INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Learning Techniques for Pseudo-Boolean Solving and Optimization

José Faustino Fragoso Fremenin dos Santos

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente: Prof. Nuno João Neves Mamede

Orientador: Prof. Vasco Miguel Gomes Nunes Manquinho

Vogais: Prof. Maria Inês Camarate de Campos Lynce de Faria

Setembro de 2008

Resumo

Os avanços recentes experimentados pelos algoritmos para Satisfação e Optimização Inteira com variáveis proposicionais (PB-SAT e PBO respectivamente) foram motivados pelo conhecimento acumulado nos algoritmos para Satisfação Proposicional (SAT), nomeadamente a extensão da aprendizagem baseada em conflitos dos algoritmos para SAT para os algoritmos para PB-SAT e PBO. Tal extensão compreende diferentes esquemas de aprendizagem. Contudo, a comunidade cientifíca não está de acordo sobre qual o melhor esquema de aprendizagem a ser utilizado por estes algoritmos. Assim sendo, a contribuição deste trabalho consiste na apresentação de um exaustivo estudo comparativo entre vários esquemas de aprendizagem diferentes sobre uma plataforma comum. Os resultados de cada esquema de aprendizagem relativos a um grande conjunto de instâncias são apresentados, tendo sido implementados sobre um algoritmo de acordo com o estado da arte, o bsolo.

Também são apresentados resultados preliminares relativos à aplicação de algoritmos de aprendizagem automática para a selecção do esquema de aprendizagem mais apropriado para cada instância dada como input. O objectivo prende-se com o desenvolvimento de um algoritmo para PBO e PB-SAT "adaptável" à instância dada como input por forma a tirar partido dos diferentes esquemas de aprendizagem.

Finalmente, vários aspectos relativos à implementação de um algoritmo para PBO e PB-SAT são discutidos e os respectivos resultados experimentais são apresentados e analisados.

Palavras Chave

Optimazação inteira com variáveis proposicionais, Satisfação proposicional, Saltos não cronológicos, Aprendizagem baseada em conflitos, Planos de corte, Reduções a restrições de cardinalidade.

Abstract

Recent advances in Pseudo-Boolean Solving and Optimization (PB-SAT and PBO respectively) have been motivated by the accumulated knowledge in Propositional Satisfiability (SAT) algorithms, namely the extension of conflict-based learning from SAT solvers to PB solvers. Such extension comprises several different learning schemes. However, it is not commonly agreed among the research community which learning scheme should be used in PB solvers. Hence, this work presents a contribution by providing an exhaustive comparitive study between several learning schemes in a common platform. Results for a large set of benchmarks are presented for the different learning schemes, which were implemented on *bsolo*, a state of the art PB solver.

Preliminary results concerning the application of machine learning algorithms to the selection of the most appropriate learning scheme for each instance given as input are also presented. The goal is to build an "instance aware" PB solver that takes advantage of all different learning schemes.

Finally, several issues concerning the implementation of an efficient PB solver are also discussed and the corresponding experimental results are presented and analysed.

Keywords

Pseudo-Boolean Optimization, Propositional Satisfiability, Non-chronological Backtracking, Conflict-based Learning, Cutting Planes, Cardinality Constraint Reductions.

Acknowledgements

My first and most heartfelt *thank you* goes to Prof. Vasco Manquinho. I thank him for being so supportive and patient, but most of all for being so committed to helping me during this last year.

A word of gratitude is due to Prof. Inês Lynce for having read a preliminary version of this work.

I must also thank Ana Sofia Graça for all her advices, help with latex and other software tools, without her help I would have surely spent much more time with technical issues.

All my love to my family and to my dearest friend, Maria José, without whom I could not have endured these last five years.

This work was funded by a research grant from Fundação para a Ciência e Tecnologia (FCT).

Contents

List of Figures

List of Tables

List of Acronyms

SAT	Propositional Satisfiability
PB	Pseudo-Boolean
PBO	Pseudo-Boolean Optimization
PB-SAT	Pseudo-Boolean Satisfiability
DPLL	Davis Putnam Longemann Loveland
FIFO	First in First out
LIFO	Last in First out

Chapter 1

Introduction

The Propositional Satisfiability (SAT) algorithms have experimented huge developments in the last two decades due to the introduction of conflict-based learning [22, 33] and watched-literal schemes for Boolean Constraint Propagation (BCP) [23]. As such, they are now applied to solve many real world problems which are encoded into SAT instances (with millions of clauses and tens of thousands of variables). However, there are many problems which cannot be directly (or naturally) encoded into SAT instances, thus requiring a huge number of clauses. Therefore, in these situations, clauses may not be the most efficient type of constraint and other kinds of constraints must be used. Since Pseudo-Boolean (PB) Constraints are much more expressive than clauses [14], they have been successfully used to encode many problems in different research areas such as Logic Synthesis and Verification, Operations Research, Bioinformatics and many more. Additionally, PB functions are increasingly used as objective functions in optimization applications.

The use of an algorithm based on the Davis Putnam Longemann Loveland (DPLL) Procedure for solving the Pseudo-Boolean Optimization (PBO) problem was proposed by P. Barth in 1995 in its seminal paper "A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization" [4]. The enormous progresses of SAT solvers contributed to the development of more powerful pseudo-Boolean solvers which exhibit the same structure of a typical SAT solver. Moreover, PB solvers also benefit from all the knowledge and experience from Integer Linear Programming research. The contributions of SAT solvers to PB solvers, particularly the introduction of conflict-based learning, are the main topic of this thesis and will be explained with detail in the following chapters.

1.1 Organization

This thesis is divided in four main chapters. Chapter 2 presents a self-contained description of a generic satisfiability algorithm. This description is, however, focused on the conflict analysis procedure. In chapter 3, we introduce the Pseudo-Boolean Optimization (PBO) problem and its corresponding deci-

sion problem, the Pseudo-Boolean Satisfiability (PB-SAT) problem. Then, we explain how to extend the conflict analysis and the propagation procedures, already discussed for SAT solvers, to PB solvers. Again, our discussion is focused on the conflict analysis procedure and, as such, four different learning techniques are described with detail. Chapter 4 is focused on implementation-related issues, it describes the main issues and concerns raised when implementing all the learning schemes presented in chapter 3 and explains the adopted solutions. In chapter 5, we present the experimental results corresponding to several versions of the solver, namely the versions which implement each different learning scheme.

This thesis concludes in chapter 6 with an overview of the work already developed and a description of several proposals for future work.

1.2 PB-SAT and PBO Applications

Many problems can be encoded as PBO or PB-SAT instances. This section presents two simple examples in which it will be used a PBO encoding and PB-SAT encoding respectively.

The Progressive Party

A progressive party [30] is a party that is organized during a yacht competition. Since it is a yacht competition, the party takes place on several yachts. Some yachts are selected to be hosts. Naturally, the crew of each host yacht must stay on board to receive the guests. The crews of the remaining boats visit the host boats for T successive time periods during the evening. Note that each guest crew must stay together as a unit. Every boat can only receive a limited number of guests at a time and crew sizes are different. A guest crew cannot revisit a host and guest crews cannot meet more than once. The goal of the party organizer is to minimize the number of host boats.

The Boolean variables used in the problem are the following:

- $\delta_i = 1$ iff boat i corresponds to a host.
- $\gamma_{ikt} = 1$ iff boat k is a guest of boat i in period t.
- $\eta_{klt} = 1$ iff guest boats k and l meet in period t.

For each boat i, c_i represents the number of elements of the crew and K_i represents the total capacity of the boat. Naturally, the objective function is:

$$\sum_i \delta_i$$

This problem requires the following constraints:

- A boat can only be visited if it is a host boat. So, for all i, k, t, such that $i \neq k$:

$$\gamma_{ikt} - \delta_i \leq 0$$

- The capacity of a host boat cannot be exceeded. So, for all i, t:

$$\sum_{k \neq i} c_k \cdot \gamma_{ikt} \leq K_i - c_i$$

- Each crew must always have a host or be a host (but cannot be a host and have a host at the same time). So, for all k, t:

$$\sum_{i \neq k} \gamma_{ikt} + \delta_k = 1$$

- A guest crew cannot visit a host boat more than one. For all i, k, such that $i \neq k$:

$$\sum_{t} \gamma_{ikt} \leq 1$$

- For all i, k, l, t:

$$\gamma_{ikt} + \gamma_{ilt} - \eta_{klt} \leq 1$$

- Every pair of guest crews cannotmeet more than once. For all k, l:

$$\sum_{t} \eta_{klt} \leq 1$$

With B boats and T time periods, the problem has $O(BT)$ variables and $O(BT)$ constraints.

A Round-robin tournament

In a round-robin tournament [24], the problem scenario is again a sports tournament in which participate n teams and which lasts $n-1$ weeks. Each week is divided into $n/2$ periods. The tournament must satisfy the following three constraints:

- Every team plays exactly once a week.
- Every team plays at most twice in the same period over the tournament.
- Every team plays against every other team only once.

This problem requires only one type of Boolean variable:

$$v_{wpxy}$$

Such that: $1 \leq w \leq n-1$, $1 \leq p \leq n/2$ and $1 \leq x < y \leq n$. For each variable v_{wpxy}, $v_{wpxy} = 1$ *iff* team x plays against team y in period p of week w. This problem requires the following constraints:

- In each period of each week occurs exactly one game.

  For all $1 \leq w \leq n-1$ and $1 \leq p \leq n/2$:

$$\sum_{x=1}^{n} \left(\sum_{y=x+1}^{n} v_{wpxy} \right) = 1$$

- Each pair of teams play exactly one time during all the tournament. For all $1 \leq x < y \leq n$:

$$\sum_{w=1}^{n-1} \left(\sum_{p=1}^{n/2} v_{wpxy} \right) = 1$$

- Each team plays exactly once per week. For all $1 \leq w \leq n-1$ and $1 \leq x \leq n$:

$$\sum_{p=1}^{n/2} \left(\sum_{y=1}^{x-1} v_{wpxy} + \sum_{y=x+1}^{n} v_{wpxy} \right) = 1$$

- Each team plays at most twice per period. For every $1 \leq x \leq n$ and $1 \leq p \leq n/2$:

$$\sum_{w=1}^{n-1} \left(\sum_{y=1}^{x-1} v_{wpyx} + \sum_{y=x+1}^{n} v_{wpxy} \right) \leq 2$$

Note that as opposed to the previous problem which was encoded as an optimization problem, this problem was encoded as a decision problem.

Chapter 2

The SAT problem

2.1 Preliminaries

Definition 1. A *Boolean variable* is any symbol to which we can assign one of the truth values 0 and 1, also denoted by FALSE and TRUE, respectively.

Definition 2. Let X be a countable set of Boolean variables. The class of the *Boolean formulas* over X is the smallest class which is inductively defined as follows:

- The Boolean constants **0** and **1** are Boolean formulas.
- Every Boolean variable x in X is a Boolean formula.
- If φ and ψ are Boolean formulas then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ and $(\neg\varphi)$ are Boolean formulas.

Definition 3. A *literal* is a Boolean formula which is either a Boolean variable, or the complement of a Boolean variable.

Definition 4. A *clause* is a disjunction of literals.

Definition 5. A Boolean formula φ is said to be in *conjunctive normal form* (CNF) if it is a conjunction of clauses.

Definition 6. A *Boolean assignment* is a mapping $A_{X'} : X' \to \{0, 1\}$, where $X' \subseteq X$. If $X = X'$ then we say that the assignment $A_{X'}$ is *complete*, otherwise $A_{X'}$ is said to be a *partial* assignment.

Remark 1. Note that we have used **0** and **1** to denote the syntactic constants and 0 and 1 to denote their semantic interpretation.
Usually we represent an assignment as a set of pairs. For instance:

$$A_{X'} = \{(x_1, 0), \ldots, (x_n, 1)\}$$

Definition 7. Given a Boolean formula φ, we can obtain a corresponding *simplified Boolean formula* applying the following procedure recursively:

- Delete all clauses in which occur the Boolean constant **1**.
- Remove all occurrences of the Boolean constant **0** from the clauses in which they occur.
- If the obtained formula is different from the one considered in the beginning of the procedure, then another recursive step must be applied.

Definition 8. Given a Boolean formula φ and an assignment $A_{X'}$, the restricted formula $\varphi[A_{X'}]$ is the simplified Boolean formula obtained from φ replacing the variables for which $A_{X'}$ is defined with their assigned values. Note that an assignment is a mapping from the set of variables to the set $\{0, 1\}$, each element of this set corresponds to the semantic interpretation of each of the Boolean constants. Therefore, when we say that each variable in the formula is replaced with its assigned value, we are talking about the corresponding Boolean constant (not its semantic interpretation).

Definition 9. Given a Boolean formula φ and a complete assignment A_X, the value of φ under A_X denoted as $v(\varphi[A_X])$ is the Boolean value inductively defined as follows:

1. $v(\varphi[A_X]) = 0$ if $\varphi = \mathbf{0}$.
2. $v(\varphi[A_X]) = 1$ if $\varphi = \mathbf{1}$.
3. $v(\varphi[A_X]) = A_X(x)$ if $\varphi = x$ for $x \in X$.
4. If $\varphi = (\psi \vee \delta)$, then $v(\varphi[A_X]) = 1$ if $v(\psi[A_X]) = 1$ **or** $v(\delta[A_X]) = 1$.
5. If $\varphi = (\psi \wedge \delta)$, then $v(\varphi[A_X]) = 1$ if $v(\psi[A_X]) = 1$ **and** $v(\delta[A_X]) = 1$.

Remark 2. Note that given a complete assignment A_X and a Boolean formula φ, the restricted Boolean formula $\varphi[A_X]$ will correspond to one of the Boolean constants **0** or **1**. As such, $v(\varphi[A_X])$ can also be defined in the following manner:

1. If $\varphi[A_X] = \mathbf{1}$, $v(\varphi[A_X]) = 1$
2. If $\varphi[A_X] = \mathbf{0}$, $v(\varphi[A_X]) = 0$

Definition 10. Given a formula φ and a partial assignment $A_{X'}$, $v(\varphi[A_{X'}])$ is defined as follows:

- If $\varphi[A_{X'}] = \mathbf{1}$, $v(\varphi[A_{X'}]) = 1$
- If $\varphi[A_{X'}] = \mathbf{0}$, $v(\varphi[A_{X'}]) = 0$
- If none of the equalities presented above hold, $v(\varphi[A_{X'}])$ is said to be undefined.

Definition 11. Given a Boolean formula φ and an assignment (complete or partial) $A_{X'}$, there are three possible situations:

- $v(\varphi[A_{X'}]) = 0$ in which case we say that φ is *unsatisfied* under the assignment $A_{X'}$.
- $v(\varphi[A_{X'}]) = 1$ in which case we say that φ is *satisfied* under the assignment $A_{X'}$.
- $v(\varphi[A_{X'}])$ is not defined, in which case we say that φ is *unresolved* under the assignment $A_{X'}$. This situation can only happen if $A_{X'}$ is a partial assignment.

Definition 12. Given a Boolean formula φ, we say that φ is *satisfiable* if there is a complete assignment A_X which satisfies φ. If φ is satisfied under every truth assignment, we say that φ is a *tautology*.

Definition 13. The *Propositional Satisfiability Problem* (SAT problem) asks whether a given Boolean formula φ is satisfiable. In formal language terms:

$$\text{SAT} = \{\langle\phi\rangle : \phi \quad \text{is a satisfiable Boolean formula}\}$$

Where $\langle\phi\rangle$ represents the encoding of ϕ.

Definition 14.

$$\text{CNF-SAT} = \{\langle\phi\rangle : \phi \quad \text{is a satisfiable Boolean formula in conjunctive normal form}\}$$

Proposition 1. For each Boolean formula φ there exists another Boolean formula ψ in CNF which can be computed from φ in polynomial time and such that φ is satisfiable if and only if ψ is satisfiable. In other words, SAT$\leq_p$CNF-SAT [8].

This proposition states that given an instance of the SAT problem, we can compute in polynomial time an instance of the CNF-SAT problem such that the former formula is satisfiable if and only if the first formula is satisfiable. Therefore, CNF-SAT solvers are indeed SAT solvers. The algorithm presented later in this chapter will receive as an argument a Boolean formula in conjunctive normal form. So, from now on, we will use the terms formula and CNF-formula interchangeably. In this context, it is natural to think of a formula φ as a set of clauses.

Proposition 2 (Cook's Theorem). CNF-SAT is NP-complete [8].

2.2 A Generic Satisfiability Algorithm

Since SAT is known to be NP-Complete, it is unlikely that there exists any SAT algorithm with polynomial time complexity. Nevertheless, SAT instances that encode real world problems seem to have some structure that enables efficient solution [33]. In this section we will present a generic satisfiability algorithm based on the GRASP algorithm [22].

Given a CNF formula φ, an algorithm for the satisfiability problem must decide if φ is satisfiable and, if it is, provide an assignment that satisfies φ. Using this assignment, a satisfiability checker can verify in polynomial time that φ is indeed satisfiable.

Most of the successful SAT solvers [22, 23, 13] are based on the Davis Putnam Longemann Loveland (DPLL) algorithm. In addition to DPLL, these algorithms use a pruning technique based on the recording of nogood clauses and perform non-chronological backtracking. The DPLL algorithm embodies three usefull rules [27] which are also used by modern SAT solvers:

1. **Early Termination:** Given a partial assignment $A_{X'}$, we can detect whether a given clause w is satisfied under all the assignments that contain $A_{X'}$, testing if w contains a true literal. Hence, the formula as a whole can be found satisfiable even before a complete assignment is found.
   Similarly, given a partial assignment $A_{X'}$, we can detect whether a given clause w is unsatisfied under all the assignments that contain $A_{X'}$, testing if all literals in w are assigned to 0. If a clause is found unsatisfied under a partial assignment $A_{X'}$, we can infer that the corresponding formula is unsatisfied under all the assignments which include $A_{X'}$.

2. **Pure Literal Rule:** A pure literal is a literal that always appears with the same sign in all clauses. It is easy to see that if a formula has a satisfying assignment, then it has a satisfying assignment that assigns all the pure literals to 1.

3. **Unit Clause Rule:** Given a partial assignment $A_{X'}$, a clause w is said to be unit under that assignment if it is unresolved and the number of its unassigned literals (which are commonly referred to as *free literals*) is one. Given a unit clause $w = (l_1 \vee l_2 \vee \cdots \vee l_j \vee \cdots \vee l_k)$ and a partial assignment $A_{X'}$, such that l_j is the only free literal, l_j must be assigned to 1 in order to guarantee that w is not unsatisfied under $A_{X'}$. This procedure is called the unit clause rule. The iterated application of the unit clause rule to a given formula until the set of unit clauses becomes empty or more clauses become unsatisfied is called *Boolean Constraint Propagation* (BCP).

It was already stated that given a CNF formula φ, an algorithm for the satisfiability problem must decide if there exists an assignment A_X such that $\varphi[A_X] = \mathbf{1}$, denoted as a *satisfying assignment*. Starting from an empty truth assignment, a backtrack algorithm examines the space of truth assignments in order to find a satisfying assignment. It organizes the search for a satisfying assignment by implicitly maintaining a *decision tree*. Each node in the decision tree specifies an elective assignment for an unassigned variable called a *decision assignment*. A *decision level* is associated with each decision assignment to denote its depth in the decision tree. The first decision assignment is associated with decision level 1.

In each iteration of the search process the following steps are followed:

1. Extend the current partial assignment by making a *decision assignment*.

2. Extend the current partial assignment performing BCP. The assignments made in this step are referred to as *implied assignments*. The deduction process may lead to the identification of one or more unsatisfied clauses, which imply that the current assignment is not a satisfying one. Such occurrence is called a *conflict* and the associated unsatisfying assignment is called a *conflicting assign-*

ment, the clauses which are unsatisfied under the current assignment are referred to as *conflicting clauses*.

3. If the current assignment is a conflicting assignment another one must be tried. Therefore, the current one must be undone. The backtracking mechanism enables the algorithm to retreat from regions of the search space that do not correspond to satisfying assignments.

Henceforth, the following notation will be used:

1. $\delta(x)$ denotes the level at which x is assigned and $v(x)$ denotes the value assigned to x.
2. $x = v(x)@\delta(x)$ specifies that the value $v(x)$ is assigned to x at decision level $\delta(x)$.

Algorithm 1 follows the steps previously identified. The procedures that are invoked in this algorithm

Algorithm 1 Generic Algorithm for the Satisfiability Problem

```
if preprocess()=TRUE then
  return CONFLICT;
end if
while TRUE do
  if decide() then
    while deduce()=CONFLICT do
      blevel ⇐ analyseConflict()
      if blevel ≤ 0 then
        return UNSATISFIABLE;
      else
        backtrack(blevel);
      end if
    end while
  else
    return SATISFIABLE;
  end if
end while
```

will be explained in great detail in the following sections. However some brief clarifications are stated bellow:

1. preprocess() performs several changes on the formula received as input to make the search easier. For instance, it can apply the pure literal rule.
2. decide() makes a decision assignment.
3. deduce() corresponds to the application of BCP.
4. analyseConflict() is invoked when a conflicting assignment occurs. It analyses the conflict and returns a decision level (blevel), prior to the current level, at which the current conflict is not verified. If such level cannot be found, analyseConflict() returns 0.
5. backtrack() just performs the necessary updates so that the algorithm can backtrack to the decision level identified by analyseConflict() and proceed the search process.

2.3 Choosing a Decision Assignment

In each step of the search, the algorithm must choose a decision assignment. There are several strategies that can be followed to make such a choice:

1. Select randomly an unassigned literal (this heuristic is commonly denoted as *RAND*).
2. Select the literal which appears most frequently in unresolved clauses, this strategy is called the Dynamic Largest Individual Sum, *DLIS* [22].
3. Choose the literal that directly satisfies the largest number of clauses.
4. Choose the literal which simplifies the largest number of clauses and can lead to more implications during BCP.

But how can the effectiveness of these decision heuristics be evaluated? Given two decision heuristics we can say that one is better than the other if, in general, given a SAT instance the algorithm using the first one makes less decision assignments than using the former one. Fewer decisions ought to mean smarter decisions were made. Nevertheless, not all decisions yield an equal number of BCP operations and as such a shorter sequence of decision assignments may lead to more BCP operations than a longer one. Additionally, when choosing a decision heuristic we must also consider its overhead which cannot be very heavy.

The decision heuristic used by Chaff [23], the Variable State Independent Decaying Sum heuristic (*VSIDS*) was the first heuristic suited for lazy data structures which will be introduced in section 2.4. As such, VSIDS is briefly described bellow:

1. Each variable is associated with a counter representing its activity.
2. When a clause is added to the formula (which only happens when a conflict occurs) the activity of each one of its literals is incremented.
3. The literal with the highest activity is selected.
4. Ties are broken randomly.
5. Periodically, the activity of each variable in the system is multiplied by a constant smaller than 1. Therefore, this strategy guarantees that recent increments count more than old ones.

In order to choose the literal with highest activity more quickly, the algorithm must maintain a priority queue in which the literals are ordered by their activity.

One of the main advantages of learning lies in the fact that the learned clauses drive the search. Therefore, using this strategy, the algorithm tries to satisfy the conflict clauses, particularly the more recent ones so as to explore their implications more quickly.

This heuristic helps the algorithm solve the more difficult instances and since it has a very low overhead, it does not compromise its performance on the easier ones.

2.4 The Deduction Procedure

When a decision assignment is made, the algorithm must perform BCP, that is, it must apply the unit clause rule recursively. However, checking all clauses each time a decision assignment is made to determine if one of them becomes unit is extremely inefficient. Zhang et al [23] suggested a different method. Given a clause w, they noted that if we pick any two literals not assigned to 0 in w, we can guarantee that w is not unit[1] until one of those two literals (the *watched literals*) is set to 0. As such, when assigning a literal to 0[2], the deduction procedure only needs to analyse the clauses in which it is being watched. Therefore, for each literal the algorithm keeps the list of clauses in which it is being watched, the *watcher list*. These are the clauses which may become unit if the literal is set to 0.

When the deduction procedure analyses a clause after one of its watched literals is set to 0, one of the following situations may hold:

1. All literals except for the other watched literal are set to 0 and the remaining watched literal is unassigned. The clause is unit and therefore the remaining unassigned literal is immediately implied.

2. There is a non-false non-watched literal. This literal is chosen to replace the one just assigned to 0.

3. The other watched literal is set to 1. Do nothing.

Figure 2.4 illustrates a sequence of updates of the watched literals of a given clause during a certain search process.

2.5 Conflict Analysis

2.5.1 Implication Graph

Let the assignment of a variable x be implied due to a clause $w = (l_1 \vee \cdots l_k)$, which is referred to as the *antecedent clause* [22] of x. The *antecedent assignment* of x, denoted as $A^w(x)$, is defined as the set of assignments to variables other than x with literals in w. Informally we can say that $A^w(x)$ denotes the set of assignments which imply the assignment of x. Naturally, the antecedent assignment of a decision variable is empty. A variable assignment can be expressed as the literal which evaluates to 1 under that assignment. As such, the antecedent assignment of a variable can be expressed as a set of literals.

The decision level of an implied variable is related to the decision level of the variables in $A^w(x)$ in the following way:

$$\delta(x) = \max\{\delta(y) \mid (y, v(y)) \in A^w(x)\}$$

[1] Recall that that w is said to be unit if all but one of its literals are set to 0.
[2] Note that when we assign a literal to 1, we are assigning its complement to 0.

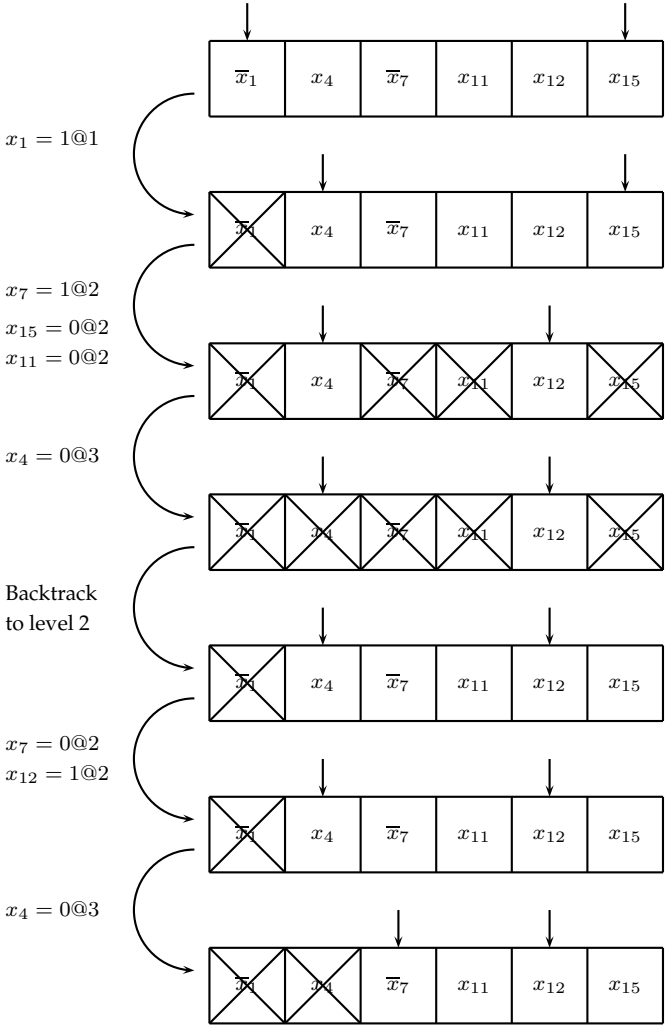

Figure 2.1: A sequence of updates of the watched literals of a given clause during a certain search process

The implication relationships of variable assignments during the SAT solving process can be expressed as an *implication graph* [22]. An implication graph I is defined as follows:

- Each vertex in I corresponds to a variable assignment $x = v(x) @\delta(x)$ or to a conflict.
- The predecessors of the vertex $x = v(x) @\delta(x)$ are the assignments in $A^w(x)$. The directed edges from the vertices in $A^w(x)$ to the vertex $x = v(x) @\delta(x)$ are labeled with w. Hence, vertices with no predecessors correspond to decision assignments.
- Each time a conflict occurs a special vertex is added to I (the letter k is generally used as an identifier of this vertex). If a clause w becomes unsatisfied under the current assignment, in which case w will be referred to as the *conflicting clause* [22], $A^w(k)$ denotes the set of assignments to variables in w. One could say that these are the assignments responsible for the conflict k. Hence, the predecessors of k in I are the vertices in $A^w(k)$. Again, the edges from the vertices in $A^w(k)$ to k are labelled with w.

In actual implementations, the implication graph is maintained by associating each assigned non-decision variable with a pointer to its antecedent clause. By following the antecedent pointers, the implication

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2, \cdots\}$

Current Decision Assignment: $\{x_1 = 1@6\}$

Constraints:

$\omega_1 : \bar{x}_1 \vee x_2$

$\omega_2 : \bar{x}_1 \vee x_3 \vee x_9$

$\omega_3 : \bar{x}_2 \vee \bar{x}_3 \vee x_4$

$\omega_4 : \bar{x}_4 \vee x_5 \vee x_{10}$

$\omega_5 : \bar{x}_4 \vee x_6 \vee x_{11}$

$\omega_6 : \bar{x}_5 \vee \bar{x}_6$

$\omega_7 : x_1 \vee x_7 \vee \bar{x}_{12}$

$\omega_8 : x_1 \vee x_8$

$\omega_9 : \bar{x}_7 \vee \bar{x}_8 \vee \bar{x}_{13}$

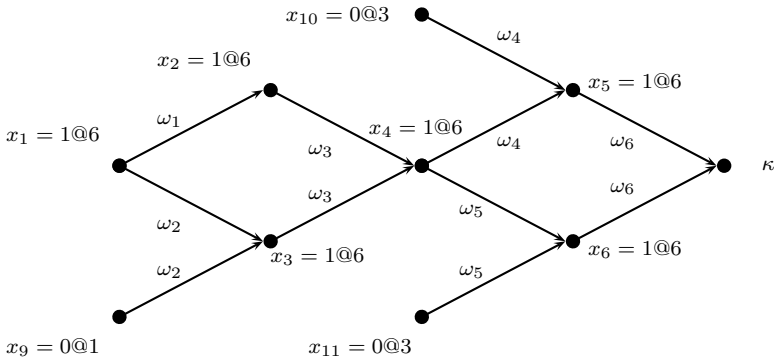

Figure 2.2: Example of a formula, an assignment and the corresponding implication graph

graph can be constructed when needed.

In an implication graph, we say that a vertex a *dominates* a vertex b iff any path from the vertex corresponding to the decision assignment at the decision level of vertex a to vertex b needs to go through a. A vertex a at decision level dl is said to be a *Unique Implication Point* (UIP) [22] if it dominates the conflict vertex, which means that any path from the vertex corresponding to the decision assignment at decision level dl to the conflict vertex needs to go through a. Intuitively, we can say that an UIP at a certain decision level is the only assignment at that decision level that implies the conflict. The UIPs of each decision level are ordered starting from the conflict. Therefore, the first UIP of the current decision level refers to the UIP at the current decision level which is closest to the conflict vertex.

In figure 2.2, besides x_1, the only UIP at level 6 is x_4. Observe that every path from x_1 to k must go through x_4. As such, x_4 is the first UIP at decision level 6 and x_1 is the second.

2.5.2 Learning

When a conflict occurs, the structure of the implication graph (that is, its connected component which contains the conflict vertex) is analysed to determine those variable assignments which are responsible for the conflict. The conjunction of those variable assignments is a sufficient condition for the conflict to arise. If we take the complement of this new formula we will obtain a clause that is consistent with the formula and not satisfied under the current assignment. This clause, the *conflict clause* [22], and its corresponding assignments are denoted by $w_C\,(k)$ and $A^{w_C}\,(k)$ respectively. Using the conflict clause, the algorithm identifies a decision level to which it can safely backtrack. Then, the conflict clause is added to the formula, this process is called *learning* [22]. Therefore, conflict analysis is the procedure that finds a reason for a conflict and tries to resolve it. It tells the SAT solver that there exists no solution for the problem in a certain search space, and indicates a new search space to continue the search.

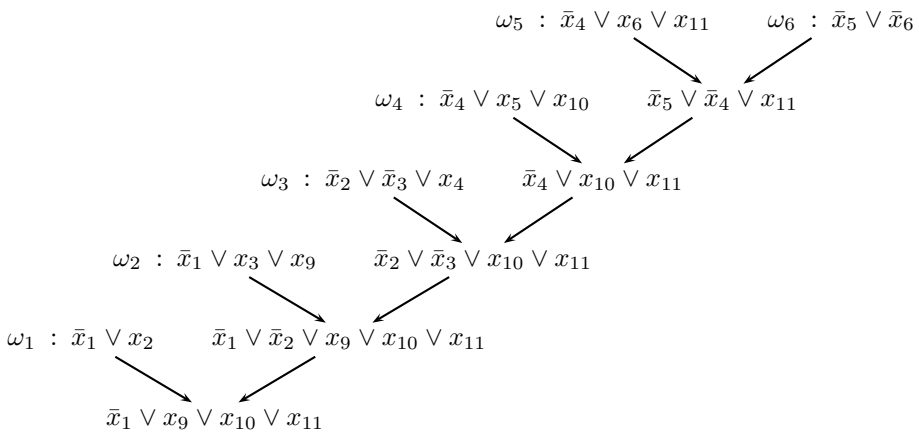

Figure 2.3: A sequence of resolution steps starting from the conflict vertex of the implication graph presented in figure 2.2

A new clause is said to be *consistent* [22] with a given formula if all the assignments that satisfy all the clauses in the formula also satisfy the new clause (note that the opposite is not always true, there can be assignments which satisfy the new clause and do not satisfy all the clauses in the formula).

Consider the vertex k from figure 2.2, its antecedent clause is w_6, if we replace in w_6 one of its literals by its antecedent literals (that is, the other literals which occur in its antecedent clause), we obtain a new clause that is consistent with the current formula. As such, we can replace $\overline{x}_5$ by x_{10} and $\overline{x}_4$, obtaining the clause $(\overline{x}_6 \vee x_{10} \vee \overline{x}_4)$. It is important to note that the procedure just described corresponds to performing a *resolution step* on clauses w_6 and w_4. Hence, the correction of this rule follows directly from the correction of resolution [26].

When a conflict occurs, resolution may be used to learn a new clause consistent with the formula which is unsatisfied under the current assignment. Note that to obtain such a clause, the learning procedure must perform a sequence of resolution steps starting from the conflicting clause and following the implication graph. If the learning procedure starts from an intermediate vertex and not from the conflict, it will not obtain a conflict clause. For instance, in figure 2.2, consider the antecedent clause of x_4, w_3, after replacing all the occurrences of x_2 and x_3 in w_3 by their antecedent literals, x_1 and x_9, the following clause is obtained: $(\overline{x}_1 \vee x_9 \vee x_4)$. Figure 2.3 presents a sequence of resolution steps starting from the conflict vertex of the implication graph introduced in figure 2.2.

However, when applying a sequence of resolution steps it is necessary to know where to stop. Therefore, a bipartition of the implication graph must be established [33]. This bipartition has all the decision assignments in one side (the *reason side*) and the conflict vertex on the other side (the *conflict side*). Such a bipartition is called a *cut*. When applying resolution, we only replace literals corresponding to assignments that occur on the conflict side (resolution steps are only used to eliminate literals belonging to the conflict side). As such, choosing the cut is the main feature which distinguishes different learning schemes.

The conflict clause also helps the algorithm finding the level to which it can safely backtrack. There-

fore, the learning procedure must use a cut such that the learned clause has only one literal assigned at the current decision level. When the algorithm backtracks, the learned clause becomes unit and this literal is forced to flip. Such a clause that causes a flip of the variable is called an *asserting clause*. The learning procedure must use a cut such that there is only one vertex on the reason side at the current decision level that has edges to the conflict side. In other words, any admissible cut must put on the reason side an UIP of the current decision level and all literals assigned at the current decision level before the chosen UIP, all the other literals assigned at the current decision level (the ones which were assigned after the UIP) must be put on the conflict side [33]. Thus, after backtracking, the UIP vertex becomes unit and makes the clause an asserting clause.

There are several different types of cuts which enable the learning procedure to obtain an assertive clause. Bellow we describe very briefly some of them:

- *Last UIP Cut:* All vertices corresponding to assignments made at the current decision level are put on the conflict side, except for the decision assignment which is put on the reason side. The decision assignment and all the assignments made in previous decision levels are put on the reason side.
- *Decision Cut:* All decision assignments are put on the reason side, everything else is put on the conflict side.
- *1UIP Cut:* Every variable implied after the first UIP of the current decision level (the one which is closest to the conflict) is put on the conflict side. The others are put on the reason side.
- *2UIP Cut:* The current decision level and the previous decision level have their first UIPs just before the partition.
- *3UIP Cut:* It is what we expect it to be.
- *All UIPs Cut:* Every decision level has its first UIP just before the partition.
- *Min Cut:* It can also be beneficial to make the conflict clause as short as possible. To learn a clause as short as possible the learning procedure must use a cut such that the number of edges which cross it is minimum. This problem can be approached using a typical max-flow/min-cut algorithm.

It is very important to note that the algorithm cannot learn everything, since the learned clauses will slow down the deduction procedure. It must learn as much as it can as long as the performance of the deduction procedure is not very heavily compromised. Learning must be selective [33].

Generally speaking, a short clause contains more information than a longer one. So, we can say that it is better to learn shorter clauses than longer ones. However, the effectiveness of certain learning schemes can only be determined through empirical data. Zhang et al [33] suggested that the 1UIP learning scheme is the most effective one when learning just a single clause.

2.5.3 Backtracking

It remains yet to clarify how can one use the conflict clause to determine the level to which the algorithm must backtrack. As explained in the previous section after identifying the conflict (and the conflict clause) the algorithm must erase all the assignments made at the current decision level (including the decision assignment). It was also noted that the algorithm must use a learning scheme, such that after erasing all the assignments made at the current decision level, the learned clause becomes unit and forces the unit literal to flip. Modern SAT solvers backtrack to the highest decision level lower than the current one, such that the conflict clause has at least one literal assigned at that level [22]. For instance, if the algorithm uses a 1UIP learning scheme in figure 2.2, the conflict clause is $(x_{10} \vee \overline{x}_4 \vee x_{11})$. So, the algorithm backtracks to decision level 3.

Let β denote the *backtrack decision level*. If $\beta = d - 1$, where d is the current decision level, the search process backtracks *chronologically* to the immediately preceding decision level. However, if $\beta < d - 1$, the search process backtracks *nonchronologically* by jumping back over several levels in the decision tree.

2.5.4 Clause Deletion

Adding clauses to the formula has two major drawbacks:

1. It slows down the deduction process.

2. The size of the formula grows with the number of backtracks. In the worst case, such a growth can be exponential in the number of variables.

The first issue is not very worrying, since it was empirically verified [22] that for some classes of instances the advantages of learning surpass by far its drawbacks as far as the performance of the algorithm is concerned. However, when solving large SAT instances, the solver may run out of memory. So periodically the algorithm must remove a few learned clauses.

The formula is divided in two parts: the problem clauses and the learned clauses. Generally, the problem clauses cannot be removed. However, they can under specific circumstances, for instance subsumption [32].

To choose what clauses to remove, the algorithm may apply one of the following strategies:

1. Assume that an integer parameter k is given. Conflict clauses whose size (number of literals) is no greater than k are marked green and handled normally. Conflict clauses of size greater than k are marked red and kept around while they are satisfied, unsatisfied or unit clauses.

2. When a clause is added, it is examined in order to estimate its power to generate new implications. Shorter clauses convey more information that longer ones. Similarly, if a clause has very few unassigned literals, it will probably be used in the search before a clause that has a large number of unassigned literals. So, the algorithm must choose a number n (typically between 100 and 200)

such that when the number of unassigned literals in a given clause is greater than n, that clause is deleted.

3. MINISAT [13] uses for clauses a heuristic very similar to the one used by Chaff in the decision procedure, VSIDS. Each time a learned clause is used to imply an assignment, its activity is incremented. Over time the activities of all clauses are divided by a constant. When the formula gets too big, the algorithm deletes the clauses which have the lowest activity.

2.6 Restarts

Modern solvers use *restarts* [15, 23, 13, 22] to escape difficult regions of the search tree. A restart corresponds to a halt in the search process and a restart of the analysis maintaining at least some of the learned clauses. As such, when the algorithm restarts the search, it does not simply repeat what was done before, because the learned clauses will drive the search in a different direction. Moreover, some randomness can be added to the decision procedure so as to guarantee that each time the solver restarts it follows a different path. Almost all recent SAT solvers use restarts. A simple restart policy consists in establishing a fixed limit k such that after every k conflicts the solver restarts.

Using restarts only provides a modest improvement in typical performance, but substantially improves robustness. That is, they significantly reduce the variability in running time found over collections of similar instances [17]. To ensure completeness, solvers increment the restart interval each time the solver restarts. Hence, the solver will eventually perform a complete search with no restart.

Chapter 3

Pseudo-Boolean Satisfiability and Optimization

3.1 The Problem

3.1.1 Basic Definitions

A *pseudo-Boolean (PB) function* is a function that maps n Boolean variables to a real number (for any positive integer n). Naturally, an *integer pseudo-Boolean function* maps n Boolean variables to an integer. A *linear Pseudo-Boolean constraint* (commonly denoted as *PB constraint* or *LPB constraint*) over a set of Boolean variables $X = \{x_1, \cdots, x_n\}$ is an inequality that has the following form:

$$\sum_{j=1}^{n} a_j \cdot l_j \triangleright b$$

such that for each $j \in \{1, \cdots, n\}$, a_j is an integer coefficient and l_j is a literal, b is an integer coefficient and $\triangleright$ is one of the common relational operators ($=, \geq, \leq, > and <$). The right side of the constraint is denoted as the *degree* of the constraint[1]. The addition operator and the other relational operators have their usual arithmetic meaning.

Given a set of PB constraints W over a set of Boolean variables X, we say that W is satisfiable if there is an assignment A_X, such that all constraints in W are satisfied under A_X.

Definition 15. The PB-SAT problem asks whether a given set of PB constraints is satisfiable. In formal language:

$$\text{PB-SAT} = \{\langle W \rangle : W \quad \text{is a satisfiable set of PB constraints}\}$$

Where $\langle W \rangle$ denotes the encoding of W.

[1] It is also commonly referred to as rhs.

Definition 16. The PBO Problem (the Pseudo-Boolean Optimization problem) consists in finding a satisfying assignment to a set of PB constraints that minimizes a given pseudo-Boolean objective function.

3.1.2 Constraints

A linear PB constraint is said to be in normal form when expressed as:

$$\sum_{i=1}^{n} a_i \cdot l_i \geq b$$

such that for each $i \in \{1, \cdots, n\}$, $a_i \in Z^+$ and l_i is a literal and $b \in Z^+$. Any PB constraint w can be converted into the normal form applying the rules specified bellow iteratively:

1. If the relational operator of w is "=", w is replaced by two new constraints w_1 and w_2 which are equal to w except for the relational operator that in one case corresponds to $\leq$ and in the other to $\geq$.

2. If the relational operator of w is ">", we replace ">" by "$\geq$" and increment the right hand side (rhs) in one unity.

3. If the relational operator of w is "<", we replace "<" by "$\leq$" and decrement the right hand side (rhs) in one unity.

4. Every literal l_i in w associated with a negative coefficient is replaced by $\left(1 - \overline{l}_i\right)$[2].

5. If none of the operations presented above can be applied and the rhs of the inequality is negative, then w is always satisfied and therefore can be removed from the formula.

6. If the relational operator of w is "$\leq$" we must multiply both members of the inequality by -1.

This procedure allows the mapping in linear time of all pseudo-Boolean constraints into the normalized formulation. As such, from now on, when talking about a pseudo-Boolean constraint we will always mean a linear pseudo-Boolean constraint in the normal form.

Given a partial assignment $A_{X'}$ and a constraint $w = \sum_i a_i \cdot l_i \geq b$, w is said to be:

- *satisfied* if $\sum_{l_i=1} a_i \geq b$;
- *unsatisfied* if $\sum_{l_i \neq 0} a_i < b$;
- *unresolved* if $\sum_{l_i \neq 0} a_i \geq b$.

[2]Note that for any literal l_i, $l_i = \left(1 - \overline{l}_i\right)$.

Cardinality Constraints

A *cardinality constraint* is a constraint on the number of literals which are true among a given set of literals. There are three kinds of cardinality constraints:

- The constraint $atleast\,(k, \{l_1, \cdots, l_n\})$ requires that *at least* k literals among $\{l_1, \cdots, l_n\}$ be true. This constraint corresponds to the Pseudo-Boolean constraint: $\sum_i l_i \geq k$.
- The constraint $atmost\,(k, \{l_1, \cdots, l_n\})$ requires that *at most* k literals among $\{l_1, \cdots, l_n\}$ be true, which corresponds to Pseudo-Boolean constraint: $\sum_i l_i \leq k$.
- The constraint $exactly\,(k, \{l_1, \cdots, l_n\})$ requires that *exactly* k literals among $\{l_1, \cdots, l_n\}$ be true, which corresponds to Pseudo-Boolean constraint: $\sum_i l_i = k$.

It can easily be proved that all three types of cardinality constraints can be expressed as an *atleast* constraint. Note that:

$$atmost\,(k, \{l_1, \cdots, l_n\}) \equiv atleast\,\left(n-k, \{\bar{l}_1, \cdots, \bar{l}_n\}\right)$$
$$exactly\,(k, \{l_1, \cdots, l_n\}) \equiv atleast\,(k, \{l_1, \cdots, l_n\}) \wedge atmost\,(k, \{l_1, \cdots, l_n\})$$

Therefore, any cardinality constraint can be expressed as a pseudo-Boolean constraint in the normal form.

Conversely a Pseudo-Boolean constraint w in the normal form such that all its coefficients are equal is actually a cardinality constraint. For instance, let w be $\sum_i a_0 \cdot l_i \geq b$, then w is equivalent to the following constraint:

$$\sum_i l_i \geq k$$

where $k = \lceil \frac{b}{a_0} \rceil$. Hence, this constraint is equivalent to the cardinality constraint $atleast\left(\lceil \frac{b}{a_0} \rceil, \{l_1, \cdots, l_n\}\right)$. As such, a cardinality constraint is a special kind of pseudo-Boolean constraint.

It can be showed that a general PB constraint can be encoded as a conjunction of cardinality constraints [14]. Similarly a cardinality constraint can be encoded as a conjunction of clauses. In both cases it can also be showed that the size of the encoding can be exponential in the number of variables. Thus a cardinality constraint encoding can be exponentially smaller than a clause encoding and a general PB encoding can be exponentially smaller than a cardinality constraint encoding.

Clauses

The clause $l_1 \vee l_2 \vee \cdots \vee l_n$ can be express as the cardinality constraint $atleast\,(1, \{l_1, \cdots, l_n\})$. Indeed, every clause can be expressed as a cardinality constraint.

3.1.3 Inference Rules

One of the most important features of modern PB-SAT solvers is the ability to derive a new PB constraint from a set of PB constraints.

An *Inference Rule* is a procedure which allows us to infer a new constraint α from a finite set of constraints Δ (with a specific structure). If for any proper set of premises Δ the inference rule produces a conclusion α such that all the assignments which satisfy Δ also satisfy α we say that the inference rule is *sound*. Additionally, if the set of assignments which satisfy α is equal to the set of assignments which satisfy Δ we say that the inference rule does not convey any *loss of information*.

Using inference rules allows adding new constraints to the formula. These learned constraints will help the algorithm identify future conflicts more quickly and therefore they help pruning the search space. If there are assignments that satisfy the premises but do not satisfy the conclusion, adding the conclusion to the formula will possibly change the solution set. As such, the soundness of every inference must be proved before using it in the search process.

Let x denote any given literal, the following inference rules can be stated:

- **Negation:**

$$\frac{}{\overline{x} = 1 - x}$$

- **Idempotence:**

$$\frac{}{x \cdot x = x}$$

- **Bounds:**

$$\frac{}{\begin{array}{c} x \geq 0 \\ -x \geq -1 \end{array}}$$

The correction of these first three rules follows directly from the definition of literal. Note that the negation rule was used when the procedure for clause normalization was introduced.

The following two rules are extensively used in normal integer arithmetic.

- **Addition:**

$$\frac{\begin{array}{c} \sum_i a_i \cdot l_i \geq b \\ \sum_i c_i \cdot l_i \geq d \end{array}}{\sum_i (a_i + c_i) \cdot l_i \geq (b + d)}$$

- **Multiplication:**

$$\frac{\begin{array}{c} \sum_i a_i \cdot l_i \geq b \\ \alpha > 0 \\ \alpha \in N \end{array}}{\sum_i \alpha \cdot a_i \cdot l_i \geq \alpha \cdot b}$$

Consider a constraint similar to a pseudo-Boolean constraint but such that its coefficients can be real numbers, $\sum r_i \cdot l_i \geq b$. Firstly, it is important to note that $\forall r_i \in R, \quad \sum_i \lceil r_i \rceil \geq \lceil \sum_i r_i \rceil$, from which the following inequalities can be directly inferred:

$$\sum_i \lceil r_i \cdot l_i \rceil \geq \left\lceil \sum_i r_i \cdot l_i \right\rceil \geq \lceil b \rceil$$

If we also note that given a real number r_i and a literal l_i, $\lceil r_i \rceil \cdot l_i = \lceil r_i \cdot l_i \rceil$, the following rule can be easily obtained:

- **Rounding:**

$$\frac{\sum_i r_i \cdot l_i \geq b}{\sum_i \lceil r_i \rceil \cdot l_i \geq \lceil b \rceil}$$

- **Division:**

$$\frac{\begin{array}{c}\sum_i a_i \cdot l_i \geq b \\ \alpha > 0 \\ \alpha \in N\end{array}}{\sum_i \lceil \frac{a_i}{\alpha} \rceil \cdot l_i \geq \lceil \frac{b}{\alpha} \rceil}$$

- **Multiplication/Division:**

$$\frac{\begin{array}{c}\sum_i a_i \cdot l_i \geq b \\ \alpha > 0\end{array}}{\sum_i \lceil \alpha \cdot a_i \rceil \cdot l_i \geq \lceil \alpha \cdot b \rceil}$$

Example 1. In this example it is illustrated the application of the division rule followed by the rounding rule:

$$\frac{\frac{3x_1 + x_2 + x_3 + x_4 + x_5 \geq 6}{x_1 + \frac{1}{3}x_2 + \frac{1}{3}x_3 + \frac{1}{3}x_4 + \frac{1}{3}x_5 \geq 2}}{x_1 + x_2 + x_3 + x_4 + x_5 \geq 2}$$

The addition and the multiplication rules can be used together as a single rule such that given a set of PB constraints it will allow us to infer any linear combination of this set of constraints. This new rule, which will be referred to as the *Cutting Planes* Rule [7], can be used to eliminate one or more variables that occur in the premises and as such it can be viewed as an extension of resolution from clauses to PB constraints.

- **Cutting Planes:**

$$\frac{\begin{array}{c}\sum_i a_i \cdot l_i \geq b \\ \sum_i c_i \cdot l_i \geq d \\ \alpha > 0 \\ \beta > 0\end{array}}{\sum_i (\alpha \cdot a_i + \beta \cdot c_i) \cdot l_i \geq (\alpha \cdot b + \beta \cdot d)}$$

Generally, when applying this rule, α and β are chosen in order to eliminate one variable which occurs simultaneously in both constraints (in one of them it must occur its corresponding positive literal and in the other its negative one). Suppose $a \cdot x$ appears in the first constraint and $c \cdot \overline{x}$ appears in the second one and we want to eliminate x from both constraints then α and β must be chosen in the following way:

- $\alpha = \frac{lcm(a,c)}{a} = \frac{c}{gcd(a,c)}$
- $\beta = \frac{lcm(a,c)}{c} = \frac{a}{gcd(a,c)}$

Where $lcm\,(a, c)$ and $gcd\,(a, c)$ denote the least common multiple and the greatest common divisor between a and c respectively.[3]

Example 2. This example presents two different applications of the cutting planes rule.

$$\frac{\begin{array}{c}1 \cdot (x_4 + 3x_5 + 2x_3 \geq 3) \\ 2 \cdot (x_1 + x_2 + \overline{x}_3 \geq 2)\end{array}}{2x_1 + 2x_2 + x_4 + 3x_5 \geq 5}$$

$$\frac{\begin{array}{c}1 \cdot (3x_1 + 2\overline{x}_2 + x_3 + x_4 \geq 3) \\ 1 \cdot (\overline{x}_3 + x_4 \geq 1)\end{array}}{3x_1 + 2\overline{x}_2 + 2x_4 \geq 3}$$

- **Saturation:**

$$\frac{\begin{array}{c}\sum_i a_i \cdot l_i \geq b \\ a_j > b\end{array}}{b \cdot l_j + \sum_{i \neq j} a_i \cdot l_i \geq b}$$

Given a PB constraint, if one of its literals has a coefficient greater than the rhs, then if this literal is assigned to 1, the constraint is immediately satisfied. However, if the corresponding coefficient is equal to the rhs, the constraint is also immediately satisfied. As such, all coefficients greater than the rhs can be replaced by the rhs of the constraint.[4]

Example 3. This example illustrates the application of the saturation rule and one application of the Multiplication/Division rule which yields the same result.

$$\frac{3x_1 + x_2 + x_3 \geq 2}{2x_1 + x_2 + x_3 \geq 2}$$

[3] We have used the following identity: $lcm\,(a, c) = \frac{a \cdot c}{gcd(a,c)}$

[4] The saturation rule can be replaced by iterated applications of the multiplication/division rule, with α chosen in such a way that all coefficients greater than the rhs are decremented in at least one unity and all the others are left unchanged. These two conditions are satisfied when $(b-1)/b < \alpha \leq b/(b+1)$.

$$\dfrac{\dfrac{\dfrac{3x_1 + x_2 + x_3 \geq 2}{\frac{2}{3}(3x_1 + x_2 + x_3 \geq 2)}}{2x_1 + \frac{2}{3}x_2 + \frac{2}{3}x_3 \geq \frac{4}{3}}}{2x_1 + x_2 + x_3 \geq 2}$$

It is important to emphasize that the saturation rule corresponds to a sequence of applications of the Multiplication/Division rule, in this example the corresponding sequence has only one element, which does not always happen.

- **Coefficient Reduction:**

$$\dfrac{\begin{array}{c}\sum_i a_i \cdot l_i \geq b \\ a_j > 0\end{array}}{\sum_{i \neq j} a_i \cdot l_i \geq (b - a_j)}$$

- **Partial Coefficient Reduction:**

$$\dfrac{\begin{array}{c}\sum_i a_i \cdot l_i \geq b \\ a_j > 0 \\ a_j > a > 0\end{array}}{(a_j - a) \cdot l_j + \sum_{i \neq j} a_i \cdot l_i \geq (b - a)}$$

Both this two previous rules can be obtained combining the cutting planes rule with the bounds rule.

Example 4.

$$\dfrac{x_1 + x_2 + x_4 + 2x_7 + 2x_8 \geq 5}{x_1 + x_2 + 2x_7 \geq 2}$$

- **Cardinality Constraint Reduction**

$$\dfrac{\begin{array}{c}\sum_{i=1}^{n} a_i \cdot l_i \geq b \\ \sum_{i=1}^{\beta-1} a_i < b \leq \sum_{i=1}^{\beta} a_i\end{array}}{\sum_{i=1}^{n} l_i \geq \beta}$$

In the inference rule stated above it is assumed that literals in a constraint are ordered decreasingly according to the value of their coefficients. As such, a cardinality constraint reduction derives a cardinality constraint from a general PB constraint. It calculates the minimum number of literals that must be assigned to 1 for the constraint to be satisfied.

Example 5.

$$\dfrac{6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 + x_6 \geq 17}{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 4}$$

Chai and Kuehlmann further note [7] that some of the literals with the smallest coefficients may be

safely removed from the derived cardinality constraint. This procedure will be carefully explained later in this chapter.

3.2 A Generic Pseudo-Boolean Satisfiability Algorithm

The huge progresses which SAT solvers have experimented in the last decade contributed decisively to the development of efficient PB solvers which exhibit the same general structure of SAT solvers. Due to its great similarity to the SAT problem, we will only consider the PB-SAT problem in this first approach.

A PB-SAT solver, just like any SAT solver, must search the space of all Boolean assignments. This search is organized in the exact same way that was explained in section 2.2. Therefore, the generic algorithm presented in that section can also be used by any PB-SAT solver. It was already stressed many times that the main features of an efficient SAT solver that uses a DPLL-like algorithm are efficient BCP (which is implemented in the deduce() procedure) and conflict-based learning (which is implemented in the analyseConflict() procedure). Naturally, the deduce and the analyseConflict() procedures must be adapted in order to handle PB constraints. This adaptation is far from obvious and it will be carefully explained in the following sections.

3.3 Extending BCP to Pseudo-Boolean Constraints

Suppose w is a PB constraint:

- $L_T[w] = \{l_i \mid l_i = 1 \text{ and } l_i \text{ occurs in } w\}$
- $L_F[w] = \{l_i \mid l_i = 0 \text{ and } l_i \text{ occurs in } w\}$
- $L_U[w] = \{l_i \mid l_i \text{ is an unassigned literal which occurs in } w\}$
- $S_T[w] = \sum_{l_i \in L_T} a_i$
- $S_F[w] = \sum_{l_i \in L_F} a_i$
- $S_U[w] = \sum_{l_i \in L_U} a_i$
- $a_{\max}[w] = \max\{a_i \mid l_i \text{ occurs in } w\}$
- $l_{\max}[w]$ denotes the literal associated with $a_{\max}[w]$
- $a^U_{\max}[w] = \max\{a_i \mid l_i \in L_U[w]\}$
- $l^U_{\max}[w]$ denotes the literal associated with $a^U_{\max}[w]$

From now on, when talking about the concepts defined above it is assumed that they are associated with a constraint w and as such w is omitted. For instance, instead of using $L_T[w]$, it is used L_T.

Given a partial assignment $A_{X'}$ and a PB constraint w, we define s as the *slack* of w under $A_{X'}$ in the following way: $s = S_T + S_U - b$, where b denotes the rhs of the constraint. It is quite easy to see that, given a partial assignment $A_{X'}$ and a constraint w, w is *unsatisfied* under $A_{X'}$ if its slack is negative, this condition can be formally expressed as follows:

$$S_T + S_U - b < 0$$

$$S_T + S_U < b \tag{3.1}$$

Given an assignment $A_{X'}$, a Pseudo-Boolean constraint is said to be *unit* under $A_{X'}$, if it is unresolved under that assignment and at least one of its literals must be assigned to 1 for the constraint to be satisfied.

It is easy to see that if the slack of a certain constraint is lower than $a^U_{\max}$ the corresponding literal, $l^U_{\max}$, must be immediately implied since assigning it to 0 would make the constraint unsatisfied. We can express this condition as follows:

$$s < a^U_{\max}$$
$$S_T + S_U - b < a^U_{\max}$$
$$S_T + S_U < b + a^U_{\max} \tag{3.2}$$

The fastest known method for BCP in SAT is based on the watch-literal strategy. This strategy exploits the fact that a clause cannot become unit as long as two of its literals remain unassigned. So the algorithm must only watch two literals in each clause. Similarly, a cardinality constraint w with degree b cannot become unit while $(n - (b + 1))$ of its literals are not assigned to 0, where n denotes the number of literals in w. Thus, it is only necessary to watch $b + 1$ literals in w[5].

As opposed to cardinality constraints and clauses in which it is only necessary to watch a **fixed** number of literals, in general PB constraints a variable number of literals must be watched. Moreover, it is important to minimize the number of watched literals in each constraint so as to simplify the propagation. As such, to extend the watch-literal strategy from clauses to general PB constraints, it is necessary to specify a procedure that given a PB constraint w selects a set of literals, say L_W, such that while none of those literals is assigned to 0, w cannot become unit. Given a set of watched literals L_W, the *Watch Sum* (S_W) is defined as the sum of the coefficients of the literals in L_W:

$$S_W = \sum_{l_i \in L_W} a_i$$

Note that $S_T + S_U \geq S_W$. So, it follows directly from (3.1) that a constraint is not unsatisfied while:

$$S_T + S_U \geq S_W \geq b$$

[5]Note that when a cardinality constraint w becomes unit all its remaining unassigned literals are immediately implied.

$$S_W \geq b \tag{3.3}$$

Similarly, it follows directly from (3.2) that a constraint is not unit while:

$$S_T + S_U \geq S_W \geq b + a^U_{\max}$$

$$S_W \geq b + a^U_{\max} \tag{3.4}$$

Therefore, it can be concluded that the algorithm must watch in each constraint enough literals in order to satisfy (3.4).

Using this strategy the algorithm maintains for each literal a list of the constrains in which it is being watched, the *watcher list*, and when a literal is assigned to 0 the algorithm must analyse its watcher list in order to find which constraints become unit.

For each constraint it must check if $S_W \geq b + a^U_{\max}$. If it is not, the steps presented bellow must be followed:

1. Increase the value of S_W by watching more literals. As such, new positive or unassigned literals must be added to L_W until (3.4) holds.

2. Decrease the value of $a^U_{\max}$ by implying more literals. If after adding all unassigned and positive literals to L_W, (3.4) remains unsatisfied, the algorithm must successively imply the unassigned literals in L_W in decreasing order of coefficient. Therefore, a unit PB constraint can imply more than one literal.

Algorithm 2 illustrates the steps that must be followed when analyzing a constraint after one of its literals (which will be referred to as l_t) is assigned to 0, whereas a practical use of this algorithm can be checked in figure 3.1.

Algorithm 2 BCP with Pseudo-Boolean Constraints

$L_W \Leftarrow L_W \backslash \{l_t\}$
$S_W \Leftarrow S_W - a_t$
$a^U_{\max} \Leftarrow \max\{a_i \mid l_i \in L_U\}$
while $S_W < b + a^U_{\max} \wedge L_W \neq L_T \cup L_U$ **do**
  $a_s \Leftarrow \max\{a_i \mid l_i \in (L_T \cup L_U) \backslash L_W\}$
  $S_W \Leftarrow S_W + a_s$
  $L_W \Leftarrow L_W \cup \{l_s\}$
end while
if $S_W < b$ **then**
  return CONFLICT
end if
while $S_W < b + a^U_{\max}$ **do**
  $imply\,(l_{\max})$
  $a^U_{\max} \Leftarrow \{a_i \mid l_i \in L_W \wedge l_i \neq 1\}$
end while
return NOCONFLICT

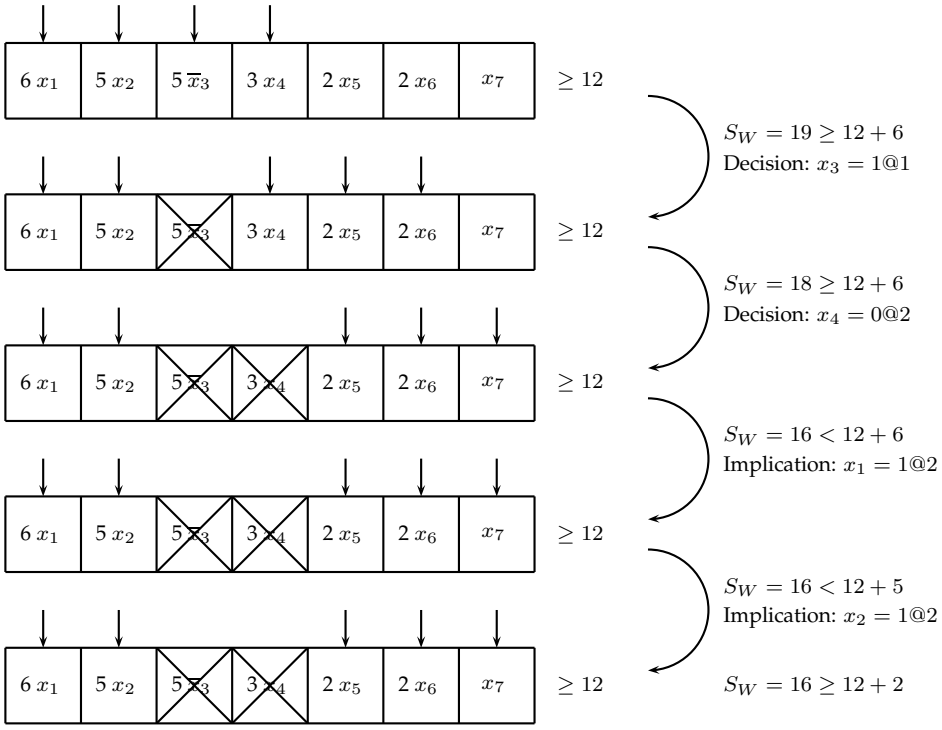

Figure 3.1: A sequence of updates of the watched literals of a certain PB constraint

It must be noted that each time a literal is implied the value of $a^U_{\max}$ may change in each constraint in which it is being watched[6] and if it does, it must be updated. Updating $a^U_{\max}$ has, however, a very large overhead. Chai and Kuehlmann, who proposed this watch-literal strategy [7] concluded that despite being very effective with clauses and cardinality constraints, the watch-literal strategy (at least the one they presented) could not be efficiently applied to general PB constraints. As such, they implemented in their solver (*Galena*) a Watch All Literals scheme, similar to the one previously suggested by Aloul et al [2].

Sakallah and Sheini proposed the following alteration to this watch-literal strategy [29]. Instead of using $a^U_{\max}$ in (3.4), they use $a_{\max}$. Hence, it is no longer necessary to compute $a^U_{\max}$ and as such propagation becomes much faster. Note that this strategy is a compromise between the Watch All Literals scheme and the one previously explained since it requires that the algorithm to watch in each constraint more literals than strictly necessary ($a_{\max} \geq a^U_{\max}$).

3.4 Conflict Analysis

In this section the learning techniques used in SAT are extended to the PB-SAT problem.

[6]Note that in algorithm 2 literals are added to L_W in decreasing order of coefficient and as such $l^U_{\max}$ is always watched.

3.4.1 Implication Graph

In this section it will be used the definition of Implication Graph presented in section 2.5.1 so it will not be repeated. Therefore, we will only clarify some subtleties.

When a clause w becomes unit, its remaining unassigned variable, say x, is immediately implied and $A^w(x)$ will contain all the assignments to variables other than x corresponding to literals in w. All those literals are assigned to 0 and the conjunction of all those assignments implies the assignment of x. However, when a variable implication is triggered by a general PB constraint w, its antecedent assignment will only contain assignments to variables corresponding to false literals in w. Note that even if w is unit, there can be other unassigned variables in w besides the implied variables and there can even be assignments corresponding to true literals in w. Naturally, these assignments must not be included in the antecedent assignments of the implied variables.

Similarly, when a PB constraint w becomes unsatisfied, the antecedent assignment of its corresponding conflict will be the set of all assignments corresponding to false literals in w.

Using this slightly modified definition of antecedent assignment, the implication graph can be constructed as detailed in section 2.5.1:

1. Every assignment $x = v(x) @\delta(x)$ corresponds to a vertex.
2. Given an assignment $x = v(x) @\delta(x)$, all vertices corresponding to assignments in $A^w(x)$ have edges to the vertex associated with this assignment.
3. Given a conflict vertex k, all vertices corresponding to assignments in $A^w(k)$ have edges to k.

In figure 3.2 it is illustrated a PB-SAT problem with clauses and PB constraints and an implication graph which shows the sequence of implications triggered by the assignment to x_1.

3.4.2 Clause Learning

The clause learning scheme here presented is equal to the learning scheme used by SAT solvers.

First of all, we must establish a partition of the implication graph. We will always use a 1UIP cut, since it is considered the most effective one [33].

Starting from $A^w(k)$ the algorithm applies recursively the following step:

- Replace in $A^w(k)$ all assignments which occur in the conflict side of the partition by their antecedent assignments.

The recursive application of this procedure corresponds to a backward traversal of the implication graph starting at k. At the end, $A^w(k)$ will contain one unique assignment made at the current level. Since the conjunction of those assignments yields a conflict, the clause corresponding to its negation must be consistent with the PB formula.

Current Truth Assignment: $\{x_5 = 1@1, x_{10} = 0@2, x_1 = 0@3, \cdots\}$

Current Decision Assignment: $\{x_1 = 0@3\}$

Constraints:

$\omega_1 : 3x_1 + 2\bar{x}_2 + x_3 + \bar{x}_4 \geq 3$ $\omega_4 : x_2 + \bar{x}_5 + x_6 \geq 1$

$\omega_2 : 2x_2 + x_8 + x_9 + 2x_{10} \geq 2$ $\omega_5 : \bar{x}_6 + \bar{x}_4 + \bar{x}_9 \geq 1$

$\omega_3 : \bar{x}_3 + x_4 + x_7 \geq 2$ $\omega_6 : x_1 + x_3 + x_4 \geq 1$

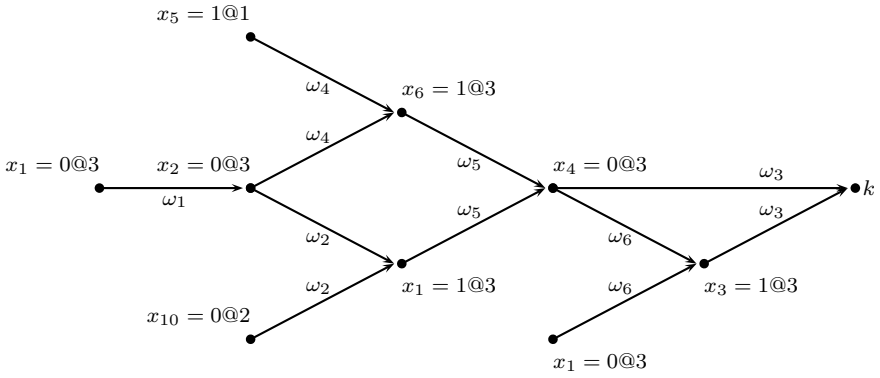

Figure 3.2: Example of a PB formula, a partial assignment, a decision assignment and the corresponding implication graph

As discussed in section 2.5.2, this kind of implication graph analysis can be viewed as a sequence of resolution steps guided by the original implication sequence in reverse order. In each resolution step one implied variable is eliminated and, since the implied variables are eliminated in reverse order, one can safely remove them from the current assignment[7]. If we denote the violated constraint by V and the unit implying constraints leading to the conflict by $U_1, \ldots, U_k$, the conflict analysis procedure can be expressed as follows:

$$R_1 = resolve\,(V, U_1, x_1)$$
$$R_2 = resolve\,(R_1, U_2, x_2)$$
$$\cdots$$
$$R_k = resolve\,(R_{k-1}, U_k, x_k)$$

Where R_i denotes the resolvent at each resolution step (also denoted as *Accumulator Constraint*) and x_i the variable to be eliminated. It must be stressed that the implied variables $x_1, \cdots, x_k$ are considered in reverse order and as such after each resolution step the corresponding eliminated variable can be removed from the current assignment. The sequence of clauses generated by the resolution steps correspond to a sequence of cuts in the implication graph[8].

Therefore, the correction of the learned clause (i.e., its consistency with the PB formula) is guaranteed by the correction of resolution [26]. However, when considering general PB constraints instead of

[7] Note that when erasing an assignment, the sequence of assignments it triggered must also be erased.

[8] Any of these clauses can reproduce the conflict and thus could be added to the PB formula.

clauses, resolution cannot be directly applied. So, in order to be able to apply resolution, a preprocessing step must be performed before each resolution step.

Firstly, it must be noted that every PB constraint can be trivially reduced to a clause using the following rule:

$$\frac{\sum_i a_i l_i \geq b}{\vee_{i=1}^{n} l_i}$$

This rule can be viewed as a corollary of the division rule. In the corresponding application of the division rule one must choose $\alpha \geq a_{\max}$.

Before applying this rule, the coefficient reduction rule must be used to eliminate all positive and unassigned literals except for the implied literal. When performing coefficient reduction on a general PB constraint it is possible to weaken too much the constraint, obtaining a tautology. Since the constraint that is being considered is unit, $\left(S_T + S_U - a^U_{\max}\right) < b$ is true. As our goal is to eliminate all literals in $(L_T \cup L_U) \setminus \{l^U_{\max}\}$ from the constraint, after applying coefficient reduction the right side of the inequality will be decremented in $S_T + S_U - a^U_{\max}$ which is lower than b. Therefore, reducing an unit PB constraint to a clause in the way just described, never yields a tautology[9].

In example 6 we resume the example presented in figure 3.2 by showing an application of clause learning to the implication graph there presented.

Example 6.

Step 0	$A(k) = \{x_3, \overline{x}_4\}$
Step 1	$A(k) = \{\overline{x}_1, \overline{x}_4\}$
Step 2	$A(k) = \{\overline{x}_1, x_6, x_9\}$
Step 3	$A(k) = \{\overline{x}_1, \overline{x}_2, x_6, \overline{x}_{10}\}$
Step 4	$A(k) = \{\overline{x}_1, \overline{x}_2, x_5, \overline{x}_{10}\}$
Step 5	$A(k) = \{\overline{x}_1, x_5, \overline{x}_{10}\}$

- Current Assignment: $\{\overline{x}_1, \overline{x}_2, x_3, \overline{x}_4, x_5, x_6, x_9, \overline{x}_{10}\}$

  Variable to Eliminate: x_3

  $w_3 : \overline{x}_3 + x_7 + x_4 \geq 2$

  $w'_3 : \overline{x}_3 \vee x_4$

  $w_6 : x_1 \vee x_3 \vee x_4$

  $R_1 = Res(w'_3, w_6, x_3) = x_1 \vee x_4$

- Current Assignment: $\{\overline{x}_1, \overline{x}_2, \overline{x}_4, x_5, x_6, x_9, \overline{x}_{10}\}$

  Variable to Eliminate: x_4

  $w_5 : \overline{x}_6 \vee \overline{x}_4 \vee \overline{x}_9$

  $R_2 = Res(R_1, w_5, x_4) = x_1 \vee \overline{x}_6 \vee \overline{x}_9$

[9] A PB constraint is a tautology if its rhs is lower than 1.

- Current Assignment: $\{\overline{x}_1, \overline{x}_2, x_5, x_6, x_9, \overline{x}_{10}\}$

  Variable to Eliminate: x_9

  $w_2 : 2x_2 + 2x_{10} + x_8 + x_9 \geq 2$

  $w_2' : x_2 \vee x_{10} \vee x_9$

  $R_3 = Res\,(R_2, w_2', x_9) = x_1 \vee x_2 \vee \overline{x}_6 \vee x_{10}$

- Current Assignment: $\{\overline{x}_1, \overline{x}_2, x_5, x_6, \overline{x}_{10}\}$

  Variable to Eliminate: x_6

  $w_4 : x_2 \vee \overline{x}_5 \vee x_6$

  $R_4 = Res\,(R_3, w_4, x_6) = x_1 \vee x_2 \vee \overline{x}_5 \vee x_{10}$

- Current Assignment: $\{\overline{x}_1, \overline{x}_2, x_5, \overline{x}_{10}\}$

  Variable to Eliminate: x_2

  $w_1 : 3x_1 + 2\overline{x}_2 + x_3 + \overline{x}_4 \geq 3$

  $w_1' : x_1 \vee \overline{x}_2$

  $R_5 = Res\,(R_4, w_1', x_2) = x_1 \vee \overline{x}_5 \vee x_{10}$

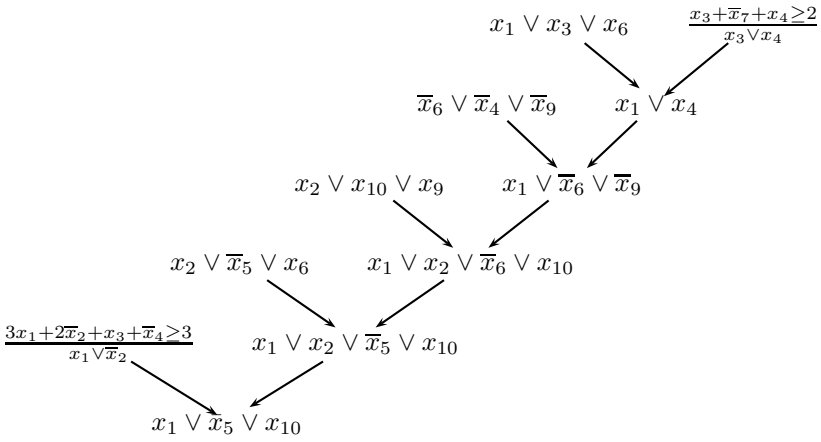

Figure 3.3: A sequence of resolution steps starting form the conflict vertex of the implication graph presented in example 6

Finally, it is important to emphasize that using this learning scheme the algorithm can always learn an *assertive constraint*, since after erasing all the assignments made at the current level, the learned constraint becomes unit.

3.4.3 Pseudo-Boolean Learning

In modern SAT solvers and PB-SAT solvers the learned constraints occupy a central role in the search process by pruning the search. There is an exponential gap between clauses and general PB constraints [14]. So, from a theoretical point of view, it is likely that general PB constraints have a greater pruning power than clauses.

When a conflict occurs the algorithm uses the learned constraints to:

Current Truth Assignment: $\{x_8 = 1@1, \cdots\}$

Current Decision Assignment: $\{x_6 = 0@2\}$

Constraints:

$\omega_1 : 3x_1 + 2x_2 + x_7 + 2\overline{x}_8 \geq 3$

$\omega_2 : 3\overline{x}_1 + x_3 + x_5 + x_9 \geq 3$

$\omega_3 : \overline{x}_2 + \overline{x}_3 + x_6 \geq 2$

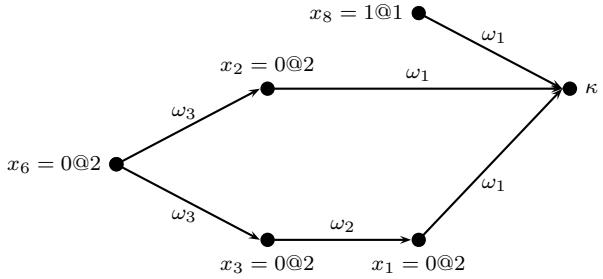

Figure 3.4: A PB formula, a PB partial assignment and the corresponding implication graph

- Determine to which level it must backtrack.
- Imply a new assignment after backtracking.

Therefore each learned constraint must exhibit two properties:

- It must be unsatisfied under the current assignment.
- It must be an assertive constraint.

The learning scheme presented in section 3.4.2 satisfies both those properties.

The operation on PB constraints which corresponds to clause resolution is *Cutting Planes*. As such, to learn a general PB constraint, the algorithm must perform a sequence of cutting plane steps instead of a sequence of resolution steps. Again, in each cutting plane step one implied variable is eliminated. The implied variables are considered in reverse order, i.e., we start at the last assignment and finish at the first UIP.

However, when performing cutting planes, the resulting constraint may not be unsatisfied under the current assignment [7]. In this situation, the learned constraint will not be able to flip any variables after erasing all the assignments made at the current decision level, which is essential for driving the search forward.

To understand why it is possible to get a constraint which is not unsatisfied under the current assignment during the sequence of cutting plane steps, we examine a concrete example based on the implication graph presented in figure 3.4.

Example 7.

- Current Assignment: $\{\overline{x}_1, \overline{x}_2, \overline{x}_3, \overline{x}_6, x_8\}$

$$\begin{array}{cc} 1 \cdot w_1 \ : \ 1\left(3x_1 + 2x_2 + x_7 + 2\overline{x}_8 \geq 3\right) & slack = -2 \\ 1 \cdot w_2 \ : \ 1\left(3\overline{x}_1 + x_3 + x_5 + x_9 \geq 3\right) & slack = 2 \\ \hline 2x_2 + x_3 + x_5 + x_7 + 2\overline{x}_8 + x_9 \geq 3 & slack = 0 \end{array}$$

- Current Assignment: $\{\overline{x}_2, \overline{x}_3, \overline{x}_6, x_8\}$

$$\begin{array}{cc} 1\left(2x_2 + x_3 + x_5 + x_7 + 2\overline{x}_8 + x_9 \geq 3\right) & slack = 0 \\ 1 \cdot w_3 \ : \ 1\left(\overline{x}_2 + \overline{x}_3 + x_6 \geq 2\right) & slack = 0 \\ \hline x_2 + x_5 + x_6 + x_7 + 2\overline{x}_8 + x_9 \geq 3 & slack = 0 \end{array}$$

- Current Assignment: $\{\overline{x}_2, \overline{x}_6, x_8\}$

$$\begin{array}{cc} 1\left(x_2 + x_5 + x_6 + x_7 + 2\overline{x}_8 + x_9 \geq 3\right) & slack = 0 \\ 1 \cdot w_3 \ : \ 1\left(\overline{x}_2 + \overline{x}_3 + x_6 \geq 2\right) & slack = 0 \\ \hline \overline{x}_3 + x_5 + 2x_6 + x_7 + 2\overline{x}_8 + x_9 \geq 4 & slack = 0 \end{array}$$

Note that in the example presented in figure 3.4 one of the implying constraints (w_1) is oversatisfied, therefore after combining it with the accumulator constraint, the slack of the resulting constraint is not negative. So, before performing each cutting plane step, the conflict analysis procedure must examine both constraints to determine if the slack of the resulting constraint is still negative.

Consider the application of a cutting plane step to two arbitrary constraints w_1 with slack s_1 and w_2 with slack s_2 and suppose α and β are used as the multiplying factors. In this situation, the slack of the resulting constraint, here denoted by s_r, is given by linearly combining the slacks of w_1 and w_2:

$$s_r = (\alpha \cdot s_1) + (\beta \cdot s_2)$$

Using the procedure just presented, before the application of each cutting plane step, the learning algorithm verifies if the resulting constraint is still unsatisfied under the current assignment. If it is not, the implied constraint must be reduced to lower its slack. This process is guaranteed to work since the repeated reduction of constraints will eventually lead to a simple clause with slack 0 [10].

An iterative procedure to weaken an over-satisfied implying constraint w_i is suggested by Chai and Kuehlmann [7]. In each iteration of this procedure $s_{acc}(n+1) = (\alpha \cdot s_{acc}(n)) + (\beta \cdot s_i)$ is evaluated[11]. If $s_{acc}(n+1) < 0$, then the cutting plane step may be applied. If it is not, then another coefficient reduction must be performed in order to lower the slack of w_i in at least $\lceil (s_{acc}(n+1) + 1)/\beta \rceil$ unities. After applying this reduction the values of α and β must be updated, since they depend on the value of the coefficient of the variable being eliminated, say x_i. However the extent of the next possible reduction depends on the new values of α and β. Therefore, a conservative over-approximation of the slack value is suggested in order to break this cyclic dependency.

[10] This was already discussed in section 3.4.2.

[11] The variables α and β correspond to the multiplying factors used in the application of the cutting plane step.

If the procedure just described is applied to example 7, before the first cutting plane step, a coefficient reduction must be performed on constraint w_2, since $2 \cdot 1 + (-2) \cdot 1 = 0$. Therefore, the slack of w_2 must be decremented in at least one unity ($0 + 1 = 1$), which can be easily done removing one of its non-false literals, say x_9. As a result, the following constraint is generated:

$$w'_2 \ : \ 2\overline{x}_1 + x_3 + x_5 \geq 2 \quad slack = 1$$

The values of α and β must be updated to 2 and 3 respectively. The algorithm may now apply the cutting plane step, since:

$$\alpha \cdot s_1 + \beta \cdot s'_2 = 2 \cdot (-2) + 3 \cdot 1 = (-1) < 0$$

Sakallah and Sheini implemented in their solver, *Pueblo* [29], a different approach. Before each cutting plane step, they test the slack of the resulting constraint; if its slack is not negative, they always reduce the implying constraint to a clause. Note that when reducing an implying constraint to a clause, the slack of the obtained clause is always 0.

In the beginning of this subsection it was stressed the importance of learning an assertive constraint, that is a constraint for which there is a decision level in which the given constraint becomes unit. However the conflict analysis procedure may learn a non-assertive constraint (this happens both in *Pueblo* and *Galena*). Consider the following constraint:

$$\overline{x}_1 \, (0@4) + \overline{x}_2 + x_3 \, (0@3) + x_6 \, (0@3) + x_0 \, (0@3) \geq 3$$

It is easy to see that if the algorithm backtracks to decision level 3, after erasing all the assignments made at that decision level, this constraint does not become unit. In fact, this constraint cannot become unit, no matter the level to which the algorithm backtracks.

Chai and Kuehlmann [7] suggested the following scheme: when analyzing the implication graph instead of stopping the sequence of cutting plane steps at the first UIP, they only stop when the learned constraint is an assertive one. However, they do not provide further details. Moreover, Daniel Le Berre and Anne Parrain [5] who implement in their solver, *sat4jPseudo*, a PB learning scheme similar to the one just described, state: 'For PB constraints, we do not have an efficient incremental way to do it [detect assertive constraints].'

3.4.4 Cardinality Constraint Learning

It was already discussed that learning general PB constraints slows down the deduction procedure because the watch literal strategy is not as efficient with general PB constraints as it is with clauses or cardinality constraints [7]. Note that in a clause, as well as in a cardinality constraint, it is only necessary to watch a fixed number of literals, whereas in a general PB constraint the number of watched literals varies during the execution of the algorithm.

In theory, a general PB constraint corresponds to an exponential number of cardinality constraints and a cardinality constraint corresponds to an exponential number of clauses. However, in practice the gap between cardinality constraints and clauses is typically larger than the gap between general PB constraints and cardinality constraints when considering learned constraints which are highly redundant [7].

In *Galena*, Chai and Kuehlmann choose to learn cardinality constraints instead of general PB constraints. The method used to learn cardinality constraints in *Galena* is similar to the method used to learn general PB constraints described in section 3.4.3. Additionally, it is introduced a post-reduction procedure, which converts the learned constraint into a weaker cardinality constraint.

Algorithm 3 converts a general PB constraint into a cardinality constraint. This algorithm first determines the minimum number of literals that must be assigned to 1 for the constraint to be satisfied and then it drops as many low-coefficient literals as possible.

Algorithm 3 Cardinality Constraint Reduction Algorithm

$x \leftarrow 0$
$k' \leftarrow 0$
$L' \leftarrow L$
while $x < b \vee L \neq \emptyset$ **do**
  $a_{\max} \leftarrow \max\{a_i | l_i \in L\}$
  $last \leftarrow a_{\max}$
  $L \leftarrow L \backslash \{l_{\max}\}$
  $x \leftarrow x + a_{\max}$
  $k' \leftarrow k' + 1$
end while
$guard = x - last$
while $guard + \min\{a_i | l_i \in L'\} < b$ **do**
  $a_{\min} = \min\{a_i | l_i \in L'\}$
  $guard += a_{\min}$
  $L' = L' \backslash \{l_{\min}\}$
end while
return $\sum_{l_i \in L'} l_i \geq k'$

Note that the first loop of algorithm 3 corresponds to the cardinality constraint reduction presented in section 3.1.3, whereas the second loop allows the algorithm to strengthen the cardinality constraint obtained after the execution of the first loop (which is implicitly maintained). Therefore, it is important to clarify the rationale behind the second loop.

Consider the application of the cardinality constraint reduction described in section 3.1.3 to a general PB constraint:

$$\sum_{i=1}^{n} a_i \cdot l_i \geq b$$

Suppose the resulting constraint is:

$$\sum_{i=1}^{n} l_i \geq m$$

To obtain a stronger cardinality constraint, we must apply to the original PB constraint successive coeffi-

cient reductions, so as to drop as many low-coefficient literals as possible, obtaining a new PB constraint:

$$\sum_{i=1}^{n'} a_i \cdot l_i \geq b'$$

However, it must be guaranteed that when applying a cardinality constraint reduction to the new PB constraint, the obtained cardinality constraint has also degree m. Therefore, the following inequalities must hold:

$$\sum_{i=1}^{m-1} a_i < b' \tag{3.5}$$

$$\sum_{i=1}^{m} a_i \geq b' \tag{3.6}$$

Observe that when performing a coefficient reduction, the value of b is decremented, so $b' \leq b$. Therefore, (3.6) is immediately satisfied. It is also important to note that (3.5) limits the number of possible coefficient reductions, since the value of b can only be decremented in $b - b'$ unities. Using (3.5), it can easily be obtained an upper bound for the extent of the coefficient reductions:

$$\begin{aligned} \sum_{i=1}^{m-1} a_i &< b' \\ -\left(\sum_{i=1}^{m-1} a_i\right) &> -b' \\ b - \left(\sum_{i=1}^{m-1} a_i\right) &> b - b' \\ \left(\sum_{i=1}^{m-1} a_i\right) + (b - b') &< b \end{aligned} \tag{3.7}$$

Note that (3.7) is used as the guard of the second loop of algorithm 3.

Example 8.

$$\frac{6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 + x_6 \geq 17}{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 4}$$

$$(b - b') < b - \textstyle\sum_{i=1}^{3} a_i$$
$$(b - b') < 17 - 15$$
$$(b - b') \leq 1$$

$$\frac{\dfrac{6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 + x_6 \geq 17}{6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 \geq 16}}{x_1 + x_2 + x_3 + x_4 + x_5 \geq 4}$$

In example 8 we resume example 5, showing the application of the strengthening technique just presented to a given cardinality constraint. In this example it is performed a normal cardinality constraint reduction, then it is calculated the extent of the possible coefficient reduction. Since $b - b' = 1$, we can only eliminate from the original PB constraint literal x_6. After applying this coefficient reduction, a new cardinality constraint reduction is applied, which yields a cardinality constraint stronger than the one previously obtained.

It is possible that the resulting cardinality constraint is not unsatisfied under the current assignment. In this situation the algorithm proceeds as was described in section 3.4.3.

3.4.5 Backtracking

Suppose the conflict analysis procedure is always able to learn an assertive constraint. After learning such a constraint, the algorithm can use it to determine to which decision level it must backtrack. It must backtrack to a decision level at which the learned constraint becomes unit[12].

However, a general PB constraint may become unit in multiple decision levels. Consider the following constraint:

$$w_i \ : \ 3x_1 \, (x_1 = 0@3) + 2x_2 \, (x_2 = 0@2) + x_3 \geq 4$$

If the algorithm backtracks to decision level 3, w_i becomes unit and the assignment $x_1 = 1$ is implied. However, if the algorithm backtracks to decision level 2, again, w_1 becomes unit and $x_1 = 1$ is also implied.

Galena performs the largest possible backtrack to an earlier decision level. This allows the algorithm to maintain as an invariant the fact that each variable is implied at the earliest possible decision level.

It is important to note that as opposed to general PB constraints which may become unit in multiple decision levels, cardinality constraints and clauses can only become unit in one decision level. Therefore,

[12]Since we are considering that the conflict analysis procedure always learns an assertive constraint, there must be a level at which the constraint becomes unit.

backtracking is easier when handling clauses or cardinality constraints.

3.4.6 A Hybrid Approach

Sheini and Sakallah [29] noted that any solver which performs PB learning can be modified to additionally perform clause learning with no significant extra overhead. Moreover, despite the greater pruning power of PB learning, clause learning has its own advantages: it always produces an assertive constraint and it does not compromise as heavily the propagation procedure as general PB learning. As such, in their solver *Pueblo*, they implement a hybrid learning method. Each time a conflict occurs, not only does *Pueblo* learn a general PB constraint, but it also learns a clause.

In order to perform clause learning efficiently, during the sequence of cutting plane steps, *Pueblo* maintains a special set X_F which stores the false literals in the accumulator constraint (note that these literals correspond to the current cut in the implication graph). At each cutting plane step, the algorithm eliminates one implied variable. Simultaneously, this implied variable is replaced in X_F by its antecedent assignment. The overall learning process terminates as soon as the first UIP is reached which is automatically detected inside the clause learning procedure. At this point, the clause stored in X_F is unit and the accumulator constraint corresponds to the learned PB constraint. The backtracking level is determined processing together the learned PB constraint and the learned clause.

As opposed to *Galena* which does not always stop the implication graph analysis at the first UIP (it continues the backward search until it finds an assertive constraint), *Pueblo* can always stop at the first UIP, since at the first UIP, the learned clause is guaranteed to be an assertive constraint.

Before applying a cutting plane step, *Pueblo* checks if the slack of the resulting constraint is negative. If it is not, *Pueblo* immediately reduces the current implying constraint to a clause with slack 0, thereby avoiding the iterated application of coefficient reduction steps described in section 3.4.3.

In example 9 it is presented an application of the *Pueblo's* learning algorithm to the conflict illustrated in figure 3.5.

Example 9.

- **Step 0:**

  Current Partial Assignment: $\{x_1, \overline{x}_3, \overline{x}_4, x_5, \overline{x}_6, \overline{x}_7, x_8, \overline{x}_9, \overline{x}_{10}, x_{11}\}$

  Variable to Eliminate: x_{11}

  Accumulator Constraint: $\underbrace{\overline{x}_2 + \overline{x}_5 + \overline{x}_8 + \overline{x}_{11} \geq 3}_{w_4} \quad slack = -2$

  Implying Constraint: $\underbrace{x_{11} + x_9 + x_{10} \geq 1}_{w_1} \quad slack = 0$

  $X_F = \{\overline{x}_5, \overline{x}_8, \overline{x}_{11}\}$

Current Truth Assignment: $\{x_3 = 0@1, x_6 = 0@2, x_9 = 0@3, \cdots\}$

Current Decision Assignment: $\{x_1 = 1@4\}$

Constraints:

$\omega_1 : x_9 + x_{10} + x_{11} \geq 1$

$\omega_2 : x_6 + x_7 + x_8 \geq 1$

$\omega_3 : x_3 + x_4 + x_5 \geq 1$

$\omega_4 : \overline{x}_2 + \overline{x}_5 + \overline{x}_8 + \overline{x}_{11} \geq 3$

$\omega_5 : \overline{x}_1 + \overline{x}_4 + \overline{x}_7 + \overline{x}_{10} \geq 3$

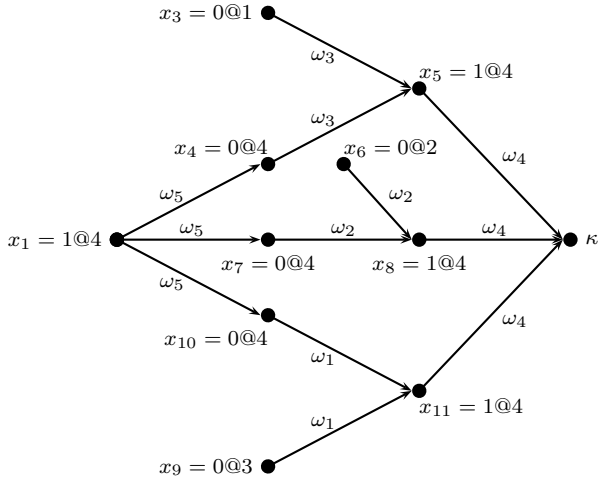

Figure 3.5: A PB formula, a PB partial assignment and the corresponding implication graph

$$\begin{array}{rl} 1\,(\overline{x}_2 + \overline{x}_5 + \overline{x}_8 + \overline{x}_{11} \geq 3) & slack = -2 \\ 1\,(x_9 + x_{10} + x_{11} \geq 1) & slack = 0 \\ \hline \overline{x}_2 + \overline{x}_5 + \overline{x}_8 + x_9 + x_{10} \geq 3 & slack = -2 \end{array}$$

- **Step 1:**

  Current Partial Assignment: $\{x_1, \overline{x}_3, \overline{x}_4, x_5, \overline{x}_6, \overline{x}_7, x_8, \overline{x}_9, \overline{x}_{10}\}$

  Variable to Eliminate: x_8

  Accumulator Constraint: $\overline{x}_2 + \overline{x}_5 + \overline{x}_8 + x_9 + x_{10} \geq 3 \quad slack = -2$

  Implying Constraint: $\underbrace{x_6 + x_7 + x_8 \geq 1}_{w_2} \quad slack = 0$

  $X_F = \{\overline{x}_5, \overline{x}_8, x_9, x_{10}\}$

$$\begin{array}{rl} 1\,(\overline{x}_2 + \overline{x}_5 + \overline{x}_8 + x_9 + x_{10} \geq 3) & slack = -2 \\ 1\,(x_6 + x_7 + x_8 \geq 1) & slack = 0 \\ \hline \overline{x}_2 + \overline{x}_5 + x_6 + x_7 + x_9 + x_{10} \geq 3 & slack = -2 \end{array}$$

- **Step 2:**

  Current Partial Assignment: $\{x_1, \overline{x}_3, \overline{x}_4, x_5, \overline{x}_6, \overline{x}_7, \overline{x}_9, \overline{x}_{10}\}$

  Variable to Eliminate: x_5

  Accumulator Constraint: $\overline{x}_2 + \overline{x}_5 + x_6 + x_7 + x_9 + x_{10} \geq 3 \quad slack = -2$

Implying Constraint: $\underbrace{x_3 + x_4 + x_5 \geq 1}_{w_3} \quad slack = 0$

$X_F = \{\overline{x}_5, x_6, x_7, x_9, x_{10}\}$

$$\begin{array}{cc} 1\,(\overline{x}_2 + \overline{x}_5 + x_6 + x_7 + x_9 + x_{10} \geq 3) & slack = -2 \\ 1\,(x_3 + x_4 + x_5 \geq 1) & slack = 0 \\ \hline \overline{x}_2 + x_3 + x_4 + x_6 + x_7 + x_9 + x_{10} \geq 3 & slack = -2 \end{array}$$

- **Step 3:**

  Current Partial Assignment: $\{x_1, \overline{x}_3, \overline{x}_4, \overline{x}_6, \overline{x}_7, \overline{x}_9, \overline{x}_{10}\}$

  Variables to Eliminate: x_4, x_7, x_{10}

  Accumulator Constraint: $\overline{x}_2 + x_3 + x_4 + x_6 + x_7 + x_9 + x_{10} \geq 3\ slack = -2$

  Implying Constraint: $\underbrace{\overline{x}_1 + \overline{x}_4 + \overline{x}_7 + \overline{x}_{10} \geq 3}_{w_5} \quad slack = 0$

  $X_F = \{x_3, x_4, x_6, x_7, x_9, x_{10}\}$

$$\begin{array}{cc} 1\,(\overline{x}_2 + x_3 + x_4 + x_6 + x_7 + x_9 + x_{10} \geq 3) & slack = -2 \\ 1\,(\overline{x}_1 + \overline{x}_4 + \overline{x}_7 + \overline{x}_{10} \geq 3) & slack = 0 \\ \hline \overline{x}_1 + \overline{x}_2 + x_3 + x_6 + x_9 \geq 3 & slack = -2 \end{array}$$

- **Results:**

  Learned PB constraint: $\overline{x}_1 + \overline{x}_2 + x_3 + x_6 + x_9 \geq 3 \quad slack = -2$

  Learned clause: $\overline{x}_1 \vee x_3 \vee x_6 \vee x_9$

3.4.7 Backtracking Revisited

Pueblo's learning scheme requires a specific backtracking strategy since instead of one constraint it learns two different constraints. There are two main scenarios to consider:

1. The learned PB constraint is an assertive constraint.
2. The learned PB constraint is not an assertive constraint.

The Learned PB Constraint is an Assertive Constraint

The earliest decision level in which the learned PB constraint (w_p) becomes unit is denoted by l_p and the decision level in which the learned clause (w_c) becomes unit is denoted by l_c. In this situation the backtracking level is computed in the following way: $\min(l_c, l_p)$. After backtracking:

1. If $l_c = l_p$, both constraints become unit and as such both their implications are propagated.
2. If $l_c < l_p$, only w_c becomes unit and as such only the implications triggered by w_c are propagated.

3. If $l_c > l_p$, only w_p becomes unit and as such only the implications triggered by w_p are propagated.

In example 9 the algorithm learns the following constraints:

$$\overline{x}_1\,(0@4) + \overline{x}_2 + x_3\,(0@1) + x_6\,(0@2) + x_9\,(0@3) \geq 3$$

$$\overline{x}_1\,(0@4) \vee x_3\,(0@1) \vee x_6\,(0@2) \vee x_9\,(0@3)$$

It is easy to see that in this situation $l_c = 3$ and $l_p = 2$. As such, according to the backtracking scheme just presented the algorithm must backtrack to decision level 2.

The Learned PB Constraint is not Assertive

In this situation there are two different possibilities:

1. If w_p is not unsatisfied at decision level l_c, the algorithm backtracks to decision level l_c and the implications triggered by w_c are propagated.

2. If w_p is unsatisfied at decision level l_c there are two different strategies that can be followed:

   - The algorithm backtracks to decision level l_c and in this decision level it applies the conflict analysis procedure to w_p.
   - The algorithm backtracks to the highest decision level in which w_p is not unsatisfied. In this decision level neither w_p nor w_c are unit, as such the algorithm must make a new decision assignment. If the algorithm uses this strategy, it must never erase the learned clause and PB constraint so as to ensure completeness. *Plueblo* uses this strategy.

3.4.8 Constraint Deletion

This subject was already discussed for SAT solvers in section 2.5.4. Since the problem constraints cannot generally be deleted[13], any PB solver which only performs clause learning may use one of the deletion strategies there presented. Therefore, it is important to clarify the constraint deletion strategies performed by PB solvers which implement PB learning.

First of all, it is very important to note that a general PB constraint uses much more memory than a clause, since a general PB constraint has coefficients of arbitrary size. Moreover, as was already seen, general PB constraints are much harder to propagate.

Consider the following extension of the clause deletion strategy used by MINISAT:

1. Each PB constraint has associated an activity counter.

2. Each time a PB constraint is used to imply a conflict its activity is incremented.

[13] They can under certain circumstances. For instance, subsumption.

3. Over time the activities of all PB constraints are divided by a constant.

4. When the activity of a certain PB constraint drops below a certain threshold, that constraint is erased.

   - If a hybrid approach is being used the corresponding clause is not erased from the PB formula. This strategy is implemented in *Pueblo*.
   - If a pure PB learning algorithm is being used, before erasing a PB constraint, it is reduced to a clause which is stored in the PB formula and only then may the algorithm erase the PB constraint. This strategy is suggested by Chai and Kuehlmann [7].

   Both this strategies limit the time spent in propagating PB constraints and delegate the role of the dropped constraints to their corresponding clauses.

5. Every time the solver restarts it increments the threshold. Hence, the amount of PB constraints that the algorithm can learn.

3.4.9 PB Learning Techniques in Practice

There are several solvers which learn cardinality or general pseudo-Boolean constraints. Chai and Kuehlmann [7] implement in *Galena* a cardinality constraint learning scheme. Daniel Le Berre and Anne Parrain [5] implement in *sat4jPseudo* a pure PB learning scheme, very similar to the one implemented in *Galena*. However they do not perform the final post-reduction that is performed in *Galena* and that converts the learned PB constraint into a cardinality constraint. Finally, Sheini and Sakallah [29] use in *Pueblo* a hybrid approach.

In "The First Evaluation of Pseudo-Boolean Solvers" [20], *Pueblo* was found to be the best solver on decision problems. It also performed very well on optimization problems, except for PBO instances containing big integers (integers bigger than 2^{30}), on which it was not evaluated. Vasco Manquinho and Olivier Roussel conclude that: 'All in all, in this combined categories [proving optimality and proving unsatisfiability], *Pueblo* has the best results'.

Galena, however, performed very poorly, both on decision and optimization problems. Since *Galena* uses a watch all literals scheme and *Pueblo* uses a more sophisticated watch literal scheme, cardinality constraint learning cannot be just yet discarded as an efficient learning technique.

In "The second Evaluation of Pseudo-Boolean Solvers", *Pueblo* was again the best PB solver on decision problems and it also performed reasonably well on optimization problems. However, *sat4jPseudo* 'was well behind of other solvers for optimal answers'. Daniel Le Berre and Anne Parrain evaluated *sat4jPseudo* with clause learning instead of PB learning under the same conditions and got much better results. They argue that since most of the benchmarks are not pure pseudo-Boolean problems[14], clause

[14] they have clauses, cardinality constraints and PB constraints.

learning is only needed to deal with the clausal part of such problems and PB learning is not needed at all.

The results obtained by Anne Parrain and Daniel Le Berre are very disturbing when considering the excellent results of *Pueblo*. Nevertheless, they provide three major reasons for the bad performance of their solver:

- It is very difficult to detect an assertive PB constraint.
- In the conflict analysis, maintaining a conflicting constraint needs additional processing.
- Manipulation of PB constraints is very costly, since they use arbitrary precision integers.

The learning scheme used by *Pueblo*, solves the first two issues and does not have to cope with the third one, since *Pueblo* does not deal with big integers.

Hence, the main goal of this thesis is to help understanding the usefulness of learning techniques in solving *PBO* and *PB-SAT* instances.

Chapter 4

Implementation Issues

A PB solver is a very complex piece of software which tries to solve efficiently a very hard problem. Hence, every implementation detail plays an important role in the solver overall performance, thus requiring carefull study and examination. This chapter does not address all the implementation issues which arose in the course of this work, but only those considered to be more important, not only because of their impact in the performance of the solver but also because of their theoretical interest.

4.1 Generating the Implication Graph

It is important to emphasize that when a literal is implied to 0, all the constraints in which it is being watched must be analysed (according to the procedure described in algorithm 2 presented in section 3.3) and other literals may have to be implied. These literals are added to a list - the *unit list*. In each iteration of the propagation procedure one literal is removed from the *unit list* and all the operations that must be done in order to imply this literal are carried away. Each literal kept in the *unit list* must be associated with the constraint by which it is implied, in order to implicitly maintain the implication graph. The propagation procedure finishes when this list becomes empty.

Naturally, the generated implication graph depends on the behaviour of the *unit list*:

- If the *unit list* has a FIFO (first in first out) behaviour, the implication graph will be built in a breadth-first search way.
- If the *unit list* has a LIFO (last in first out) behaviour, the implication graph will be built in a depht-first search way.

No matter what kind of learning scheme is being used, the learned constraint will always depend on the structure of the implication graph. As such, we implemented in *bsolo* both these strategies in order to draw conclusions about their relative efficiency.

Figure 4.1 presents a formula and partial assignment, whereas figures 4.2 and 4.3 present two different possible implication graphs corresponding to the given formula and partial assignment. The first implication graph is built in a depth-first search way, while the second one is built in a breath-first search way. Both implication graphs are preceded by the sequence of *unit lists* that were considered during the propagation procedure.

When generating the implication graph in a breath-first search way, one can guarantee that there is no other possible implication graph such that the length of the longest path between the decision assignment vertex and the conflict vertex is lower than the one considered. As such, the learned constraint is probably determined using a smaller number of constraints. Considering the formula and decision assignment presented in figure 4.1, it is possible to verify that if the implication graph is generated in a breath-first search way, the learned constraint is determined using five constraints. If it is generated ia a depth-first search way, the learned constraint is determined using nine constraints (ω_1 is used twice).

Current Truth Assignment: $\{x_1 = 0@1\}$
Current Decision Assignment: $\{x_1 = 0@1\}$
Constraints:

$\omega_1 : x_1 + x_2 + x_3 + x_4 \geq 3$
$\omega_2 : \bar{x}_2 + x_6 \geq 1$
$\omega_3 : \bar{x}_3 + \bar{x}_6 + x_7 \geq 1$
$\omega_4 : \bar{x}_4 + x_5 \geq 1$
$\omega_5 : \bar{x}_5 + \bar{x}_6 + \bar{x}_7 \geq 1$
$\omega_6 : \bar{x}_5 + x_8 \geq 1$
$\omega_7 : \bar{x}_8 + x_9 \geq 1$
$\omega_8 : \bar{x}_9 + x_{10} \geq 1$
$\omega_9 : x_6 + \bar{x}_8 + \bar{x}_9 + \bar{x}_{10} \geq 1$

Figure 4.1: Example of a PB formula, a partial assignment and a decision assignment

4.2 Dealing with Large Coefficients

When performing general PB Learning or any learning scheme that requires performing a sequence of cutting plane steps each time a conflict occurs, the coefficients of the learned constraints may grow very fast. Note that in each cutting plane step two PB constraints are linearly combined. Given two constraints: $\sum_i a_i \cdot l_i \geq b$ and $\sum_i c_i \cdot l_i \geq d$, the size of the largest coefficient of the resulting constraint may be $\max\{b \cdot d, \max_i\{a_i\} \cdot \max_i\{c_i\}\}$ in the worst case. Therefore, it is easy to see that during a sequence of cutting plane steps the size of the coefficients of the accumulator constraint may, in the worst case, grow exponentially in the number of cutting plane steps (which is of the same order of the number of literals assigned at the current level).

Hence, one problem that may occur during the resolution of the formula and particularly in the conflict analysis procedure is integer overflow. To ensure that this problem does not occur it was established in our versions of *bsolo* a maximum coefficient size (we have used 10^6). Therefore, every time the solver performs a cutting plane step all the coefficients of the resulting constraint are checked in order to find if one of them is bigger than the established limit. If it is, the solver repeatedly divides the constraint by

$Unit\ List\ 1\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(x_4,\omega_1)\}$

$Unit\ List\ 2\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(x_5,\omega_4)\}$

$Unit\ List\ 3\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(x_8,\omega_6)\}$

$Unit\ List\ 4\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(x_9,\omega_7)\}$

$Unit\ List\ 5\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(x_{10},\omega_8)\}$

$Unit\ List\ 6\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(x_6,\omega_9)\}$

$Unit\ List\ 7\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(\bar{x}_7,\omega_5)\}$

$Unit\ List\ 8\ :\ \{(x_2,\omega_1),(x_3,\omega_1),(\bar{x}_3,\omega_3)\}$

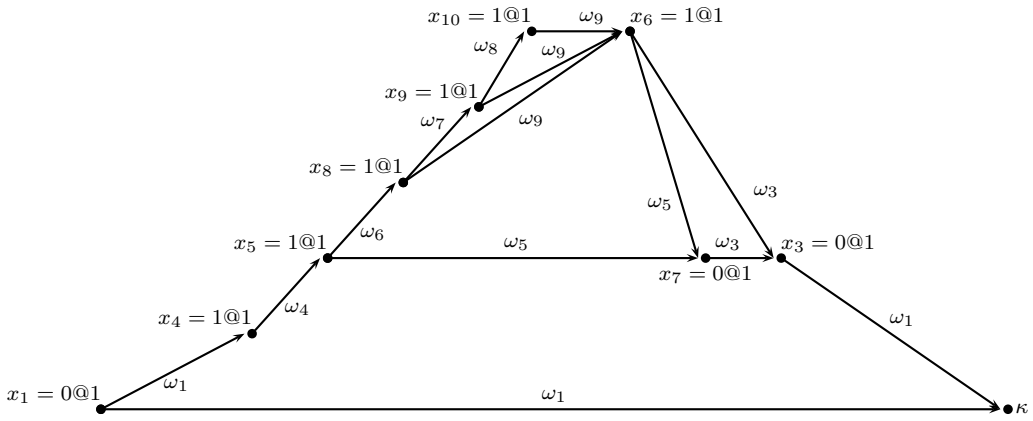

Figure 4.2: Example of an implication graph built in a depth-first search way

2 until its largest coefficient is lower than a second maximum coefficient size (we have used 10^5). When dividing a constraint by 2, it is being used the division rule presented in section 3.1.3.

During the conflict analysis the accumulator constraint must always have negative slack. However the division rule does not preserve the slack of the resulting constraint, since it does not guarantee that the slack of the resulting constraint is equal to the slack of the original one, which can be verified in example 10.

Example 10.

$$\frac{3x_1(0@1) + 3x_2(0@1) + 3x_3(1@1) + x4(1@1) \geq 5 \quad slack = -1}{2x_1(0@1) + 2x_2(0@1) + 2x_3(1@1) + x4(1@1) \geq 3 \quad slack = 0}$$

As such, before dividing by 2 a coefficient associated with a slack contributing literal, the solver must check if it is odd. In this case it must perform a coefficient reduction step before the division (note that the coefficient reduction rule when applied to slack contributing literals preserves the slack). This technique is illustrated in example 11.

Example 11.

$$\frac{\dfrac{3x_1(0@1) + 3x_2(0@1) + 3x_3(1@1) + x4(1@1) \geq 5 \quad slack = -1}{3x_1(0@1) + 3x_2(0@1) + 2x_3(1@1) \geq 3 \quad slack = -1}}{2x_1(0@1) + 2x_2(0@1) + x_3(1@1) \geq 2 \quad slack = -1}$$

$Unit\ List\ 1\ :\ \{(x_2, \omega_1), (x_3, \omega_1), (x_4, \omega_1)\}$

$Unit\ List\ 2\ :\ \{(x_3, \omega_1), (x_4, \omega_1), (x_6, \omega_2)\}$

$Unit\ List\ 3\ :\ \{(x_4, \omega_1), (x_6, \omega_2)\}$

$Unit\ List\ 4\ :\ \{(x_6, \omega_2), (x_5, \omega_4)\}$

$Unit\ List\ 5\ :\ \{(x_5, \omega_4), (x_7, \omega_3)\}$

$Unit\ List\ 6\ :\ \{(x_7, \omega_3), (\bar{x}_7, \omega_5), (x_8, \omega_6)\}$

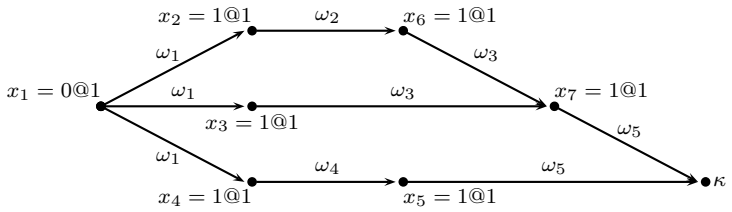

Figure 4.3: Example of an implication graph built in a breath-first search way

4.3 Eliminating non-false Literals

Besides all the learning schemes described in section 3.4 we tried yet a different one, in which a final sequence of coefficient reduction steps is applied to the learned constraint in order to eliminate all slack contributing literals. As explained in section 4.2, the coefficient reduction rule, when applied to eliminate slack contributing literals, preserves the slack of the constraint to which it is applied. As such, each time a conflict occurs, this learning scheme allows the solver to learn a smaller contraint that also captures the conflict. Since smaller constraints are easier to propagate this learning scheme appears to be another compromise between clause learning and general PB Learning.

When using a general PB learning scheme, not only does the learned constraint eliminate partial assignments which only include literals that imply the conflict, but it can also eliminate partial assignments that include slack contributing literals, that is, non-false literals. Consider the implication graph presented in figure 4.4. If the solver applies a general PB learning scheme, it is able to learn the following constraint:

$$x_3 + 2x_4 + \bar{x}_5 \geq 2$$

It is easy to see that this constraint excludes the partial assignment which caused the conflict:

$$A^{wc}(k) = \{x_5 = 1, x_4 = 0\}$$

However this constraint also elminates an additional conflicting assignment: it excludes the assignment $\{x_3 = 0, x_4 = 0\}$, which includes a slack contributing literal (x_3). When using a clause learning scheme or even a learning scheme similar to the one proposed in this section, the learned constraint prevents the solver from repeating a partial assignment that led to a conflict. However, when using a general PB learning scheme, the learned constraint can additionally prevent conflicting partial assignments before

Current Truth Assignment: $\{x_3 = 1@1, x_5 = 1@2\}$

Current Decision Assignment: $\{x_4 = 0@3\}$

Constraints:

$\omega_1 : x_1 + x_4 + \overline{x}_5 \geq 1$

$\omega_2 : \overline{x}_1 + x_2 + x_3 \geq 2$

$\omega_3 : x_4 + \overline{x}_2 \geq 1$

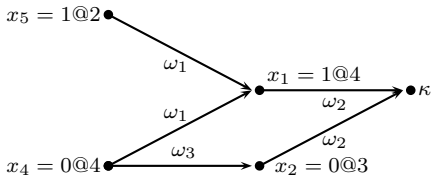

Figure 4.4: A PB formula, a PB partial assignment and the corresponding implication graph

they are tried [12]. Hence, after eliminating all non-false literals, the prunning power of the resulting constraint is lower than it is before. Nevertheless, its overhead to maintain is smaller.

4.4 Heuristic Backtracking

Heuristic backtracking [6, 18] consists of determining the backtrack level using heuristic information about the most recently recorded constraint. In this work, we have implemented two different heuristics:

1. **H1:** The backtrack level is determined selecting the decision level with the largest number of occurrences in the learned constraint.

2. **H2:** The backtrack level is determined selecting the decision level corresponding to the literal with the highest activity. The activity of each literal is computed using a VSIDS-like heuristic.

Both heuristic backtracking schemes were implemented on top of a version of the solver which uses a hybrid learning scheme. Every time a conflict occurs, if the backtracking level associated with the PB constraint is not lower than the one associated with the clause, the solver performs a heuristic backtracking step. Our work concerning heuristic backtracking was mainly exploratory, since both these versions did not show promising results, it was not carried away.

When performing heuristic backtracking the completeness of the algorithm is not guaranteed. Note that complete algorithms can establish unsatisfiability if given enough CPU time, whereas incomplete algorithms cannot.

Completeness techniques for heuristic backtracking can be organized in two classes:

1. Making recorded constraints non-deletable. This solution may yield an exponential growth in the number of recorded constraints.

2. Increasing the interval between applications of different backtracking schemes. Using this scheme the algorithm normally performs conflict directed backtracking and every time a given threshold is reached it performs an heuristic backtracking step, after this step the threshold is incremented.

4.5 An Initial Classification Step

In the course of this thesis we have implemented different versions of the solver corresponding to different learning schemes. Each one of these versions proved to be more effective than the others for some group of instances. One can easily conclude that versions of the solver using different learning schemes behave better than the others depending on some characteristics of the instances given as input. For instance, a cardinality constraint learning scheme is more appropriate to deal with an instance with a large number of cardinality constraints than a clause learning scheme. As such, we used the algorithm C4.5 [25] to generate a decision tree that given a PB instance, determines which version of the solver is more appropriate to it, in order to develop an "instance aware" PB solver. The classification of each instance is done according to the following attributes:

- Number of variables;
- Number of constraints;
- Number of clauses;
- Number of cardinality constraints;
- Number of general PB constraints;
- Number of literals;
- Number of watched literals.

Since these are the attributes which define the structure of the formula.

The algorithm C4.5 outputs a file containing the description of the decision tree, using this file our program generates the code of a C++ function which receives as input the attributes stated above and outputs the corresponding classfication. Using this C++ file, we developed a new version of the solver which aggregates the best versions of the solver.

Chapter 5

Experimental Resutls

5.1 Results Presentation

This section presents the experimental results of applying different learning schemes to several classes of instances, which only include small integers (that is, integers smaller than 2^{30}). Experimental results concerning heuristic backtracking are also presented and compared with the standard conflict directed backtracking approach. All these learning schemes as well as the different backtracking heuristics were implemented on top of *bsolo*, a state of the art PB solver. All versions of the solver were run on a Intel Xeon 5160 server (3.0GHZ, 1333Mhz, 4GB) running Red Hat Enterprise Linux WS 4. The CPU time limit for each instance was set to 1800 seconds.

The solver can output one of four possible answers [20]:

1. *"s SATISFIABLE"*: the solver has found a solution but either there is no function to optimize or it cannot prove that this solution gives the least value of the objective function.
2. *"s OPTIMUM FOUND"*: the solver has found a model and it can prove that no other solution will give a value of the objective function strictly less than the one obtained with this model.
3. *"s UNSATISFIABLE"*: the solver can prove that the formula has no solution.
4. *"s UNKNOWN"*: the solver is unable to tell anything about the formula.

The experimental results presented in tables 5.1, 5.2, 5.3, 5.4 and 5.6 correspond to the small integer non-optimization benchmarks from PB'07 evaluation [20]. As such, the results in each cell denote the number of benchmarks SAT/UNSAT/UNKNOWN for each class of instances. The experimental results presented in tables 5.7 and 5.8 correspond to the small integer optimization benchmarks from PB'07 evaluation [20]. Therefore, the results in each cell denote the number of benchmarks OPT/SAT/UNSAT/UNKNOWN for each class of instances.

Table 5.1 presents the results obtained for three different versions of the solver which perform a hybrib learning scheme very similiar to the one performed by *Pueblo*. While version PB1.1 generates the

Table 5.1: Multiple approaches to general PB learning

Benchmark	PB1.1	PB1.2	PB2
armies	**3**/0/9	5/0/7	5/0/7
dbst	15/0/0	15/0/0	15/0/0
FPGA	34/**20**/3	**36**/**20**/1	35/1/21
pigeon	0/11/9	0/**18**/2	0/1/19
progressive party	4/0/2	4/0/2	4/0/2
reduced	14/0/0	14/0/0	14/0/0
robin	3/0/3	3/0/3	**4**/0/2
tsp	40/1/59	40/5/55	40/**43**/17
uclid	1/40/9	1/42/7	1/**43**/6
vdw	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0
Total	147/140/98	151/153/81	151/**156**/78

Table 5.2: Heuristic Backtracking

Benchmark	PB1.2	PB3.1	PB3.2
armies	5/0/7	5/0/7	**4**/0/8
dbst	15/0/0	15/0/0	15/0/0
FPGA	36/20/1	36/21/0	36/20/1
pigeon	0/18/2	0/**19**/1	0/14/6
progressive party	**4**/0/2	1/0/5	1/0/5
reduced	14/0/0	14/0/0	14/0/0
robin	1/0/5	**2**/0/4	1/0/5
tsp	**40**/**5**/55	39/2/59	39/2/59
uclid	1/42/7	1/40/9	1/**49**/0
vdw	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0
Total	**151**/**153**/81	146/150/89	144/153/88

implication graph in a depth-first search way, versions PB1.2 and PB2 do it in a breath-first search way. Additionally, version PB2 performs a final sequence of coefficient reductions in order to eliminate all non-false literals.

On top of version PB1.2 we developed two new versions which perform heuristic backtracking. Version PB3.1 implements the backtracking heuristic introduced in section 4.4 and identified as H1 while version PB3.2 implements H2. Results can be checked in table 5.2.

On top of versions PB1.2 and PB2 we implemented a final cardinality constraint reduction corresponding to versions CARD1.2 and CARD2 respectively. Results can be checked in table 5.3.

In table 5.4 the results of all different learning schemes are presented. Version CL1 corresponds to the original version of the solver in which the implication graph is generated in a depth-first search way, while in version CL2 it is generated in a breath-first search way. In version COMB it is applied an initial classification step in order to select the best fitting learning scheme.

In table 5.5 we present the number of instances that each one of the best versions can solve for each class of instances followed by the total time that it takes to solve them (note that the times corrrespond-

Table 5.3: Cardinality Constraint Learning

Benchmark	PB1.2	CARD1.2	PB2	CARD2
armies	5/0/7	**6**/0/6	5/0/7	**6**/0/6
dbst	15/0/0	15/0/0	15/0/0	15/0/0
FPGA	36/20/1	**36**/**21**/0	35/1/21	**36**/1/20
pigeon	0/18/2	0/**19**/1	0/1/19	0/1/19
progressive party	4/0/2	4/0/2	4/0/2	3/0/3
reduced	14/0/0	14/0/0	14/0/0	14/0/0
robin	3/0/3	2/0/4	**4**/0/2	3/0/3
tsp	40/5/55	**40**/**44**/16	40/43/17	**40**/**44**/16
uclid	1/42/7	1/**43**/6	1/**43**/6	1/42/7
vdw	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0
Total	151/153/81	151/**195**/39	151/156/78	151/156/78

Table 5.4: An overview of the results of all the different learning schemes

Benchmark	CL1	CL2	CARD1.2	PB1.2	PB2	COMB
armies	5/0/7	4/0/8	6/0/6	5/0/7	5/0/7	**7**/0/5
dbst	15/0/0	15/0/0	15/0/0	15/0/0	15/0/0	15/0/0
FPGA	36/2/19	36/2/19	36/**21**/0	36/20/1	35/1/21	36/20/1
pigeon	0/2/18	0/2/18	0/**19**/1	0/18/2	0/1/19	0/18/2
progressive party	**4**/0/2	3/0/3	**4**/0/2	**4**/0/2	**4**/0/2	3/0/3
reduced	14/0/0	14/0/0	14/0/0	14/0/0	14/0/0	14/0/0
robin	3/0/3	**4**/0/2	2/0/4	3/0/3	**4**/0/2	**4**/0/2
tsp	40/33/27	40/**50**/10	40/44/16	40/5/55	40/43/17	40/**50**/10
uclid	1/39/10	1/40/9	1/**43**/6	1/42/7	1/**43**/6	1/42/7
vdw	1/0/4	1/0/4	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0	32/68/0	32/68/0
Total	151/144/90	150/162/73	151/195/39	151/153/81	151/156/78	**153**/**198**/34

ing to the instances that a version of the solver is unable to solve are not considered in the total time).

In table 5.6 the results of the best known solvers can be checked and compared with our best version.

Finally, table 5.7 presents the results obtained for the best versions of the solver, whereas table 5.8 presents the results of the best version of the solver and of several other solvers.

5.2 Results Analysis

5.2.1 Generating the Implication Graph

It is easy to verify that the versions of the solver which generate the implication graph in a breath-first search way exhibit better results than the other versions. This has probably to do with the fact that, as discussed in section 4.1, when generating the implication graph in a breath-first search way, the learned constraint is probably generated using a smaller number of constraints.

Table 5.5: Time results for the best learning schemes

Benchmark	CL2	CARD1.2	PB1.2	PB2	COMB
armies	4/1540	6/2642	5/419	5/3922	7/2400
dbst	15/**2061**	15/2066	15/2070	15/2074	15/2072
FPGA	38/557	57/26	56/34	36/1342	56/15
pigeon	2/501	19/361	18/331	1/334	18/1571
progressive party	3/27	4/17	4/75	4/**12**	3/28
reduced	14/0	14/0	14/0	14/0	14/0
robin	4/**225**	2/2	3/217	4/1722	4/1382
tsp	**90**/46832	84/46589	45/8682	83/45587	90/46895
uclid	41/2597	44/5099	43/5537	44/**4957**	43/4746
vdw	1/185	1/**178**	1/182	1/183	1/181
wnqueen	100/3482	100/3847	100/871	100/3612	100/**782**
Total	312/58005	346/60827	304/18419	307/63746	351/60077

Table 5.6: The Results of other solvers

Benchmark	bsolo	Pueblo	minisat+	PBS4
armies	7/0/5	6/0/6	8/0/4	**9**/0/3
dbst	**15**/0/0	**15**/0/0	7/0/8	**15**/0/0
FPGA	**36**/20/1	**36**/**21**/0	33/3/21	26/21/10
pigeon	0/18/2	0/13/7	0/2/18	0/**20**/0
progressive party	3/0/3	**6**/0/0	5/0/1	3/0/3
reduced	14/0/0	14/0/0	14/0/0	14/0/0
robin	**4**/0/2	3/0/3	**4**/0/2	3/0/3
tsp	40/50/10	**40**/**60**/0	39/46/15	40/52/8
uclid	1/42/7	1/42/7	1/**46**/3	1/44/5
vdw	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0
Total	153/198/34	**154**/203/28	144/165/76	144/**205**/36

5.2.2 Heuristic Backtracking

Both versions of the solver which perform heuristic backtracking presented worse results than the version on top of which they were implemented. These results may have been caused by the fact that in both these versions the algorithm performs a heuristic backtracking step very often (check section 4.4). Since our work with heuristic backtracking was mainly exploratory, these results are preliminary and further study must be carried away.

5.2.3 Eliminating non-false Literals

As explained in section 4.3 the learning scheme implemented in version PB2 is a compromise between general PB learning and clause learning. As such, version PB2 must be compared with version CL2. Version PB2 surpasses version CL2 in 3 classes of instances. However, version CL2 exhibits better results in 3 classes of instances and its overall results are also better than version PB2. This has probably to do with the fact that the additional prunning power obtained by learning general PB constraints using the

Table 5.7: The results of the main versions of the solver for the optimization benchmarks

Benchmark	CL1	CARD1.2	PB1.2
aksoy	25/52/0/2	25/53/0/1	**26**/52/0/1
course-ass	0/5/0/0	1/4/0/0	**2**/3/0/0
domset	0/15/0/0	0/15/0/0	0/15/0/0
garden	1/2/0/0	1/2/0/0	1/2/0/0
haplotype	0/8/0/0	0/8/0/0	0/8/0/0
kexu	0/40/0/0	0/40/0/0	0/40/0/0
logic-synthesis	**52**/22/0/0	51/23/0/0	51/23/0/0
market-split	4/16/4/16	4/16/4/16	**5**/16/4/15
mps-v2-20-10	**14**/13/2/3	12/17/2/1	10/17/2/3
numerical	**12**/18/0/4	**12**/19/0/3	11/19/0/4
primes-dimacs-cnf	69/36/8/17	69/35/8/18	**74**/30/8/18
radar	6/6/0/0	6/6/0/0	6/6/0/0
reduced	17/62/72/122	**33**/55/**99**/86	14/63/70/126
routing	9/1/0/0	**10**/0/0/0	**10**/0/0/0
synthesis-ptl-cmos-circuits	6/2/0/0	6/2/0/0	6/2/0/0
testset	6/0/0/0	6/0/0/0	6/0/0/0
ttp	2/6/0/0	2/6/0/0	2/6/0/0
vtxcov	0/15/0/0	0/15/0/0	0/15/0/0
wnq	0/15/0/0	0/15/0/0	0/15/0/0
Total	223/334/86/164	**238**/331/**113**/125	224/332/84/167

learning scheme implemented in version PB2 does not compensate the additional computational effort concerning the propagation procedure.

5.2.4 General PB learning versus Cardinality Constraint Learning

Version PB1.2 is the best version of the solver which performs general PB learning. This version obtained worse results than version CL2, which implements a clause learning scheme. However, version PB1.2 is dramatically better than version CL2 for some classes of instances, such as the Pigeon-Hole and FPGA benchmarks. Moreover, version PB1.2 is better than version CL2 in the majority of classes of instances. Version CL2 is only drastically better than PB1.2 in the TSP benchmarcks. Therefore, version CARD1.2 seems to be a reasonable compromise between these two versions since it performs as well as version PB2 in the Pigeon-Hole and FPGA benchmarks and almost as well as CL2 in the TSP benchmarks.

5.2.5 An Initial Classification Step

As was expected the version that combines all the best learning schemes (COMB) obtained the best results. However, since there are 27 instances that none of these versions can solve and the best version (CARD1.2) is unable to solve 39 instances, version COMB could not improve the results much more. Nevertheless, this version is able to solve more instances than version CARD1.2 and it even takes less time to do so, which corroborates the quality of the classifier obtained using algorithm C4.5.

Our work concerning the application of machine learning algorithms for selecting the appropriate

Table 5.8: The results of other solvers for the optimization benchmarks

Benchmark	bsolo	PBS4	minisat+	Pueblo
aksoy	**25**/53/0/1	11/63/0/5	**25**/45/0/9	15/64/0/0
course-ass	1/4/0/0	0/4/0/1	**2**/1/0/2	0/5/0/0
domset	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
garden	1/2/0/0	0/3/0/0	1/2/0/0	1/2/0/0
haplotype	0/8/0/0	0/8/0/0	**8**/0/0/0	0/8/0/0
kexu	0/40/0/0	0/40/0/0	**11**/29/0/0	0/40/0/0
logic-synthesis	**51**/23/0/0	19/55/0/0	31/41/0/2	32/42/0/0
market-split	**4**/16/**4**/16	0/20/0/20	0/20/0/20	4/16/4/16
mps-v2-20-10	**12**/17/**2**/1	7/21/1/3	10/15/1/6	10/18/1/3
numerical	12/19/0/3	12/11/0/11	10/9/0/15	**14**/17/0/3
primes-dimacs-cnf	69/35/8/18	69/36/8/17	**79**/26/8/17	75/30/8/17
radar	**6**/6/0/0	0/12/0/0	0/12/0/0	0/12/0/0
reduced	**33**/55/**99**/86	14/77/14/168	17/10/23/213	14/92/18/149
routing	10/0/0/0	4/6/0/0	10/0/0/0	10/0/0/0
synthesis-ptl-cmos-circuits	**6**/2/0/0	0/8/0/0	1/7/0/0	1/7/0/0
testset	**6**/0/0/0	4/2/0/0	5/1/0/0	**6**/0/0/0
ttp	2/6/0/0	2/6/0/0	2/6/0/0	2/6/0/0
vtxcov	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
wnq	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
Total	**238**/331/**113**/125	142/417/23/225	212/264/35/301	184/404/31/188

learning scheme is very preliminary. As future work, we intend to use other attributes in the generation of the decision tree and try different machine learning algorithms.

5.2.6 Optimization Benchmarks

As happened with the non-optimization benchmarks, the cardinality constraint learning scheme was also found the best learning scheme for the optimization benchmarks. However, the gap between the cardinality constraint learning scheme and the original clause learning scheme is lower for the optimization benchmarks than it is for the non-optimization benchmarks. Nevertheless, in the majority of the benchmarks for which none of the different versions was able to provide the optimum solution, version CARD1.2 reached the lowest value of the objective function.

It is important to stress that, in the PB'07 evaluation, *bsolo* was found the best solver on small integer non-optimization benchmarks and, even so, we were able to improve those results.

Chapter 6

Conclusions and Future Work

This thesis was focused on the contribution of SAT solvers to PB solvers, especially on the generalization of conflict-based learning. Thus, besides the clause learning scheme, a broad range of learning methods based on cutting planes were covered and compared with the original clause learning scheme implemented in *bsolo*, a state of the art PB solver.

It is commonly known that, general PB constraints are more expressive than clauses and cardinality constraints. However, the additional pruning power obtained by learning general PB constraints may not compensate the additional effort concerning the propagation procedure. Note that general PB constraints are much harder to propagate than clauses, or even cardinality constraints.

It is not agreed among the research community which is the best learning scheme. In 2007's PB evaluation, *Pueblo* [29], a solver which performs a hybrid learning scheme, was found the best solver on non-optimization benchmarks. There are, however, contradictory results. Chai and Kuehlmann [7] implemented in their solver, *galena*, each one of the main learning schemes. The cardinality constraint learning scheme obtained the best results. Nevertheless, in 2005's PB evaluation *galena* output several wrong answers. Moreover, *galena* uses a watch all literals strategy, thus making the propagation procedure slower, which increases the cost of learning general PB constraints. Additionally, Parrain and Le Berre [5] evaluated their solver, *sat4jPseudo*, with clause learning instead of PB learning and obtained much better results.

Considering the disparity between the results concerning the application of learning techniques in several state of the art PB solvers, the main goal of this work is to help clarifying which is the best learning scheme.

Our results show that cardinality constraint learning is the most effective and robust learning scheme. Moreover, it obtained much better results than the original clause learning scheme both on the small integer non-optimization and optimization benchmarks from PB'07 evaluation. Cardinality constraints are easier to propagate than general PB constraints and are also more expressive than clauses. Therefore, this learning scheme seems a reasonable compromise between general PB learning and clause learning.

As such, these results were not unexpected and confirm the results obtained by Chai and Kuehlmann [7] using their PB solver, *galena*.

Since cardinality constraints and general PB constraints may generate many implications, the order in which those implications are followed affects the conflict analysis procedure, since the learned constraint depends on the structure of the implication graph. It was noted that when generating the implication graph in a breadth-first search way the number of constraints that implied the conflict was generally lower than it was when using a depth-first search approach. Experimental results supported this observation. Nevertheless, there are other possible ways to generate the implication graph that should be explored as future work. For instance, the literals to imply may be maintained in a priority queue, using the activity of each literal as its priority in the queue.

Two new versions of the solver that implement heuristic backtracking were developed and compared with the standard conflict directed backtracking approach. Since these versions obtained worse results than the version on top which they were implemented, this work was not carried away. We plan to resume our work concerning local search techniques, such as heuristic backtracking and restarts, in the future, since they were found to be of the utmost importance for the performance of SAT solvers.

Finally, after processing the experimental results obtained for the best versions of the solver, the C4.5 algorithm was used to generated a decision tree in order to classify any instance given as input according to the learning scheme more appropriate to it. Using this decision tree, an "instance-aware" version of the solver was developed. This version presented better results than those corresponding to the best learning scheme.

Different learning techniques show better results for different classes of instances. As such, using machine learning algorithms to identify the more appropriate learning technique for each instance seems a vibrant are of research. Different instance parameters (attributes) must be tried in the generation of the decision tree and different machine learning algorithms (such as neural networks and support vector machines) must be used and compared.

Bibliography

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Generic ILP versus Specialized 0-1 ILP: An Update. In *Proceedings of the International Conference on Computer Aided Design*, pages 450–457, November 2002.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A Backtrack-Search Pseudo-Boolean Solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 346–353, 2002.

[3] P. Barth. A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science, 1995.

[4] P. Barth. *Logic-Based 0-1 Constraint Programming*. Kluwer Academic Publishers, 1995.

[5] D. Le Berre and A. Parrain. On Extending SAT-solvers for PB Problems. *14th RCRA workshop Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion(RCRA07)*, July 2007.

[6] A. Bhalla, I. Lynce, J.T. de Sousa, and J. Marques-Silva. Heuristic-Based Backtracking for Propositional Satisfiability. In *Procedding of the Portuguese Conference on Artificial Intelligence*, pages 116–130, 2003.

[7] D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. In *Proceedings of the Design Automation Conference*, pages 830–835, 2003.

[8] S. Cook. The Complexity of Theorem Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, May 1971.

[9] T. Cormen, R. Rivest, and C. Leiserson. *Introduction to Algorithms*. MIT Press, Englewood Cliffs, New Jersey, 1991.

[10] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Journal of the Association for Computing Machinery*, 5:394–397, 1962.

[11] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[12] H. Dixon and M. Ginsberg. Inference Methods for a Pseudo-Boolean Satisfiability Solver. In *Proceedings of the National Conference on Artificial Intelligence*, pages 635–640, 2002.

[13] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[14] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006. Special Issue on SAT 2005 competition and evaluations.

[15] E. Golberg and Y. Novikov. BERKMIN: a Fast and Robust SAT-Solver. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 142–149, March 2002.

[16] Y. Gurevich. A SAT Solver Primer. In *Bull. Euro. Ass. Theoret. Comput. Science 85*, pages 112–133, 2005.

[17] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic Restart Policies. In *Proceedings of the National Conference on Artificial Intelligence*, pages 674–681, 2002.

[18] I. Lynce and J. Marques-Silva. Complete unrestricted backtracking algorithms for satisfiability. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 214–221, May 2002.

[19] Y.S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient SAT solver. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 360–375, June 2004.

[20] V. Manquinho and O. Roussel. The First Evaluation of Pseudo-Boolean Solvers (PB'05). *Journal on Satisfiability, Boolean Modeling and Computation*, 2:103–143, 2006. Special Issue on SAT 2005 competition and evaluations.

[21] J. P. Marques-Silva and I. Lynce. Towards Robust CNF Encodings of Cardinality Constraints. *International Conference on Principles and Practice of Constraint Programming*, pages 483–497, September 2007.

[22] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, pages 220–227, November 1996.

[23] M. Moskewicz, C. Madigan, Y. Zhao, and L. Zhang. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

[24] S. Prestwich and C. Quirke. Boolean and Pseudo-Boolean Models for Scheduling. *Second International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2003.

[25] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.

[26] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[27] S. Russel and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall series in Artificial Intelligence. Englewood Cliffs, New Jersey, 1995.

[28] H. Sheini and K. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT Solver. In *Proceedings of the Design and Test in Europe Conference*, pages 684–685, March 2005.

[29] H. Sheini and K. Sakallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:157–181, 2006. Special Issue on SAT 2005 competition and evaluations.

[30] B. M. Smith, S. C. Brailsford, P. M. Hubbard, and H. P. Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Principles and Practice of Constraint Programming - CP'95*, 1:36–52, 1995.

[31] J. Walser. Solving Linear Pseudo-Boolean Constraint Problems with Local Search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 269–274, 1997.

[32] L. Zhang. On subsumption removal and on-the-fly CNF si. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 2005.

[33] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, 2001.