Integration of pseudo-Boolean constraint learning in a state-of-the-art Conflict-Driven Answer Set Solver

---

Bachelor's Thesis

by

Michael Görner

Potsdam University
Institute of Computer Science
Knowledge Processing and Information Systems

Supervisors:
Prof. Dr. Torsten Schaub
Benjamin Kaufmann

Potsdam, September 25, 2012

Görner, Michael
v4hn@cs.uni-potsdam.de
Integration of pseudo-Boolean constraint learning in a state-of-the-art Conflict-Driven Answer Set Solver
Bachelor's Thesis, Institute of Computer Science
Potsdam University, October 2014

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, September 25, 2012

_____

Michael Görner

Zusammenfassung

Antwortmengenprogrammierung (engl. Answer Set Programming/ASP) beschäftigt sich mit der Modellierung von Problemstellungen durch Mengen logischer Regeln und der Suche nach stabilen Modellen(Antwortmengen) der modellierten Probleminstanzen. Um solche Modelle effizient zu finden, nutzen aktuelle Suchverfahren den Resolutionskalkül und lernen aus einmal erkannten Konflikten durch Konfliktanalyse neue Einschränkungen für mögliche Modelle. Allerdings lassen sich in ASP-Modellierungen auch komplexere Strukturen wie Kardinalitäts- und Gewichtsbedingungen nutzen, die sich ähnlich wie Pseudo-Boolesche Ungleichungen verhalten. Da mit dem Cutting Planes System ein für diese Art von Bedingungen leistungsfähigerer Inferenzkalkül bekannt ist, liegt es nahe diesen Kalkül bei der Analyse von Problemen, die solche Konstrukte enthalten, einzusetzen und die resultierende Pseudo-Boolsche Ungleichung zu lernen. Die vorliegende Arbeit beschäftigt sich mit den Möglichkeiten eines solchen Einsatzes, stellt eine Implementierung in den ASP-Solver *clasp* vor und diskutiert Benchmark-Resultate.

Abstract

In Answer Set Programming (ASP) one models problems by sets of logical rules and afterwards looks for stable models (i.e. answer sets) of the modeled problem instances. In order to find such models efficiently state-of-the-art ASP-solvers utilize the resolution calculus to learn new valid constraints from each conflict they hit (Conflict Driven Clause Learning). However, ASP also allows for a number of more complex structures like cardinality and weight constraints, which are essentially similar to linear pseudo-Boolean constraints. While these structures can be weakened to clauses so as to apply resolution to them, a more powerful inference calculus (i.e. the cutting planes system) is known for pseudo-Boolean problems. Therefore, it's worth a try to utilize this calculus in conflict analysis when dealing with these more complex constraints. The resulting pseudo-Boolean constraint can then be learnt just like a normal conflict resolvent. This thesis discusses possible applications of the cutting planes system in conflict analysis, describes an implementation based on the ASP-solver *clasp* and presents benchmarks.

Contents

1 Introduction

1.1 Motivation

In the field of satisfiability testing, i.e. "SAT solving", there have been a lot of major improvements in recent years. Some of these are the 2 watched literals scheme, variable state independent decaying sum(VSIDS) heuristics and conflict clause minimization [9, 14]. Because solving problems modelled with Answer Set Programming (ASP) is in many aspects similar to SAT solving, most of these techniques can be used profitably with ASP solvers as well. One highly optimized solver which implements all mentioned techniques (and many others) is *clasp*, the solver of the open-source ASP suite *potassco* [7].

Another subfield of SAT solving which was a subject of research lately is pseudo-Boolean problem solving and the related technique of pseudo-Boolean Constraint (PBC) learning [5, 1, 2, 6, 11]. Because ASP also includes structures which behave similar to such pseudo-Boolean constraints, it is of interest to see whether or not the corresponding learning technique improves the overall performance of an ASP solver.

1.2 Objective

The objective of this thesis is to extend clasp to learn valid pseudo-Boolean constraints in the context of decision problems, i.e. finding **one** solution of a given problem instance or proving it unsatisfiable, and to evaluate whether or not it is feasible to add PBC learning capabilities to a conflict driven ASP solver. In order to do so, the next chapter first of all formally introduces linear PBCs and reviews a number of relevant properties. It also explains where PBCs can appear when modelling problems with ASP. The following chapter then discusses the *cutting planes system* and *variable elimination* and their usability in conflict analysis. It also discusses problems that arise with this usage. Chapter 4 afterwards describes the actual implementation of this learning mechanism. Finally chapter 5 presents and evaluates benchmarks of this implementation.

2 Linear Pseudo-Boolean Constraints

Pseudo-Boolean constraint problems are also referred to as *0-1 integer programming problems*, as they are a special case of Integer Linear Programming (ILP) problems. An ILP constraint is normally defined for constants $n, a_1, \ldots, a_n$ and a bound b as follows

$$\sum_{0 \le i \le n} a_i x_i \bowtie b \quad , a_i, b \in \mathbb{Z}, x_i \in \mathbb{N}_0$$

where $\bowtie$ is one of $<, \le, =, \ge$ and $>$. However, due to the integral nature of all involved variables and the possibility to multiply the equation by -1 it is enough to restrict the relational operator to $\ge$[1]. Also note that all variables x_i range over all natural numbers.

A linear pseudo-Boolean constraint on the other hand restricts these values to $\{0, 1\}$ so as to parallel the range $\{\bot, \top\}$ of Boolean variables. Because we work with *literals*[2] instead of just variables, we also need to define the negation $\bar{x}$ of a variable x in this setting. Easily enough, the definition $\bar{x} = (1 - x)$ works fine, even though we need to keep the additional 1 in mind later on. Using this definition we can now observe $-ax \ge b \Leftrightarrow -a(1 - \bar{x}) \ge b \Leftrightarrow -a + a\bar{x} \ge b \Leftrightarrow a\bar{x} \ge b + a$. This enables us to further restrict the form of the constraint to allow only positive coefficients on the left-hand side, because we can simply invert all negative coefficients, switch the sign of the corresponding literal and add the new coefficient to the bound. Also, if the right-hand side of a PBC is negative and all coefficients on the left are positive, then this constraint is trivially satisfied and we can replace it by 0. For ease of implementation we also sort the left-hand side products by the size of their coefficients in decreasing order. The rest of this thesis assumes the following form of PBCs and any new constraint that doesn't fit this definition will be normalized.

$$\sum_{0 \le i \le n} a_i l_i \ge b \quad , a_i \in \mathbb{N}^+, b \in \mathbb{N}_0, l_i \in \{x_i, \overline{x_i}\}, i > j \Rightarrow (a_i \le a_j \wedge l_i \ne l_j)$$

where all x_i are valid binary variables.

One more interesting thing to mention is that this form is an obvious generalization of Boolean *clauses*, which emerge when forcing all constants to 1 (or actually any fixed number):

$$\sum_{0 \le i \le n} l_i \ge 1 \quad \Leftrightarrow \quad \bigvee_{0 \le i \le n} l_i$$

However, general linear PBCs are exponentially stronger than clauses. That is to say, in order to represent one PBC an exponential number of clauses is needed relative to the number of involved variables (See section 2.2).

[1] $\sum_{0 \le i \le n}[a_i l_i] > b \Leftrightarrow \sum_{0 \le i \le n}[a_i l_i] \ge b - 1$
$\sum_{0 \le i \le n}[a_i l_i] < b \Leftrightarrow \sum_{0 \le i \le n}[-a_i l_i] > -b$
$\sum_{0 \le i \le n}[a_i l_i] = b \Leftrightarrow \sum_{0 \le i \le n}[a_i l_i] \ge b \wedge \sum_{0 \le i \le n}[a_i l_i] \le b$

[2] A Boolean variable or its negation

2.1 Valid Operations

This section presents some well known ([2, 11, 1]) correct operations on PBCs that are used later on.

Saturation This operation is also named *Coefficient Reduction* in some literature. Presume there is a PBC with $a_i > b$ for some i. Because the corresponding binary factor can only either add a_i to the overall sum - so the constraint is satisfied even if all other literals are false - or make it completely irrelevant for the constraint, it is appropriate to reduce a_i to b. This way the behaviour of the constraint does not change, but the coefficients are always smaller than or equal to the bound of the constraint.

$$\textsc{Sat}\ \frac{\sum_{i\in I} a_i \cdot l_i \geq b \qquad \exists j \in I. a_j > b}{b \cdot l_j + \sum_{i \in I \setminus \{j\}} a_i \cdot l_i \geq b}$$

This irreversibly changes the constraint and therefore removes some information. But take a look at the loss: Even if l_i is mapped to 1 and all other literals of the unchanged constraint remain undecided, the constraint is not just satisfied, but is **oversatisfied** by $a_j - b$. After applying saturation the clause will be satisfied exactly in this case without being oversatisfied. As we'll see in 3.2 oversatisfied constraints cause problems with the conflict analysis, so we'll try to circumvent as many of these problems as possible.

Multiplication and Rounding We are obviously free to multiply and divide the constraint by any constant without a loss of information as long as all resulting constants remain integers. But it is also correct to multiply by **any** factor as long as we round all constants up afterwards. Rounding the constants of a PBC may drop valuable information, but is still correct and can be used effectively in some cases (e.g. saturation actually is a special case of rounding [2]):

$$\textsc{Mul}(\lambda)\ \frac{\sum_{i\in I} a_i \cdot l_i \geq b \qquad \lambda \in \mathbb{R}}{\sum_{i\in I} \lceil \lambda \cdot a_i \rceil \cdot l_i \geq \lceil \lambda \cdot b \rceil}$$

Reduction One more useful operation is to simply forget about a literal of a constraint. This way the resulting constraint tells us what remains to be checked if this particular literal is mapped to 1.

$$\textsc{Red}(l_j)\ \frac{\sum_{i\in I} a_i \cdot l_i \geq b \qquad j \in I}{\sum_{\substack{i\in I \\ i \neq j}} a_i \cdot l_i \geq b - a_j}$$

2.2 Constraint Propagation with PBCs

When applying conflict driven clause learning (CDCL) as described in [12] and unit propagation two of the most important properties an involved constraint must exhibit are the following:

1. Given any partial non-conflicting assignment of involved variables (represented by a set A of all currently true literals) the constraint needs to be able to specify (i.e. propagate) any literals which **has** to be true in order to make the constraint remain satisfiable and non-conflicting.

2. It needs to provide a reason, that is to say a number of currently true literals, why it implies a specific literal.

For normal clauses this is quite easy. Given a clause $\bigvee_{i\in I} l_i$ and a partial assignment A, this clause is said to be unit if and only if there is exactly one $j \in I$ such that neither l_j nor $\bar{l}_j$ is in A and $\forall k \in I\backslash\{j\}.\, \bar{l}_k \in A$. That is to say, all but one literals of the clause are false w.r.t. A. In this case the clause can propagate the remaining undecided literal l_j, because there would be no way left to make the clause true if $\bar{l}_j$ would be set to true. The reason for this propagation is the fact that all other literals of the clause are false, which can be represented by the set $\{\bar{l}_k \mid k \in I\backslash\{j\}\} \subseteq A$. Note that each clause can become unit only **once**, because it is satisfied after the unit literal is propagated.

Now this is not as simple for PBCs. As already mentioned, a PBC can subsum a large number of clauses and more than one of these can become unit over time. Consider the following constraint:

$$c_1 : 2x_1 + 1x_2 + 1\bar{x}_3 \geq 3$$

Given the empty partial assignment $A = \varnothing$ the constraint can propagate x_1 because the inequality would be violated in any case if x_1 would be mapped to 0 (i.e. $\bar{x}_1$ would be mapped to 1). If we then also add $\bar{x}_2$ to the assignment (so $A = \{x_1, \bar{x}_2\}$), the constraint implies yet another assignment: $\bar{x}_3$, because again there would be no way left to satisfy the constraint if $\bar{x}_3$ would be mapped to 0. Now the constraint actually *is* satisfied and can't propagate any more literals w.r.t. $A = \{x_1, \bar{x}_2, \bar{x}_3\}$ (or any non-conflicting extension of it).

In order to formalize this behaviour, people came up with the following value (called *slack* or *poss* [1, 11]) associated with a partial assignment A and a constraint $c : \sum_{i\in I} a_i l_i \geq b$.

$$s_c^A = \sum_{\substack{i\in I \\ \bar{l}_i \notin A}} a_i - b$$

This value represents the amount by which c is oversatisfied if all undecided literals in this constraint would be mapped to 1. This way a literal assignment is implied for an undecided literal l_j if and only if $a_j > s_c^A$ for the corresponding coefficient a_j. This is, as already mentioned above, because if l_j would be mapped to 0 there would be no way left to satisfy the constraint. Also it is important to see that a negative slack marks a constraint as violated under the current assignment. Using this knowledge we can exactly specify the set of clauses that represent a PBC c.

$$c : \sum_{0\leq i\leq n} a_i l_i \geq b \quad \Leftrightarrow \quad \left\{ \bigvee_{i\in I} l_i \mid I \subseteq \{0,\ldots,n\},\, s_c^{\{\bar{l}_i \mid i\in I\}} < 0 \right\}$$

Now that we know how to detect whether a literal is propagated by the constraint (i.e. one of the subsumed clauses is unit), we still need to provide a reason for this propagation. A valid

reason that is easy to compute is the set of all currently false literals of the constraint. Because the coefficient of each of these literals was subtracted from the initial slack of the constraint, the current slack is smaller than the coefficient of the propagated literal. However, note that this is not necessarily the strongest (in the sense of smallest) reason for this propagation as the following example illustrates: Given the constraint $c : 3x_1 + 2x_2 + 1x_3 + 1x_4 \geq 3$ and the partial assignment $A = \{\overline{x}_4, \overline{x}_1\}$ the constraint propagates x_2 (as well as x_3 later on) with the reasonset A. The strongest reason for propagating x_2 is only $\{\overline{x}_1\}$ though. Nevertheless, for ease of computation, clasp as well as the implementation in this thesis uses the first way to compute the reason.

2.3 Weight Constraints in clasp

Normally ASP does not directly relate to pseudo-Boolean problem solving or pseudo-Boolean constraints. Even so, the authors of [8] use PBCs to capture the behaviour of *weight constraints* (as defined by [13]) in clasp. An ASP weight constraint is actually very similar to a linear PBC with one crucial difference: The constraint itself is handled like a Boolean variable and **can become false**. A weight constraint consists of a lower bound ℓ, a number of atoms $a_0, \ldots, a_n$ and weights $w_0, \ldots, w_n$ associated with the atoms:

$$\ell \{a_0 = w_0, \ldots, a_m = w_m, \text{not } a_{m+1} = w_{m+1}, \ldots, \text{not } a_n = w_n\}$$

The literal W, which represents the truth value of this weight constraint, has to be mapped to 1 if the sum of weights of true atoms and false negated atoms reaches the lower bound ℓ and has to be mapped to 0 otherwise. clasp represents this logical connection by the following dual PBCs.

$$\begin{aligned} PB_{\omega}: &\quad \ell \cdot \overline{W} + w_0 \cdot a_0 + \ldots + w_m \cdot a_m + w_{m+1} \cdot \overline{a}_{m+1} + \ldots + w_n \cdot \overline{a}_n \geq \ell \\ PB_{\overline{\omega}}: &\quad (s^{\{W\}}_{PB_{\omega}} + 1) \cdot W + w_0 \cdot \overline{a}_0 + \ldots + w_m \cdot \overline{a}_m + w_{m+1} \cdot a_{m+1} + \ldots + w_n \cdot a_n \geq (s^{\{W\}}_{PB_{\omega}} + 1) \end{aligned}$$

These constraints restrict the possible assignments of $W, a_0, \ldots, a_n$ such that $\overline{W}$ has to be mapped to 1 if the sum of weights of true literals does not exceed the lower bound ℓ (PB_{ω}) and W has to be mapped to 1 if the sum of false literals of the weight constraint does not exceed the maximum slack of the weight constraint ($PB_{\overline{\omega}}$). Once the value of W is known, they also assure the sum of true literals in the weight constraint is greater or equal or respectively less than the bound ℓ.

Therefore, propagations because of weight constraints in clasp are actually propagations because of one of these two associated pseudo-Boolean constraints and the reasons provided are reasons computed from these constraints.

By forcing the literal W to 1, this concept degenerates to a normal pseudo-Boolean constraint and clasp uses this to solve pseudo-Boolean problems using CDCL technology as well.

3 Cutting Planes Inference

Just as linear pseudo-Boolean constraints are a proper generalization of Boolean clauses the corresponding inference system (i.e. the *cutting planes system* [3]) is a proper generalization of the resolution calculus ([10]). Let's take the following resolution inference in propositional logic as an example:

$$\textsc{Res}(l)\ \frac{(a \vee b \vee l) \qquad (c \vee b \vee \bar{l})}{(a \vee b \vee c)}$$

This inference is isomorphic to simply adding the inequalities corresponding to these clauses and saturating the result.

$$\begin{array}{l} \textsc{Add}\ \dfrac{1a + 1b + 1l \geq 1 \qquad 1c + 1b + 1(1-l) \geq 1}{2b + 1a + 1c \geq 1} \\ \textsc{Sat}\ \dfrac{}{1b + 1a + 1c \geq 1} \end{array}$$

A linear combination of a number of PBCs is called a *cutting plane*, as it "cuts off" a part of the reachable search space without removing any 0-1 solution of the overall set of constraints. This also gives rise to the primary inference rule for PBCs:

$$\text{CP}(\alpha, \beta)\ \frac{\sum_{i \in I}[a_i l_i] \geq b_1 \qquad \sum_{j \in J}[c_j l'_j] \geq b_2 \qquad \alpha, \beta \in \mathbb{N}}{\sum_{i \in I}[\alpha \cdot a_i l_i] + \sum_{j \in J}[\beta \cdot c_j l'_j] \geq b_1 + b_2}$$

Keep in mind that the resulting constraint probably has to be renormalized and potentially saturated, because if for some $i \in I, j \in J$ there is $l_i = \overline{l'}_j$ then negation as defined in the previous chapter leaves an amount of $\min(a_i, c_i)$ on the left-hand side after adding the summands, that needs to be subtracted.

This thesis only uses CP so as to remove a literal from a PBC. Therefore we will always use $\text{CP}(\alpha, \beta)$ with two PBCs which contain summands $c_1 \cdot l$ and $c_2 \cdot \bar{l}$ respectively such that $c_1 \cdot \alpha = c_2 \cdot \beta$. This usage is also dubbed *variable elimination (VE)* [11].

3.1 CP Conflict Analysis

Let's consider a simple search conflict involving these two constraints:

$$\begin{array}{llllllll} (c_1) & 3x_1 & +2x_2 & +2x_3 & +1x_4 & & \geq 3 \\ (c_2) & 3x_2 & +2x_5 & +1\bar{x}_1 & +1x_3 & +1x_6 & \geq 3 \end{array}$$

Figure 3.1 demonstrates a possible search path leading to a conflict and gives the reasons for the propagated literals as described in section 2.2. At first the decision is made to propagate $\bar{x}_6$

constraint	reason	literal	decision level
decision	$\varnothing$	$\rightarrow \overline{x}_6$	@1
decision	$\varnothing$	$\rightarrow \overline{x}_2$	@2
c_2	$\{\overline{x}_6, \overline{x}_2\}$	$\rightarrow x_5$	@2
decision	$\varnothing$	$\rightarrow \overline{x}_3$	@3
c_1	$\{\overline{x}_2, \overline{x}_3\}$	$\rightarrow x_1$	@3
c_2	$\{\overline{x}_6, \overline{x}_2, \overline{x}_3\}$	$\rightarrow \overline{x}_1$	@3
$\lightning$			

Figure 3.1: search history leading to a conflict

(thus entering the first decision level[1]). This does not trigger any further propagation, so the next decision is made on the next decision level and forces $\overline{x}_2$. This time the current slack of (c_2) reduces to 1 and therefore the constraint propagates x_5. Because again no conflict was found and no further literal is propagated yet another decision is made on the next level and this time forces $\overline{x}_3$. Because of this the slacks of (c_1) and (c_2) reduce to 1 and 0 resp. and both constraints propagate complementary literals leading to a conflict.

A normal CDCL conflict analysis now proceeds by converting the conflicting implication $(\overline{x}_6 \wedge \overline{x}_2 \wedge \overline{x}_3 \rightarrow \overline{x}_1)$ to a clause $(x_6 \vee x_2 \vee x_3 \vee \overline{x}_1)$ and resolves out literals in reverse order of propagation. To do so resolution is applied to the conflict clause and the clauses which correspond to the particular implications until only one literal on the current decision level is left:

$$\textsc{Res}(x_1)\ \frac{(x_6 \vee x_2 \vee x_3 \vee \overline{x}_1) \qquad (x_2 \vee x_3 \vee x_1)}{(x_6 \vee x_2 \vee x_3)}$$

The resulting clause is called the first unique implication point (First-UIP) clause [12]. This clause is then learnt and the search jumps back to the first level the clause is unit (i.e. the First-UIP level) and propagates the unit literal (in this case the search jumps back to the second decision level and the clause propagates x_3).

What happens if we apply CP to the actual PBCs instead of just RES to their weakened implications?

$$\begin{array}{rl} \text{CP}(3,1) & \dfrac{3x_2 + 2x_5 + 1\overline{x_1} + 1x_3 + 1x_6 \geq 3 \qquad 3x_1 + 2x_2 + 2x_3 + 1x_4 \geq 3}{11x_2 + 6x_5 + 5x_3 + 3x_6 + 1x_4 \geq 9} \\ \textsc{Sat} & \dfrac{}{9x_2 + 6x_5 + 5x_3 + 3x_6 + 1x_4 \geq 9} \end{array}$$

The produced constraint also is unit on decision level 2 and also propagates x_3 there. However, it is strictly stronger than the First-UIP clause because it actually subsums four clauses:

$$9x_2 + 6x_5 + 5x_3 + 3x_6 + 1x_4 \geq 9 \Leftrightarrow \begin{Bmatrix} x_2 \vee x_3 \vee x_5, & x_2 \vee x_5 \vee x_6, \\ x_2 \vee x_4 \vee x_5, & x_2 \vee x_3 \vee x_6 \end{Bmatrix}$$

Especially note the clauses $x_2 \vee x_3 \vee x_6$ and $x_2 \vee x_4 \vee x_5$. Whereas resolution based conflict analysis yields one of the four clauses and another one is subsumed by (c_2), the search didn't know

[1] the number of a decision level simply represents the amount of uninformed decisions propagated up to this point

$$(c_1') \quad 2x_3 \quad +1x_1 \quad +1x_2 \quad \geq 2$$
$$(c_2') \quad 2\bar{x}_3 \quad +1x_1 \quad +1x_2 \quad \geq 2$$

constraint	reason	literal	decision level
decision	$\varnothing$	$\rightarrow x_1$	@1
decision	$\varnothing$	$\rightarrow \bar{x}_2$	@2
c_1'	$\{\bar{x}_2\}$	$\rightarrow x_3$	@2
c_2'	$\{\bar{x}_2\}$	$\rightarrow \bar{x}_3$	@3
$\lightning$			

Figure 3.2: search history leading to a conflict with oversatisfied constraints

about these two clauses before and the analyzed conflict has nothing to do with them. In fact, learning this new PBC which the CP analysis produced enables the search to avoid conflicts it never actually hit. This is a tremendous advantage of CP based analysis over resolution based analysis. Also [3] proofs that there exists at least one class of unsatisfiable problem instances (i.e. the well known *pigeonhole problems*) which resolution based algorithms can't refute in a polynomial number of steps, but which CP based algorithms can refute polynomially.

However, there are a number of disadvantages as well and some of them are discussed in the next section.

3.2 Problems to Overcome

3.2.1 Keeping Constraints Violated

Let's take a look at another example presented in figure 3.2. The search decides to force x_1 and $\bar{x}_2$ and hits a conflict involving x_3. As explained in the previous section we can now analyze this conflict with the CP method:

$$\text{CP}(1,1)\ \frac{2\bar{x}_3 + 1x_1 + 1x_2 \geq 2 \qquad 2x_3 + 1x_1 + 1x_2 \geq 2}{2x_1 + 2x_2 \geq 2}$$

However, the resulting constraint doesn't conflict with the current assignment and does not propagate literals on any earlier decision level. Therefore we can't use it to continue the search on an earlier decision level and remove the current assignment from the search space. What happened? Taking another look at the definition of slack in 2.2 it becomes clear that when adding two PBCs which do not contain unassigned complementary literals, the corresponding slacks add up as well. Because the constraint implying x_3 is oversatisfied ($s_{c_1'}^{\{x_1,\bar{x}_2,x_3\}} = 1$) and the conflicting constraint is only violated by one ($s_{c_2'}^{\{x_1,\bar{x}_2,x_3\}} = -1$) the resulting slack of the resolvent is 0 and hence it is not violated. To keep the resolvents violated in conflict analysis the authors of [2] propose to use the reduction operation defined earlier to reduce the slack of the implying constraint. To do so, they apply the reduce operation to remove non-false (and non-implied) literals until the sum of slacks of both constraints remains negative. This is guaranteed to suffice, because the repeated application of the reduction operation eventually results in the unit clause which implied the literal

and unit clauses by definition can't have a slack greater than 0. The following is an analysis which includes such a weakening:

$$\dfrac{\dfrac{2\bar{x}_3 + 1x_1 + 1x_2 \geq 2 \qquad \dfrac{\dfrac{2x_3 + 1x_1 + 1x_2 \geq 2}{2x_3 + 1x_2 \geq 1}\,\text{RED}(x_1)}{1x_3 + 1x_2 \geq 1}\,\text{SAT}}{3x_2 + x_1 \geq 2}\,\text{CP(1,2)}}{2x_2 + x_1 \geq 2}\,\text{SAT}$$

The resulting constraint is still violated on the current decision level and implies x_2 on the previous level. However, note that without further restrictions the resulting constraint might remain violated, but does **not** need to be implicative on an earlier level [11].

Instead of iteratively weaken the constraint it is also sound to simply replace it by the implying clause and this is the approach used in the implementation presented in this thesis. In order to detect whether weakening is needed, different approaches were used in the past. [5] checks whether or not one of the coefficients is equal to 1 and if not calculates a weakened version together with the original constraint. [11] applies CP and reverts it if a non-negative slack is detected afterwards. The implementation described here checks if the sum of the slacks is non-negative and applies weakening if it is not. This is conservative though, because if there are unassigned complementary literals in the involved constraints, the smaller of both coefficients is counted twice as indicated above. So the actual slack of the resolvent is always smaller or equal the sum of the slacks of the CP operands.

3.2.2 Overhead

An obvious disadvantage of the cutting planes analysis is the additional overhead that is needed to save and process the coefficients of the literals. Normal resolution based analysis only needs to mark literals in order to find the First-UIP clause, but CP analysis requires the multiplication and addition of coefficients and space to save them. This is the nature of cutting planes and there's no way to change these requirements.

A much bigger problem is assignment propagation taking longer. Whereas propagation on clauses exploits the highly efficient 2-watched-literals scheme [9], there is no real equivalent for PBCs. In order to find out whether or not a PBC c is implicative, we need to calculate the current slack s_c^A and check if for some coefficient a_i of an **undecided** literal in c there is $a_i > s_c^A$ - this is simplified by sorting the literals by their coefficient. The most basic approach just watches all literals in c and subtracts the respective coefficient from the slack, whenever one of them is set to false. It remains to check each time, if there's an undecided literal with a greater coefficient. This technique is labelled the "counted" approach and is implemented in clasp as well as this thesis. Profiling reveals that this technique uses between 10-40% of the overall runtime of the solver, which really is a lot. Therefore it is advisable to look for alternatives here.

Also, watching all literals of a constraint results in strongly growing watch lists and contradicts the idea of a watched literals scheme. Watch schemes rely on watching always just enough literals to be sure that the constraint can't be implicative.

In order to be sure that a PBC is not implicative, one needs to watch a set of true or undecided

literals W, such that the sum of coefficients of these literals (let's call this S_W) is greater or equal to the sum of the bound and the largest coefficient of an undecided literal $a^U_{max} = \max\{a_i \mid \bar{l}_i \notin A\}$.

$$S_W \geq b + a^U_{max} \tag{3.1}$$

Techniques based on this idea are labelled "watched" approaches.

This theoretically sounds fine, but it was found to be mostly inferior to the counted approach by the authors of [1] and [2]. One big problem with this approach is that the value of a^U_{max} might change between **each** propagation and one therefore needs to look it up each time. In effect this technique needs to look at all (undecided) literals each time a watched literal is propagated.

A possible improvement concerning this problem is proposed by [11]. Instead of watching just enough literals to satisfy (3.1), they relax this requirement and use the largest coefficient of c instead of the largest one of an undecided literal.

$$S_W \geq b + a_{max} \tag{3.2}$$

This way they most likely end up watching more literals than needed, but they don't need to calculate a^U_{max} anymore. As a result propagation became "considerably faster" as the authors mention, though they do not compare it to an implementation of the counted approach. Also this scheme possibly degenerates into a counted approach for a large number of constraints as described in section 2.3, because these contain exactly one coefficient equal to the lower bound. It therefore still needs to be evaluated whether such a scheme proofs superior.

Another issue is that PBCs produced by CP analysis tend to include a number of clauses that are also subsumed by the operands (e.g. in 3.1 the clause $x_2 \vee x_3 \vee x_5$ is subsumed by (c_2)). Hence, the constraint database becomes highly redundant instead of storing only distinct clauses and a lot of constraints quickly become obsolete and remain pure overhead in propagation. Therefore people proposed to drop learnt PBCs after a rather short time period ([11]) or don't even learn the derived PBC but only an extracted violated cardinality constraint ([2]). Because of this redundancy another deletion scheme which might reveal itself profitable is to drop learnt PBCs after they where involved in a set number of conflicts, because the new learnt constraints probably subsum a high number of clauses of the old constraint.

3.2.3 When to Stop the Analysis

The conflict analysis should stop as soon as the resolved constraint becomes implicative on an earlier (in the best case the earliest possible) decision level. While a resolution based analysis exploits that the First-UIP is easy to recognize (only one literal in the resolvent is assigned on the conflict level), there is no such easy test known for PBCs. So in order to check whether or not the constraint is implicative, the solver needs to compute the slack of the constraint on each earlier decision level **after each** cutting plane step. This is computationally expensive and the authors of [1] mention their solver spends as much as about 50% of its running time here.

3.2.4 Learnt Constraint(s) and Backjumping

As explained above, because learning one PBC for each conflict adds an immense overhead to propagation and uses a lot of space, one might prefer to remove these constraints again after a short period of time or not even learn them at all but learn only a weakened cardinality constraint instead. However, when discarding constraints which were produced in conflict analysis the solver risks to hit the same conflict again later in the search and it is not guaranteed that the search will eventually terminate[2].

That's one reason why the SAT solver *Pueblo* ([11]) learns not only a PBC, but the relevant conflicting clause as well. Therefore the analysed conflict is still avoided after the PBC is dropped. Based on the insight that the set of false literals in a PBC derived in CP analysis corresponds to the currently derived conflicting clause it can also be avoided to run both analyses separately. It is enough to extend the resolution analysis such that it also applies CP inference steps as needed. Because such an analysis ends when the First-UIP clause is found, the generated PBC is not necessarily implicative on any earlier decision level as mentioned in section 3.2.1. Hence, a couple of different scenarios are possible when the solver tries to continue the search after the analysis. Examples of these different scenarios can be found in [11].

If the derived PBC is implicative on some decision level then the solver can jump back to the earlier of this level and the First-UIP level and continue the search by propagating the respective literal(s).

If, however, the constraint is **not** implicative, then two more scenarios are possible: Either the constraint is satisfiable but not unit on the First-UIP level, in this case the search can continue on this level using the conflict clause, or the constraint is still violated. *Pueblo* handles this later case by jumping back to the last level where both constraints are satisfied and taking a new uninformed decision.

The implementation presented in this thesis uses the above approach with one significant change: In case the derived PBC is still violated on the First-UIP level, but is not implicative on an earlier level, this is handled as a conflict on the First-UIP level which again is analyzed. Note that this scenario seems to happen only rarely though.

[2] Due to hardware limitations solvers normally have to discard learnt conflict clauses as well. However, PBCs would have to be discarded much faster in order to keep the overhead low.

Algorithm 1 PBConstraint interface

```
class PBConstraint : LearntConstraint {
    PBConstraint(Solver& s, Literal p, Antecedent& a, bool conflict);

    // p was just set to true - propagate consequences
    bool propagate(Solver&, Literal p);

    // Write reason for propagating p earlier to lits
    void reason(Solver&, Literal p, LitVec& lits);

    // Eliminate variable from constraint using VE
    void varElimination(Solver& s, Literal lit);

    // Normalize & Saturate constraint
    int64 canonicalize(Solver& s);

    // Weaken constraint to relevant clause
    void weaken(Solver& s);

    // Integrate into current search
    bool integrate(Solver& s);

    // Is literal at position idx known to be false
    bool litSeen(uint32 idx);
    void toggleLitSeen(uint32 idx);

    // update slack + watches + undo stack
    void updateConstraint(Solver& s, Literal p);

    WeightLitVec lits_; // literals of constraint
    int64   bound_;     // lower bound
    int64   slack_;     // slack of PB constraint

    uint32   up_;       // size of undo stack
    UndoInfo* undo_;    // undo stack
}
```

4 Implementation

First of all please note that this chapter discusses only a part of the implementation and omits/ simplifies a number of less relevant details/optimizations for ease of understanding. If you are interested in the full implementation please read the actual source code.

In order to extent clasp to use pseudo-Boolean constraint learning the most important thing to add was a new class of learnt constraints which represents pseudo-Boolean constraints. The previous page presents an outline of the implemented class `PBConstraint`. In many ways this class mimics the already existing class `WeightConstraint` which is used for weight constraints as discussed in section 2.3, though a number of adjustments were required because this class does not implement the `LearntConstraint` interface. An instance of the class consists of a vector of weight literals `lits_` (i.e. pairs of a literal and a weight), a lower bound `bound_` and the current slack `slack_`. In order to provide reasons as discussed in section 2.2, each instance also maintains a stack `undo_` of `UndoInfo` objects. Each of these provides the local index (via a method `idx()`) of a literal propagated as false during search.

The first interesting thing about this design is the argument list of the constructor. Whereas one would probably expect the lower bound and a list of literals, a `PBConstraint` is constructed from a `Literal` and an `Antecedent`, which is the clasp abstraction for objects providing reasons. However, because PBCs are only created from conflicts and in conflict analysis and this implementation does not make use of an additional builder class, it is justified to construct a PBC from a forced literal and its reason. The additional flag `conflict` is required, because conflicts are detected the moment a false literal is forced to true and the conflicting constraint did not yet update its slack accordingly. The conflicting PBC needs to have a negative slack though.

Algorithm 2 PBConstraint::propagate() and PBConstraint::reason()

```
1  bool PBConstraint::propagate(Solver& s, Literal p){
2      updateConstraint(s, p);
3      uint32 reasonData= up_;
4      for (uint32 n= 0; n < size() && weight(n) > slack_; ++n) {
5          if (!litSeen(n) && !s.force(lit(n), this, reasonData)) {
6                  return false;
7          }
8      }
9      return true;
10 }
11
12 void PBConstraint::reason(Solver& s, Literal p, LitVec& lits){
13     uint32 stop = s.reasonData(p);
14     for (uint32 i= 0; i != stop; ++i) {
15         lits.push_back( ~lit(undo_[i].idx()) );
16     }
17 }
```

The algorithms for propagation and computation of valid reasons for resolution are listed in algorithm 2. Because the PBC watches all negations of literals it contains, the solver calls `propagate` with the respective literal whenever one of these becomes true (i.e. the corresponding literal of the constraint becomes false). `updateConstraint` then removes the coefficient of that literal from the current slack, calls `toggleLitSeen` and adds an `UndoInfo` for that literal to the undo stack. Because the slack decreased, the code now checks whether some coefficient of a non-false literal is greater than the new slack and forces (i.e. schedules for propagation) all such literals (`force` simply ignores all requests to force literals which are already true). When forcing some literal, the solver notes down the current size of the undo stack in `reasonData`, so the constraint can look up that value when it is requested to provide a reason for the implication via `reason`. The reason is then build up from the lower part of the undo stack up to this size.

In order to run the CP analysis in addition to the normal resolution based analysis, the solver keeps the current CP resolvent in the property `aggregator_` which just points to a `PBConstraint` object. This property is initialized with a PBC derived from the conflicting constraint whenever a conflict is found in `setConflict()`. During the resolution based conflict analysis `analyzeConflict()` calls `aggregator_->varElimination()` for each literal that is about to be resolved out and that is present in the aggregator. The latter condition is due to the fact that a single CP step can actually remove more than one literal, and therefore can remove literals from the aggregator which are still present in the corresponding resolution resolvent.

Algorithm 3 PBConstraint::varElimination()

```
1  void PBConstraint::varElimination(Solver& s, Literal lit){
2      PBConstraint eliminator(s, lit, s.reason(lit));
3
4      weight_t mel,mag;
5
6      get_multipliers(lit, *this, eliminator, mel, mag);
7      weaken_on_overflow(mel, *this);
8      weaken_on_overflow(mag, eliminator);
9      get_multipliers(lit, *this, eliminator, mel, mag);
10
11     if (mag*eliminator.slack_ + mel*this->slack_ >= 0 ){
12         eliminator.weaken(s);    mel= 1; mag= this->weight(~lit);
13     }
14
15     multiply(mel);
16     eliminator.multiply(mag);
17
18     // this might be overestimated and is adjusted via canonicalize
19     slack_= eliminator.slack_ + slack_;
20     bound_= eliminator.bound_ + bound_;
21
22     lits_.append(eliminator.lits_);
23     slack_-= canonicalize(s);
24 }
```

After the conflict analysis is finished, the solver jumps back to the First-UIP level of the derived conflict clause and learns this clause in `resolveConflict()`. If the derived PBC possibly subsums more than this conflict clause (i.e. the bound of the aggregator is greater 1), it then calls `aggregator_->integrate()`, which adds the aggregator to the solver as well. This can trigger a new conflict though and therefore `resolveConflict()` repeats the conflict analysis until no further conflict is found or the problem is proven to be unsatisfiable. The particular algorithms are (simplified) listed in Appendix A and the rest of this section concentrates on the procedures implementing variable elimination and constraint integration.

`varElimination()` (listed in algorithm 3) first of all builds a new `PBConstraint` object called `eliminator` from the literal `lit` that is supposed to be eliminated and its `Antecedent`. If the propagating constraint already is a acPBC, that constraint is copied, if it is a weight constraint, the relevant PBC is extracted and else a pseudo-Boolean representation of the implying clause is used.[1] Next the required multipliers for eliminating `lit` are calculated. If the multiplication or the addition of the constraints could possibly cause an integer overflow, the respective constraint is weakened to the relevant clause.

This is a very primitive overflow handling though, because in many cases it should be possible to apply the multiplication and/or weakening operations defined in section 2.1 and keep most of the information of the constraint.

However, applying multiplication with a multiplier $0 < \lambda < 1$ is only possible if the implicit rounding does not significantly change the slack of the constraint and weakening involves more computation and might still produce the relevant clause. Also the eliminator is weakened if the slack of the resulting constraint could become non-negative. The relevant logic was already discussed in section 3.2.1.

At last the constraints are multiplied and added up together. To do so, the slacks and bounds are added up and the weighted literals of the `eliminator` are appended to the weighted literals of this constraint. The following call to `canonicalize()` renormalizes and saturates the constraint and adjusts the slack by the amount described in 3.2.1.

The `integrate()` function listed on the next page in algorithm 4 is called for `aggregator_` once the conflict analysis finished and the solver integrated the conflict clause on the First-UIP level. The key idea of this function is to initialize the constraint as if all literals were currently undecided and then manually propagate all false literals. In order to do so the first loop calculates the maximal slack of the constraint, sets up all required watches and notes down all false literals. Afterwards the constraint is added to clasps constraint database.

The crucial part of the integration is to get the `UndoInfo` objects corresponding to false literals in the right order so as to propagate them in the same way the solver originally propagated them. Because sorting the literals by their appearance in the propagation trail of the solver is computationally expensive and cache intense, the code takes advantage of the facts that clasp always undoes **whole** decision levels instead of single assignments and that reasons stay valid if additional true literals are added to them.

[1] Be aware of the fact that this actually allocates memory which is really time consuming and should be avoided if possible.

Keeping these things in mind, it is enough to sort the objects by the decision level of their respective literals. At last the false literals are propagated one by one and if the constraint is found to be implicative at some point, the relevant literals are forced **on the level of the last propagated false literal**. This might undo further decision levels and can even produce a new conflict as explained earlier.

When either all currently false literals are propagated or the search jumped back to an earlier level to force some implied literal the constraint is fully integrated and the function returns true unless a new conflict was found.

Algorithm 4 PBConstraint::integrate()

```
1  bool PBConstraint::integrate(Solver& s){
2      uint32 todo= 0;
3      slack_      = -bound_;
4
5      for (i= 0; i < lits_.size(); ++i){
6          slack_+= weight(i);
7          s.addWatch(~lit(i), this);
8          if (s.isFalse(lit(i))) {
9              undo_[todo++]= UndoInfo(i);
10         }
11     }
12     s.addLearnt(this, lits_.size());
13
14     sort_by_decision_level(undo_, undo_+todo);
15
16     for (uint32 n= 0, thisDL= 0; n < size(); ++up_) {
17         while(n != end && weight(n) > slack_){
18             // constraint is unit on thisDL
19             if (!litSeen(n) && !s.force(lit(n), thisDL, this, up_)) {
20                 return false;
21             }
22             n++;
23         }
24
25         if (up_ == todo || !s.isFalse(lit(undo_[up_].idx()))) {
26             break;
27         }
28
29         uint32 idx = undo_[up_].idx();
30         Var    var = lit(idx).var();
31         thisDL     = s.level(var);
32         slack_    -= weight(idx);
33         toggleLitSeen(idx);
34     }
35     return true;
36 }
```

Benchmark Class	clasp-default			clasp-cp-default		
	time(to)	conflicts	choices	time(to)	conflicts	choices
Fastfood	19.0(0)	220,043	253,734	2,873.0(1)	26,084	29,240
WeightBoundDS	5,271.3(2)	144,763,680	230,718,880	23,501.6(16)	285,133	438,468
WireRouting	1,625.2(1)	3,932,646	5,632,039	8,744.5(6)	547,548	841,043
KnightTour	1,010.2(0)	591,340	25,380,012	217.3(0)	77,483	4,405,257
GraphPart	134.4(0)	2,307,040	3,153,471	2552.1(2)	1,086,051	1,254,199
HardwareASP	39,394.1(28)	132,446,099	334,359,106	52,712.5(43)	4,060,163	50,531,961
PBC11	86,342.3(64)	122,475,616	396,239,900	107,423.8(85)	33,748,832	94,916,266
Pigeonhole	4,170.9(3)	430,446,762	456,478,172	0.1(0)	60	335
Benchmark Class	**clasp-pbc12**			**clasp-cp-pbc12**		
	time(to)	**conflicts**	**choices**	**time(to)**	**conflicts**	**choices**
Fastfood	9.6(0)	303,202	344,813	808.4(0)	47,577	53,333
WeightBoundDS	14,000.3(11)	440,026,352	767,239,789	26,541.8(20)	168,920	209,592
WireRouting	96.1(0)	365,868	699,240	9,734.6(4)	545,304	865,006
KnightTour	4.7(0)	19,945	423,959	2.9(0)	2,721	281,791
GraphPart	1,255.2(1)	33,294,762	42,493,026	2,308.5(1)	3,574,126	3,973,199
HardwareASP	1,595.3(1)	5,186,675	34,923,926	15,692.1(10)	285,534	20,714,946
PBC11	84,655.5(64)	284,632,309	2,085,980,173	96,682.9(68)	144,304,222	972,595,931
Pigeonhole	4875.3(4)	158,373,919	167,742,662	0.05(0)	60	340

Table 5.1: Benchmarks with default and PBC12 configuration

5 Experimental Analysis

5.1 Benchmarks

In order to benchmark the effects of PBC learning a variety of problem classes were evaluated. This includes a number of ASP problem classes[1] which include many weight or cardinality constraints and a sample of 100 (small integer) pseudo-Boolean problems taken from the *PB competition 2011*. The experiments were conducted on Intel Xeon 2.26GHz machines with 48GB of RAM running Linux - Kernel 2.6.18. and each run was given up to 3GB of RAM and 1200 seconds before timeout (to).

The measured solver configurations include clasp using its default configuration (*-default*) and the configuration which was used in the PB competition 2012 (*-pbc12*) as well as the corresponding configurations using the described cutting planes conflict analysis. The above table 5.1 lists for each problem class the summed number of conflicts, choices, the runtime and timeouts, which are accounted for by the maximal runtime of 1200 seconds.

5.2 Evaluation

First of all it is easy to see that the cutting planes implementation produces many timeouts in the presented benchmark and hence the reported numbers of conflicts and choices have to be

[1] most of them taken from the *ASP competition 2009* [4]

evaluated with care[2]. However, only part of the bad runtime performance of the system is due to the inherent complexity of the cutting planes analysis. For example profiling revealed, that the CP solver spends up to 90% of its runtime in the function `varElimination` described in algorithm 3 when solving instances of *FastFood*, *WeightBoundedDominatingSet* or *HardwareASP*. This is likely due to the fact that this function (which is run at minimum 100 times per second) actually allocates memory in order to provide a common interface for the constraint which implies `lit`. An implementation using a builder class which keeps the current resolvent in static memory and a common constraint interface to extract the implying PBC should remove this requirement and therefore reduce the runtime drastically.

The second most time consuming task in most problem classes is executing `propagate()` and `updateConstraint()` which use 10-40% of the runtime as mentioned earlier. This is due to the counted propagation approach and the high number of learnt constraints which implement this approach. The only way to reduce this overhead without implementing a watched scheme is to remove possibly valuable information. For example the solver could learn only constraints which force at least one literal, trigger large backjumps or add new literals to decision level 0. Also one can think of removing true variables with low heuristical values from the constraint before learning it or learning only extracted cardinality constraints (this last technique was implemented in [2]).

The most astonishing observation about this benchmark though, is the fact that the learnt PBCs often reduce the number of conflicts and choices by an order of magnitude. This is immense and if even part of this learning scheme can be implemented to run fast enough, it is definitely worth the effort.

Also note there are problem classes like *KnightTour* and *Pigeonhole* which the solver solves much faster even with the non-optimized CP analysis implemented in this thesis. This is probably due to problem structures where a single CP analysis can produce a PBC which subsums a large number of conflict clauses, whereas a solver using only resolution analysis needs to analyze these conflicts all separately.

[2] On timeout the solver reports the current number of conflicts and choices and it remains unknown how many more conflicts are required to solve the problem

6 Conclusion

This thesis discussed a number of possible ways how pseudo-Boolean constraint learning can be used within an ASP solver and also implemented a simple cutting planes conflict analysis within the solver clasp. It was shown that such an analysis has the potential to significantly boost the performance of the search, though one has to find ways of minimizing the additional overhead. Some possible ways to do so were pointed out.

Many things are still left to do: An efficient cutting planes analysis could be implemented and a number of different tactics for minimizing the overhead of variable elimination and propagation should be investigated. It also remains to see whether a watched literals scheme proves useful. In the end CP analysis still has to be implemented and evaluated for optimization problems because much more improvements are expected in this category of problems.

A Interaction between PBConstraint and Solver

Algorithm 5 setConflict()

```
1  void Solver::setConflict(Literal p, const Antecedent& a, uint32 data)
       {
2      conflict_.push_back(~p);
3      if (!a.isNull()) {
4          if (data == UINT32_MAX) {
5              a.reason(*this, p, conflict_);
6              aggregator_= new PBConstraint(*this, p, a, true);
7          }
8          else {
9              uint32 saved = assign_.data(p.var());
10             assign_.setData(p.var(), data);
11             a.reason(*this, p, conflict_);
12             aggregator_= new PBConstraint(*this, p, a, true);
13             assign_.setData(p.var(), saved);
14         }
15     }
16 }
```

Algorithm 6 resolveConflict()

```
1  bool Solver::resolveConflict() {
2      while (decisionLevel() > 0) {
3          uint32 uipLevel = analyzeConflict();
4          undoUntil( uipLevel );
5          if (aggregator_->bound() <= 1) {
6              // Subsumed by clause
7              aggregator_->destroy();
8              aggregator_ = 0;
9          }
10         if (integrateClause(cc_) &&
11                 (!aggregator_ || aggregator_->integrate(*this))) {
12             return true;
13         }
14     }
15     return false;
16 }
```

Algorithm 7 analyzeConflict()

```
1  uint32 Solver::analyzeConflict() {
2      uint32 onLevel= 0; // number of literals left on level
3      Literal p;         // literal to be resolved out next
4      cc_.assign(1, p);  // placeholder for asserting literal
5      for (;;) {
6          for (uint32 i= 0; i != conflict_.size(); ++i) {
7              Literal& q= conflict_[i];
8              uint32 cl=  level(q.var());
9              if (!seen(q.var())) {
10                 markSeen(q.var());
11                 if (cl == decisionLevel()) {
12                     ++onLevel;
13                 }
14                 else {
15                     cc_.push_back(~q);
16                 }
17             }
18         }
19         // find the last assigned literal in resolvent
20         while (!seen(assign_.last().var())) {
21             assign_.undoLast();
22         }
23         if (--onLevel == 0) {
24             conflict_.push_back(~p);
25             break;
26         }
27
28         // p is resolved out next
29         p = assign_.last();
30         clearSeen(p.var());
31
32         // an earlier elimination might have removed p already!
33         if (aggregator_->weight(~p) > 0){
34             aggregator_->varElimination(*this, p);
35         }
36         reason(p, conflict_);
37     }
38
39     cc_[0] = ~assign_.last(); // store the 1-UIP
40     clearSeen(cc_[0].var());
41
42     return uipLevel(cc_);
43 }
```

List of Figures

List of Tables

List of Algorithms

Abbreviations

ASP Answer Set Programming

CDCL conflict driven clause learning

First-UIP first unique implication point

ILP Integer Linear Programming

PBC pseudo-Boolean Constraint

Bibliography

[1] Daniel Le Berre and Anne Parrain. On extending sat solvers for pb problems.

[2] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver, 2003.

[3] W. Cook, C.R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[4] Marc Denecker, Joost Vennekens, Stephen Bond, and Martin Gebser. The second answer set programming competition, 2009.

[5] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver, 2002.

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

[7] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[8] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, 2009.

[9] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[10] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[11] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-boolean sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:2006, 2006.

[12] Joso L Marques Silva. Grasp - a new search algorithm for satisfiability. pages 220–227, 1996.

[13] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181 – 234, 2002.

[14] Niklas Sörensson and Niklas Een. Minisat v1.13 - a sat solver with conflict-clause minimization. Technical report, 2002.