond to setting x_n to false and true, respectively. The process is repeated until nodes corresponding to trivial constraints are reached, which are encoded as true or false. For each node $D_{i,b}$ with children $D_{i-1,b}$ and $D_{i-1,b-a_i}$, the following four clauses are added:

$$\begin{array}{ll} D_{i-1,b-a_i} \rightarrow D_{i,b} & \overline{D}_{i-1,b} \rightarrow \overline{D}_{i,b} \\ \overline{D}_{i-1,b-a_i} \wedge x_i \rightarrow \overline{D}_{i,b} & D_{i-1,b} \wedge \overline{x}_i \rightarrow D_{i,b} \end{array}$$

Example 4.27. *The encoding of [BBR06a] on* $2x_1 + \cdots + 2x_{10} + 5x_{11} + 6x_{12} \leqslant 10$ *is illustrated in Figure 4.9. Node* $D_{10,5}$ *represents* $2x_1 + 2x_2 + \cdots + 2x_{10} \leqslant 5$*, whereas node* $D_{10,4}$ *represents* $2x_1 + 2x_2 + \cdots 2x_{10} \leqslant 4$*. The method fails to identify that these two PB constraints are equivalent and hence subtrees B and C will not be merged, yielding a much larger representation than with ROBDDs.*

The resulting encoding is arc-consistent, but an example of a PB constraint family is given for which their kind of *non-reduced* BDDs, with *their concrete variable ordering* is exponentially large. However, as we have shown in Section 4.3.2, ROBDDs for this family are polynomial.

Several important new contributions were presented in the paper by the MiniSAT team [ES06]. The paper describes three encodings, all of which are implemented in the MiniSAT+ tool. The first one is a standard ROBDD construction for Pseudo-Boolean constraints. This is done in two steps: first, they suggest a simple dynamic

programming algorithm for constructing a non-reduced BDD, which is later reduced. The result is a ROBDD, but the first step may take exponential time even if the final ROBDD is polynomial. Once the ROBDD is constructed, they suggest to encode it into SAT using 6 ternary clauses per node. The paper showed that, given a concrete variable ordering, the encoding is arc-consistent. Regarding the ROBDD size, the authors cite [BBR06a] to state the BDDs are exponential in the worst case. As we have seen before, the citation is not correct because the method of [BBR06a] do not construct ROBDDs.

The second method is similar to the one of [War98] in the sense that the construction relies on a network of adders. First of all coefficients are decomposed into binary representation. For each bit i, a bucket is created with all variables whose coefficient has bit i set to one. The i-th bit of the left-hand side of the constraint is computed using a series of full adders and half adders. Finally, the resulting sum is lexicographically compared to k. The resulting encoding is neither consistent nor arc-consistent and uses a number of adders linear in the sum of the number of digits of the coefficients.

The last method they suggest is the use of sorting networks. Numbers are expressed in unary representation and coefficients are decomposed using a mixed radix representation. The smaller the number in this representation, the smaller the encoding. In this setting, sorting networks are used to play the same role of adders, but with better propagation properties. If N is smaller than the sum of the digits of all coefficients in base 2, the size of the encoding is $O(N \log^2 N)$. Whereas this encoding is not arc-consistent for arbitrary Pseudo-Boolean constraints, arc-consistency is obtained for cardinality constraints.

The first polynomial and arc-consistent encoding for Pseudo-Boolean constraints, called Watch-Dog, was introduced in [BBR09]. Again, numbers are expressed in unary representation and totalizers are used to play the role of sorting networks. In order to make the comparison with the right hand side trivial, the left-hand side and k are incremented until k becomes a power of two. Then, all coefficients are decomposed in binary representation and each bit is added independently, taking into account the corresponding carry. The encoding uses $O(n^2 \log n \log a_{max})$ variables and $O(n^3 \log n \log a_{max})$ clauses. In the same paper, another encoding which is only consistent and uses $O(n \log n \log a_{max})$ variables and $O(n^2 \log n \log a_{max})$ clauses is also presented.

Finally, it is worth mentioning the work of [BB03b]. The authors deal with the more general case in which the variables x_i are not Boolean, but bounded integers $0 \leqslant x_i < 2^b$. They suggest a BDD-based approach very similar in flavor to our method of Section 4.4, but instead of decomposing the coefficients as we do, they decompose the variables x_i in binary representation. The BDD ordering starts with the first bit of x_1, then the first bit of the x_2, etc... After that, the second bit is treated in a similar fashion, and so on. The resulting BDD has $O(n \cdot b \cdot \sum a_i)$ nodes and nothing is mentioned about propagation properties. For the case of Pseudo-Boolean constraints, i.e. $b = 1$, their approach amounts to standard BDDs.

Encoding	Reference	Clauses	Consist.	Arc-Consist.
Warners	[War98]	$\mathcal{O}(n \log a_{max})$	NO	NO
Non-reduced BDD	[BBR06a]	Exponential	YES	YES
ROBDD	[ES06]	Exponential (6 per node)	YES	YES
Adders	[ES06]	$\mathcal{O}(\sum \log a_i)$	NO	NO
Sorting Networks	[ES06]	$\mathcal{O}((\sum \log a_i) \log^2(\sum \log a_i)$	YES	NO
Watch Dog (WD)	[BBR09]	$\mathcal{O}(n^2 \log n \log a_{max})$	YES	NO
Arc-consist. WD	[BBR09]	$\mathcal{O}(n^3 \log n \log a_{max})$	YES	YES

Table 4.1: Summary comparing the different encodings.

Table 4.1 summarizes the different encodings of PB constraints into SAT.

4.8 Experimental Results

The goal of this section is to assess the practical interest of the encodings we have presented in the paper. Our aim is to evaluate to which extent BDD-based encodings are interesting from the practical point of view. For us, this means to study whether they are competitive with existing techniques, whether they show good behavior in general or are only interesting for very specific types of problems, or whether they produce smaller encodings.

For that purpose, first of all, we compare our encodings with other SAT encodings in terms of encoding time, number of clauses and number of variables. After that, we also consider total runtime (that is, encoding time plus solving time) of these encodings and we compare it with the runtime of state-of-the-art Pseudo-Boolean solvers. Finally, we briefly report on some experiments with sharing, that is, trying to encode several Pseudo-Boolean constraints in a single ROBDD.

All experiments were performed on a 2Ghz Linux Quad-Core AMD with a time limit of 1800 seconds per benchmark. The benchmarks used for these experiments were obtained from the Pseudo-Boolean Competition 2011 (see `http://www.cril.univ-artois.fr/PB11/`), category *no optimization, small integers, linear constraints (DEC-SMALLINT-LIN)*. For compactness and clarity, benchmarks that come from the same source into families are grouped. However, individual results can be found at
`http://www.lsi.upc.edu/~iabio/BDDs/results.ods`.

4.8.1 The Bergmann Test

In order to summarize the experiments and make it easier to extract conclusions, every experiment is accompanied with a *Bergmann-Hommel non-parametric hypothesis test* [BH88] of the results with a confidence level of 0.1.

The Bergmann-Hommel test is a way of comparing the results of n different

methods over multiple independent data sets. It gives us two interesting pieces of information. First of all, it sorts the methods by giving them a real number between 1 and n, such that the lower the number the better the method. Moreover, it indicates, for each pair of methods, whether one method significatively improves upon the other. As an example, Figure 4.10 is the output of a Bergmann-Hommel test. BDD-1 is the best method but there is not significant difference between this method and BDD-2 (this is illustrated by a thick line connecting the methods). On the other hand, the Bergmann-Hommel test indicates that BDD-1 is significantly better than Adder, since there is no thick line connecting BDD-1 and Adder. The same can be said for BDD-1 and WD-1, BDD-1 and BDD-3, BDD-1 and WD-2, BDD-2 and Adder, etc.

We will now give a quick overview of how a Bergmann-Hommel test is computed. The remaining of this section can be skipped if the reader is not interested in the details of the test. On the other hand, for more detailed information, we refer the reader to [BH88].

Let us assume we have n methods and m data sets, and let $C_{i,j}$ be the result (time, number of variables or any other value) of the i-th method in the j-th benchmark. For every data set, we assign a number to every method: the best method in that data set has a 1, the second has a 2, and so on. Then, for every method, we compute the average of these values in the different data sets. The obtained value is denoted by R_i and is called the *average rank* of the i-th method. A method with smaller average rank is better than a method with a bigger one.

These average ranks make it possible to rank the different methods. However, we are also interested in detecting whether the differences between the methods are significant or not: this is computed in the second part of the test. Before that, we need some previous definitions.

Given $i, j \in N = \{1, 2, \ldots, n\}$, we denote by $p_{i,j}$ the p-value[3] of $z_{i,j}$ with respect a normal distribution $N(0, 1)$, where $z_{i,j} = \frac{R_i - R_j}{\sqrt{n(n-1)/(6m)}}$. A *partition* of $N = \{1, 2, \ldots, n\}$ is a collection of sets $P = \{P_1, P_2, \ldots, P_r\}$ such that (i) the P_i's are subsets of N, (ii) $P_1 \cup P_2 \cup \cdots \cup P_r = N$ and (iii) $P_i \cap P_j = \emptyset$ for every $i \neq j$. Given P a partition of N, we define

$$L(P) = \sum_{i=1}^{r} \frac{|P_i|(|P_i| - 1)}{2}$$

and $p(P)$ as the minimum $p_{i,j}$ such that i and j belong to the same subset $P_k \in P$.

The Bergmann-Hommel test ensures (with a significance level of α) that the methods i and j are not significantly different if and only if there is a partition P with $p(P) > \alpha L(P)$ such that i and j belong to the same subset $P_k \in P$. Hence, it is a time-consuming test since the number of partitions can be very large.

In our case, the data sets are the families of benchmarks. We have to use the families instead of the benchmarks because the data sets must be independent.

[3]The p-value of z with respect to a normal distribution $N(0, 1)$ is the probability $p[\ |Z| > |z|\ ]$, where the random variable $Z \sim N(0, 1)$.

4.8.2 Encodings into SAT

We start by comparing different methods for encoding Pseudo-Boolean constraints into SAT. We have focused on the time spent by the encoding, the number of auxiliary variables used and the number of clauses. Moreover, for each benchmark family, we also report the number of PB-constraints that were encoded into SAT.

The encodings we have included in the experimental evaluation are: the adder encoding as presented at [ES06] (Adder), the consistent WatchDog encoding of [BBR09] (WD-1), its arc-consistent version (WD-2), the encoding into ROBDDs without coefficient decomposition, using Algorithm 4.1 and the encoding from Section 4.6 (BDD-1); the encoding into ROBDDs after coefficient decomposition as explained in Section 4.4.2 (BDD-2), with Algorithm 4.1 and the encoding from Section 4.6; and the arc-consistent approach from Section 4.4.3 (*BDD-3*), also with Algorithm 4.1 and the encoding from Section 4.6. Notice that BDD-1 method is very similar to the ROBDDs presented at [ES06]. However, since Algorithm 4.1 produces every node only once, BDD-1 should be faster. Also, the encoding of Section 4.6 only creates two clauses per BDD node, as opposed to six clauses as suggested at [ES06].

Table 4.2 shows the number of problems that the different methods could encode without timing out. The first column corresponds to the family of problems. The second column shows the number of problems in this family. The third and fourth columns contain the average number of SAT and Pseudo-Boolean constraints in the problem. For the experiments, we considered a constraint to be SAT if it is a clause or has at most 3 variables. Small PB constraints do not benefit from the above encodings and hence for these constraints a naive encoding into SAT was always used. The remaining columns correspond to the number of encoded problems without timing out. Time limit was set to 1800 seconds per benchmark. Every method but WD-2 and BDD-3 were able to encode all constraints without timing out.

Table 4.3 shows the time spent to encode the benchmarks by the different methods. As before, the first columns correspond to the family of problems, the number of problems in this family and the average number of SAT and Pseudo-Boolean constraints in the problems. The remaining columns correspond to the average encoding time (in seconds) per benchmarks of each method. Timeouts are counted as 1800 seconds in the average computation.

Table 4.4 shows the average number of auxiliary variables required for encoding the PB constraints (SAT constraints are not counted). The meaning of the first 4 columns is the same as before, and the others contain the average number of auxiliary variables (in thousands) of the benchmarks that did not time out.

Finally, Table 4.5 contains the average number (in thousands) of clauses needed to encode the problem. As before, we have only considered the benchmarks that have not timed out, and clauses due to the encoding of SAT constraints are not counted.

Figures 4.10, 4.11 and 4.12 represent the Bergmann-Hommel tests of these tables.

Family	**Pr**	**SAT**	**PB**	**Adder**	**WD-1**	**WD-2**	**BDD-1**	**BDD-2**	**BDD-3**
lopes	200	502,671	592,715	188	188	118	**197**	**197**	188
army	12	192	451	**12**	**12**	**12**	**12**	**12**	**12**
blast	8	6,510	1,253	**8**	**8**	**8**	**8**	**8**	**8**
cache	9	181,100	4,507	**9**	**9**	**9**	**9**	**9**	**9**
chnl	21	0	125	**21**	**21**	**21**	**21**	**21**	**21**
dbstv30	5	326,200	2,701	**5**	**5**	0	**5**	**5**	0
dbstv40	5	985,200	4,801	**5**	**5**	0	**5**	**5**	0
dbstv50	5	2,552,000	7501	**5**	**5**	0	**5**	**5**	0
dlx	3	20,907	857	**3**	**3**	**3**	**3**	**3**	**3**
elf	5	46,446	1,399	**5**	**5**	**5**	**5**	**5**	**5**
fpga	36	0	687	**36**	**36**	**36**	**36**	**36**	**36**
j30	17	13,685	270	**17**	**17**	**17**	**17**	**17**	**17**
j60	18	30,832	309	**18**	**18**	**18**	**18**	**18**	**18**
j90	17	50,553	337	**17**	**17**	8	**17**	**17**	11
j120	28	104,147	516	**28**	**28**	11	**28**	**28**	18
neos	4	1,451	3,831	**4**	**4**	**4**	**4**	**4**	**4**
ooo	19	95,217	4,487	**19**	**19**	**19**	**19**	**19**	**19**
pig-card	20	0	113	**20**	**20**	18	**20**	**20**	**20**
pig-cl	20	161,150	58	**20**	**20**	**20**	**20**	**20**	**20**
ppp	6	29,846	1,023	**6**	**6**	**6**	**6**	**6**	**6**
robin	6	0	761	**6**	**6**	2	**6**	**6**	**6**
13queen	100	8	93	**100**	**100**	**100**	**100**	**100**	**100**
11tsp11	100	2,662	45	**100**	**100**	**100**	**100**	**100**	**100**
vdw	5	8,978	267,840	**5**	**5**	**5**	**5**	**5**	**5**
TOTAL	669			657	657	540	**666**	**666**	626

Table 4.2: Number of problems encoded (without timing out) by the different methods.

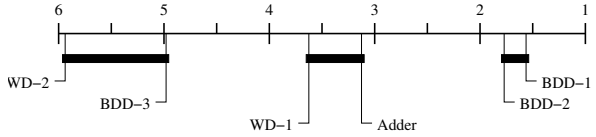

Figure 4.10: Statistical comparison of the results of Table 4.3, time spent by the different methods in encoding.

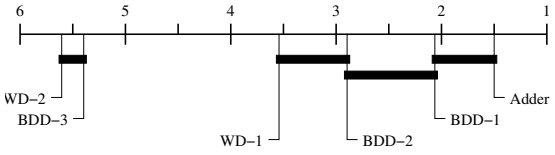

Figure 4.11: Statistical comparison of the results of Table 4.4, number of auxiliary variables used by the different encodings.

Family	Pr	SAT	PB	Adder	WD-1	WD-2	BDD-1	BDD-2	BDD-3
lopes	200	502,671	592,715	335.23	292.14	996.07	165.75	**163.66**	316.35
army	12	192	451	0.37	0.43	39.98	**0.19**	**0.19**	10.26
blast	8	6,510	1,253	3.89	2.45	40.41	2.20	**1.89**	23.15
cache	9	181,100	4,507	23.08	18.74	81.65	16.19	**15.74**	47.78
chnl	21	0	125	0.54	1.05	87.08	**0.13**	**0.13**	2.68
dbstv30	5	326,200	2,701	57.77	97.21	—	**45.85**	83.09	—
dbstv40	5	985,200	4,801	211.51	210.25	—	**105.62**	165.96	—
dbstv50	5	2,552,000	7,501	547.30	552.99	—	**272.02**	468.51	—
dlx	3	20,907	857	3.73	3.05	8.41	2.76	**2.75**	6.19
elf	5	46,446	1,399	7.37	6.53	21.68	**5.19**	5.90	13.42
fpga	36	0	687	1.90	2.46	69.90	**0.30**	**0.30**	3.75
j30	17	13,685	270	3.64	4.62	81.03	**3.13**	3.67	42.44
j60	18	30,832	309	**6.85**	10.69	466.07	8.19	8.77	252.69
j90	17	50,553	337	**14.81**	31.02	1,277.28	28.20	27.76	1,155.18
j120	28	104,147	516	**19.25**	47.62	1,305.55	21.68	25.50	967.10
neos	4	1,451	3,832	10.43	12.65	257.97	**3.46**	5.32	77.04
ooo	19	95,217	4,487	13.48	9.67	71.20	**7.76**	7.88	26.35
pig-card	20	0	113	0.97	3.29	517.51	0.22	**0.21**	9.52
pig-cl	20	161,150	58	7.73	8.78	284.15	7.35	**7.31**	10.79
ppp	6	29,846	1,024	6.13	5.09	33.26	**3.17**	3.23	9.83
robin	6	0	761	12.03	67.41	1,315.96	2.94	**2.82**	301.11
13queen	100	8	93	0.19	0.45	100.29	**0.14**	0.17	18.48
11tsp11	100	2,662	45	0.46	0.51	24.42	**0.30**	0.33	6.30
vdw	5	8,978	267,840	170.33	109.42	441.21	47.15	**46.32**	125.91
Average				110.57	99.79	510.40	**55.90**	57.66	223.41

Table 4.3: Average time spent on the encoding by the different methods.

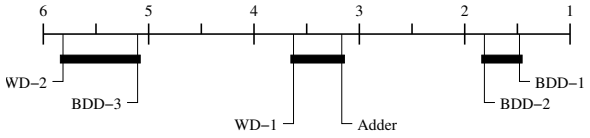

Figure 4.12: Statistical comparison of the results of Table 4.5, number of clauses used by the different methods.

Family	**Pr**	**SAT**	**PB**	**Adder**	**WD-1**	**WD-2**	**BDD-1**	**BDD-2**	**BDD-3**
lopes	200	502,671	592,715	**1,744**	3,566	5,479	2,394	2,394	7,735
army	12	192	451	**4.63**	10.96	246	6.36	6.36	480
blast	8	6,510	1,253	**27.77**	62.22	1,394	36.74	39.67	761
cache	9	181,100	4,507	**146**	339	2,394	201	211	1,503
chnl	21	0	125	8.39	24.55	1,008	**6.76**	**6.76**	185
dbstv30	5	326,200	2,701	**220**	710	—	442	1,696	—
dbstv40	5	985,200	4,801	**2,468**	6,564	—	4,282	7,226	—
dbstv50	5	2,552,000	7,501	**6,135**	16,365	—	11,111	19,723	—
dlx	3	20,907	857	**10.4**	21.62	248	12.40	13.89	127
elf	5	46,446	1,399	**20.37**	42.78	571	24.62	28.13	307
fpga	36	0	687	21.15	53.96	1,074	**13.27**	**13.27**	242
j30	17	13,685	270	**18.15**	50.8	1,191	44.96	59.82	1,154
j60	18	30,832	309	**37.03**	112	4,776	158	180	7,286
j90	17	50,553	337	**65.4**	217	6,544	554	554	19,793
j120	28	104,147	516	**159**	540	5,714	612	806	22,247
neos	4	1,451	3,832	**73.74**	186	3,543	79.33	123	2,004
ooo	19	95,217	4,487	**118**	274	2,248	162	169	1,316
pig-card	20	0	113	15.26	50.75	2,967	**11.93**	**11.93**	632
pig-cl	20	161,150	58	7.68	25.25	1,984	**4.01**	**4.01**	310
ppp	6	29,846	1,024	**57.13**	141	624	81.57	82.86	383
robin	6	0	761	171	628	3,634	**159**	**159**	16,566
13queen	100	8	93	**2.2**	6.17	462	5.63	7.08	791
11tsp11	100	2,662	45	**3.37**	8.83	171	5.71	6.51	221
vdw	5	8,978	267,840	1,895	3,357	12,819	**1,392**	**1,392**	5,876
Average				**591**	1,266	1,876	893	998	3,807

Table 4.4: Average number of auxiliary variables (in thousands) used.

Family	**Pr**	**SAT**	**PB**	**Adder**	**WD-1**	**WD-2**	**BDD-1**	**BDD-2**	**BDD-3**
lopes	200	502,671	592,715	10,644	7,472	22,082	3,049	**3,049**	9,746
army	12	192	451	26.09	34.5	2,156	**10.87**	**10.87**	925
blast	8	6,510	1,253	185	102	2,108	70.9	**65.46**	1,265
cache	9	181,100	4,507	981	551	3,652	**272**	275	2,419
chnl	21	0	125	56.82	117	4,936	**11.23**	**11.23**	286
dbstv30	5	326,200	2,701	1,497	3,368	—	**857**	3,282	—
dbstv40	5	985,200	4,801	17,185	16,917	—	**5,527**	11,259	—
dbstv50	5	2,552,000	7,501	42,797	44,311	—	**14,400**	31,279	—
dlx	3	20,907	857	65.25	35.68	378	23.04	**22.59**	209
elf	5	46,446	1,399	129	71.28	881	46.3	**46.07**	507
fpga	36	0	687	139	176	3,615	**15.65**	**15.65**	278
j30	17	13,685	270	122	165	3,890	**89.53**	116	2,244
j60	18	30,832	309	**253**	495	22,843	311	351	14,355
j90	17	50,553	337	**450**	1,286	34,137	1,106	1,095	39,112
j120	28	104,147	516	**1,103**	3,803	26,205	1,187	1,571	44,069
neos	4	1,451	3,832	471	595	12,410	**139**	220	3,681
ooo	19	95,217	4,487	793	442	3,378	**219**	228	2,126
pig-card	20	0	113	104	367	20,711	**19.86**	**19.86**	959
pig-cl	20	161,150	58	52.41	180	14,641	**4.07**	**4.07**	314
ppp	6	29,846	1,024	393	272	1,804	**101**	103	654
robin	6	0	761	1,186	6,916	19,695	**281**	**281**	28,876
13queen	100	8	93	14.73	38.91	5,068	**10.84**	13.73	1,574
11tsp11	100	2,662	45	23.31	25.35	1,336	**7.76**	9.35	434
vdw	5	8,978	267,840	10,886	6,564	24,274	**1,663**	**1,663**	7,263
Average				3,676	2,971	8,297	**1,174**	1,380	5,844

Table 4.5: Average number of clauses (in thousands) used.

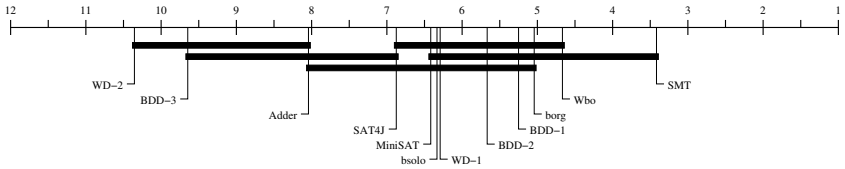

Figure 4.13: Statistical comparison of the results of Table 4.7, runtime of the different methods.

They show that BDD-1, BDD-2 and Adders are the best methods in terms of time, variables and clauses. It is worth mentioning that BDD-1 and BDD-2 are faster and use significantly less clauses than Adder. However, Adders uses significantly less auxiliary variables than BDD-2. Notice that BDD-1 is arc-consistent, BDD-2 is only consistent and Adder is not consistent, so at least theoretically BDD-1 clauses have more unit propagation power than BDD-2 clauses, and BDD-2 clauses are better than Adder clauses. Hence, BDD-1 is the best method using these criteria and BDD-2 is better than Adder. Regarding the other methods, it seems clear that encoding n different constraints in order to obtain arc-consistency, as it is done in WD-2 and BDD-3, is not a good idea in terms of variables and clauses.

4.8.3 SAT vs. PB

In this section we compare the state-of-the-art solvers for Pseudo-Boolean problems and some encodings into SAT. For the SAT approach, once the encoding has been done, the SAT formula is given to the SAT Solver Lingeling [Bie10b] version 276. We have considered the same SAT encodings as in the previous section. Regarding Pseudo-Boolean solvers, we have considered MiniSAT+ [ES06] and the best non-parallel solvers in the *No optimization, small integers, linear constraints* category of the Pseudo-Boolean Competition 2010: borg [SM10] version pb-dec-11.04.03, bsolo [MS06] version 3.2, wbo [MML10] version 1.4 and SAT4J [BP10] version 2.2.1. We have also included the SMT Solver Barcelogic [BNO$^+$08] for PB constraints, which couples a SAT solver with a theory solver for PB constraints.

Table 4.6 shows the number of instances solved by each method. Table 4.7 shows the average time spent by all these methods. For the SAT encodings, times include both the encoding and SAT solving time. As before, a time limit of 1800 seconds per benchmark was set, and for the average computation, a timeout is counted as 1800 seconds. Both tables include a column VBS (Virtual Best Solver), which represents the best solver in every instance. This gives an idea of which speedup we could obtain with a portfolio approach.

Figure 4.13 shows the result of the Bergmann test: SMT is the best method, whereas Adder, BDD-3 and WD-2 are the worst ones. There are no significant difference between the other methods. The main conclusion we can infer is that BDD encodings are definitely a competitive method. Also, there is no technique that out-

Family	**Adder**	**WD-1**	**WD-2**	**BDD-1**	**BDD-2**	**BDD-3**	**bsolo**	**MiniSAT**	**SAT4J**	**Wbo**	**borg**	**SMT**	**VBS**
lopes	42	54	40	56	57	61	39	**66**	23	63	37	43	77
army	9	**12**	7	10	11	5	6	6	6	6	10	5	12
blast	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	8
cache	**9**	**9**	**9**	**9**	**9**	3	7	8	6	6	6	**9**	9
chnl	3	3	2	5	5	3	**21**	3	1	3	**21**	0	21
dbstv30	**5**	**5**	0	**5**	**5**	0	**5**	**5**	**5**	**5**	**5**	**5**	5
dbstv40	0	**5**	0	**5**	**5**	0	**5**	**5**	**5**	**5**	**5**	**5**	5
dbstv50	0	**5**	0	**5**	**5**	0	**5**	**5**	**5**	**5**	**5**	**5**	5
dlx	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	3
elf	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	5
fpga	25	**36**	**36**	**36**	**36**	**36**	**36**	33	**36**	**36**	**36**	**36**	36
j30	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	17
j60	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	17
j90	**17**	**17**	7	**17**	**17**	8	**17**	**17**	**17**	**17**	**17**	**17**	17
j120	14	**16**	9	**16**	**16**	11	13	12	**16**	**16**	**16**	**16**	17
neos	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	2
ooo	15	**19**	16	18	**19**	17	14	15	14	15	14	17	19
pig-card	2	2	2	2	2	1	19	2	2	2	**20**	0	20
pig-cl	2	1	2	1	1	2	3	2	2	2	**5**	0	5
ppp	4	3	4	3	4	4	4	4	4	3	**5**	4	6
robin	3	3	2	3	3	**6**	3	3	4	3	3	4	6
13queen	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	100
11tsp11	**100**	**100**	96	**100**	**100**	75	72	90	93	**100**	**100**	**100**	100
vdw	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	2
TOTAL	403	443	385	444	448	391	422	429	392	440	**458**	419	514

Table 4.6: Number of problems solved by different methods.

Family	Adder	WD-1	WD-2	BDD-1	BDD-2	BDD-3	bsolo	MiniSAT	SAT4J	Wbo	borg	SMT	VBS
lopes	1,5145	1,420	1,561	1,408	1,401	1,435	1,509	**1,344**	1,661	1,364	1,555	1,464	1,249
army	660	**139**	1,141	543	469	1,298	1,028	913	1,127	1,084	438	1,066	86.29
blast	6.12	2.56	46.78	2.42	1.99	27.63	0.12	0.51	0.84	0.08	2.13	**0.03**	0.03
cache	253	123	396	**75.49**	115	375	653	395	670	606	636	266	63.95
chnl	1,543	1,543	1,716	1,508	1,508	1,681	**0.55**	1,551	1,751	1,673	3.78	—	0.47
dbstv30	1,049	128	—	91.66	192	—	59.28	32.6	99.81	1.54	9.87	**1.28**	1.28
dbstv40	—	366	—	198	324	—	187	72.25	9.74	5.69	45.33	**4.44**	4.44
dbstv50	—	935	—	629	792	—	200	430	21.22	16.13	121	**11.36**	11.36
dlx	7.06	4.88	25.72	4.29	4.34	19.58	3.47	1.29	1.6	0.55	3.15	**0.17**	0.17
elf	13.87	10.14	44.09	7.97	9	30.03	28.58	2.97	2.31	1.42	11.61	**0.69**	0.69
fpga	586	5.27	113	0.92	0.92	37.64	0.27	242	1.47	5.17	3.04	**0.1**	0.07
j30	16.7	7.79	116	5.94	8.42	77.88	6.53	4.6	14.57	0.53	1.93	**0.28**	0.28
j60	137	114	551	113	116	398	110	115	105	101	104	**101**	101
j90	24.18	36.63	1,303	39.72	39.46	1,233	0.9	3.96	1.42	0.41	3.32	**0.15**	0.15
j120	978	854	1,364	839	851	1,262	967	1,031	849	839	841	**814**	756
neos	1,023	936	1,405	910	915	1,073	1,106	1,276	1,038	**901**	976	925	901
ooo	479	190	493	**151**	176	488	645	453	575	486	512	259	126
pig-card	1,620	1,620	1,680	1,620	1,620	1,725	114	1,626	1,749	1,685	**3.92**	—	1.92
pig-cl	1,624	1,715	1,693	1,718	1,718	1,721	1,658	1,623	1,705	1,742	**1,369**	—	1,367
ppp	631	1,001	656	906	858	646	605	919	602	901	**390**	601	210
robin	938	921	1,353	913	913	719	936	971	778	920	963	**605**	444
13queen	47.52	**1.64**	264	4.63	4.51	643	54.82	238	18.92	5.9	20.35	1.92	1.28
11tsp11	28.36	8.29	429	23.86	18.32	731	855	369	503	229	27.64	**1.81**	1.51
vdw	1,645	1,568	1,545	1,493	1,493	1,612	1,478	1,448	1,596	1,441	1,450	**1,441**	1,186
Average	783	669	958	667	667	1,003	764	772	849	710	**613**	696	475

Table 4.7: Time spent by different methods on solving the problem (in seconds).

performs the others in all benchmark families, and hence portfolio strategies would make a lot of sense in this area, as witnessed by the performance of Borg, which implements such an approach. Finally, we also want to mention that the possible exponential explosion of BDDs rarely occurs in practice and hence, coefficient decomposition does not seem to pay off in practical situations.

Regarding the Best Virtual Solver, the 52% of the problems solved are due to SMT. In the 25% of the cases the best solution was given by a specific PB solver. Among them, Wbo contribute with the 10% of the problems and bsolo with the 8%. Finally, the encoding methods give the best solution in the 23% of the cases: 14% of the times due to Watchdog methods and 8% of the times due to BDD-based methods.

4.8.4 Sharing

One of the possible advantages of using ROBDDs to encode Pseudo-Boolean constraints is that ROBDDs allow one to encode a set of constraints, and not just one. It would seem natural to think that if two constraints are *similar enough*, the two individual ROBDDs would be similar in structure, and merging them into a single one would result in a ROBDD whose size is smaller than the sum of the two individual ROBDDs. However, the main difficulty is to decide which constraints should be encoded together, since a bad choice could result in a ROBDD whose size is larger than the sum of the ROBDDs for the individual constraints.

We performed initial experiments where the criteria of similarity between constraints only took into account which variables appeared in the constraints. We first fixed an integer k and chose the constraint with the largest set of variables. After that, we looked for a constraint such that all but k variables appeared in the first constraint. The next step was to look for another constraint such that all but k variables appeared in any of the two previous constraints and so on, until reaching a fixpoint. Finally, all selected constraints were encoded together.

We tried this experiment on all benchmarks with different values of k and it rarely gave any advantage. However, we still believe that there could be a way of encoding different constraints into a single ROBDD, but different criteria for selecting the constraints should be studied. We see this as a possible line of future research.

4.9 Conclusions and Future Work

Both theoretical and practical contributions have been made. Regarding the theoretical part, we have negatively answered the question of whether all PB constraints admit polynomial BDDs by citing the work of [HTY94] which, to the best of our knowledge, is largely unknown in our research community. Moreover, we have given a simpler proof assuming that NP is different from co-NP, which relates the subset sum problem and the ROBDDs' size of PB constraints.

At the practical level, we have introduced a ROBDD-based polynomial and arc-consistent encoding of PB constraints and developed a BDD-based arc-consistent encoding of monotonic functions that only uses two clauses per BDD node. We have also presented an algorithm to efficiently construct all these ROBDDs and proved that the overall method is competitive in practice with state-of-the-art encodings and tools. As future work at the practical level, we plan to study which type of Pseudo-Boolean constraints are likely to produce smaller ROBDDs if encoded together rather than being encoded individually.

5 Conflict-Directed Lazy Decomposition

5.1 Introduction

Compared with other systematic constraint solving tools, SAT solvers have many advantages for non-expert users. They are extremely efficient off-the-shelf black boxes that require no tuning regarding variable (or value) selection heuristics. However, propositional logic cannot directly deal with complex constraints: we need either *to enrich the language* in which the problems are defined, or *to reduce the complex constraints* to propositional logic.

Lazy clause generation (LCG) or *SAT Modulo Theories* (SMT) approaches correspond to an enrichment of the language: the problem can be expressed in first-order logic instead of propositional logic. A specific theory solver for that (kind of) constraint, called a *propagator*, takes care of the non-propositional part of the problem, propagating and explaining the propagations, whereas the SAT Solver deals with the propositional part. On the other hand, reducing the constraints to propositional logic corresponds to *encoding* or *decomposing* the constraints into SAT: the complex constraints are replaced by an equivalent set of auxiliary variables and clauses.

The advantages of the propagator approach is that the size of the propagator and its data structures are typically quite small (in the size of the constraint) compared to the size of an encoding, and we can make use of specific global algorithms for efficient propagation. The advantages of the encoding approach are that the resulting decomposition uses efficient SAT data structures and are inherently incremental, and more importantly, the auxiliary variables give the solver more scope for learning appropriate reusable nogoods.

In this chapter[1] we examine how to get the best of each approach, and illustrate

[1]Based on the paper "Conflict Directed Lazy Decomposition" from the *18th International Conference on Principles and Practice of Constraint Programming* [AS12]

our method on the two fundamental constraints considered in the previous sections: cardinality and Pseudo-Boolean constraints.

An SMT solver generates lazily an encoding of the constraint. However, this encoding has no auxiliary variables: this can be a bad option since encodings without variables needs usually a huge number of clauses and, above all, the SAT Solvers cannot *decide* on these auxiliary variables, which usually implies that the search space is explored in a worse way. Lazy Decomposition allows to lazily generate the decomposition but introducing the auxiliary variables: in this way, a constraint is decomposed only when it is useful but the decomposition is done in a better way than in the SMT case: it also introduces auxiliary variables.

As said, in this chapter we deal with cardinality and Pseudo-Boolean constraints. Regarding cardinality constraints, the two previously mentioned approaches for solving complex constraints have been studied. In the literature one can find different encodings using adders [War98], binary trees [BB03a] or sorting networks [ES06], among others. The best encoding, to our knowledge, was the cardinality network-based encoding explained in Section 3.3 [ANORC11b] and improved in Section 3.5. On the other hand, we can use a propagator to deal with these constraints, either an SMT Solver [NOT06] or a LCG Solver [OSC09].

Regarding Pseudo-Boolean constraints, in the literature one can find different encodings for them using adders [War98, ES06], BDDs or similar tree-like structures [ES06, BBR06a, ANORC11a] (see Chapter 4) or sorting networks [ES06]. As before, LCG and SMT approaches are also possible.

To see why both approaches, both SMT and encoding, have advantages consider the following two scenarios:

- Consider a problem with hundreds of large cardinality constraints where all but 1 never cause failure during search. Decomposing each of these constraints will cause a huge burden on the SAT solver, adding many new variables and clauses, all of which are actually useless. The propagation approach will propagate much faster, and indeed just the encoding step could overload the SAT solver.

- Consider the problem with the cardinality constraint $x_1 + \cdots + x_n \leq K$ and some propositional clauses implying $x_1 + \cdots + x_n \geq K + 1$. The problem is obviously unsatisfiable, but if we use a propagator for the cardinality constraint, it will need to generate all the $\binom{n}{k}$ explanations possible in order to prove the unsatisfiability. However with a decomposition approach the problem can be solved in polynomial time due to the auxiliary variables.

In conclusion it seems likely that in every problem there are *some* auxiliary variables that will produce more general reasons and will help the SAT solver, and *some* other variables that will only increase the search space size, making the problem more difficult. The intuitive idea of Lazy Decomposition is to try to generate only the *useful* auxiliary variables. The solver initially behaves as a basic SMT solver. If it observes that an auxiliary variable would appear in many reasons, the solver generates it.

While there is plenty of research on combining SAT and propagation-based methods, for example all of SAT modulo theories and lazy clause generation, we are unaware of any previous work where a complex constraint is partially decomposed. There is some recent work [MP11] where the authors implement an incremental method for solving Pseudo-Boolean constraints with SAT, by encoding the Pseudo-Booleans one by one. However, they do not use propagators for dealing with the non-decomposed constraints, since in every step they consider the problem involving only the decomposed ones. Moreover, the decomposition for a single constraint is done in one step.

The remainder of the chapter is organized as follows. The next section is devoted to the SMT propagators for cardinality and Pseudo-Boolean constraints: lazy decomposition propagators are in some sense an extension of SMT propagators, since they also lazily provide an encoding to the SAT Solver, but they also introduce auxiliary variables. In Section 5.3 we define a framework for lazy decomposition propagators, and instantiate it for cardinality and Pseudo-Boolean constraints. In Section 5.4 we show the results of experiments, and in Section 5.5 we conclude.

5.2 Preliminaries

In this section we present the propagators for cardinality and Pseudo-Boolean constraints. As explained in Section 2.5.2, a propagator needs to perform the following actions: (i) return a set of the inferred literals when the SAT Solver assigns a value to an undefined literal, (ii) return to the previous state when the SAT Solver unassigns a literal, and, when the SAT Solver requires it, (iii) give a reason for a propagation done.

5.2.1 A Propagator for Cardinality Constraints

A cardinality constraint takes the form $x_1 + \cdots + x_n \ \# \ K$, where the K is an integer, the x_i are variables, and the relation operator $\#$ belongs to $\{\leqslant, \geqslant, =\}$.

For a $\leqslant$ constraint, the propagator keeps a count of the number of literals of the constraint which are *true* in the current assignment. The propagator increments this value every time the SAT solver assigns *true* a literal of the constraint. The count is decremented when the SAT solver unassigns one of these literals. When this value is equal to K, no other literal can be *true*: the propagator sets to *false* all the remaining literals. The reason for setting a literal x_j to *false* can be built by searching for the K literals $\{x_{i_1}, \ldots, x_{i_K}\}$ of the constraint which are *true* to give the reason $x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_K} \rightarrow \overline{x_j}$.

Similarly, in a $\geqslant$ constraint the propagator keeps a count of the literals which are *false* in the current assignment. When this value is equal to $n - K$, the propagator sets to *true* the non-propagated literals. A propagator for an equality constraint keeps track of both values. All these propagators are arc-consistent.

5.2.2 A Propagator for Pseudo-Boolean Constraints

PB constraints are a generalization of cardinality constraints. They take the form $a_1x_1+\cdots+a_nx_n \# K$, where K and a_i are integers, the x_i are literals, and the relation operator $\#$ belongs to $\{\leqslant, \geqslant, =\}$. In this chapter we assume that the operator $\#$ is $\leqslant$ and the coefficients a_i and K are positive. Other cases can be easily reduced to this one (see [ES06]).

The propagator must keep the current sum s during the search, defined as the sum of all coefficients a_i for which x_i is true. This value can be easily incrementally computed: every time the SAT solver sets a literal x_i of the constraint to true, the propagator adds a_i to s, and when the literal is unassigned by the SAT solver it subtracts a_i. For each $i \in \{1, \ldots, n\}$ such that x_i is unassigned and $K - s < a_i$, the propagator sets x_i to false. The propagator can produce reasons in the same way as in the cardinality case: if it has propagated x_j to false, $x_{i_1} \wedge \cdots \wedge x_{i_r} \rightarrow \overline{x_j}$, is returned as the reason, where $x_{i_1}, \cdots, x_{i_r}$ are all the literals of the constraint with true polarity. It is easy to see that this propagator is arc-consistent.

As an optimization, the propagator can compute a value A defined as an upper bound of the coefficients of the unassigned variables. In other words, for every i with x_i unassigned, it holds $A \geqslant a_i$. Initially, $A = \max\{a_i\}$. When $K - s \geqslant A$, the propagator does not need to visit any coefficient for checking if some literal can be propagated to false. This value A can be updated when the visit to the coefficients is necessary, i.e., when $K - s < A$, and when a literal x_i is unassigned.

5.3 Lazy Decomposition

The idea of lazy decomposition is quite simple: a Lazy Decomposition (LD) solver is, in some sense, a combination of a SMT Solver and an eager encoding. LD solvers, as SMT solvers, are composed of a SAT solver engine (that deals with the propositional part of the problem) and propagators, each one in charge of a complex constraint. The difference between SMT and LD solvers lies in the role of the propagators: SMT propagators only propagate and give explanations. LD propagators, in addition, detect which variables of the eager encoding would be helpful. These variables and the clauses from the eager encoding involving them are added to the SAT solver engine.

Here we design the LD for cardinality and Pseudo-Boolean constraints, but the idea of LD is not specific to them. Given a complex constraint type and an eager encoding method for it, we can create a LD solver for them if we can design a LD propagator able to perform the following actions:

- **Identify (dynamically) which parts of the encoding would be helpful to learning:** LD can be seen as a combined methodology that aims at taking advantage of the most profitable aspects of SMT and eager encoding. This point assures that the solver moves to the encoding when it is the best option.

- **Propagate the constraint when any subset of the encoding has been added:** The propagator must work either without encoding or with a part of it.
- **Avoid propagation for the constraint which is handled by the current encoding:** auxiliary variables from the eager encoding have their own meanings. The propagator must use these meanings in order to efficiently propagate the constraint when it is partially encoded. For example, if the entire encoding is added, we want the propagator to do no work at all.

In this chapter we present two examples of LD propagators: the first one, for cardinality constraints, is based on the eager encoding of Cardinality Networks of Section 3.3 [ANORC11b]. The second one is a propagator for Pseudo-Boolean constraints, is based on the BDD decomposition of Chapter 4 [ANORC11a].

5.3.1 Lazy Decomposition Propagator for Cardinality Constraints

In this section we describe the LD propagator for a cardinality constraint of the form $x_1 + x_2 + \ldots + x_n \leqslant K$. LD propagators for $\geqslant$ or $=$ cardinality constraints can be defined similarly.

According to Section 3.2, the encoding of a cardinality constraint based on cardinality networks consists in the encoding of 2-comparators into SAT. A key property of the 2-comparator $(y_1, y_2) = \text{2-Comp}(x_1, x_2)$ of Figure 3.1 is that $x_1 + x_2 = y_1 + y_2$. This holds since $y_1 = x_1 \vee x_2$ and $y_2 = x_1 \wedge x_2$. Thus we can define a *2-comparator decomposition step* for the 2-comparator $(y_1, y_2) = \text{2-Comp}(x_1, x_2)$ as replacing the current cardinality constraint $x_1 + x_2 + x_3 + \ldots + x_n \leqslant K$ by $y_1 + y_2 + x_3 + \ldots + x_n \leqslant K$ and adding a SAT encoding for the 2-comparator. The resulting constraint system is clearly equivalent. The encoding introduces the new variables y_1 and y_2.

The propagation of the LD propagator works just as in the SMT case. As decomposition occurs, the cardinality constraint that is being propagated changes by substituting newly defined encoding variables for older variables.

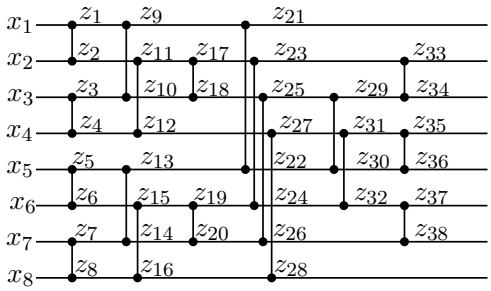

Figure 5.1: An 8-cardinality constraint of size 8, as described at Section 3.3. We are representing the 2-comparators as described at Section 3.2.

Example 5.1. *Figure 5.1 shows an 8-cardinality network with all its auxiliary variables and 2-comparators. A LD propagator for the constraint* $x_1+\ldots+x_8 \leqslant 5$ *initially behaves as an SMT propagator for that constraint. When variables* $z_1, z_2, \ldots, z_{12}$ *are introduced by decomposing the corresponding six 2-comparators, the substitutions result in the cardinality constraint* $z_5 + z_6 + z_7 + z_8 + z_9 + z_{10} + z_{11} + z_{12} \leqslant 5$.

A LD propagator must determine parts of the decomposition that should be added to the SAT solver. For simplicity, our LD solver adds variables only when it performs a restart: restarts occurs often enough for generating the important variables *not too late*, but occasionally enough to not significantly affect solver performance. Moreover, it is much easier to add variables and clauses to the solver at the root node, when the current assignment is empty.

The propagator assigns a natural number act_i, the *activity*, to every literal x_i of the constraint. Every time a reason is constructed, the activity of the literals belonging to the reason is incremented by one. Each time the solver restarts, the propagator checks if the activities of the literals of the constraint are greater than λN, where N is the number of conflicts since the last restart and λ is a parameter of the LD solver.

If $act_i < \lambda N$ then $act_i := act_i/2$. This is done in order to focus on the recent activity. If $act_i \geqslant \lambda N$, there are three possibilities:

- If x_i is not the input of a 2-comparator (i.e. an output of the cardinality network) nothing is done.
- If x_i is an input of a 2-comparator $(y_1, y_2) = \text{2-Comp}(x_i, x_j)$, and its other input x_j has already been generated by the lazy decomposition, we perform a decomposition step on the comparator.
- If x_i is an input of a 2-comparator $(y_1, y_2) = \text{2-Comp}(x_i, x_j)$, and its other input x_j has not been generated by the decomposition yet, we proceed as follows: let $S = \{x_{k_1}, x_{k_2}, \ldots, x_{k_s}\}$ be the literals in the current constraint that, after some decomposition steps, can reach x_j. We perform a decomposition step on all the comparators whose inputs both appear in S. Thus x_j is "closer" to being generated by decomposition.

Example 5.2. *Assume the LD propagator for the constraint* $x_1 + \ldots + x_8 \leqslant 5$ *has generated some variables so that the current constraint is* $z_9 + z_{17} + z_{18} + z_{12} + z_5 + z_{15} + z_7 + z_{16} \leqslant 5$. *The remaining undecomposed cardinality network is shown in Figure 5.2(a).*

In a restart, if the activity of z_{12} *is greater or equal than* λN *we decompose the comparator* $(z_{27}, z_{28}) = \text{2-Comp}(z_{12}, z_{16})$ *generating new literals* z_{27} *and* z_{28} *and using them to replace* z_{12} *and* z_{16} *in the constraint.*

However, if the activity of z_{18} *is greater or equal than* λN, *we cannot decompose* $(z_{25}, z_{26}) = \text{2-Comp}(z_{18}, z_{20})$ *since* z_{20} *has not been generated yet. The literals reaching* z_{20} *are* z_5, z_{15} *and* z_7 *(see Figure 5.2(b)). Since* z_5 *and* z_7 *are the inputs*

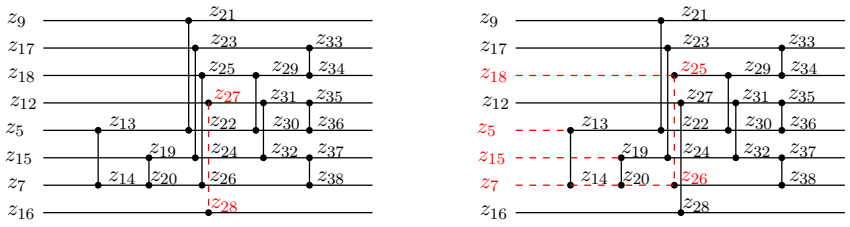

Figure 5.2: The remaining undecomposed sorting network after decomposing some 2-comparators with (a) $(z_{27}, z_{28}) = \text{2-Comp}(z_{12}, z_{16})$ shown dotted, and (b) inputs leading to $(z_{25}, z_{26}) = \text{2-Comp}(z_{18}, z_{20})$ shown dotted.

of a 2-comparator $(z_{13}, z_{14}) = \text{2-Comp}(z_5, z_7)$*, this comparator is encoded:* z_{13} *and* z_{14} *are introduced and they replace* z_5 *and* z_7 *in the constraint.*

Notice the variables are only replaced in the cardinality constraints: when a variable is removed of the constraint, it is not removed from the SAT Solver, so the lemmas where it appears are still valid.

Algorithms for the LD Propagator

In this section we reproduce in detail the algorithms explained above, i.e., algorithms that a lazy decomposition propagator for cardinality constraints needs to implement. In this section, if t is an array, then t_i will denote the i-th element of t.

The attributes of a propagator for a $\leqslant$-cardinality constraint are:

- An array of literals x containing the literals of the constraint.
- An integer s containing the number of true literals of the constraint in the current assignment.
- An array of integers *act* with the activities of the literals of the constraint.
- An integer K with the bound of the constraint.
- An integer n, the constraint size.

Additionally, another global (not specific for a single constraint but for all of them) auxiliary structure is needed, in order to associate every literal with the constraints it appears and the position of it in the constraint. We will assume that functions **assign**(y, C, i), that assigns the literal y to the i-th position of the constraint C, and **position**(y, C), which returns the position of y in the constraint C if y belongs to C or -1 if not, are implemented.

Algorithm 5.1 is executed when a cardinality constraint is read from the input. It creates a propagator for this constraint: the other algorithms are methods of this *object*.

Algorithm 5.1 Initialize the Propagator

Require: Constraint $C : x'_1 + x'_2 + \ldots + x'_{n'} \leqslant K'$
Ensure: Returns $\mathcal{P}$ a LD propagator for C.
1: $n \leftarrow n'$.
2: $s \leftarrow 0$.
3: $K \leftarrow K'$.
4: **for all** i such that $1 \leqslant i \leqslant n'$ **do**
5: $x_i \leftarrow x'_i$.
6: **assign**(x'_i, C, i).
7: $act_i \leftarrow 0$.
8: **end for**
9: **return** $\mathcal{P}$.

Every time the SAT Solver assigns or unassigns a literal, the following methods 5.2 or 5.3 of the propagator $\mathcal{P}$ are called, respectively.

Algorithm 5.2 Propagate a literal

Require: A literal y.
Ensure: Returns L a set of the propagated literals.
1: $L \leftarrow \emptyset$.
2: $i \leftarrow$**position**(y, C).
3: **if** $i \geqslant 0$ **then**
4: $s \leftarrow s + 1$.
5: **if** $s = K$ **then**
6: $L \leftarrow \{\overline{x}' \ : \ x'$ is a non-propagated literal of $C \ \}$.
7: **end if**
8: **end if**
9: **return** L.

When the SAT Solver needs a reason for a propagation, algorithm 5.4 is called. Finally, every restart method 5.5 is called.

5.3.2 Lazy Decomposition Propagator for PB Constraints

In this section we describe the LD propagator for a PB constraint of the form $C \equiv a_1x_1 + \cdots + a_nx_n \leqslant K$ with $a_i > 0$, since other PB constraints can be reduced to this one. We also assume $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$.

Suppose $\mathcal{B}$ is the BDD for PB constraint C. For simplicity, we work with quasi-reduced ordered BDDs, this is, ordered BDDs such that isomorphic trees are merged but nodes with identical children are NOT removed. The algorithms explained here can be adapted to ROBDDs, but it requires some technical details that could compromise the understanding of the section. Moreover, the overhead of using quasi-reduced ordered BDDs is negligible.

Algorithm 5.3 Unassign a literal

Require: A literal y.

$i \leftarrow$**position**(y, C).
if $i \geqslant 0$ **then**
  $s \leftarrow s - 1$.
end if

Algorithm 5.4 Give the reason for a propagation

Require: A literal y propagated by $\mathcal{P}$.
Ensure: Returns a clause L representing the reason why y was propagated.

$L \leftarrow \{\overline{y}\}$.
$i \leftarrow$**position**(y, C). // Notice $i > 0$ by requirement.
$act_i \leftarrow act_i + 1$.
for all x_j such that x_j is a true propagated literal of C **do**
  $act_j \leftarrow act_j + 1$.
  $L \leftarrow L \cup \{\overline{x_j}\}$.
end for
return L.

Algorithm 5.5 Generates useful literals

Let N be the number of conflicts since the last restart.
for all i such that $1 \leqslant i \leqslant n$ **do**
  if $act_i \geqslant \lambda N$ **then**
    if x_i is not an output of the network **then**
      Let x' be the other input of a 2-comparator having x_i as input.
      if x' has not been generated yet **then**
        DefineRecursive(x').
      else
        $j \leftarrow$**position**(x', C).
        Define(i, j).
      end if
    end if
  else
    $act_i \leftarrow act_i/2$.
  end if
end for

Algorithm 5.6 Define

Require: Integers i and j such that x_i and x_j are the inputs of a 2-comparator.
Ensure: Decompose the 2-comparator having x_i and x_j as input.
1: **Let** y, y' be the outputs of the 2-comparator having (x_i, x_j) as input.
2: **AddVariableToSATSolver**(y).
3: **AddVariableToSATSolver**(y').
4: **AddClauseToSATSolver**$(x_i \to y)$.
5: **AddClauseToSATSolver**$(x_j \to y)$.
6: **AddClauseToSATSolver**$(x_i \wedge x_j \to y')$.
7: $act_i \leftarrow 0$.
8: $act_j \leftarrow 0$.
9: **assign**$(x_i, C, -1)$.
10: **assign**$(x_j, C, -1)$.
11: **assign**(y, C, i).
12: **assign**(y', C, j).
13: $x_i \leftarrow y$.
14: $x_j \leftarrow y'$.

Algorithm 5.7 DefineRecursive

Require: A variable x which has not been generated yet.
1: **Let** (y, y') be the input of the 2-comparator having x as an output.
2: **if** y and y' have been both generated **then**
3: $i \leftarrow$**position**(y, C).
4: $j \leftarrow$**position**(y', C).
5: **Define**(i, j).
6: **else**
7: **if** y has not been generated yet **then**
8: **DefineRecursive**(y).
9: **end if**
10: **if** y' has not been generated yet **then**
11: **DefineRecursive**(y').
12: **end if**
13: **end if**

According Corollary 4.25, the encoding of the constraint works as follows: if ν is a node with selector variable x_i and interval $[\alpha, \beta]$, ν is set to true if $a_1x_1 + \cdots + a_{i-1}x_{i-1} \geqslant K-\beta$. If ν is set to false, the encoding assures that $a_1x_1+\cdots+a_{i-1}x_{i-1} \leqslant K - \beta + 1$. The LD propagator must maintain this property for nodes ν which have been created as a literal via decomposition.

In our LD propagator, the BDD is lazily encoded from bottom to top: all nodes with the same selector variable are encoded together, thus removing a layer from the bottom of the BDD. Therefore, the LD propagator must deal with the nodes ν at some level i which all represent expressions of the form $a_ix_i + \cdots + a_nx_n \leqslant \beta_\nu$ or equivalently $a_1x_1 + \cdots + a_{i-1}x_{i-1} \geqslant K - \beta_\nu$. Suppose ν' is the node at level i with highest β_ν where ν' is currently false. The decomposed part of the original PB constraint thus requires that $a_1x_1 + \cdots + a_{i-1}x_{i-1} \leqslant K - \beta_{\nu'} - 1$. Define $K_i = K - \beta_{\nu'} - 1$, and $node_i = \nu'$.

The LD propagator works as follows. The propagator maintains the current sum (lower bound) of the expression $s = a_1x_1 + \cdots + a_{i-1}x_{i-1}$, just as in the SMT case. If this value is greater than $K - \beta_\nu$ for some leaf node ν with selector variable x_i and interval $[\alpha_\nu, \beta_\nu]$, this node variable ν is set to *true*. If some leaf node ν (with selector variable x_i and interval $[\alpha_\nu, \beta_\nu]$) is set to *false*, we set

$$K_i \leftarrow \min\{K_i, K - \beta_\nu - 1\} \text{ and } node_i \leftarrow \begin{cases} \nu & \text{if } K_i = K - \beta_\nu - 1 \\ node_i & \text{otherwise.} \end{cases}$$

If, at some moment, $s + a_j$ for some $1 \leq j < i$ such that x_j is undefined is greater than K_i, the propagator sets x_j to *false*. The reason is the literals in $x_1, \ldots, x_{i-1}$ that are true and $\overline{node_i}$. As in the SMT case, the propagator may maintain an upper bound $A \geqslant \max\{a_j \ : \ j < i, \ x_j \text{ is undefined}\}$.

The policy for lazy decomposition is as follows. Every time a reason is generated that requires explanation from the PB constraint c, an activity act_c for the constraint c is incremented. If at restart $act_c \geqslant \mu N$ where N is the number of conflicts since last restart, and μ is a parameter of the solver, we decompose the bottom layer of c and set $act_c = 0$. Otherwise $act_c := act_c/2$.

Note that the fact that the coefficients a_i in c are in increasing order is important. Big coefficients are more important to the constraint and hence their corresponding variables are likely to be the most valuable for decomposition.

Algorithms for the LD Propagator

In this section we reproduce in detail the algorithms that a lazy decomposition propagator for Pseudo-Boolean needs to implement, explained in the previous section.

The attributes of a propagator for a $\leqslant$ Pseudo-Boolean constraint are:

- An array of literals x containing the literals of the constraint.
- An array of integers a with the coefficients of the literals of the constraint.

- A set of nodes $\mathcal{N}$ containing the leaf nodes.
- An integer s representing the sum of the coefficients of all the true literals of the constraint in the current assignment.
- An Integer *act* with the activity of the constraint.
- An integer K with the initial bound of the constraint.
- An integer K_{cur} with the current bound of the constraint (i.e., the value K_i).
- An integer n, the constraint size.
- An integer A with an upper bound of the non-propagated coefficients' literals.
- An integer A_{ini} with an upper bound of all the coefficients (for efficiency, it is better to precompute this value and set $A = A_{\text{ini}}$ than recompute A every time the SAT Solver unassigns a literal of the constraint).

As before, additionally, another global (not specific for a single constraint but for all of them) auxiliary structure is needed, in order to associate every SAT Solver literal with the constraints it appears and, either the position of it in the constraint (if it is a literal x_i) or with the interval it has (if it is a leaf node of the constraint).

As before, we will assume that the following functions are implemented:

- **assign**(y, C, i), which assigns the literal y to the i-th position of the constraint C.
- **assign**$(y, C, [\alpha, \beta])$, which assigns the literal y to the level n node having interval $[\alpha, \beta]$.
- **position**(y, C), which returns the position of y in the constraint C if y is a literal of C or -1 if not.
- **interval**(y, C), which returns the interval of the node y if y is a leaf node of C, or $\emptyset$ if not.

Algorithm 5.8 is called to initialize a propagator with a constraint. It creates a propagator for this constraint: the other algorithms are methods of this *object*. When the SAT Solver assigns or unassigns a literal, methods 5.9 and 5.10 are called. When the SAT Solver needs a reason for a propagation, algorithm 5.11 is called. In the restarts, SAT Solver calls method 5.12 of every propagator.

5.4 Experimental results

The goals of this section are, firstly, to check that Lazy Decomposition solvers do in fact significantly reduce the number of auxiliary variables generated and, secondly, to compare them to the SMT and eager encoding solving approaches. For

Algorithm 5.8 Initialize the Propagator

Require: Constraint $C : a'_1x'_1 + a'_2x'_2 + \ldots + a'_{n'}x'_{n'} \leqslant K'$.
Ensure: Returns $\mathcal{P}$ a LD propagator for C.

$x \leftarrow [x'_1, x'_2, \ldots, x'_{n'}]$.
$a \leftarrow [a'_1, a'_2, \ldots, a'_{n'}]$.
$\mathcal{N} := \emptyset$.
$s \leftarrow 0$.
$act \leftarrow 0$.
$K \leftarrow K'$.
$K_{\text{cur}} \leftarrow K'$.
$n \leftarrow n'$.
$A \leftarrow \max\{a_i \ : \ 1 \leqslant i \leqslant n'\}$.
$A_{\text{ini}} \leftarrow A$.
for all i such that $1 \leqslant i \leqslant n'$ **do**
  assign(x_i, C, i).
end for
return $\mathcal{P}$.

Algorithm 5.9 Propagate a literal

Require: A literal y.
Ensure: Returns L a set of the propagated literals.

$L \leftarrow \emptyset$.
$i \leftarrow$**position**(y, C).
if $i \geqslant 0$ **then**
  $s \leftarrow s + a_i$.
  for all $\nu \in \mathcal{N}$: $s > K - \beta_\nu$ and ν is not true **do**
    $L \leftarrow L \cup \{\nu\}$.
  end for
  if $s + A > K_{\text{cur}}$ **then**
    $L \leftarrow L \cup \{\overline{x_j} \ : \ x_j$ has not been propagated yet and $s + a_j > K_{\text{cur}}\}$.
    $A \leftarrow \max\{a_j \ : \ x_j$ has not been propagated yet and $x_j \notin L\}$.
  end if
else if $\overline{y} \in \mathcal{N}$ **then**
  $[\alpha, \beta] \leftarrow$ **interval**$(\overline{y}, C)$.
  if $K_{\text{cur}} > K - \beta - 1$ **then**
    $K_{\text{cur}} \leftarrow K - \beta - 1$
    if $s + A > K_{\text{cur}}$ **then**
      $L \leftarrow L \cup \{\overline{x_j} \ : \ x_j$ has not been propagated yet and $s + a_j > K_{\text{cur}}\}$.
      $A \leftarrow \max\{a_j \ : \ x_j$ has not been propagated yet and $x_j \notin L\}$.
    end if
  end if
end if
return L.

Algorithm 5.10 Unassign a literal

Require: A literal y.
1: $i \leftarrow$**position**(y, C).
2: **if** $i \geqslant 0$ **then**
3: $\quad s \leftarrow s - a_i$.
4: $\quad A \leftarrow A_{\text{ini}}$.
5: **else if** $\overline{y} \in \mathcal{N}$ **then**
6: $\quad K_{\text{cur}} \leftarrow \min\{K, K - \beta_\nu - 1 \ : \ \nu \in \mathcal{N} \text{ is false } \}$.
7: $\quad A \leftarrow A_{\text{ini}}$.
8: **end if**

Algorithm 5.11 Give the reason for a propagation

Require: A literal y propagated by $\mathcal{P}$.
Ensure: Returns a clause L representing the reason why y was propagated.
1: $L \leftarrow \{\overline{y}\}$.
2: $act \leftarrow act + 1$.
3: $i \leftarrow$**position**(y, C).
4: **if** $i \geqslant 0$ and $K_{\text{cur}} < K$ **then**
5: $\quad$ **Let** $\nu \in \mathcal{N}$ be the node such that $K_{\text{cur}} = K - \beta_\nu - 1$.
6: $\quad L \leftarrow L \cup \{\nu\}$.
7: **end if**
8: $L \leftarrow L \cup \{\overline{x_i} \ : \ x_i \text{ is a true propagated literal of } \mathcal{P}\}$.
9: **return** L.

Algorithm 5.12 Generates a level of the BDD if needed

Let N be the number of conflicts since the last restart.
if $act \geqslant \mu N$ **then**
  for all $\nu \in \mathcal{N}$ **do**
    remove$(\nu, \mathcal{N})$.
    assign$(\nu, C, \emptyset)$.
  end for
  for all Node ν at level n of the BDD of C **do**
    Let ν_f and ν_t be respectively the false and true children of ν in the BDD.
    AddVariableToSATSolver(ν).
    AddClauseToSATSolver$(\nu \to \nu_f)$.
    AddClauseToSATSolver$(\nu \wedge x_n \to \nu_t)$.
    insert$(\nu, \mathcal{N})$.
    Let $[\alpha, \beta]$ be the interval of ν in the BDD
    assign$(\nu, C, [\alpha, \beta])$.
  end for
  $act \leftarrow 0$.
  $A \leftarrow \max\{a_i \ : \ 1 \leqslant i \leqslant n-1\}$.
  $A_{\text{ini}} \leftarrow A$.
  remove(x_n, x).
  remove(a_n, a).
  $n \leftarrow n - 1$.
else
  $act \leftarrow act/2$.
end if

some problems we include other related solving approaches to illustrate we are not optimizing a very slow system.

All the methods are programmed in the Barcelogic SAT solver [BNO+08]. All experiments were performed on a 2Ghz Linux Quad-Core AMD. All the experiments used a value of $\lambda = 0.3$ and $\mu = 0.1$. We experimented with different values and found values for λ between 0.1–0.5 give similar performance, while values for μ between 0.05–0.5 also give similar performance. While there is more to investigate here, it is clear that no problem specific tuning of these parameters is required.

5.4.1 Cardinality Optimization Problems

Many of the benchmarks on which we have experimented are pure SAT problems with an optimal cardinality function (i.e., an objective function $x_1 + \cdots + x_n$) to minimize.

These problems can be solved by branch and bound: first, we search for an initial solution solving the SAT problem. Let Ω be the value of $x_1 + \cdots + x_n$ in this solution. Then, we include the cardinality constraint $x_1 + \cdots + x_n \leqslant \Omega - 1$. We repeatedly solve replacing the cardinality constraint by $x_1 + \cdots + x_n \leqslant \Omega - 1$, where Ω is the last solution found. The process finishes when the last problem is unsatisfiable, which means that Ω is the optimal solution.

Notice that this process can be used for all approaches considered. In the cardinality network encoding approach, the encoding is not re-generated every time a new solution is found: we just have to add a unit clause setting the Ω-th output variable of the network to false. SMT and LD solvers can also easily be adapted as branch and bound solvers, by modifying the bound on the constraint.

For all the benchmarks of this section we have compared the SMT solver for cardinality constraints (SMT), the eager cardinality constraint encoding approach of Section 3.3 (ENC), our Lazy Decomposition solver for cardinality constraints (LD), and the three best solvers for industrial partial MaxSAT problems in the past Partial MaxSAT Evaluation 2011: versions 1.1 (QMaxSAT1.1) and 4.0 (QMaxSAT4.0) of QMaxSAT [KZFH12] and Pwbo solver, version 1.2 (Pwbo) [MML11].

Partial MaxSAT

The first set of benchmarks we used were obtained from the MaxSAT Evaluation 2011 (see `http://maxsat.ia.udl.cat/introduction/`), industrial partial MaxSAT category. The benchmarks are encodings of different problems: filter design, logic synthesis, minimum-size test pattern generation, haplotype inference or maximum-quartet consistency.

We can easily transform these problems into SAT problems by introducing one fresh variable to any soft clause. The objective function is the sum of all these new variables. Time limit was set to 1800 seconds per benchmark as in the Evaluation. Table 5.1 shows the number of problems (up to 497) solved by the different methods after, respectively, 15 seconds, 1 minute, etc.

Method	15s	1m	5m	15m	30m
ENC	211	296	367	**382**	386
SMT	144	209	265	275	279
LD	**252**	**319**	**375**	381	**386**
QMaxSAT4.0	191	274	352	370	377
Pwbo	141	185	260	325	354
QMaxSAT1.1	185	278	356	373	383

Table 5.1: Number of instances solved of 497 partial MaxSAT benchmarks.

In these problems the eager encoding approach is much better than the SMT solver. Our LD approach has a similar behavior to the encoding approach, but LD is faster in the easiest problems. Notice that with these results we would be the best solver in the evaluation, even though our method for solving these problems (adding a fresh variable per soft clause) is a very naive one!

Discrete-Event System Diagnosis Suite

The next benchmarks we used are for discrete-event system (DES) diagnosis. In these problems, we consider a plant modeled by a finite automaton. Its transitions are labeled by the events that occur when the transition is triggered. A sequence of states and transitions on the DES is called a trajectory; it models a behavior of the plant. Some events are observable, that is, an observation is emitted when they occur. The goal of the problem is, knowing that there is a set of faulty events in the DES, find a trajectory consistent with the observations that minimizes the number of faults.

We have considered the problems from [AG09]. These problems consists in a set of clauses and a cardinality constraint to optimize. The paper, moreover, propose an encoding for these cardinality constraint which we have included (denoted by SARA-09). This encoding is specific for these problems: uses the set of input clauses of the problem in order to simplify it. Table 5.2 shows the number of benchmarks solved by the different methods after 15 seconds, 1 minute, etc.

The best method is that described in [AG09]. However, ENC and LD methods are not far from it. This is a strong argument for these methods, since SARA-09 is a specific method for these problems while eager encoding and lazy decomposition are general methods. On the other hand, SMT does not perform well in these problems, and LD performs more or less as ENC. Both versions of QMaxSAT also performs very well on these problems.

Close Solution Problems

Another type of optimization problems are those considered in Chapter 6. In these problems, we have a set of SAT clauses and a model, and we want to find the most

Method	15s	1m	5m	15m
ENC	409	490	530	541
SMT	151	186	206	228
LD	370	482	528	539
QMaxSAT4.0	275	421	534	**557**
Pwbo	265	361	423	446
QMaxSAT1.1	378	488	**537**	556
SARA-09	**411**	**501**	**537**	549

Table 5.2: Number of instances solved of 600 DES benchmarks.

Method	15s	1m	5m	15m	60m
ENC	18	24	**31**	**34**	34
SMT	16	18	24	27	30
LD	**19**	**26**	**31**	**34**	**36**
QMaxSAT4.0	9	14	18	20	22
Pwbo	5	6	7	7	7
QMaxSAT1.1	6	11	16	17	19

Table 5.3: Number of instances solved of the 40 original close-solution problems.

similar solution (with respect to the Hamming distance) to the given model if we add some few extra clauses. Table 5.3 contains the number of solved instances of the original paper after different times.

For the original problems LD is slightly better than eager encoding (ENC) and much better than the other approaches.

Since the number of instances of the original paper was small, we created more instances. We selected the 55 satisfiable instances from SAT Competition 2011, industrial division, that we could solve in 10 minutes. These problems have much more variables than the ones from the original paper. For each of these 55 problems, we generated 10 *close-solution* benchmarks adding a single randomly generated new clause (with at most 5 literals) that falsified the previous model. 100 of the 550 benchmarks were unsatisfiable, so we removed them (searching the closest solution does not make sense in an unsatisfiable problem). Table 5.4 shows the results on the remaining 450 instances.

For the new problems SMT and LD are the best methods with similar behavior. Notice that for these problems the cardinality constraint size involves all the variables of the problem, so it can be huge. In a few cases, the encoding approach runs out of memory since the encoding needed more than 2^{25} variables. We considered these cases as a timeout.

Method	15s	1m	5m	15m	60m
ENC	143	168	208	226	243
SMT	181	223	242	255	268
LD	**187**	**230**	**252**	**262**	**279**
QMaxSAT4.0	55	55	63	69	80
Pwbo	102	144	179	204	215
QMaxSAT1.1	54	55	57	57	64

Table 5.4: Number of instances solved of 450 new close-solution problems.

Method	15s	1m	5m	15m
ENC	190	282	352	411
SMT	123	168	212	241
LD	**263**	**336**	**410**	**435**

Table 5.5: Number of families solved from 479 non-trivial MSU4 problems.

5.4.2 MSU4

Another type of cardinality benchmarks also comes from the MaxSAT Evaluation 2008. In this case we solved them using the *msu4* algorithm [MSP09], which transforms a partial MaxSAT problem into a set of SAT problems with multiple cardinality constraints.

We have grouped all the problems that came from the same partial MaxSAT problem, and we set a timeout of 900 seconds for solving all the family of problems. We had 1883 families of problems (i.e., there were originally 1883 partial MaxSAT problems), but in many cases all the problems of the family could be solved by any method in less than 5 seconds, so we removed them. Table 5.5 contains the results on the remaining 479 benchmarks.

In these problems the LD approach is clearly the best, particularly in the first minute. The reason is that for most of the problems, ENC is faster than SMT, and LD performs similarly to ENC. But there are some problems where SMT is much faster than ENC: in these cases, LD is also faster than SMT, so in total it beats both other methods. Moreover, in some problems some constraints are dynamically discovered to be important. While some constraints should be encoded, others should not. The LD approach can do this, while ENC and SMT methods either encode all the constraints or none.

5.4.3 PB Competition Problems

To compare PB propagation approaches we used benchmarks from the Pseudo-Boolean Competition 2011 (`http://www.cril.univ-artois.fr/PB11/`), category

Method	15s	1m	5m	15m	60m
ENC	318	354	390	407	427
SMT	**372**	387	400	415	433
LD	369	382	401	423	439
borg	280	**406**	**438**	**445**	**467**

Table 5.6: Number of instances solved from 669 problems PB Competition-2011.

DEC-SMALLINT-LIN (no optimization, small integers, linear constraints). In these problems we have compared the SMT, ENC and LD approaches for PB constraints and the winner of the Pseudo-Boolean Competition 2011, the solver borg (borg) [SM10] version pb-dec-11.04.03. Table 5.6 contains the number of solved instances (up to 669) after 15 seconds, 1 minute, etc.

In this case, SMT approach is better than ENC, while LD is slightly better than SMT and much better than ENC since presumably it is worth decomposing some of the PB constraints to improve learning, but not all of them. The borg solver is clearly the best, but again it is a tuned portfolio solver specific for Pseudo-Boolean problems and makes use of techniques (as in linear programming solvers) which treat all PB constraints simultaneously.

5.4.4 Variables Generated

One of the goals of Lazy Decomposition is to reduce the search space of the problem. In this section we examine the "raw" search space size in terms of the number of Boolean variables in the model.

Table 5.7 shows the results of all the problem classes. ENC gives the multiplication factor of Boolean variables created by eager decomposition. For example if the original problem has 100 Boolean variables and the decomposition adds 150 auxiliary variables, we have 250 Boolean variables in total and the multiplication factor will be 2.5. LD gives the multiplication factor of Boolean variables resulting from lazy decomposition. Finally *aux. %* gives the percentage of auxiliary decomposition variables actually created using lazy decomposition. The values in the table are the average over all the problems in that class.

In the optimization problems, there is just one cardinality constraint and most of the time is devoted to proving the optimality of the best solution. Therefore, the cardinality constraint appears in most reasons since we require many explanations to prove the optimality of the solution. For these classes, the number of auxiliary variables we need is high (35-60 %). Still this reduction is significant.

In the MSU4 and PB problems, on the other hand, there are lots of complex constraints. Most of them have little impact in the problem (i.e., during the search they cause few propagations and conflicts). These constraints are not decomposed in the lazy approach. The LD solver only decomposes part of the most active

Class of problems	ENC	LD	*aux. %*
Partial MaxSAT	7.46	5.41	61.72
DES	1.55	1.16	26.62
Original close-solution	12.21	7.48	45.33
New close-solution	24.55	12.38	35.88
MSU4	1.77	1.01	2.18
PB Competition	44.21	17.52	3.24

Table 5.7: The average variable multiplication factor for (ENC) eager encoding and (LD) lazy decomposition, and the average percentage of auxiliary decomposition variables created by lazy decomposition.

constraints, so, the number of auxiliary variables generated in these problems is highly reduced.

5.5 Conclusions and Future Work

We have introduced a new general approach for dealing with complex constraints in complete methods for combinatorial optimization, that combines the advantages of decomposition and global constraint propagation. We illustrate this approach on two different constraints: cardinality and Pseudo-Boolean constraints. The results show that, in both cases, our new approach is nearly as good as the best of the eager encoding and global propagation approaches, and often better. Note that the strongest results for lazy decomposition arise when we have many complex constraints, since many of them will not be important for solving the problem, and hence encoding is completely wasteful. But we can see that for the important constraints it is worthwhile to decompose.

There are many directions for future work. First we can clearly improve our policies for when and what part of a constraint to decompose. We will also investigate how to decide the right form of decomposition for a constraint during execution rather than fixing on an encoding prior to search. We also plan to create lazy decomposition propagators for other complex constraints such as linear integer constraints, and incorporate the technology into a full SMT solver.

6 Close Solutions

6.1 Introduction

For many practical problems, good encodings into propositional logic exist that make them amenable to be solved with SAT. Due to techniques such as conflict-driven backjumping, lemma learning and restarts, state-of-the-art SAT solvers can in many cases efficiently solve large and hard real-world instances. For problems that have no good or compact direct encodings into propositional logic, several extensions of SAT are emerging. One of these extensions is *SAT Modulo Theories* (SMT), where atoms need not be propositional symbols, but may belong to *theories*, like, for example, linear arithmetic, as in the formula $x \leq 2 \wedge (x+y \geq 10 \vee 2x+3y \geq 30) \wedge y \leq 4$. In SMT, a SAT solver cooperates with *theory solvers* that can handle *conjunctions* of theory atoms (see, for instance, [NOT06] for details). Another extension of SAT is the Lazy Clause Generation approach of [OSC09], where new propositional clauses are generated on demand each time a given constraint propagates, thus frequently reducing the number of clauses needed in comparison with a direct *a priori* SAT encoding.

SAT and SAT-like solving approaches almost universally make use of *activity-based* search heuristics, which roughly speaking, select the variables that have been involved in many recent conflicts. A drawback of activity-based heuristics is that they make the search behave *chaotically* (explaining why is out of the scope of this document), i.e., extremely sensitive to the initial conditions, the so-called *butterfly effect*.

But in practice it is almost always important that the new solution is "close" to the original one. For example, analyzing a solution may take time and effort and include discussions with other people. If someone, inspired by the solution, suggests adding a few new constraints, it is undesirable that a new solution for the extended problem has nothing in common with what was analyzed previously. Something similar happens in the context of *rescheduling*, where a solution that was intended to be used for a period of time has to be adapted due to unforeseen circumstances: changes should be minimal since many resources (people, vehicles, machines) are

already allocated according to the original solution.

In this chapter[1], Section 6.2 gives an accurately definition of the problem and the distance metrics, e.g., what it means for a solution to be *close*. Section 6.3 presents the experimental setting and the large set of real-world benchmarks used along the chapter. In particular, in Section 6.4 we use them to experimentally demonstrate the extremely chaotic behavior of SAT Solvers, and in Section 6.5 to evaluate a naive attempt for finding close solutions inspired by local search methods.

Since this method does not solve the problem, in Section 6.6 we introduce a new approach. It combines a polarity heuristic, incremental SAT and branch-and-bound. In Section 6.7 we compare our method with (i) SAT-based optimization and Max-SAT solvers; (ii) modeling the problem as a 0-1 integer optimization problem and using CPLEX on it. As we shall see, approaches (i) and (ii) behave very poorly, but our new approach obtains close solutions very quickly. In fact, it typically finds the optimal (i.e., closest) solution in only 25% of the time the solver takes in solving the original problem.

Finally, Section 6.8 gives a factor analysis of our approach: experiments reveal that all ingredients contribute. Related work is discussed and conclusions are given in Section 6.9.

6.2 Problem definition

Assume we have found a solution *Sol* to a problem defined by a formula (a set of clauses) F and we are given a small set of additional clauses δ. We wish to find a solution Sol' that is *close* to *Sol* for the clause set $F \cup \delta$.

One way for defining solutions' proximity is by considering their Hamming distance (the number of variables which take a different value). As many problems have some *hidden* auxiliary variables in their SAT encoding F, it is frequently useful to consider only the *visible* (i.e., non hidden) variables for the distance definition.

Certain applications can require slightly more involved cost functions instead of just Hamming distance. For example, a single property of the solution, seen by the user, may depend on *combinations* of visible variables. For example, in the sports scheduling problems we will use later, a property like a match may depend on a variable m_{ijr} saying that these two teams i and j meet on round r, and another two h_{ir} and h_{jr} saying whether team i and j plays at home on round r. A more accurate cost function to capture "nearness to the existing solution" in this case would count a distance of 1 if either of m_{ijr} or h_{ir} differ from their previous values, but not count 2 if both differ.

However, in this section we have only considered Hamming distance cost functions for simplicity in the computations. In the majority of the practical cases, a

[1]Based on the paper "Reducing Chaos in SAT-Like Search: Finding Solutions Close to a Given One" from the *14th international conference on Theory and application of satisfiability testing* [ADNS11]

close solution for some distance is also a close solution for the Hamming distance (see the previous example).

6.3 Benchmarks

We have considered 40 instances of real-world benchmarks coming from five different families. Each instance consists of a different SAT formula F, the first solution Sol, and a number of required additional constraints δ. The first four families are for scheduling a double round-robin tournament among N (16, 20 or 24) teams:

`r16`: 10 instances with about 3000 variables and around 55000 clauses each;

`r20`: 10 instances with around 5000 variables and 180000 clauses each;

`R20`: 10 instances with around 5000 variables, 140000 clauses each;

`r24`: 4 instances with around 9000 variables, 270000 clauses each.

All teams meet each other once in the first $N-1$ weeks and again in the second $N-1$ weeks, with exactly one match per team each week. A given pair of teams must play at the home of one team in one half, and at the home of the other in the other half, and such matches must be spaced at least a certain minimal number of weeks apart. Additional constraints include, e.g., that no team ever plays at home (or away) three times in a row, other (public order, sportive, TV revenues) constraints, blocking given matches on given days, etc. Instances are rather different among each other, but most of them have around 10% hidden variables. The `R20` instances are also different in that their δs contain more constraints and hence the closest solution is usually not as close (see below).

The fifth family of benchmarks has six problems `tt0 - tt5` coming from real-world hard curriculum-based course timetabling problems, from the International Timetabling Competition, see the Barcelogic results on formulation 2 at `http://tabu.diegm.uniud.it/ctt`. These problems are very different from the `r` ones. Their numbers of (visible) variables and clauses are:

instances	variables	visible variables	clauses
`tt0`	12537	1500	71919
`tt1`	137688	6314	667470
`tt2`	60968	3150	305601
`tt3`	556569	9810	3372803
`tt4`	125029	4494	1001737
`tt5`	124330	3381	612475

For each instance, we consider Hamming distance on the visible variables as the cost function. All experiments were performed on a 2.66MHz Xeon.

6.4 Chaotic behavior of SAT

In this section we analyze what happens when simply re-executing the Solver with the new input $F \cup \delta$. Table 6.1 contains results on all 40 instances.

Here *Time original* denotes the time (in seconds) spent to compute the original solution *Sol*, *Time re-execution* denotes the time spent in the computation of Sol'. d_{opt} denotes the minimal Hamming distance from the original solution *Sol* to any solution of $F \cup \delta$. *Time ratio* is defined by the ratio between the re-execution time and the original time.

The *quality* of a solution Sol' at distance d of *Sol* is a real number between 0 and 1 defined by d_{opt}/d. For example, if d_{opt} is 10, then a solution at distance 50 has quality 0.2.

These experiments show the chaotic behavior of SAT Solvers: re-running the same solver with the same set of clauses except one or two added at the end of the input file causes the solver to perform a completely different search, giving a very different execution in terms of distance of the solutions and also in computation time. In particular, qualities are typically below 0.1, that is, ten times more distant than the optimal solution.

6.5 Trying a local search-like solution

In local search techniques, to find close solutions one usually resumes the search at the point where the original solution was found with the hope that another solution is found in the nearby neighborhood. Therefore, at first sight, mimicking local search might seem a good option for overcoming the chaotic behavior of SAT.

More specifically, we want to re-execute the solver in the region of the search tree where the original solution was found. A simple way of implementing this idea is by changing the *variable selection heuristics* as follows. We remember the ordered sequence of decision literals of the original solution, and when the solver is re-launched with the new constraints, it always decides on the first undefined literal of the sequence, with the same polarity, until the first conflict occurs. After that, we fall back to the standard decision heuristic. Note that this will always find the same solution *Sol* if *Sol* is also a solution of $F \cup \delta$.

Unfortunately, the results do not improve significantly upon re-running from scratch as described in the previous section. Table 6.2 contains the results of this method. We have obtained similar results with some variations of this method (keeping the lemmas of the original execution as in the next section, keeping this heuristic, or a combination of both).

6.6 Our Barcelogic approach

As we have seen in the previous sections, the naive approaches are not effective for solving this problem in practice. The good news is that an adequate combination of

Instance	Time original	d_{opt}	Quality	Time re-execution	Time ratio
r16-0	0.88	12	0.03	0.93	1.06
r16-1	1.58	14	0.04	1.27	0.80
r16-2	1.66	8	0.02	0.74	0.45
r16-3	0.97	8	0.02	1.63	1.68
r16-4	3.56	64	0.14	7.09	1.99
r16-5	0.03	12	0.22	0.04	1.33
r16-6	0.02	14	0.03	0.05	2.50
r16-7	0.4	18	0.04	0.69	1.72
r16-8	3.55	8	0.02	1.27	0.36
r16-9	1.39	12	0.03	0.61	0.44
r20-0	12.23	24	0.04	12.37	1.01
r20-1	59.6	8	0.01	20.00	0.34
r20-2	9.47	12	0.02	9.65	1.02
r20-3	12.82	14	0.03	2.83	0.22
r20-4	20.15	18	0.19	20.03	0.99
r20-5	20.48	16	0.02	8.82	0.43
r20-6	8.81	18	0.04	2.09	0.24
r20-7	10.88	20	0.03	13.46	1.24
r20-8	13.52	16	0.04	8.95	0.66
r20-9	7.04	12	0.03	12.39	1.76
R20-0	1.77	8	0.02	3.56	2.01
R20-1	2.37	88	0.17	6.30	2.66
R20-2	6.69	96	0.19	9.53	1.42
R20-3	9.46	8	0.01	5.30	0.56
R20-4	5.4	136	0.25	1.14	0.21
R20-5	1.14	1	0.00	7.04	6.18
R20-6	7.71	104	0.19	4.95	0.64
R20-7	5.45	26	0.05	0.62	0.11
R20-8	0.61	82	0.16	7.03	11.52
R20-9	7.49	94	0.16	1.78	0.24
r24-0	227.97	42	0.04	143.51	0.63
r24-1	124.28	58	0.05	315.14	2.54
r24-2	277.49	14	0.01	226.80	0.82
r24-3	200.53	8	0.01	416.14	2.08
tt-0	1.62	10	0.03	0.36	0.22
tt-1	0.96	10	0.07	0.93	0.97
tt-2	0.38	6	0.04	0.28	0.74
tt-3	16.3	8	0.01	14.20	0.87
tt-4	27.42	26	0.04	16.17	0.59
tt-5	1.75	8	0.02	1.73	0.99

Table 6.1: Results of re-execution.

Instance	Time original	d_{opt}	Quality	Time re-execution	Time ratio
r16-0	0.88	12	0.03	0.80	0.91
r16-1	1.58	14	0.03	1.87	1.18
r16-2	1.66	8	0.02	0.66	0.40
r16-3	0.97	8	0.02	2.81	2.90
r16-4	3.56	64	0.13	3.82	1.07
r16-5	0.03	12	0.03	0.08	2.67
r16-6	0.02	14	0.03	0.00	0.00
r16-7	0.4	18	0.04	0.18	0.45
r16-8	3.55	8	0.02	0.85	0.24
r16-9	1.39	12	0.03	1.27	0.91
r20-0	12.23	24	0.04	9.50	0.78
r20-1	59.6	8	0.31	0.03	0.00
r20-2	9.47	12	0.55	0.03	0.00
r20-3	12.82	14	0.03	6.64	0.52
r20-4	20.15	18	0.33	0.03	0.00
r20-5	20.48	16	0.03	21.12	1.03
r20-6	8.81	18	0.03	17.50	1.99
r20-7	10.88	20	0.03	6.69	0.61
r20-8	13.52	16	0.03	2.15	0.16
r20-9	7.04	12	0.02	6.96	0.99
R20-0	1.77	8	0.02	2.43	1.37
R20-1	2.37	88	0.16	5.20	2.19
R20-2	6.69	96	0.15	2.26	0.34
R20-3	9.46	8	0.02	4.77	0.50
R20-4	5.4	136	0.26	6.34	1.17
R20-5	1.14	1	0.00	5.84	5.12
R20-6	7.71	104	0.20	6.18	0.80
R20-7	5.45	26	0.05	11.27	2.07
R20-8	0.61	82	0.16	4.94	8.10
R20-9	7.49	94	0.17	2.82	0.38
r24-0	227.97	42	0.04	134.14	0.59
r24-1	124.28	58	0.05	3574.00	28.76
r24-2	277.49	14	0.01	157.08	0.57
r24-3	200.53	8	0.01	296.43	1.48
tt-0	1.62	10	0.03	0.21	0.13
tt-1	0.96	10	0.29	0.29	0.30
tt-2	0.38	6	0.60	0.13	0.34
tt-3	16.3	8	0.01	18.13	1.11
tt-4	27.42	26	0.04	11.88	0.43
tt-5	1.75	8	0.01	2.36	1.35

Table 6.2: Results of a local-search-like approach.

three quite well-known ingredients does obtain close solutions very quickly.

The first ingredient is a polarity selection heuristic: the SAT solver uses its standard heuristic for picking the next variable to decide upon, but it sets this variable's polarity as in the original solution *Sol* (other optimization tools do this too for the visible variables: first try those values that minimize the cost function; however, it is experimentally better to do that in all the variables; it is also related to, but different from, *phase saving* [PD10]).

Second, a branch-and-bound wrapper is placed around the standard SAT loop. Each time the cost of the best solution discovered so far is exceeded by the current partial assignment, due to literals $l_1 \dots l_n$ (on visible variables) that disagree with *Sol*, a backjump is forced from a conflict analysis on an "explanation" $\overline{l_1} \vee \dots \vee \overline{l_n}$ of why the cost is currently too high. In particular, this is done each time a better model is found, in order to find, from then on, only lower-cost models. Here this explanation clause need not be learned. The backjump clause itself is learned as usual. Eventually this process terminates by discovering unsatisfiability—that there is no "better" solution to the best already found. As is well-known, it may require far more time to prove optimality than it does to find an optimal solution.[2] However, good solutions can often be found in a short time. See, e.g., [MMS04, LNORC09, LNORC11] and references of these for many more details and an abstract framework for Boolean optimization. Notice that the branch-and-bound can be implemented with an SMT, an eager encoding or a Lazy Decomposition Solver. In the original paper we implemented an SMT, but, as seen in the previous chapter, the best results are achieved with the LDS approach

Third, the lemmas the SAT Solver generated when finding the original solution are added; this is sound since there are only *additional* constraints, no removed ones; this latter idea is also used in the context of incremental SAT solving for, e.g., verification applications.

6.7 Experimental comparison with Cplex and other tools

In this section we compare experimentally our approach with other tools. We first encoded $F \cup \delta$ together with the cost function as a Pseudo-Boolean (0-1 Integer Programming) optimization problem and tried the state-of-the-art Pseudo-Boolean solver *Bsolo* [MMS04] and the well-known commercial CPLEX solver.

We also tried several state-of-the-art Max-SAT solvers. MiniMaxSAT Solver [HLO08] found close solutions only in a few cases. The unsatisfiable-core-based MaxSAT solvers *msuncore* [MSP09] and *PM2* [ABL09] were not competitive either, among other reasons because unsat-core-based solvers find no solution before the optimal one. We do not report here on these MaxSAT solvers' results: they were always much worse than the listed ones.

[2] In fact, for some of the benchmarks in this examples *proving* optimality took days of CPU time.

We also tried Barcelogic omitting its ingredients one by one, i.e., without keeping the lemmas from the first run or without the modified polarity heuristic. The results are described in the next section. Bsolo and CPLEX results are without the lemmas: the number of lemmas was much bigger than the number of original constraints and these solvers perform much worse if we add them.

The results are given in Table 6.3.

Solution quality: As before, the table lists *solution qualities* as real numbers between 0 and 1: d_{opt} denotes the minimal Hamming distance from the original solution Sol to any solution of $F \cup \delta$ and again we say that a solution Sol' at distance d of Sol has *quality* d_{opt}/d.

Entries in the table: The table gives results on all 40 instances for Barcelogic, Bsolo and CPLEX. For each instance, column 2 lists the time T the (Barcelogic) SAT solver took to compute the initial solution Sol. The third column indicates the cost of the optimal solution, d_{opt}. For each approach, the table lists the quality of the solution found after 25% of T, after 50% of T, etc., up to 800% of T. Moreover, the two average rows show the average of, respectively, the first 20 problems and the 20 other (harder) ones. The two plots of Figure 6.1 represent graphically these averages. They also give some intuition about how the approaches scale.

6.8 Factor analysis of the Barcelogic approach

In this section we evaluate separately the different ingredients used in our approach. More specifically, we show the experimental results of our solver with just a Branch and Bound ("B&B" in the table; first column), adding the lemmas ("B&B + lemmas"; second column), with the modified polarity heuristic ("B&B + polarity"; third column) and finally "B&B + All" (fourth column). The results are given in Table 6.4. As in the previous section, the table shows the quality of the solution found after 25%, 50%, etc. of the time spent in solving the original problem.

Clearly, the polarity decision heuristic hugely improves the method. On the other hand, keeping the lemmas helps significantly for the hard problems, while on the easier ones the overhead of reading the additional clauses frequently does not pay off.

Again, the two plots of Figure 6.2 represent graphically the results of the table for average solution qualities of, respectively, the first 20 instances, and the other much harder 20 ones.

6.9 Related work and conclusions

We have studied, from a practical point of view, the problem of, given a SAT formula F with a model Sol, and a small set of additional clauses δ, finding a model of $F \cup \delta$ that is *close* to Sol.

Similar problems were studied before in a more theoretical (complexity) setting. [HHOW05] examine the problem of finding a set of diverse or similar solutions

	Time	d_{opt}	Barcelogic						Bsolo						Cplex					
			25	50	100	200	400	800	25	50	100	200	400	800	25	50	100	200	400	800
r16-0	0.88	12	1	1	1	1	1	1	0	0	.67	.67	1	1	0	0	0	1	1	1
r16-1	1.58	14	1	1	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	1
r16-2	1.66	8	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1
r16-3	0.97	8	1	1	1	1	1	1	0	0	.67	1	1	1	0	0	1	1	1	1
r16-4	3.56	64	.86	.86	.94	1	1	1	.67	.67	.67	.67	.67	.67	0	0	0	0	0	0
r16-5	0.03	12	0	0	.50	.60	1	1	0	0	0	0	0	0	0	0	0	0	0	0
r16-6	0.02	14	0	0	0	0	.12	.64	0	0	0	0	0	0	0	0	0	0	0	0
r16-7	0.4	18	.82	.82	.82	.82	1	1	0	0	0	.36	.36	.36	0	0	0	0	0	0
r16-8	3.55	8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
r16-9	1.39	12	1	1	1	1	1	1	0	.60	.60	.60	.60	1	0	0	1	1	1	1
r20-0	12.23	24	1	1	1	1	1	1	.34	.34	.34	.50	1	1	0	0	0	0	0	1
r20-1	59.6	8	1	1	1	1	1	1	.67	1	1	1	1	1	1	1	1	1	1	1
r20-2	9.47	12	1	1	1	1	1	1	.27	.38	.38	.60	.60	.75	0	1	1	1	1	1
r20-3	12.82	14	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
r20-4	20.15	18	1	1	1	1	1	1	.41	.41	.41	.43	.43	.64	0	0	.38	.38	1	1
r20-5	20.48	16	1	1	1	1	1	1	.57	.57	.57	.89	.89	1	0	0	0	0	.24	1
r20-6	8.81	18	1	1	1	1	1	1	.30	.82	.82	.82	.82	1	0	0	0	.90	.90	1
r20-7	10.88	20	1	1	1	1	1	1	.83	.83	.83	.91	1	1	0	0	.32	.32	.32	1
r20-8	13.52	16	1	1	1	1	1	1	.35	.35	.35	.35	.35	1	0	0	0	0	1	1
r20-9	7.04	12	1	1	1	1	1	1	0	.22	.25	.55	.60	.86	0	1	1	1	1	1
Av.	-	-	**.88**	**.88**	**.91**	**.92**	**.96**	**.98**	**.32**	**.46**	**.58**	**.67**	**.72**	**.81**	**.10**	**.30**	**.43**	**.53**	**.62**	**.80**
R20-0	1.77	8	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1
R20-1	2.37	88	.57	.57	.57	.57	.72	.75	0	0	0	0	.66	.66	0	0	0	0	0	0
R20-2	6.69	96	.74	.74	.74	.80	.84	.89	0	.53	.53	.55	.55	.70	0	0	0	0	0	0
R20-3	9.46	8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R20-4	5.4	136	.65	.65	.86	.86	.91	.97	0	0	0	0	0	0	0	0	0	0	0	0
R20-5	1.14	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
R20-6	7.71	104	.80	.80	.88	.88	.88	.88	0	0	0	.42	.57	.57	0	0	0	0	0	0
R20-7	5.45	26	.93	.93	1	1	1	1	.68	.68	.68	.68	.68	.68	0	1	1	1	1	1
R20-8	0.61	82	.84	.84	.84	.85	.98	.98	0	0	0	0	.60	.60	0	0	0	0	0	0
R20-9	7.49	94	.64	.77	.77	.84	.90	.90	0	.43	.59	.59	.59	.59	0	0	0	0	0	0
r24-0	227.97	42	1	1	1	1	1	1	0	0	0	0	.57	.57	0	0	0	0	0	0
r24-1	124.28	58	.58	.58	.58	.74	.74	.74	0	.42	.42	.42	.42	.42	0	0	0	0	0	0
r24-2	277.49	14	1	1	1	1	1	1	.37	.37	.37	.37	.37	.37	0	1	1	1	1	1
r24-3	200.53	8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
tt-0	1.62	10	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1
tt-1	0.96	10	0	.36	.36	.36	.36	.36	0	0	0	0	0	0	0	0	0	0	0	0
tt-2	0.38	6	0	.60	.60	.60	.75	.75	0	0	0	0	0	0	0	0	0	0	0	0
tt-3	16.3	8	.57	.57	.57	.57	.67	.67	0	0	0	0	0	0	0	0	0	0	0	0
tt-4	27.42	26	.10	.10	.50	.50	.50	.50	0	0	0	0	0	0	0	0	0	0	0	0
tt-5	1.75	8	0	0	0	0	.13	.14	0	0	0	0	0	0	0	0	0	0	0	0
Av.	-	-	**.67**	**.73**	**.76**	**.78**	**.82**	**.83**	**.15**	**.22**	**.28**	**.35**	**.45**	**.46**	**.10**	**.20**	**.25**	**.30**	**.30**	**.35**

Table 6.3: Comparative results of the three most competitive approaches: Barcelogic, Bsolo and CPLEX.

	Basic B&B						B&B + lemmas						B&B + polarity						B&B + All					
	25	50	100	200	400	800	25	50	100	200	400	800	25	50	100	200	400	800	25	50	100	200	400	800
r16-0	0	0	0	.04	.04	.04	.03	.03	.03	.04	.04	.04	1	1	1	1	1	1	1	1	1	1	1	1
r16-1	0	0	.04	.04	.04	.04	0	0	.04	.04	.04	.04	1	1	1	1	1	1	1	1	1	1	1	1
r16-2	0	.02	.02	.02	.02	.02	.02	.02	.02	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
r16-3	0	0	0	.03	.03	.03	0	0	.02	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
r16-4	0	0	0	.14	.16	.16	0	0	.14	.14	.15	.16	.80	.80	.82	.82	.86	.91	.86	.86	.94	1	1	1
r16-5	0	0	.38	.38	.38	.38	0	0	0	.03	.03	.03	0	0	.24	1	1	1	0	0	.50	.60	1	1
r16-6	0	0	0	.03	.03	.04	0	0	0	0	0	.03	0	.12	1	1	1	1	0	0	0	0	.12	.64
r16-7	0	0	0	.04	.05	.05	0	0	0	.05	.05	.05	.69	.82	.90	.90	1	1	.82	.82	.82	.82	1	1
r16-8	0	.02	.02	.02	.02	.02	0	.02	.02	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
r16-9	0	.04	.04	.04	.04	.04	0	0	0	.03	.03	.03	1	1	1	1	1	1	1	1	1	1	1	1
r20-0	0	0	0	.05	.05	.06	.04	.04	.05	.05	.05	.05	1	1	1	1	1	1	1	1	1	1	1	1
r20-1	0	.01	.01	.01	.01	.02	.01	.01	.01	.01	.01	.01	1	1	1	1	1	1	1	1	1	1	1	1
r20-2	0	0	0	.02	.02	.02	0	0	.02	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
r20-3	.03	.03	.03	.03	.03	.03	0	0	.02	.02	.03	.03	1	1	1	1	1	1	1	1	1	1	1	1
r20-4	0	0	0	.20	.20	.20	0	0	.04	.04	.04	.04	1	1	1	1	1	1	1	1	1	1	1	1
r20-5	0	.02	.02	.03	.03	.03	0	.03	.03	.03	.03	.03	1	1	1	1	1	1	1	1	1	1	1	1
r20-6	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	1	1	1	1	1	1	1	1	1	1	1	1
r20-7	0	0	0	.03	.03	.05	.06	.06	.06	.06	.06	.06	.91	.91	1	1	1	1	1	1	1	1	1	1
r20-8	0	0	.04	.04	.04	.04	0	.03	.03	.03	.03	.03	1	1	1	1	1	1	1	1	1	1	1	1
r20-9	0	0	0	.03	.03	.03	0	0	.02	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
Av.	**0**	**.01**	**.03**	**.06**	**.06**	**.07**	**.01**	**.01**	**.03**	**.04**	**.04**	**.04**	**.87**	**.88**	**.95**	**.99**	**.99**	**1**	**.88**	**.88**	**.91**	**.92**	**.96**	**.98**
R20-0	0	0	0	0	.02	.02	0	0	0	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
R20-1	0	0	0	0	.18	.18	0	0	0	.15	.15	.15	0	0	.64	.64	.64	.71	.57	.57	.57	.57	.72	.75
R20-2	0	0	0	.19	.19	.20	0	0	.22	.25	.25	.25	.52	.69	.73	.75	.86	.91	.74	.74	.74	.80	.84	.89
R20-3	0	0	.01	.02	.02	.02	0	.02	.02	.02	.02	.02	1	1	1	1	1	1	1	1	1	1	1	1
R20-4	.25	.25	.25	.25	.26	.26	.23	.25	.25	.25	.25	.27	0	.44	.77	.77	.79	.85	.65	.65	.86	.86	.91	.97
R20-5	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
R20-6	0	0	.20	.20	.26	.26	0	.16	.16	.23	.23	.23	.81	.81	.87	.87	.90	.91	.80	.80	.88	.88	.88	.88
R20-7	.05	.05	.06	.06	.06	.06	0	.05	.05	.05	.05	.06	.93	.93	1	1	1	1	.93	.93	1	1	1	1
R20-8	0	0	0	0	0	0	0	0	0	0	.13	.13	.50	.50	.50	.59	.72	.93	.84	.84	.84	.85	.98	.98
R20-9	0	.16	.20	.20	.20	.20	.18	.21	.21	.21	.24	.24	.71	.81	.85	.85	.89	.89	.64	.77	.77	.84	.90	.90
r24-0	0	0	.04	.04	.04	.04	0	0	0	.04	.04	.04	1	1	1	1	1	1	1	1	1	1	1	1
r24-1	0	0	0	0	.05	.05	0	.05	.05	.06	.06	.06	.67	.67	.76	.76	.76	.76	.58	.58	.58	.74	.74	.74
r24-2	0	0	.01	.01	.01	.01	0	.01	.01	.01	.01	.01	1	1	1	1	1	1	1	1	1	1	1	1
r24-3	0	0	0	0	.01	.01	.01	.01	.01	.01	.01	.01	1	1	1	1	1	1	1	1	1	1	1	1
tt-0	.03	.03	.03	.03	.04	.04	0	0	0	0	.03	.03	.05	.14	.14	.19	.25	.50	1	1	1	1	1	1
tt-1	0	0	.07	.07	.07	.07	0	0	.04	.04	.04	.04	0	.36	.36	.36	.36	.36	0	.36	.36	.36	.36	.36
tt-2	0	0	.04	.04	.04	.04	0	0	0	.02	.02	.02	0	.60	.60	.60	.60	1	0	.60	.60	.60	.75	.75
tt-3	0	0	.01	.01	.01	.01	0	0	0	.01	.01	.01	0	0	.01	.04	.11	.29	.57	.57	.57	.57	.67	.67
tt-4	0	0	.04	.04	.04	.04	0	.04	.04	.04	.04	.04	0	0	.06	.41	.41	.41	.10	.10	.50	.50	.50	.50
tt-5	0	0	0	.02	.02	.02	0	0	.02	.02	.02	.02	0	0	0	0	.14	.14	0	0	0	0	.13	.14
Av.	**.02**	**.02**	**.05**	**.06**	**.08**	**.08**	**.02**	**.04**	**.05**	**.07**	**.08**	**.08**	**.51**	**.60**	**.66**	**.69**	**.72**	**.78**	**.67**	**.73**	**.76**	**.78**	**.82**	**.83**

Table 6.4: Results of the factor analysis.

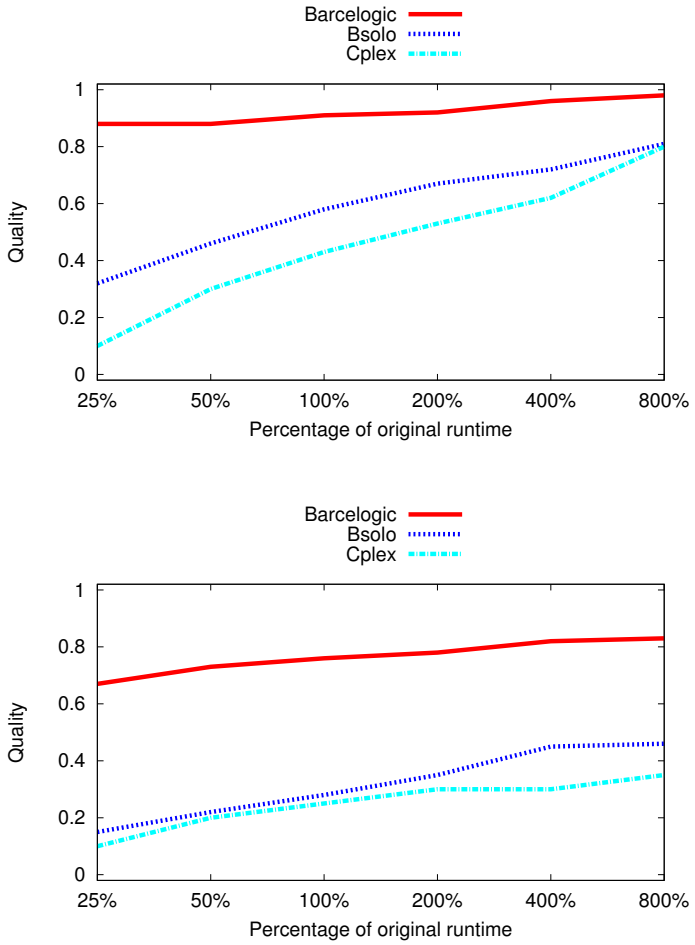

Figure 6.1: Average quality of the different approaches on the first 20 problems (top) and the second 20 harder ones (bottom).

for a single problem using constraint programming. Their MostClose question is very similar to the problem we examine looking for the closest solution to an existing solution, but both solutions are for the same problem. They outline two approaches: a reformulation approach that at least doubles the size of the problem, and a more efficient heuristic approach which is simply a branch and bound search. Our results show that this by itself is not enough in the SAT context. Distance-SAT [BM06] explores the decision problem, given a formula G and an *arbitrary* partial interpretation I, is there a model of G that disagrees with I on at most k variables? [BM06] tries on random and handcrafted problems two algorithms based on the

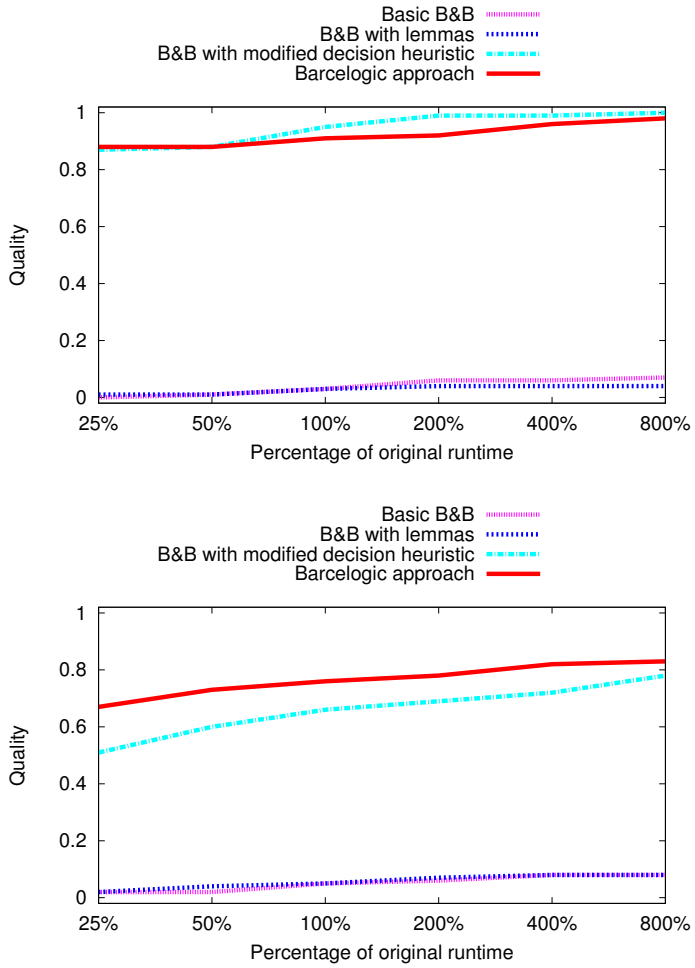

Figure 6.2: Average quality of the factor analysis on the first 20 problems (top) and the second 20 harder ones (bottom).

classical Davis/Logemann/Loveland (DLL) procedure [DLL62], but a translation into CNF is reported to work better. For our case, where deciding SAT for G is already hard, such a translation is rather hopeless. One clearly needs to exploit that in our problem I is a model of a *known* subformula of G that is almost the same as G.

Indeed, our experiments reveal that, while state-of-the-art Boolean optimization solvers behave poorly, our Barcelogic approach behaves very well, frequently finding the optimal (i.e., closest) solution in only 25% of the time the SAT solver took in solving the original problem.

7

Sport League Scheduling

7.1 Introduction

This chapter deals with the problem of scheduling a round-robin tournament. This problem may seem trivial, since a simple round-robin league can be scheduled with basic combinatorial methods; however, a professional league schedule must satisfy a lot of additional (non-structural) constraints, which make this problem much harder.

Different approaches have been applied to it: integer programming [DGM+07, CO06], constraint programming [Rég01, HMT04], tabu search [HH00], simulated annealing [WT94, AMHV06], mixed approaches using some of the previously described [Tri01], among others. See [RT08] for a survey on this field. Here we present the first (to our knowledge) SAT-based approach for solving this problem.

Its most difficult aspect is the set of combinatorial constraints that define a compact round-robin league. However, the additional constraints usually make the problem easier instead of harder: thus, problems with a lot of additional constraints can easily and quickly be solved with our method whereas pure combinatorial problems, i.e., problems with few additional constraints, turn out to be hard. This is the big difference with respect to the other methods, for which it usually holds that the fewer constraints, the easier the problem. Therefore, we think our approach may be able to deal with problems that are out of reach for the other methods.

Another convenient aspect of SAT-based technology is that, in optimization problems, the optimality of the solution is proven. This is also true in IP-like methods, but it is not the case of local-search-like methods. Local-search methods, however, can deal with bigger problems than the other methods: they usually are the only alternative for leagues with more than 24 teams.

Section 7.2 contains the terminology used throughout this chapter. In Section 7.3 we describe the constraints of the problem. Section 7.4 contains the variables needed in our approach for encoding the problem into SAT. Section 7.5 explains the different approaches for dealing with All-Different and Symmetric-All-Different constraints in SAT. Section 7.6 contains our encoding of the problem. In Section 7.7 we describe some minimal modifications we can make to our SAT Solver for solving more effi-

ciently these problems. Experimental results are presented in Section 7.8. Finally, we conclude in Section 7.9.

7.2 Terminology

This section introduces the terminology used throughout the chapter. Unfortunately, a standard terminology does not exist. We have followed the terminology introduced in [RT08].

Let n be the number of teams in the considered competition. In this document, n is assumed to be even. A *slot* is every period of time when any team can play at most one match.

Every team is associated to a *venue*, this is, a location where a match can take place. When a team plays at its own venue, plays at *home*, otherwise it plays *away*. In the latter case, it must play at the other team's venue.

A *round-robin league* is a tournament where all teams meet each other a fixed number of times. Here we consider *double round-robin* leagues, this is, teams meet twice, once at each venue.

A tournament can be *compact*, when every team plays a match in every slot, or *relaxed*, if teams do not play in every slot. Here we only consider compact tournaments: Notice that these tournaments require $2n - 2$ slots: the first $n - 1$ slots correspond to the *first half* of the competition, whereas the last $n-1$ ones correspond to *second half* slots.

Given a team, the sequence of home and away matches it plays during the league is known as a *home-away pattern* or simply *pattern*. It is represented by a vector of ones (for home matches) and zeros (for away ones). Two consecutive equal numbers in the pattern (this is, two consecutive home matches or two consecutive away matches) is called a *break* or *double*. Usually, it is desired to have few breaks in the patterns. Two patterns are *complementary* if they differ in all the values, this is, the first team plays at home when the second plays away and vice versa. The *home away pattern set*, or simply *pattern set*, is the set of patterns of every team. Table 7.1 shows an example of pattern set. Every row is a pattern. Patterns for teams 1 and 5 are complementary. Team 1 (and, of course, team 5) has breaks at slots 4, 6 and 9.

The league schedule can be presented as a timetable. Each row corresponds to a team and each column correspond to a slot. Entry of row i and column s corresponds to the opponent of the team i in the slot s; the pattern set corresponds to the signs in the timetable: the entry of the row i and column s is positive if the team i play at home, and negative if not. A schedule is *mirrored* when the first and the second half are identical except the home matches and away matches are exchanged. Table 7.2 contains a representation of a mirrored double round-robin league schedule.

Slots	1	2	3	4	5	6	7	8	9	10
Team 1	0	1	0	0	1	1	0	1	1	0
Team 2	1	0	0	1	0	0	1	1	0	1
Team 3	1	0	1	0	1	0	1	0	1	0
Team 4	0	1	1	0	1	1	0	0	1	0
Team 5	1	0	1	1	0	0	1	0	0	1
Team 6	0	1	0	1	0	1	0	1	0	1

Table 7.1: Example of a pattern set.

Slots	1	2	3	4	5	6	7	8	9	10
Team 1	-2	3	-4	-5	6	2	-3	4	5	-6
Team 2	1	-6	-5	3	-4	-1	6	5	-3	4
Team 3	4	-1	6	-2	5	-4	1	-6	2	-5
Team 4	-3	5	1	-6	2	3	-5	-1	6	-2
Team 5	6	-4	2	1	-3	-6	4	-2	-1	3
Team 6	-5	2	-3	4	-1	5	-2	3	-4	1

Table 7.2: Example of a schedule representation.

7.3 Constraints of the Schedule

In this section we present the usual constraints that can appear in these problems. However, not all constraints necessarily appear in every league scheduling problem (as in fact some subsets of them are incompatible). For clarity, we have not included the most technical constraints, since they can be encoded in a way similar to the ones explained here.

7.3.1 Structural Constraints

Some of the constraints are needed in order to ensure that the schedule corresponds to the desired type of tournament: in our case, a compact double round-robin league. Constraints for defining a mirrored schedule or restrictions in the patterns and in the breaks are also included in this group of constraints.

Here we list the structural constraints of these problems:

- **Double round-robin league:** Given a team, it must play once against every other team in the first half of the tournament and once in the second half.
- **Compact schedule:** In every slot, all teams must play a match.
- **Mirrored tournament:** In a mirrored tournament, the slots of the second half contain the same matches as in the first half, except that the venues are interchanged.

- **Minimum distance to return match:** In non-mirrored tournaments, a usual constraint is that if two teams play in a slot s, they cannot play again against each other in slots $s+1, s+2, \ldots, s+d-1$ for a fixed value of d.

  This constraint prevents things like a team playing against another team twice in a short bad patch (for instance, because its best player is injured): that is especially important if it involves two popular teams, since the audience of the matches could decrease if there are injured players.

  Notice that the case $d = n-1$ corresponds to a mirrored tournament.

- **Return per slots:** In some leagues, if a match is played at slot s_1 and its return at slot s_2, then the returns of *all* matches played at slot s_1 must be scheduled at slot s_2.

- **Home and away definition:** In a match, one team must play at home and the other team must play away.

- **Every pair of teams meet once at every venue:** If a match between teams t_1 and t_2 in a slot of the first half of the tournament has been played at t_1's home, that match has to be played at the t_2's home in the second half.

- **Triples are not allowed:** Most of the European leagues do not allow two consecutive breaks for a team.

  In this way, supporters can see their teams playing at home with some regularity, increasing ticketing revenues.

- **Number of breaks:** In some tournaments, the number of breaks in a pattern is bounded by a fixed value. In some other cases, it is required that every pattern has exactly a fixed number of breaks.

  As in the previous constraint, this is required in order to get matches at each venue with some regularity, and, in this way, maximizing ticketing revenues.

- **Breaks are not allowed in some slots:** Some leagues do not allow breaks in some slots. For instance, in the first two slots (when the supporters have not seen their team for a long time) all teams want to play once at home; the same happens in the last two slots. In some tournaments, breaks are not allowed at Christmas-time.

7.3.2 Additional Constraints

In this section we describe the rest of constraints of the problem. In a professional sport league, the schedule does not only contain constraints for being a double round-robin tournament: it also contains a lot of constraints to make the competition more attractive, fairer, etc. and maximize the revenues. These constraints are imposed by different sources: TV companies, security forces, the teams themselves...

Here we list the non-structural constraints of these problems:

- **Match constraints:** These constraints forbid a match to take place in some slots. They are often imposed by TV companies, because they do not want that higher audience matches take place in days when other events can overshadow them. They can also be imposed by the teams: for instance, a team that does not want to travel to far venues after an international match will forbid playing against teams with distant venues in the corresponding slots.

- **Place constraints:** A team must play at home (or away) in a certain slot. It is usually imposed when a venue is unavailable and the team cannot play at home, or when the slot coincides with a local festivity and the team wants to play at home.

- **Top teams constraints:** In some leagues, the teams do not want to play against two strong teams on two consecutive slots.

- **Complementary constraints:** This constraint imposes that two teams have complementary patterns. It is necessary in some leagues where there are teams sharing their playing grounds (e.g., stadiums).

- **Geographical constraints:** These constraints are often imposed by the security forces, transport agencies or similar. It avoids that a lot of matches take place in a small region, which could bring it to a standstill.

- **Top matches constraints:** In some leagues, TV companies select $n - 1$ matches: every slot must contain (exactly) one of these matches. In this way, the company can broadcast a high-audience match every week and, more importantly, a top match will not overshadow another one.

7.3.3 The Optimization Problem

In some cases, in addition to the mandatory constraints (called *hard* constraints), there are some additional constraints (called *soft* constraints) that the schedule can violate. In these cases, it is desired to find the schedule that satisfies the hard constraints and violates the minimum number of soft constraints.

Additional constraints and structural constraints involving breaks can be turned into soft constraints. However, the other structural constraints are always hard.

In some problems, the teams can add a fixed number of soft constraints, with *suggestions* for the league schedule. In these cases, an additional hard constraint is usually added:

- **Maximum number of soft constraints violated per team:** This constraint imposes that the number of soft constraints proposed by a team that can be unsatisfied for the resulting schedule is bounded by a fixed value.

Soft constraints can have associated a weight (or a cost): in this case, we have to minimize the sum of the weights of unsatisfied soft constraints. The *maximum number of soft constraints violated per team* constraint is modified in a similar way.

7.4 Variables of the Encoding

In this section we list the variables needed for encoding a sport league scheduling problem. Some of the variables are only needed if some of the previously described constraints are present, whereas others are always included.

- **Match variables:** Given two teams i, j with $i < j$ and a slot s, we define the variable $G_{i,j,s}$, which is true if and only if team i plays against team j in the slot s.
- **Pattern variables:** Given a team i and a slot s, we define the variable $H_{i,s}$, which is true if and only if team i plays at home in the slot s.
- **Break variables:** Given a team i and a slot $s > 1$, we define the variable $D_{i,s}$, which is true if and only if team i has a break in the slot s (i.e., plays at twice home or twice away in the slots $s-1$ and s).

  Indeed, we need to add the following clauses for defining them:

$$\left\{ \begin{array}{r} H_{i,s-1} \wedge H_{i,s} \rightarrow D_{i,s},\ H_{i,s-1} \wedge \overline{H_{i,s}} \rightarrow \overline{D_{i,s}},\ \overline{H_{i,s-1}} \wedge H_{i,s} \rightarrow \overline{D_{i,s}}, \\ \overline{H_{i,s-1}} \wedge \overline{H_{i,s}} \rightarrow D_{i,s},\ :\ 1 \leqslant i \leqslant n, 2 \leqslant s \leqslant 2n-2 \end{array} \right\}.$$

- **Optimization variables:** Given a soft constraint c, we define the variable o_c, which is true if the constraint c is not satisfied. In this way, if the constraint c can be encoded with the clause C as a hard constraint, we encode it with the clause $C \vee o_c$ as a soft constraint.

  These variables are only needed in optimization problems.
- **Auxiliary variables:** Besides, some auxiliary variables are needed for encoding some of the constraints. Letter a will denote an auxiliary variable.

7.5 All-Different Constraints

Some of the constraints of the sport league scheduling problem can be casted as particular cases of the so-called *all-different* or *symmetric-all-different* constraints. In this section, we study these constraints and the different options to deal with them.

7.5.1 Introduction

This section deals with all-different constraint (AD in the following). This constraint forces that $u_i \neq u_j$ for all $i, j \in \{1, 2, \ldots, n\}$ with $i \neq j$, where variables u_i are finite-domain variables with domain $\{1, 2, \ldots, n\}$[1].

[1] In a general version of these constraints, the variables can have an arbitrary finite domain. However, in this document we deal with this version of the constraint.

All-different constraints appear in many combinatorial problems: some puzzle-like problems such as Sodokus [LO06], the N-queens problem [ST05], Latin squares [Ref04]; or some industrial problems such as air traffic management [Grö04] or personnel scheduling [TFM+07]. In the sport league-scheduling problem, the *double round-robin league* structural constraints are AD constraints, as we will see in the next section.

A generalization of the AD constraint is the symmetric-all-different (SAD in the following) constraint[2]. Given variables u_i that have the domain $\{1, 2, \ldots, i-1, i+1, \ldots, n\}$, this constraint forces that

- $u_i \neq u_j$ for all $i \neq j$.
- If $u_i = j$, then $u_j = i$.

In the sport league scheduling problem, the *compact schedule* structural constraints are SAD constraints, as we will see in the next section.

7.5.2 Encoding AD and SAD into SAT

In this section we present different methods for encoding AD and SAD constraints into SAT. Unfortunately, arc-consistent encodings for AD and SAD must have an exponential number of clauses [vH01]: thus, the encodings proposed here are not arc-consistent.

The structure of the encoding of AD and SAD is similar in all methods. For AD, the encodings define a set of variables

$$\{x_{i,j} \ : \ 1 \leqslant i, j \leqslant n\}.$$

A variable $x_{i,j}$ is true if and only if u_i has been assigned to the value j. The encodings also have the following constraints:

- No two u's can take the same value j:

$$x_{1,j} + x_{2,j} + \cdots + x_{n,j} \leqslant 1, \quad 1 \leqslant j \leqslant n.$$

- Every value j must be taken for at least one u:

$$x_{1,j} + x_{2,j} + \cdots + x_{n,j} \geqslant 1, \quad 1 \leqslant j \leqslant n.$$

- Every u_i must have assigned at least one value:

$$x_{i,1} + x_{i,2} + \cdots + x_{i,n} \geqslant 1, \quad 1 \leqslant i \leqslant n.$$

- Every u_i must have assigned at most one value:

$$x_{i,1} + x_{i,2} + \cdots + x_{i,n} \leqslant 1, \quad 1 \leqslant i \leqslant n.$$

[2]This constraint is sometimes called *one-factor* constraint.

Note that some of these constraints are redundant. However, including all of them has been proved to provide more propagation power. Constraints of this form are called *ALO* and *AMO*. *ALO (At least one)* constraints are constraints of the form $x_1 + x_2 + \cdots + x_n \geqslant 1$, whereas *AMO (at most one)* constraints are constraints of the form $x_1 + x_2 + \cdots + x_n \leqslant 1$.

An ALO constraint $x_1 + x_2 + \cdots + x_n \geqslant 1$ can be encoded with a single clause:

$$x_1 \vee x_2 \vee \cdots \vee x_n.$$

However, AMO constraints can be encoded in several ways: every different method for encoding AMO defines an encoding of AD constraints.

SAD encodings are similar to AD encodings. In this case, we define a set of variables

$$\{x_{i,j} \ : \ 1 \leqslant i < j \leqslant n\}.$$

Now, a variable $x_{i,j}$ is true if and only if $u_i = j$ and $u_j = i$. The encodings also have the following constraints:

- AMO constraints:

$$x_{1,j} + x_{2,j} + \cdots + x_{j-1,j} + x_{j,j+1} + \cdots + x_{j,n} \leqslant 1, \quad 1 \leqslant j \leqslant n.$$

- ALO constraints:

$$x_{1,j} + x_{2,j} + \cdots + x_{j-1,j} + x_{j,j+1} + \cdots + x_{j,n} \geqslant 1, \quad 1 \leqslant j \leqslant n.$$

As before, AMO constraints are encoded with a single clause, whereas AMO constraints can be encoded in several ways.

Encoding AMO constraints into SAT

Consider now the AMO constraint $x_1 + x_2 + \cdots + x_n \leqslant 1$. We list the different encodings for this constraint into SAT that we can find in the literature:

- **Cardinality Network encoding:** AMO constraint is a cardinality constraint, so we could encode it as in Chapter 3. We need about n auxiliary variables and $3n$ clauses.

- **Quadratic encoding:** A naive encoding for AMO consists in adding the clauses

  $$\{\overline{x_i} \vee \overline{x_j} \ : \ 1 \leqslant i < j \leqslant n\}.$$

  We need no auxiliary variables, but $n(n-1)/2$ clauses. Notice this method corresponds to the direct cardinality network encoding (see Section 3.4).

- **Logarithmic encoding:** [FPDN05] Let us define $m = \log_2 n$. The logarithmic encoding consists in adding the variables $y_1, y_2, \ldots, y_m$ and, for all i and j with $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, the clause $\overline{x_i} \vee y_j$ if the j-th digit in binary of i is 1, or $\overline{x_i} \vee \overline{y_j}$ if it is 0.

  If one variable x_i is set to true, for any $i' \neq i$ with $1 \leqslant i' \leqslant n$, the binary representations of i' and i differ in at least one digit. For example, let us assume that the j-th digit of i is 1 whereas the j-th digit of i' is 0. Therefore, y_j is propagated to true by the clause $\overline{x_i} \vee y_j$, and the clause $\overline{x_{i'}} \vee \overline{y_j}$ propagates $x_{i'}$ to false.

  This encoding needs $\log n$ variables and $n \log n$ clauses.

- **Heule encoding:** [3] Let us take an integer $k \geqslant 2$. If $k + 1 \geqslant n$, the k-Heule encoding is the quadratic encoding. If $k + 1 < n$, it consists in introducing an auxiliary variable y and encoding (with k-Heule encoding) the AMO constraints $x_1 + x_2 + \cdots + x_k + y \leqslant 1$ and $x_{k+1} + x_{k+2} + \cdots + x_n + \overline{y} \leqslant 1$.

  This encoding needs about $\frac{n}{k-1}$ variables and $\frac{(k+1)kn}{2(k-1)}$ clauses. The following table shows the encoding size for the first values of k:

k	Variables	Clauses
2	n	$3n$
3	$n/2$	$3n$
4	$n/3$	$10n/3$
5	$n/4$	$15n/4$
6	$n/5$	$21n/5$

- **Ladder encoding:** [AM05] It is a particular case of the Heule encoding with $k = 2$.

- **2-product encoding:** [Che10] Let us define $p = \lceil \sqrt{n} \rceil$ and $q = \lceil n/p \rceil$. This method introduces the variables $u_1, u_2, \ldots, u_p, v_1, v_2, \ldots, v_q$, recursively encodes the AMOs $u_1 + \cdots + u_p \leqslant 1$ and $v_1 + \cdots + v_q \leqslant 1$ and adds the clauses

$$\{\overline{x_k} \vee u_i, \overline{x_k} \vee v_j \ : \ 1 \leqslant i \leqslant p, 1 \leqslant j \leqslant q, 1 \leqslant k \leqslant n, k = (i-1)q + j\}.$$

  Notice that the recursive AMOs can be encoded with this or any other method. The method uses about $2\sqrt{n} + V_1, +V_2$ variables and $2n + C_1 + C_2$ clauses, where (V_1, C_1) are the number of variables and clauses needed for encoding the AMO $u_1 + \cdots + u_p \leqslant 1$, and (V_2, C_2) are the number of variables and clauses needed for encoding the AMO $v_1 + \cdots + v_q \leqslant 1$.

  In particular, if the AMOs are recursively encoded with the 2-product method, we need $2\sqrt{n} + O(\sqrt[4]{n})$ variables and $2n + 4\sqrt{n} + O(\sqrt[4]{n})$ clauses.

[3] Personal communication. We want to thank him for the help with this encoding.

- **Generalized product encoding:** [Che10] Given a positive integer $k > 1$, let us define

$$p_k = \left\lceil \sqrt[k]{n} \right\rceil, \quad p_{k-1} = \left\lceil \sqrt[k-1]{\left\lceil \frac{n}{p_k} \right\rceil} \right\rceil, \quad \ldots, \quad p_1 = \left\lceil \frac{n}{p_k \cdot p_{k-1} \cdots p_2} \right\rceil.$$

This method introduces the variables w^i_j, with $i = 1, 2, \ldots, k$ and $j = 0, 1, \ldots, p_i - 1$, recursively (or with any other method) encodes the AMOs $w^i_0 + w^i_1 + \cdots + w^i_{p_i-1} \leqslant 1$ for $i = 1, 2, \ldots, k$ and adds the clauses

$$\bigcup_{1 \leqslant r \leqslant n} \left\{ \overline{x_r} \vee w^i_{j_i} \ : \ 1 \leqslant i \leqslant k, r - 1 = j_1 + p_1\Big(j_2 + p_2\big(j_3 + p_3(\cdots)\big)\Big) \right\}.$$

The method uses about

$$k\sqrt[k]{n} + \sum_{i=1}^{k} V_i \ \text{ variables and } \ nk + \sum_{i=1}^{k} C_i \ \text{ clauses,}$$

where (V_i, C_i) are the number of variables and clauses needed for encoding the i-th recursive AMO constraint.

7.6 Encoding the Constraints of the Schedule

In this section we show a way for encoding all the constraints from Section 7.3 into SAT or SMT.

7.6.1 Structural Constraints

- **Double round-robin league:** Given a team, this constraint consists of two AD constraints, one per half of the league: the opponents of team i in the slots of the first half $(1, 2, \ldots, n-1)$ must be different (the same in slots of the second half). Therefore, these constraints can be encoded into SAT in different ways as explained in Section 7.5.2; or they can be dealt with by a propagator [Rég94, GMN08, Kat08].

- **Compact schedule:** Given a slot, we have to pair up the teams: it consists of a SAD constraint: any team cannot play with two different teams in the same slot, and playing a match is a symmetric relation. This SAD constraint can be encoded into SAT as explained in Section 7.5.2; or it can be dealt with a propagator [Rég99].

- **Mirrored tournament:** In a mirrored tournament, we have to add the clauses

$$\{G_{i,j,s} \to G_{i,j,n-1+s},\ \overline{G_{i,j,s}} \to \overline{G_{i,j,n-1+s}} \ : \ 1 \leqslant i < j \leqslant n, 1 \leqslant s \leqslant n-1\}.$$

However, an easier encoding consists in using only variables for the first $n-1$ slots, since the second half slots can be built from them. Nevertheless, the two methods do not significantly differ with respect to the run-time, so any of these encodings can be used.

- **Return per slots:** We have to add the auxiliary variables $a_{s,s'}$ for $1 \leqslant s \leqslant n-1$ and $n \leqslant s' \leqslant 2n-2$ and the clauses:

$$\left\{ \begin{array}{c} G_{i,j,s} \wedge G_{i,j,s'} \rightarrow a_{s,s'},\ G_{i,j,s} \wedge a_{s,s'} \rightarrow G_{i,j,s'}, \\ G_{i,j,s'} \wedge a_{s,s'} \rightarrow G_{i,j,s}\ :\ 1 \leqslant i < j \leqslant n \end{array} \right\}.$$

- **Minimum distance to return match:** Assume a minimum distance d is demanded to the return matches. In this case, we have to add the clauses

$$\{G_{i,j,s} \rightarrow \overline{G_{i,j,s'}}\ :\ 1 \leqslant i < j \leqslant n, 1 \leqslant s \leqslant n-1, n \leqslant s' \leqslant 2n-2, s'-s < d\}.$$

However, if the *return-per-slots* constraint has been introduced, we can replace the previous clauses for these ones:

$$\{\overline{a_{s,s'}}\ :\ 1 \leqslant s \leqslant n-1, n \leqslant s' \leqslant 2n-2, s'-s < d\}.$$

- **Home and away definition:** The following clauses are added:

$$\{G_{i,j,s} \wedge H_{i,s} \rightarrow \overline{H_{j,s}},\ G_{i,j,s} \wedge \overline{H_{i,s}} \rightarrow H_{j,s},\ :\ 1 \leqslant i < j \leqslant n, 1 \leqslant s \leqslant 2n-2\}.$$

- **Every pair of teams meet once at every venue:** We add the clauses

$$\left\{ \begin{array}{c} G_{i,j,s} \wedge G_{i,j,s'} \wedge H_{i,s} \rightarrow \overline{H_{i,s'}},\ G_{i,j,s} \wedge G_{i,j,s'} \wedge \overline{H_{i,s}} \rightarrow H_{i,s'}\ : \\ 1 \leqslant i < j \leqslant n, 1 \leqslant s \leqslant n-1, n \leqslant s' \leqslant 2n-2 \end{array} \right\}.$$

If the *return per slots* constraint has been added, we can replace the previous clauses for these ones:

$$\left\{ \begin{array}{c} a_{s,s'} \wedge H_{i,s} \rightarrow \overline{H_{i,s'}},\ a_{s,s'} \wedge \overline{H_{i,s}} \rightarrow H_{i,s'}\ : \\ 1 \leqslant i \leqslant n, 1 \leqslant s \leqslant n-1, n \leqslant s' \leqslant 2n-2 \end{array} \right\}.$$

In a mirrored tournament, we can use these constraints instead of any of the previous ones:

$$\left\{H_{i,s} \rightarrow \overline{H_{i,s+n-1}},\ \overline{H_{i,s}} \rightarrow H_{i,s+n-1}\ :\ 1 \leqslant i \leqslant n, 1 \leqslant s \leqslant n-1\right\}.$$

- **Triples are not allowed:** The following clauses are added:

$$\{D_{i,s} \rightarrow \overline{D_{i,s+1}}\ :\ 1 \leqslant i \leqslant n, 2 \leqslant s < 2n-2\}.$$

- **Number of breaks:** Let k be the maximum number of breaks allowed. In this case, for every team i, we have to encode the cardinality constraints

$$D_{i,2} + D_{i,3} + \cdots + D_{i,n-1} \leqslant k,\ D_{i,n} + D_{i,n+1} + \cdots + D_{i,2n-1} \leqslant k,$$

  which can be done as in Chapter 3.

- **Breaks are not allowed in some slots:** We have to add the unit clauses

$$\{\overline{D_{i,s}}\ :\ 1 \leqslant i \leqslant n\}$$

  for every slot s where doubles are not allowed.

7.6.2 Additional Constraints

- **Match constraints:** If a match between teams i and j is not allowed in any venue in a slot s, then we add a unit clause $\overline{G_{i,j,s}}$.

  However, if the constraint does not allow a match from teams i and j at slot s in one of the venues (let us say, in team i's venue), the binary clause

$$\overline{G_{i,j,s}} \vee H_{j,s}$$

  is added.

- **Place constraints:** If team i must play at home in the slot s, we have to add the unit clause $H_{i,s}$. If it has to play away, we add $\overline{H_{i,s}}$.

- **Top teams constraints:** Given a team i, let T_i be its set of *top teams*, this is, team i cannot consecutively play against two teams of T_i. Then, we have to add the constraints

$$\{G_{i,j,s} \rightarrow \overline{G_{i,j',s+1}}\ :\ j, j' \in T_i, 1 \leqslant s < 2n-2\}.$$

- **Complementary constraints:** For every pair (i, j) of complementary teams, we have to add the constraints

$$\{H_{i,s} \rightarrow \overline{H_{j,s}},\ \overline{H_{i,s}} \rightarrow H_{j,s},\ :\ 1 \leqslant s \leqslant 2n-2\}.$$

- **Geographical constraints:** Let A be the set of teams of a small area, and assume that at most k can play at home in the same slot. This constraint consists in encoding the cardinality constraints

$$\left\{\sum_{i \in A} H_{i,s} \leqslant k\ :\ 1 \leqslant s \leqslant 2n-2\right\}.$$

  These cardinality constraints can be encoded as explained in Chapter 3.

- **Top matches constraints:** This constraint is an all-different constraint: the selected matches must be played in different slots. Therefore, it can be encoded as in Section 7.5.2 or we can use an SMT solver for dealing with it.

7.6.3 The Optimization Problem

Non-Weighted Case

- **Maximum number of soft constraints violated per team:** Let C_i be the soft constraints proposed by team i, and k_i be the maximum number of constraints from C_i that the schedule can violate. Then, these constraints are the cardinality constraints

$$\left\{ \sum_{c \in C_i} o_c \leqslant k_i \ : \ 1 \leqslant i \leqslant n \right\},$$

  which can be encoded as explained in Chapter 3.

To obtain the optimal solution, the solver has to minimize the cardinality objective function

$$\sum_{c \in C_1 \cup C_2 \cup \cdots \cup C_n} o_c.$$

Therefore, a Lazy Decomposition Solver can be used, see Section 5.4.1 for more details.

Weighted Case

- **Maximum number of soft constraints violated per team:** Let C_i be the soft constraints proposed by team i, and, for every $c \in C_i$, let w_c be its weight. Let k_i be the maximum sum of weights from the violated constraints of C_i. Then, these constraints are the Pseudo-Boolean constraints

$$\left\{ \sum_{c \in C_i} w_c o_c \leqslant k_i \ : \ 1 \leqslant i \leqslant n \right\},$$

  which can be encoded as explained in Chapter 4. Alternatively, we can use a propagator for them.

To obtain the optimal solution, the solver has to minimize the Pseudo-Boolean objective function

$$\sum_{c \in C_1 \cup C_2 \cup \cdots \cup C_n} w_c o_c.$$

Since the encoding of Pseudo-Boolean proposed in Chapter 4 is not incremental (i.e., if the constraint $\sum a_i x_i \leqslant k$ is replaced by the constraint $\sum a_i x_i \leqslant k'$ the whole encoding has to be generated from scratch), the best option for finding the optimal solution of the problem is an SMT approach.

7.7 Tuning the SAT Solver

In this section we present some minimal modifications to the SAT Solver in order to solve more efficiently sport league scheduling problems.

Notice that these problems are different from typical SAT applications coming from verification; for instance, the number of variables and clauses is reduced if we compare with some other SAT application problems (a problem may have about 20.000 variables). Therefore, some techniques that have proven to be useful in some contexts may work poorly here.

7.7.1 Cleanups Policy

The problems considered here are some orders of magnitude smaller than some other solvable problems. This means we do not need a very aggressive cleanup policy: the SAT Solver does not have space issues and the propagation does not become very slow. Quite the contrary, now keeping important lemmas is more important than removing the useless ones. Therefore, cleanups can be more spaced than in some other problems. Some strategies like [AS09] are useful in this context.

7.7.2 VSIDS Heuristics

We have generated the encoding of this problem: thus, we know the meaning of the distinct variables and we can use this information, for instance, to adapt the VSIDS policy to our problem.

The idea is splitting the problem into some parts: finding the pattern set, assigning matches to slots, assigning the optimization variables... This idea has widely been used in this context, reported for first time in [Sch92]. Our approach, however, does not completely decompose the problem, but it guides the SAT Solver on focusing on some parts of the problem before some others. In this way, we combine the advantages of solving first a small piece of the problem and the advantages of a non-decomposed problem.

We can easily do that in a simple way: we assign a coefficient to each variable. The VSIDS increments of the value of a variable are multiplied by this coefficient. In this way, variables with higher coefficients tend to be assigned with a higher priority.

We have divided the problem variables in groups as in Section 7.4, and we have assigned a value to every group. We have experimentally found that the best results are obtained when the pattern and break variables have the highest coefficients, then the optimization variables, and, finally, the match and auxiliary variables. An example of a good assignment of coefficients is:

Match	Pattern	Break	Optimization	Auxiliary
1	4	4	2	1

7.7.3 Last-Phase in Optimization Problems

In optimization problems, a variation of the *last-phase* strategy [PD07] has revealed to be useful. It consists in choosing always the negative phase of the optimization variables (this is, the phase that minimizes the objective function). Other variables' phases are chosen as in the last solution found. When searching for the first solution, non-optimization variables can be picked with the standard last-phase algorithm.

7.7.4 Handling More Leagues

A common problem consists in finding the schedule of two related leagues, for instance the first and second division league of the same country. In these cases, usually there are some constraints involving the two leagues (the *mixed constraints*), while most of them involve only one league. Usually one of the leagues is more important than the other one, and it contains a lot of constraints, while the other league is underconstrained (since the profits of the important league are much higher but it also requires more constraints from TVs, etc.).

We may try to simultaneously encode the two leagues and solve the SAT problem. However, this combined problem often requires too much time to be solved. Another possible approach is an iterative approach. Let X be the set of variables of the first league that appear in mixed constraints.

1. We solve the problem of the first league.
2. If the problem is UNSAT, we have finished. If not, we have obtained a model M. Let M_X be the assignment defined by M in the variables of X.
3. We assign the variables of M_X in the mixed constraints, and try to solve the second problem.
4. If the second problem is SAT, we have finished. If it is UNSAT, we add the clause $\neg M_X$ to the first problem and repeat the process.

However, this solution cannot deal with the optimization version of the problems. Moreover, the lemmas and heuristics of the previous iterations cannot be reused in the following ones.

We can improve this method with a technique similar to the one explained in Section 7.7.2. Let X_1, X_2 be the set of variables from the first and second problem respectively. We can modify the VSIDS heuristic so that no X_2 variable is chosen until all the variables from X_1 are assigned. In this way, we are automatically solving firstly the schedule of the important league and, after that, we solve the other one. Notice that this method can also be applied in optimization problems. Moreover, all the lemmas can be reused and, more importantly, if after solving the first problem the second one is UNSAT, we do not add $\neg M_X$ as a lemma: we add a much better lemma obtained from the usual conflict analysis.

7.8 Experimental Evaluation

In this section we evaluate our encoding in some benchmarks based in real-world sport professional leagues.

7.8.1 Instances Description

We have considered 15 benchmarks based on the French Football League. We have modified some of the original French Football League constraints in order to obtain 15 different instances with the different possible constraints described in the previous sections. The first 5 instances are non-optimization problems, the second 5 are optimization problems whose soft constraints have weight 1 and the last 5 problems are optimization problems with weighted soft constraints. We run 10 different random seeds for each benchmark.

Non-optimization problems' results are presented in tables like Table 7.3: they contain the number of problems solved after 5 seconds, 15 seconds, 1 minute, 5 minutes, etc.

The results of optimization problems are presented in tables like Table 7.4: columns 2 to 5 contain the number of benchmarks where a solution with cost $1, 2\Omega$ is found after, respectively, 1, 5, 15 and 60 minutes, being Ω the optimal cost. Columns 6 to 9 contain the number of problems where the optimal solution is found after 1, 5, 15 and 60 minutes. Finally, the last 4 columns contain the number of times the optimality is proven after 1, 5, 15 and 60 minutes.

All experiments were performed on a 2.66MHz Xeon.

7.8.2 Comparing the Different Encodings for AMOs

In first place, we compare different methods for encoding the AMOs from AD and SAD constraints (see Section 7.5.2). We compare the k-product encoding for $k = 2, 3, 4, 8$ (2-Product, 3-Product, 4-Product and 8-Product), the cardinality network encoding from Chapter 3 (Cardinality), the Heule encoding with $k = 3, 4, 5, 6$ (Heule-3, Heule-4, Heule-5 and Heule-6), the ladder encoding, which corresponds to Heule encoding for $k = 2$ (Ladder), the logarithmic encoding (Logarithmic) and the quadratic encoding (Quadratic). Table 7.3 contains the results of the different encodings in the non-optimization problems. Notice we have considered every random seed as a different benchmark. Table 7.4 contains the results in the optimization problems.

In these problems Heule gives the best performance, whereas Quadratic is not a competitive approach. The other approaches give similar performance: 4-Product seems slightly better than the other methods, whereas 3-Product seems the worst of the k--Product encodings.

We picked Heule-6 encoding method as encoding method for the next experiments since it showed the best results. Notice we can solve around the 80% of non-optimization problems in less than 1 minute, and the remaining in less than

Method	5s	15s	1m	5m	15m	60m
2-Product	0	10	29	**50**	**50**	**50**
3-Product	0	11	26	48	**50**	**50**
4-Product	1	11	36	**50**	**50**	**50**
8-Product	0	12	31	49	**50**	**50**
Cardinality	1	8	31	**50**	**50**	**50**
Heule-3	1	14	33	**50**	**50**	**50**
Heule-4	**3**	**17**	39	**50**	**50**	**50**
Heule-5	1	14	**45**	**50**	**50**	**50**
Heule-6	1	16	38	**50**	**50**	**50**
Ladder	0	10	25	48	**50**	**50**
Logarithmic	0	8	34	49	**50**	**50**
Quadratic	0	1	8	23	30	39

Table 7.3: Number of instances solved of 50 non-optimization benchmarks.

	$1,2\cdot$Optimal				Optimal found				Optimal proved			
Method	1m	5m	15m	1h	1m	5m	15m	1h	1m	5m	15m	1h
2-Product	**3**	47	76	95	**1**	32	69	79	0	10	42	77
3-Product	0	29	78	93	0	22	66	82	0	3	39	**80**
4-Product	0	51	77	94	0	40	72	83	0	21	70	79
8-Product	1	43	77	94	0	31	70	83	0	12	35	79
Cardinality	**3**	47	77	93	**1**	37	68	82	0	5	26	73
Heule-3	**3**	44	**87**	**97**	**1**	31	75	81	0	19	71	78
Heule-4	0	56	79	96	0	45	72	82	0	37	68	**80**
Heule-5	1	56	83	95	0	43	74	85	0	35	71	79
Heule-6	**3**	**62**	**87**	**97**	0	**56**	**78**	**87**	0	**49**	**76**	**80**
Ladder	0	38	73	94	0	26	63	83	0	13	56	**80**
Logarithmic	2	47	78	92	0	40	68	82	0	12	35	75
Quadratic	2	16	54	84	0	12	46	74	0	3	15	54

Table 7.4: Number of instances solved of 100 optimization benchmarks.

Method	Match	Pattern	Break	Optimization	Auxiliary
App1	1	2	2	1.5	1
App2	1	1	1	1	1
App3	1	1	1	1	2

Table 7.5: VSIDS coefficients used in the different approaches.

Method	5s	15s	1m	5m	15m	60m
App1	**1**	**16**	**38**	**50**	**50**	**50**
App2	0	11	36	48	**50**	**50**
App3	0	7	24	43	46	48

Table 7.6: Number of instances solved of 50 non-optimization benchmarks.

5 minutes. Optimization problems are harder since we are not interested in any solution, but in solutions with *good costs*. We have considered a solution with a cost $\leqslant 1.2$ times the optimal one to be a fair enough solution. Using this criterium, we find a good solution in less than 5 minutes in more than the 60% of the benchmarks, and in almost all the problems if we run the solver during 1 hour. Optimal solution is found in around an 80% of the cases in 15 minutes.

We can say, then, that we are able to quickly solve the non-optimization problems. In optimization problems we usually need more time, but we can find good solutions in almost all the cases in 1 hour. In most cases, we can find the optimal one in 15 minutes.

7.8.3 VSIDS Heuristics Tuning

In this section we evaluate the importance of tuning the VSIDS heuristics as explained in Section 7.7.2. Table 7.5 contains the coefficients used in the different approaches we have compared:

The first approach (App1) was explained in Section 7.7.2: pattern and break variables have a higher coefficient so they are assigned in first place. Afterwards, the solver assigns the optimization variables, and, finally, the rest. In the second approach (App2) all the coefficients are 1: it corresponds to the classic VSIDS heuristic. Finally, the third approach (App3) correspond to a different coefficient assignment. Tables 7.6 and 7.7 show all the related results.

We can see that a suitable VSIDS tuning (App1) improves the results either of the non-optimization and optimization problems compared to App2, which does not use VSIDS tuning. However, an inappropriate one (App3) makes the results worse than App2.

Nonetheless, the suitable VSIDS coefficients depend on the problem itself. Depending on the league (the number of teams of the schedule, if it is a mirrored league

	1, 2·Optimal				Optimal found				Optimal proved			
Method	1m	5m	15m	1h	1m	5m	15m	1h	1m	5m	15m	1h
App1	**3**	**62**	**87**	**97**	0	**56**	**78**	**87**	0	**49**	**76**	**80**
App2	0	49	80	93	0	34	74	81	0	20	66	78
App3	**3**	35	68	93	0	24	53	79	0	15	52	74

Table 7.7: Number of instances solved of 100 optimization benchmarks.

	1, 2·Optimal				Optimal found				Optimal proved			
Method	1m	5m	15m	1h	1m	5m	15m	1h	1m	5m	15m	1h
LP	0	20	68	91	0	14	66	85	0	8	65	**80**
LP-Opt	0	3	57	93	0	0	52	83	0	0	47	**80**
LS	1	53	78	91	0	42	71	77	0	34	68	73
LS-Opt	**3**	**62**	**87**	**97**	0	**56**	**78**	**87**	0	**49**	**76**	**80**

Table 7.8: Number of instances solved of 100 optimization benchmarks.

or a not, etc.) we use different coefficients.

7.8.4 Phase Selection Tuning

In this section we compare several policies of *phase selection*:

- LP: All the variables are chosen with the last-phase polarity, as explained at [PD07].
- LP-Opt: Optimization variables are always chosen with the negative polarity (in order to minimize the cost function), and the other ones are chosen with the last-phase polarity.
- LS: All the variables are chosen with the polarity of the last solution found.
- LS-Opt: Optimization variables are always chosen with the negative polarity, the other ones are chosen with the polarity of the last solution found. This approach has already been explained in Section 7.7.3.

Notice that all these approaches can only be applied in optimization problems, since in the non-optimization ones there are no previous solution or optimization variables. Therefore, we only compare them in the optimization benchmarks. Table 7.8 contains the results on these problems:

The results show that the two proposed modifications of last-phase (choosing the negative phase for optimization variables and the phase from the last solution for the other variables) improve the results.

Method	5s	15s	1m	5m	15m	60m
Conflict-5	**10**	17	45	49	**50**	**50**
Conflict-10	5	20	**47**	49	**50**	**50**
Conflict-15	1	18	45	49	49	**50**
Conflict-20	0	**22**	45	48	49	**50**
Glucose-5	3	17	40	**50**	**50**	**50**
Glucose-10	0	12	40	**50**	**50**	**50**
Glucose-15	0	13	42	49	**50**	**50**
Glucose-20	1	16	38	**50**	**50**	**50**

Table 7.9: Number of instances solved of 50 non-optimization benchmarks.

	$1,2\cdot$Optimal				Optimal found				Optimal proved			
Method	1m	5m	15m	1h	1m	5m	15m	1h	1m	5m	15m	1h
Conflict-5	**9**	**66**	83	92	**6**	54	70	77	**1**	49	68	74
Conflict-10	5	63	81	91	2	**56**	71	81	1	49	69	72
Conflict-15	8	64	80	89	**6**	54	70	76	0	46	69	74
Conflict-20	6	54	78	86	3	46	69	76	**1**	41	67	72
Glucose-5	4	63	81	**99**	0	55	71	**87**	0	**50**	71	**80**
Glucose-10	2	57	85	95	2	52	**78**	84	0	49	**77**	**80**
Glucose-15	6	58	82	93	2	48	75	84	0	45	74	**80**
Glucose-20	3	62	**87**	97	0	**56**	**78**	**87**	0	49	76	**80**

Table 7.10: Number of instances solved of 100 optimization benchmarks.

7.8.5 Cleanup Policy Tuning

In this section we show that in this problems a non-aggressive cleanup policy gives better results than an aggressive policy. We also compare a Glucose-style cleanup policy [AS09] (i.e., deleting the lemmas with most different decision levels) and the conflict-based policy (i.e., deleting the lemmas that have not appear in the recent conflict analysis).

The different strategies considered are Conflict-5, Conflict-10, Conflict-15, Conflict-20, Glucose-5, Glucose-10, Glucose-15 and Glucose-20, corresponding to conflict-based and Glucose-style cleanups with the first cleanup after 5, 10, 15 or 20 thousands of conflicts respectively. See Tables 7.9 and 7.10 for the results.

We can see that Glucose-style cleanups are slightly better than the conflict-based ones. Regarding the number of cleanups, it seems no significant difference appear in having more recent cleanups.

7.9 Conclusion and Future Work

We have presented a new approach for solving the sport league scheduling problems. As far as we know, this is the first SAT-based approach for dealing with this problem. Experimentally, our approach is able to solve real-world non-optimization problems in less than 5 minutes and find a good solution of optimization problems in at most 1 hour. In most cases, our solution was the optimal, and we found it in 5-15 minutes.

As future work, we want to improve our method to solve problems of leagues involving more teams. Another interesting future work field is related with the technique explained at Section 7.7.4: in that section we explained a method for dealing with the schedule of two related leagues. We think that this method can be extended in other SAT contexts, when two related problems must be solved and there are some constraints involving the two problems.

8

Conclusion and Future Work

In this chapter we review the main contributions of the PhD project and propose several directions for future work.

The main goal of this work is solving hard industrial combinatorial problems with SAT. The most challenging part in this field consists in dealing with the complex combinatorial constraints, this is, constraints which a SAT Solver cannot directly handle. There were two different approaches for dealing with them: encode these constraints into SAT or design a propagator for upgrading the SAT Solver into an SMT Solver.

In this document, we have firstly improved the existent encodings for two of the most common complex combinatorial constraints: cardinality constraints and Pseudo-Boolean constraints. Next, we have designed a third approach for dealing with complex constraints: lazy decomposition, which combines the advantages of the other two approaches. This new tool has been used for solving two particular real-world combinatorial problems: the close-solution and the sport league scheduling problems.

In the next paragraphs we give a more detailed description of the contributions of this thesis and we list some ideas for future work.

- **New encoding for cardinality constraints:** Cardinality constraints are present in many practical SAT applications, such as cumulative scheduling or timetabling; in this document we have used the cardinality constraints as part of an encoding for All-Different constraints, and for solving MaxSAT problems like *close-solution*.

  Our new method for encoding cardinality constraints needs many fewer variables than the state-of-the-art method, while the number of clauses may increase in a controlled way. Besides, the new encoding reduces the SAT Solver

run-time with respect to the state-of-the-art encodings. As future work, we want to develop encoding techniques for cardinality constraints that do not process constraints one-at-a-time but simultaneously, in order to exploit their similarities.

- **New encoding for Pseudo-Boolean constraints:** Pseudo-Boolean constraints are omnipresent in practical SAT applications as cumulative scheduling, logic synthesis or verification. In this document, these constraints appear in the weighted optimization version of professional sport leagues scheduling problems.

  We have studied the encoding of Pseudo-Boolean constraints through BDDs. First, we have proven that not all PB constraints admit polynomial BDDs. However, we have introduced a new BDD-based polynomial and arc-consistent encoding for these constraints that improves the state-of-the-art methods. Moreover, we have developed a BDD-based arc-consistent encoding of monotonic functions that only uses two clauses per BDD node instead of six. We have also presented an algorithm to efficiently construct all these BDDs and proved that the overall method is competitive in practice with respecto to state-of-the-art encodings and tools. As future work at the practical level, we plan to study which type of Pseudo-Boolean constraints are likely to produce smaller BDDs if encoded together rather than being encoded individually.

- **Conflict-directed lazy decomposition:** Lazy decomposition is a new general method for dealing with complex constraints with SAT. It combines the advantages of the eager encoding and the SMT approach. We have designed a lazy decomposition propagator for two different types of constraints: cardinality and Pseudo-Boolean. The experimental results show that our new approach is nearly as good as the best of the eager encoding and SMT approaches, and often better.

  As future work, we want to improve our policies for deciding when and which part of a constraint will be decomposed. In particular, we want to decide the encoding method during the execution rather than fixing it prior to search. Finally, we plan to create lazy decomposition propagators for other complex constraints such as linear integer constraints.

- **Close-solution problems:** SAT solvers algorithms for finding a model of a problem use some heuristics that makes the search chaotic, i.e., the solution is extremely sensitive to the initial conditions. This means that if two similar problem must be solved, the solution for these problems will be completely different. However, in real-world problems this is not desirable, since when a previously-solved problem is slightly modified, a similar solution is expected. In this document we have proposed a method for dealing with close-solution problems It is usually able to find the most similar solution in only 25% of the time the solver took in solving the original problem.

As future work, we want to adapt our method for an SMT and Lazy Decomposition solvers.

- **Professional sport leagues scheduling:** Finally, we have proposed a new encoding for solving professional sport leagues scheduling problems. We have proved that our method can schedule real-world leagues very fast, either with or without soft constraints.

  As future work, we want find better encodings for all-different and symmetric-all-different constraints for dealing with leagues with more than 24 teams. Additionally, we want to solve sport leagues scheduling problems with a (tuned) SMT Solver for all-different and symmetric-all-different constraints.

Bibliography

[AAN12] Roberto Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, pages 1–21, February 2012.

[ABL09] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Int. Conf. Theory and Applications of Satisfiability Testing (SAT), LNCS 4501*, pages 427–440, 2009.

[ADNS11] Ignasi Abío, Morgan Deters, Robert Nieuwenhuis, and Peter J. Stuckey. Reducing chaos in sat-like search: Finding solutions close to a given one. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT '11, pages 273–286, 2011.

[AG09] Anbulagan and Alban Grastien. Importance of Variables Semantic in CNF Encoding of Cardinality Constraints. In V. Bulitko and J. C. Beck, editors, *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA '09*. AAAI, 2009.

[ALMS11] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT '11, pages 188–200, Berlin, Heidelberg, 2011. Springer-Verlag.

[AM05] Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing*, SAT '04, pages 1–15, Berlin, Heidelberg, 2005. Springer-Verlag.

[AM06] Josep Argelich and Felip Manyà. Exact Max-SAT solvers for over-constrained problems. *Journal of Heuristics*, 12(4-5):375–392, September 2006.

[AMHV06] A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. *Journal of Scheduling*, 9(2):177–193, April 2006.

[ANO+] 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*. To appear.

[ANORC09] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Int. Conf. Theory and Applications of Satisfiability Testing (SAT), LNCS 4501*, pages 167–180, 2009.

[ANORC11a] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. BDDs for pseudo-boolean constraints: revisited. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT '11, pages 61–75, Berlin, Heidelberg, 2011. Springer-Verlag.

[ANORC11b] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality Networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[ARMS02] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 450–457, New York, NY, USA, 2002. ACM.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI '09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[AS12] Ignasi Abío and Peter J. Stuckey. Conflict-directed lazy decomposition. In *18th International Conference on Principles and Practice of Constraint Programming*, CP'12, 2012. To appear.

[Ass10] Roberto Assín. *SAT-based Techniques for Combinatorial Optimitzation*. PhD thesis, Technical University of Catalonia, 2010.

[Bat68] K. E. Batcher. Sorting Networks and their Applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[BB92] M. Buro and H. Kleine Büning. Report on a SAT Competition. Technical report, University of Paderborn, Germany, 1992. Technical Report tr-ri-92-110.

[BB03a] Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In F. Rossi, editor, *Principles and Practice of Constraint Programming, 9th International Conference, CP '03*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.

[BB03b] Constantinos Bartzis and Tevfik Bultan. Construction of efficient bdds for bounded arithmetic constraints. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS '03, pages 394–408, Berlin, Heidelberg, 2003. Springer-Verlag.

[BBR06a] O. Bailleux, Y. Boufkhad, and O. Roussel. A Translation of Pseudo Boolean Constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 2(1-4):191–200, 2006.

[BBR06b] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo boolean constraints to sat. *JSAT*, 2(1-4):191–200, 2006.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New Encodings of Pseudo-Boolean Constraints into CNF. In O. Kullmann, editor, *12th International Conference on Theory and Applications of Satisfiability Testing, SAT '09*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2009.

[BC00] Per Bjesse and Koen Claessen. SAT-Based Verification without State Space Traversal. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 372–389, London, UK, UK, 2000. Springer-Verlag.

[BCC+99] 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*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM.

[BEGJ00] Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the Relative Complexity of Resolution Refinements and Cutting Planes Proof Systems. *SIAM Journal on Computing*, 30(5):1462–1484, May 2000.

[BH88] Beate Bergmann and Gerhard Hommel. Improvements of general multiple test procedures for redundant systems of hypotheses. In P. Bauer, G. Hommel, and E. Sonnemann, editors, *Multiple Hypothesenprfung - Multiple Hypotheses Testing*, pages 100–115. Springer-Verlag, 1988.

[Bie08] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT '08, pages 28–33, Berlin, Heidelberg, 2008. Springer-Verlag.

[Bie10a] Armin Biere, 2010. Lingeling SAT Solver. Available at `http://fmv.jku.at/lingeling/`.

[Bie10b] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2010. Technical Report 10/1, August 2010, FMV Reports Series.

[BJMA06] Maria Luisa Bonet, Katherine St. John, Ruchi Mahindru, and Nina Amenta. Approximating subtree distances between phylogenies. *Journal of Computational Biology*, 13(8):1419–1434, 2006.

[BKNW09] Christian Bessiere, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI '09, pages 412–418, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[BKS03] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the Power of Clause Learning. In G. Gottlob and T. Walsh, editors, *In: Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1194–1201. Morgan Kaufmann, 2003.

[BLS02] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU Logic Formulas via Boolean and Pseudo-Boolean Encodings. In *Proceedings of the International Workshop on Constraints in Formal Verification*, CFV 02, September 2002. Associated with International Conference on Principles and Practice of Constraint Programming (CP '02).

[BLS11] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE '11, pages 101–115, Berlin, Heidelberg, 2011. Springer-Verlag.

[BM06] Olivier Bailleux and Pierre Marquis. Some computational aspects of distance-sat. *Journal of Automated Reasoning*, 37(4):231–260, November 2006.

[BNO+08] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The Barcelogic SMT Solver. In

Computer-aided Verification (CAV), volume 5123 of *Lecture Notes in Computer Science*, pages 294–298, 2008.

[BP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 7(2-3):59–6, 2010.

[Bra04] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(1):52–59, 2004.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, Providence, Rhode Island, 1997.

[BS00] Luís Baptista and João P. Marques Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 489–494, London, UK, UK, 2000. Springer-Verlag.

[CE82] Edmund Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 1982.

[CG96] Chih-Ang Chen and Sandeep K. Gupta. A satisfiability-based test generator for path delay faults in combinational circuits. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 209–214, New York, NY, USA, 1996. ACM.

[Che10] Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proceedings of the Tenth International Workshop of Constraint Modelling and Reformulation*, 2010.

[CO06] Federico Della Croce and Dario Oliveri. Scheduling the Italian football league: an ILP-based approach. *Computers and Operations Research*, 33(7):1963–1974, July 2006.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CZI10] Michael Codish and Moshe Zazon-Ivry. Pairwise cardinality networks. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010.

[DGM+07] Guillermo Durán, Mario Guajardo, Jaime Miranda, Denis Sauré, Sebastián Souyris, Andres Weintraub, and Rodrigo Wolf. Scheduling the Chilean Soccer League by Integer Programming. *Interfaces*, 37(6):539–552, November 2007.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[DLF+02] Lyndon Drake, Inês Lynce, Alan Frisch, João P. Marques Silva, and Toby Walsh. Comparing SAT Preprocessing Techniques. In T. Walsh, editor, *Proceedings of the Ninth Workshop on Automated Reasoning*, 2002.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM, CACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM, JACM*, 7(3):201–215, July 1960.

[EB05] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In F. Bacchus and T. Walsh, editors, *8th International Conference on Theory and Applications of Satisfiability Testing, SAT '05*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[EC93] Luis Entrena and Kwang-Ting Cheng. Sequential logic optimization by redundancy addition and removal. In Michael R. Lightner and Jochen A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 310–315, 1993.

[ES06] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[FM06] Zhaohui Fu and Sharad Malik. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 852–859, New York, NY, USA, 2006. ACM.

[FPDN05] Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter W. Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning*, 35(1-3):143–179, oct 2005.

[Gil60] P. C. Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of Research and Development*, 4(1):28–35, January 1960.

[GMN08] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, December 2008.

[GN02] Eugene Goldberg and Yakov Novikov. BerkMin: A Fast and Robust SAT-Solver. In *2002 Conference on Design, Automation, and Test in Europe, DATE '02*, pages 142–149. IEEE Computer Society, 2002.

[Grö04] Mattias Grönkvist. A constraint programming model for tail assignment. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference (CPAIOR)*, pages 142–156, 2004.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[HH00] Jean-Philippe Hamiez and Jin-Kao Hao. Solving the sports league scheduling problem with tabu search. In Alexander Nareyek, editor, *Local Search for Planning and Scheduling*, volume 2148 of *Lecture Notes in Computer Science*, pages 24–36. Springer, 2000.

[HHOW05] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *20th National Conference on Artificial Intelligence (AAAI)*, pages 372–377, 2005.

[HJB11] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient CNF simplification based on binary implication graphs. In *Proceedings of*

the 14th international conference on Theory and application of satisfiability testing, SAT '11, pages 201–215, Berlin, Heidelberg, 2011. Springer-Verlag.

[HLO08] Federico Heras, Javier Larrosa, and Albert Oliveras. MiniMaxSAT: An efficient Weighted Max-SAT Solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

[HMT04] Martin Henz, Tobias Müller, and Sven Thiel. Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, 153(1):92–101, 2004.

[HTY94] Kazuhisa Hosaka, Yasuhiko Takenaga, and Shuzo Yajima. On the Size of Ordered Binary Decision Diagrams Representing Threshold Functions. In *Algorithms and Computation, 5th International Symposium, ISAAC '94*, pages 584–592, 1994.

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI '07, pages 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[JG03] N. K. Jha and S. Gupta, editors. *Testing of Digital Systems*. Cambridge University Press, 2003.

[JHB12] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.

[JT96] David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.

[Kat08] George Katsirelos. *Nogood processing in CSPs*. PhD thesis, University of Toronto, 2008.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[KS+97a] Joonyoung Kim, João P. Marques Silva, , Hamid Savoj, and Karem A. Sakallah. Rid-grasp: Redundancy identification and removal using grasp. In *In IEEE/ACM International Workshop on Logic Synthesis*, 1997.

[KS97b] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks*. Kluwer Academic Publishers, 1997.

[KSMS11] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT '11, pages 343–356. Springer-Verlag, 2011.

[KZFH12] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSAT: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 8(1/2):95–100, 2012.

[Lar92] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[LNORC09] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Branch and bound for boolean optimization and the generation of optimality certificates. In *12th International Conference on Theory and Applications of Satisfiability Testing, SAT '09*, Lecture Notes in Computer Science 5584, pages 453–466, 2009.

[LNORC11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, 2011.

[LO06] Inês Lynce and Joël Ouaknine. Sudoku as a SAT Problem. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM 2006)*, 2006.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, September 1993.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[MML10] Vasco M. Manquinho, Ruben Martins, and Inês Lynce. Improving Unsatisfiability-Based Algorithms for Boolean Optimization. In O. Strichman and S. Szeider, editors, *13th International Conference on Theory and Applications of Satisfiability Testing*, volume 6175 of *SAT '10*, pages 181–193. Springer, 2010.

[MML11] Ruben Martins, Vasco Manquinho, and Inês Lynce. Parallel Search for Boolean Optimization. In *RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2011.

[MMS04] Vasco M. Manquinho and João P. Marques-Silva. Satisfiability-based algorithms for boolean optimization. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):353–372, March 2004.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[MP11] Panagiotis Manolios and Vasilis Papavasileiou. Pseudo-Boolean solving by incremental translation to SAT. In *Formal Methods in Computer-Aided Design*, FMCAD '11, 2011.

[MS06] Vasco M. Manquinho and João P. Marques Silva. On Using Cutting Planes in Pseudo-Boolean Optimization. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 2(1-4):209–219, 2006.

[MSG99] João Marques-Silva and Thomas Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '99, New York, NY, USA, 1999. ACM.

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI '92, pages 459–465. AAAI Press, 1992.

[MSP08] J. Marques-Silva and J. Planes. Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In *2008 Conference on Design, Automation and Test in Europe Conference, DATE '08*, pages 408–413. IEEE Computer Society, 2008.

[MSP09] Vasco M. Manquinho, João P. Marques Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In *Proceedings of the 12th international conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 495–508, 2009.

[MSS99a] J. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[MSS99b] J. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[NMJ09] Mladen Nikolic, Filip Maric, and Predrag Janicic. Instance-Based Selection of Policies for SAT Solvers. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 326–340, Berlin, Heidelberg, 2009. Springer-Verlag.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM, JACM*, 53(6):937–977, 2006.

[Nov03] Yakov Novikov. Local search for boolean relations on the basis of unit propagation. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10810–, Washington, DC, USA, 2003. IEEE Computer Society.

[NSR99] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. Satisfiability-based layout revisited: detailed routing of complex FPGAs via search-based Boolean SAT. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 167–175, New York, NY, USA, 1999. ACM.

[OSC09] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[PD10] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3):277–301, 2010.

[Ref04] Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In *10th International Conference on Principles and Practice of Constraint Programming*, CP'04, pages 557–571, 2004.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 362–367, 1994.

[Rég99] Jean-Charles Régin. The symmetric alldiff constraint. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 420–425, 1999.

[Rég01] Jean-Charles Régin. Minimization of the number of breaks in sports scheduling problems using constraint programming. In *DIMACS workshop on on Constraint programming and large scale discrete optimization*, pages 115–130, Boston, MA, USA, 2001. American Mathematical Society.

[RT08] Rasmus V. Rasmussen and Michael A. Trick. Round robin scheduling - a survey. *European Journal of Operational Research*, 188(3):617–636, August 2008.

[Rya04] Lawrence Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, School of Computing Science, Simon Fraser University, 2004.

[SB09] Niklas Sörensson and Armin Biere. Minimizing Learned Clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sch92] Jan A.M. Schreuder. Combinatorial aspects of construction of competition dutch professional football leagues. *Discrete Applied Mathematics*, 35(3):301–312, 1992.

[Sed78] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, October 1978.

[SFSW09] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Why cumulative decomposition is not as bad as it sounds. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pages 746–761, Berlin, Heidelberg, 2009. Springer-Verlag.

[SI09] Carsten Sinz and Markus Iser. Problem-Sensitive Restart Heuristics for the DPLL Procedure. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 356–362, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sil99] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, pages 62–74, London, UK, 1999. Springer-Verlag.

[Sil00] João P. Marques Silva. Algebraic Simplification Techniques for Propositional Satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 537–542, London, UK, 2000. Springer-Verlag.

[SM10] Bryan Silverthorn and Risto Miikkulainen. Latent class models for algorithm portfolio methods. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[Sma07] J. Smaus. On Boolean Functions Encodable as a Single Linear Pseudo-Boolean Constraint. In P. Van Hentenryck and L. A. Wolsey, editors, *4th International Conference on the Integration of AI and OR Techniques in Constraint Programming, CPAIOR '07*, volume 4510 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2007.

[Soo10] Mate Soos, 2010. CryptoMiniSat Solver. Available at `http://www.msoos.org/cryptominisat2/`.

[ST05] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Principles and Practice of Constraint Programming (CP)*, pages 817–821, 2005.

[TFM+07] Edward P. K. Tsang, John A. Ford, Patrick Mills, Richard Bradwell, Richard Williams, and Paul Scott. Towards a practical engineering tool for rostering. *Annals of Operations Research*, 155(1):257–277, 2007.

[Tri01] Michael A. Trick. A schedule-then-break approach to sports timetabling. In *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III*, PATAT '00, pages 242–253, London, UK, UK, 2001. Springer-Verlag.

[Tse68] G. S. Tseitin. On the Complexity of Derivation in the Propositional Calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968.

[vE98] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '98, pages 618–623, Washington, DC, USA, 1998. IEEE Computer Society.

[VG11] Allen Van Gelder. Generalized conflict-clause strengthening for satisfiability solvers. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT '11, pages 329–342, Berlin, Heidelberg, 2011.

[vH01] Willem Jan van Hoeve. The alldifferent Constraint: A Survey. *The Computing Research Repository (CoRR)*, 2001.

[War98] Joost P. Warners. A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. *Information Processing Letters*, 68(2):63–69, 1998.

[WT94] Robert J. Willis and Bernard J. Terril. Scheduling the australian state cricket season using simulated annealing. *Journal of the Operational Research Society*, 45(3):276–280, March 1994.