Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 165-189

# Pueblo: A Hybrid Pseudo-Boolean SAT Solver

**Hossein M. Sheini** hsheini@umich.edu
**Karem A. Sakallah** karem@umich.edu
*Electrical Engineering and Computer Science Department,*
*University of Michigan, Ann Arbor, MI*

## Abstract

This paper introduces a new hybrid method for efficiently integrating Pseudo-Boolean (PB) constraints into generic SAT solvers in order to solve PB satisfiability and optimization problems. To achieve this, we adopt the cutting-plane technique to draw inferences among PB constraints and combine it with generic implication graph analysis for conflict-induced learning. Novel features of our approach include a light-weight and efficient hybrid learning and backjumping strategy for analyzing PB constraints and CNF clauses in order to simultaneously learn both a CNF clause and a PB constraint with minimum overhead and use both to determine the backtrack level. Several techniques for handling the original and learned PB constraints are introduced. Overall, our method benefits significantly from the pruning power of the learned PB constraints, while keeping the overhead of adding them into the problem low. In this paper, we also address two other methods for solving PB problems, namely Integer Linear Programming (ILP) and pre-processing to CNF SAT, and present a thorough comparison between them and our hybrid method. Experimental comparison of our method against other hybrid approaches is also demonstrated. Additionally, we provide details of the MiniSAT-based implementation of our solver Pueblo to enable the reader to construct a similar one.

KEYWORDS: *satisfiability, SAT solver, pseudo-Boolean, inference-based learning*

*Submitted October 2005; revised March 2006; published March 2006*

## 1. Introduction

Recent advances in algorithms for Boolean Satisfiability (SAT) have led to a significant increase in the capacity and applicability of SAT solvers. Many large design and analysis problems from the field of Electronic Design Automation (EDA) are now routinely cast as SAT instances (with millions of CNF clauses and tens of thousands of variables) and quickly solved using these powerful solvers.

Modern SAT solvers for systems of Boolean constraints in Conjunctive Normal Form (CNF) are based on the DLL backtrack search procedure of Davis, Logemann, and Loveland [11] augmented with powerful conflict-based learning [28] and efficient watched-literal schemes [29] for Boolean constraint propagation (BCP). While proving to be quite versatile for a wide range of applications, CNF constraints are not always the most efficient. The closely-related linear 0-1 inequalities known as Pseudo-Boolean (PB) constraints have been widely used to efficiently encode many problems ranging from logic synthesis [1] and verification [5] to numerous Operations Research applications [4]. Additionally, PB constraints are increasingly used to represent objective functions in optimization applications, such as

Max-SAT and Max-ONEs [8]. These optimization problems are solved by deciding a series of PB SAT instances, each time tightening the right-hand side of the objective function constraint. The procedure terminates when the last instance becomes unsatisfiable. Methods for lower bound estimation [26] have been effectively applied to these problems.

In order to prune the solution space during the search, many solvers introduce new constraints dynamically. In ILP, such constraints are generated by cutting plane analysis [20] inside the branch-and-bound procedure. In modern SAT and PB solvers, on the other hand, these constraints are derived by analyzing the implication graph at each conflict resulting in a CNF clause [28, 1], or by applying cutting plane analysis to produce a PB constraint [13] or a cardinality constraint [6]. Cutting planes were also introduced into the PB optimization procedure in [27] and were used to compute lower bounds on the objective function.

In this paper, we introduce a new hybrid algorithm for solving PB SAT problems based on combining cutting plane methods and implication graph analysis for conflict-based learning. This enables the learning procedure to benefit from the unique properties of the implication graph in terms of efficient and cheap recognition of unit clauses and at the same time learn a PB constraint through the cutting plane procedure. Consequently, our method creates *both* a PB constraint and a CNF clause after a conflict is detected. This strategy enables the solver to fully utilize the pruning power of the PB constraint as long as it remains active. When the PB constraint becomes inactive, it is discarded and its role is assumed by its companion CNF clause. Such a "dual learning" scheme incurs no overhead over a learning method that just creates a PB constraint. Additionally, by only watching some of the variables in each of the learned PB constraints and periodically discarding those constraints that are not active, the handling cost is curbed further.

This method allows for a particularly efficient "light" integration of PB constraints into the MiniSAT SAT solver [16] further enhancing performance on mixed CNF/PB problems. Experimental evaluation over a wide range of benchmarks demonstrates the effectiveness of our hybrid learning and backjumping strategy compared to the individual strategies of generating either a PB constraint or a CNF clause after each conflict. We also compare our hybrid approach with ILP and Boolean preprocessing methods on a wide range of benchmarks with different characteristics and examine the effectiveness of these methods on each problem set.

The paper is structured as follows. In Section 2, we cover some preliminaries. In Section 3, different methods for solving PB satisfiability and optimization problems are introduced. Section 3.3 describes our hybrid learning, backjumping and constraint management methods. Implementation details and experimental results are reported in Sections 4 and 5 and conclusions and suggestions for further work are presented in Section 6.

## 2. Preliminaries

A linear Pseudo-Boolean (PB) constraint is said to be in normal form when expressed as[1]:

$$\sum_{i=1}^{n} a_i \dot{x}_i \geq b \quad a_i, b \in \mathbb{Z}^+, \dot{x}_i \in \{0,1\} \tag{1}$$

1. Less-than-or-equal and equality constraints can be easily transformed to equivalent greater-than-or-equal forms

$\omega_1 : 3x_1 + 2x'_2 + x_3 + x'_4 \geq 3$
$\omega_2 : 2x_2 + 2x_{10} + x_8 + x_9 \geq 2$
$\omega_3 : x'_3 + x_7 + x_4 \geq 2$
$\omega_4 : x_2 \vee x'_5 \vee x_6$
$\omega_5 : x'_6 \vee x'_4 \vee x'_9$
$\omega_6 : x_1 \vee x_3 \vee x_4$

**Figure 1.** PB SAT problem with CNF and PB constraints

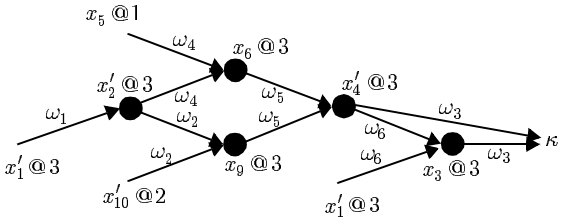


**Figure 2.** implication graph resulting in a conflict in $\omega_3$

where $\dot{x}_i$ denotes a literal $x_i$ or $x'_i$. A PB constraint in which some coefficients are negative can be transformed to normal form by noting that $\dot{x}'_i = 1 - \dot{x}_i$. The sum of the coefficients of the literals in the constraint whose values are true, false and undefined are denoted by $S_T$, $S_F$, and $S_U$ respectively. The largest coefficient in the constraint is referred to as $a_{\max}$ and its corresponding literal as $\dot{x}_{\max}$. An example PB constraint in normal form is:

$$3x_1 + 2x'_2 + x_3 + x_4 \geq 3 \tag{2}$$

If all the coefficients in a PB constraint are equal, the constraint reduces to a *cardinality constraint*; when the right-hand side and all left-hand side coefficients are equal, the constraint further reduces to a single CNF clause. Inversely, a CNF clause can be converted to a cardinality constraint with unit right-hand side:

$$\sum_{i=1}^{n} \dot{x}_i \geq 1 \Leftrightarrow \bigvee_{i=1}^{n} \dot{x}_i$$

In this paper, we will use the SAT problem of Figure 1 as a running example.

### 2.1 PB Constraint Processing

The following operations are defined for PB constraints and can be applied to them before and during the solving process:

**Coefficient Reduction -** As noted in [9], a PB constraint in normal form as in (1), such that $a_k > b$ for some $k \in \{1, \cdots, n\}$, is equivalent to the PB constraint:

$$b\dot{x}_k + \sum_{i=1, i\neq k}^{n} a_i \dot{x}_i \geq b \tag{3}$$

in the sense that they both have the same 0-1 solution set. The set of real solutions of (3) in the unit hypercube is a subset of those of the original constraint (1).

**Weakening to Cardinality Constraint -** The strongest extended clause[2.] [2] of the PB constraint (1) is the weaker cardinality constraint:

$$\sum_{i=1}^{n} \dot{x}_i \geq \beta \tag{4}$$

2. Also known as the *extended cover inequality* [36]

where $\beta$ is the smallest integer such that[3.]

$$\sum_{i=1}^{\beta-1} a_i < b \leq \sum_{i=1}^{\beta} a_i$$

For instance, the cardinality constraint representing the strongest extended clause of (2) is:

$$x_1 + x_2' + x_3 + x_4 \geq 1$$

**Variable Elimination -** A non-negative linear combination of a system of PB constraints is referred to as a cutting plane [20, 7] since it *cuts off* a portion of the feasible set obtained by relaxing the integrality requirement on the variables (in this case simply being 0 or 1) without affecting the 0-1 feasibility of the original set [7]. This construction can also be viewed as an extension of resolution from CNF clauses to 0-1 inequalities [22]; in this view the cutting plane is considered to be an implication of the set of PB constraints used to generate it. An example of this procedure is:

$$\begin{array}{lllll} 3x_1+ & 2x_2'+ & x_3+ & x_4 & \geq 3 \\ & & x_3'+ & x_4 & \geq 1 \\ \hline 3x_1+ & 2x_2'+ & & 2x_4 & \geq 3 \end{array}$$

### 2.2 PB Constraint Propagation

A PB constraint is *unit* and therefore implies $\dot{x}^U_{\max}$ when $S_T + S_U < b + a^U_{\max}$, where $\dot{x}^U_{\max}$ is the unvalued literal with the largest coefficient, $a^U_{\max}$, among all unvalued literals in the constraint. The implying assignment, in this case, is the conjunction of false literals in the PB constraint. For example, PB constraint (2) becomes unit if $x_1$ is false, implying $x_2$ to false. It later can become unit again if either $x_3$ or $x_4$ is assigned to false, implying the other. A constraint is said to be *over-satisfied* if, under its current assignment, its left-hand side is strictly greater than its right-hand side. Note that a unit CNF clause can never be over-satisfied . On the other hand, a unit PB constraint is over-satisfied when $S_T + S_U > b$. Obviously, a PB constraint is violated when $S_T + S_U < b$. Figure 3 demonstrates the status of example PB and CNF constraints under different variable assignments.

The most straightforward method for detecting that a constraint has become unit is to always update $S_T$, $S_F$, and $S_U$ after each assignment to one of its literals. This amounts to *watching all literals* in the constraint and has been dubbed the "counters" procedure in [6]. This procedure was used in the PBS PB solver [1]. Efficient propagation procedures through PB constraints [6, 14, 33] are based on various extensions of the highly effective two-literal watch strategy for CNF clauses [29]. For example, in [6] the basic idea is to watch the fewest number of non-false literals such that when the unassigned watched literal with the largest coefficient is set to false a) the constraint is still guaranteed to be satisfied and b) the constraint can identify the literals that must now be implied to true. Specifically, if $S_W$ denotes the sum of coefficients of the watched literals (the *watched sum*), the invariant that must be maintained to detect when the PB constraint becomes unit can be succinctly expressed as:

$$S_W \geq b + a^U_{\max} \tag{5}$$

3. We assume that $a_1 \geq a_2 \geq ... \geq a_n$

| constraint | $x_1, x_2, x_3$ | $S_T$ | $S_U$ | $a^U_{\max}$ | unit? $S_T + S_U < b + a^U_{\max}$ | overSAT? $S_T + S_U > b$ |
|---|---|---|---|---|---|---|
| $3x_1 + 3x'_2 + x_3 \geq 3$ | $-, -, -$ | 0 | 7 | 3 | no | yes |
| | $0, -, -$ | 0 | 4 | 3 | yes | yes |
| | $0, 0, -$ | 3 | 1 | 1 | no | yes |
| $2x_1 + x_2 + x_3 \geq 2$ | $-, -, -$ | 0 | 4 | 2 | no | yes |
| | $0, -, -$ | 0 | 2 | 1 | yes | no |
| | $0, 1, 1$ | 2 | 0 | – | no | no |
| $x_1 \vee x_2 \vee x_3$ | $-, -, -$ | 0 | 3 | 1 | no | yes |
| | $0, 0, -$ | 0 | 1 | 1 | yes | no |
| | $0, 0, 1$ | 1 | 0 | – | no | no |

**Figure 3.** Constraint status examples

When a watched literal is set to false, it must be replaced by one or more non-false literals to maintain this invariant. When that is no longer possible, the constraint becomes unit and the unassigned watched literal with the largest coefficients must be set to true to insure that the constraint is not violated. Effectively this procedure extends the two-literal watch strategy for CNF clauses [29] to unrestricted PB constraints and amount to *watching the fewest literals* necessary to perform implications and detect conflicts. It was concluded in [6], however, that while useful for CNF clauses and cardinality constraints, such a scheme is not as effective for unrestricted PB constraints due to its high overhead. Thus, for the solver in [6], they "decided to use counters to implement BCP for LPB constraints" as was suggested in [1].

### 2.3 Conflict-Based Learning Schemes

Following the introduction of no-good recording in dependency-directed backtracking method of [34], conflict learning and non-chronological backtracking techniques have been extensively studied and applied to different problems specifically to Constraint Satisfaction Problems (CSP) [12, 19, 32, 18, 24].

Noting the inevitability of conflicts during SAT search, in GRASP [28], a powerful conflict-induced learning and non-chronological backtracking for CNF propositional satisfiability problem is introduced. In this procedure, the implication graph leading to the conflict is constructed and analyzed to find a cut that includes only one assignment at the conflict level. This point in the graph is referred to as a unique implication point (UIP) [28] and its corresponding CNF clause is guaranteed to become unit when the last decision assignment is "erased".

On the other hand, the process of implication graph analysis to learn a CNF clause at the first UIP can be viewed as repeated application of resolution on the violated constraint and all the constraints that implied the conflict until a clause that becomes unit after erasing the last decision assignment is obtained [15]. In this framework, if we denote the violated

constraint by $V$ and the unit implying constraints leading to the conflict by $U_1, U_2, \cdots$ (in reverse implication order), this process can be formulated as follows:

$$\begin{aligned} R_1 &= resolve(V, U_1, \hat{x}_1) \\ R_2 &= resolve(R_1, U_2, \hat{x}_2) \\ &\cdots \\ R_k &= resolve(R_{k-1}, U_k, \hat{x}_k) \end{aligned} \tag{6}$$

where $R_i$ represents the resolvent at each resolution step and $\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_k$ denote the corresponding implied variables. This process terminates when a unit resolvent is found which, in case of CNF clauses, will be the first encountered UIP.

It is important to note that these two learning procedures, namely a) structural and b) resolution-based analysis of the implication graph, yield identical results when the only constraint types involved are CNF clauses. When PB constraints are also present in the implication sequence, these two procedures do not necessarily yield identical learned constraints. We summarize below three different learning schemes that were proposed to deal with the participation of PB constraints in conflicts.

### 2.3.1 CNF Learning

It is possible to extend the conflict-induced CNF learning method of GRASP [28] to PB constraints by treating the implications caused by PB constraints exactly the same way as the ones caused by CNF clauses. We will refer to such learning as *CNF learning*. This process basically consists of constructing the implication graph and analyzing the implied assignments regardless of the types of constraints implying them. At this point, the exact same GRASP-like implication graph analysis and backtracking process is followed that always results in learning a CNF clause at the first UIP that determines the backtrack level. Figure 2 shows the implication graph for our running example in the case where $x_5$ and $x_{10}$ are assigned to true and false at decision levels 1 and 2 respectively. At decision level 3, $x_1$ is decided to false which ends up in a conflict in the PB constraint $\omega_3$. Analyzing the implication graph yields the following CNF clause:

$$(x_1 \vee x_5' \vee x_{10}) \tag{7}$$

### 2.3.2 PB Learning

PB learning, introduced in [13, 6], aims at analyzing the implication sequences leading to conflicts in order to derive unit PB constraints. It is basically an application of resolution as in (6) and results in the step-by-step elimination of implied variables, generating a cutting plane at each step. It terminates when a unit resolvent $R_i$ is found. An example of this process is demonstrated in Figure 4 under the PB learning column.

As noted earlier, this process basically mimics the CNF learning scheme based on implication graph analysis. However, unlike CNF learning where it is guaranteed that the learned CNF clause always remains violated in the process, a resolvent, $R_i$, may become satisfied if its corresponding $U_i$ is over-satisfied. A procedure to weaken the over-satisfied unit constraint $U_i$ is suggested in [6]. In this procedure, the *slack* of a constraint, defined as $(S_T + S_U) - b$, is used to check whether the combination of $R_{i-1}$ and $U_i$ yields a non-violated

resolvent $R_i$. When this condition is detected, the unit constraint $U_i$ is weakened to insure that $R_i$ becomes violated. For example, consider the following resolution step to eliminate $x_2$ under the assignment $(x_1, x_2, x_{10}, x_5) = (0, 0, 0, 1)$ (after applying coefficient reduction):

| | | | |
|---|---|---|---|
| $V:$ | | $x_1 + 4x_2 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4$ | (slack $= -1$) |
| $U:$ | $2\times$ | $3x_1 + 2x'_2 + x_3 + x'_4 \geq 3$ | (slack $= +1$) |
| $R:$ | | $6x_1 + 4x_{10} + 2x'_5 + 2x_3 + 2x'_4 + x_7 + 2x_8 \geq 6$ | (slack $= +1$) |

Clearly the over-satisfaction of the unit constraint $U$ causes the resolvent $R$ to become non-violated. In terms of slacks, the slack of the resolvent $R$ is obtained by linearly combining the slacks of $V$ and $U$ yielding $-1 + 2 = +1$. The procedure suggested in [6] for weakening the over-satisfied unit constraint $U_i$ proceeds as follows:

| | | | | |
|---|---|---|---|---|
| 1. weakening | $3x_1 + 2x'_2 + x_3 + x'_4 \geq 3$ | (slack $= +1$) | $\rightarrow$ | $3x_1 + 2x'_2 + x_3 \geq 2$ |
| 2. coef reduc'n | $3x_1 + 2x'_2 + x_3 \geq 2$ | | $\rightarrow$ | $2x_1 + 2x'_2 + x_3 \geq 2$ |
| 3. weakening | $2x_1 + 2x'_2 + x_3 \geq 2$ | (slack $= +1$) | $\rightarrow$ | $2x_1 + 2x'_2 \geq 1$ |
| 4. coef reduc'n | $2x_1 + 2x'_2 \geq 1$ | | $\rightarrow$ | $\hat{U} : x_1 + x'_2 \geq 1$ |

Note that in each weakening step, a non-false literal is chosen to be removed in no particular order. Finally combining $V$ with the weakened constraint $\hat{U}$ yields the following resolvent after coefficient reduction, which is strictly stronger than (7):

$$4x_1 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4 \tag{8}$$

It is important to note that, cutting planes-based learning may not yield a unit constraint when PB constraints are involved. This is in contrast to the case where only CNF clauses participate in the implication sequence where it is guaranteed, as mentioned earlier, that resolution-based learning stops at the first UIP and yields a unit clause (an example of such a case will be given in Section 3.3.2).

#### 2.3.3 Cardinality Learning

Finally, learning cardinality constraints has been considered as a compromise between CNF and PB learning methods [2]. These constraints are able to retain some of the information represented by the PB constraints and, at the same time, lead to significantly cheaper propagation. In [6], a "post-reduction" procedure is introduced that converts the learned PB constraint into a weaker cardinality constraint. Applying the procedure of [6] to (8) would yield $x_1 + x_{10} + x'_5 + x_7 + x_8 \geq 1$ which is further weakened due to over-satisfaction to $x_1 + x_{10} + x'_5 \geq 1$. For more details on the this process, the reader is referred to [6].

## 3. Solving PB Constraints

### 3.1 Integer Linear Programming Methods

These methods, as used in several commercial tools such as CPLEX [23], are based on relaxing the integrality of 0-1 variables and using the Simplex procedure to solve the resulting LP problem at each node of a Branch-and-Bound (BNB) tree. The search proceeds by partitioning (branching on) the domain of each 0-1 variable at each node and terminates as

soon as it finds an integral solution or all its nodes are examined. Recent advances in this procedure are mainly due to augmenting it with the *branch-and-cut* technique [21] which is a method to find inequalities that are valid in the 0-1 problem but are violated by the LP relaxation. Hence, adding these inequalities to the LP relaxation tightens the formulation and strengthens the BNB framework. This is useful because a stronger relaxation at a node may have a larger objective function that may allow the tree to be pruned at that node. Other devices include Lagrangean relaxation, the use of reduced costs to check whether fixing a variable to 0-1 will permit the tree to be pruned, and so forth. Note that these procedures are not as efficient when applied to decision problems with no objective function.

### 3.2 Pure SAT-based Methods

The most straightforward method for solving a system of PB constraints mixed with CNF clauses is to somehow convert them to an equivalent system of CNF clauses and to submit them to a SAT solver as is done in MiniSAT+ [17]. One approach for PB-to-CNF conversion is the adoption of a circuit representation for each PB constraint to yield a CNF formula [1]. This transformation can be obtained by introducing "partial sum" variables that decompose the monolithic PB constraint into a set of smaller constraints.

In [17], the following three techniques for translating PB constraints to CNF clauses have been introduced:

1. *converting the PB constraint to a BDD*, where the BDD representation of the PB constraint is obtained and then translated into CNF clauses

2. *converting the PB constraint to a network of adders*, where the sum at the left-hand side of the PB constraint is produced as a binary number and the circuit representing the comparison of this sum against the right-hand side is created. CNF representation of this circuit is the translation of the PB constraint.

3. *converting the PB constraint to a network of sorters*, where unlike the previous technique, numbers are represented in unary instead of binary and then again the circuit for the comparison is constructed and translated to CNF clauses.

For more details on these methods, the reader is referred to [17].

### 3.3 Hybrid Methods

Extending DLL-style SAT solvers to handle PB constraints [3] is the most common SAT-based approach for solving these types of problems. In these methods, PB constraints and CNF clauses are treated as separate constraint classes. Several techniques and strategies for integrating PB constraints into SAT solvers have been developed during the past few years. Aloul et al [1] adopted the "watch all literals" and CNF learning schemes for PB propagation and learning. Conflict-induced learning based on cutting planes was first introduced to SAT solvers in [13]. Recognizing the inefficiency of handling a large number of generated PB constraints (cutting planes), the authors in [6] introduced a weakening process applied to all generated cutting planes ("post-reduction"), converting them to cardinality constraints. Also noting the significant overhead of processing a large number of generated cutting planes, the authors in [33] proposed a scheme to discard those constraints periodically. In

this section, we present a novel hybrid method for analyzing the implication graph and generating cutting planes that yields both a CNF and a PB constraint at each conflict. Two strategies for efficiently managing the PB constraints are also discussed.

### 3.3.1 Hybrid Learning

Noting the advantages of CNF learning, namely its cheap implementation and its ability to always produce a unit constraint, as well as the pruning power of PB learning, our hybrid algorithm efficiently combines the CNF and PB learning methods in order to produce both a CNF clause and a PB constraint in case of a conflict. In this combined procedure, CNF learning guarantees the correctness and completeness of the process by always producing a unit CNF clause, and is used to terminate the learning process as soon as the first UIP is reached. Additionally, the simultaneous generation of a PB constraint helps to considerably prune the search space. This is in contrast to the method of [6] in which learned PB constraints are weakened to cardinality constraints which are then used to terminate the learning process and also retained as the final learned constraints.

It is also important to note that the false literals in the resolvent, $R_i$, at each resolution step represent the cut in the implication graph at that step. Therefore they are stored in a separate set, $X_F$, and used for CNF learning. This makes CNF learning quite efficient and significantly minimizes its overhead.

The overall learning process terminates as soon as the first UIP is reached which is automatically detected inside the CNF learning procedure. At this point the learned CNF clause, stored in $X_F$, is unit and the resolvent, $R_i$, forms the learned PB constraint. The backtrack level is determined by processing both the learned CNF and PB constraints according to the conditions explained in the next section.

This process is demonstrated in Figure 4 for our running example. It starts at step 0 with the violated PB constraint. $X_F$, at this point, contains those literals in the violated constraint that are false, $x'_3$ and $x_4$. The next step eliminates $x_3$ by combining its implying clause, $x_1 \vee x_3 \vee x_4$, with the violated constraint resulting in $x_1 + 2x_4 + x_7 \geq 2$. Additionally, $x'_3$ is replaced by $x_1$ in $X_F$. In the following steps, literals $x_4$, $x_6$, and $x_9$ are eliminated from the cutting plane respectively. Note that in step 4, coefficient reduction is applied to the coefficient of $x_2$. $X_F$ at each step holds the literals in the generated cutting plane at that step that are false and cause that constraint to be violated.

While it is not necessary in this hybrid approach to eliminate over-satisfaction in the learned PB constraints, or to even make them unit, by a weakening step, we found empirically that it is beneficial to weaken them somewhat to increase their pruning power under the current truth assignment. To reduce the overhead of such weakening, we utilize the structure of the implication graph to replace an over-satisfied unit PB constraint with the implication it induced, namely the CNF clause representing the cut that corresponds to the edges labeled with that constraint. For example in Figure 4, the over-satisfied unit constraint $3x_1 + 2x'_2 + x_3 + x'_4 \geq 3$ is replaced with $x'_1 \rightarrow x'_2$ to resolve $x_2$ in $x_1 + 4x_2 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4$ yielding $4x_1 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4$ (after applying coefficient reduction). This simple weakening scheme incurs very little overhead and performed quite well in practice.

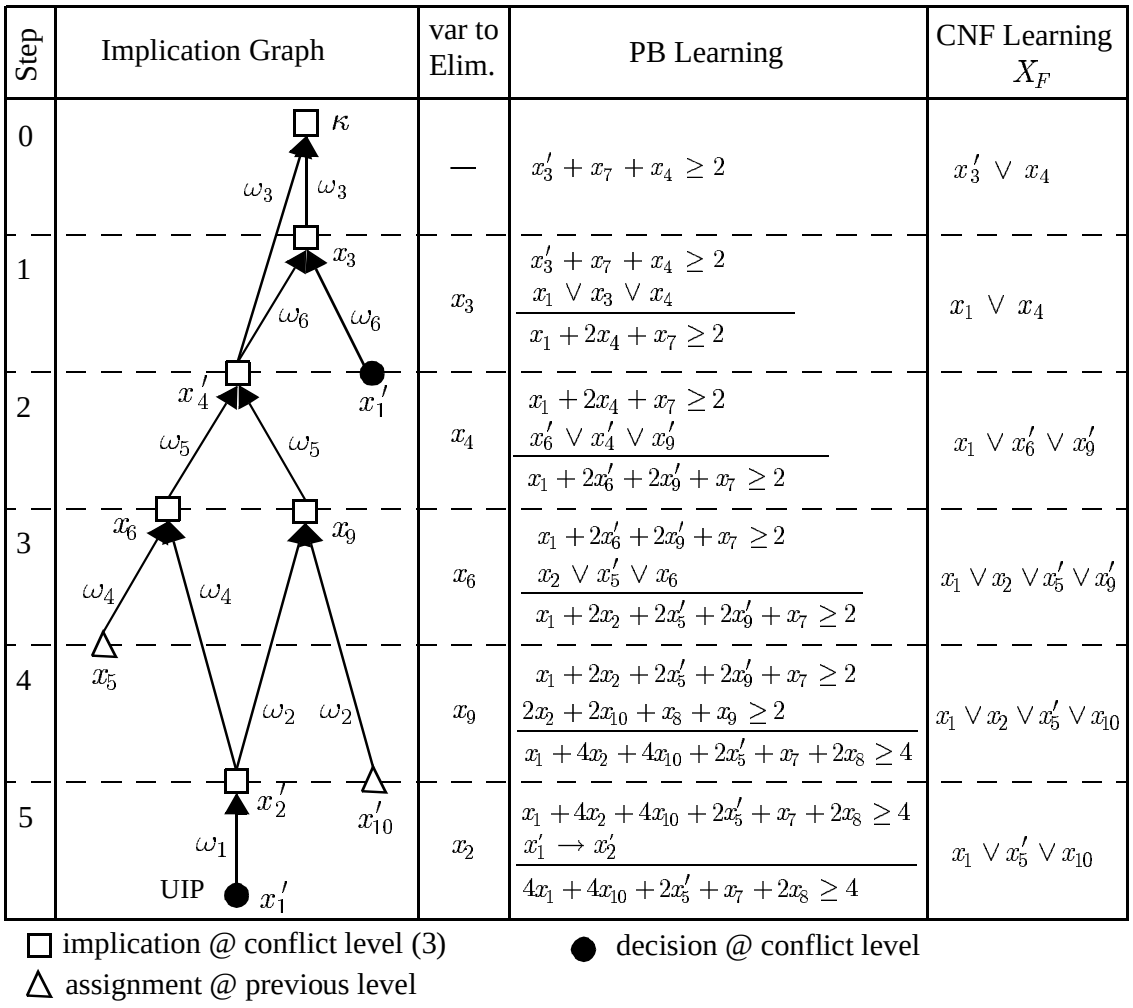


**Figure 4.** Simultaneous PB and CNF learning induced by conflict. The variable ordering is according to implication.

### 3.3.2 Hybrid Backjumping

Let $C$ and $P$ denote the CNF and PB constraints generated by the hybrid learning process described above. As mentioned earlier, $C$ is guaranteed to become unit at some earlier decision level after the current decision assignment is undone. The same cannot be said of $P$, however. In other words, after erasing the last decision assignment, it is possible for $P$ to remain non-unit at all earlier decision levels (recall that we do not require $P$ to be weakened to the point that it becomes unit). The hybrid backjumping procedure must, thus, consider the following two scenarios:

**Scenario A: The learned PB constraint can become unit.** In this case, let $l_c$ and $l_p$ denote the decision levels at which $C$ and $P$, respectively, become unit. The solver can now safely backtrack to decision level $min(l_c, l_p)$ and proceed as follows:

1. If $l_c = l_p$, then both $C$ and $P$ are unit and their corresponding implications are propagated. This was the case of our running example.
2. If $l_c < l_p$, then $C$ is unit and $P$ is satisfiable (not violated). In this case, the implication due to $C$ alone is propagated.
3. If $l_c > l_p$, then $P$ is unit and $C$ is satisfiable (not violated). In this case, the implication(s) due to $P$ alone is (are) propagated. As an example of this case, consider the following constraints whose structure is common in routing problems [1]:

$$\begin{array}{l}(C_1) x_{11} \vee x_{10} \vee x_9 \\ (C_2) x_8 \vee x_7 \vee x_6 \\ (C_3) x_5 \vee x_4 \vee x_3 \\ (C_4) x'_2 + x'_5 + x'_8 + x'_{11} \geq 3 \\ (C_5) x'_1 + x'_4 + x'_7 + x'_{10} \geq 3\end{array} \tag{9}$$

In this example, we assume that, $x_3$, $x_6$ and $x_9$ are implied to false at the decision levels 1, 2 and 3 respectively and $x_1$ is assigned to true at the current ($4^{th}$) decision level resulting in the violation of $C_4$. The learning procedure proceeds as follows:

| step | action | resulting cutting plane | conflict assignments |
|---|---|---|---|
| 1 | $C_4$ | $x'_2 + x'_5 + x'_8 + x'_{11} \geq 3$ | $x_1 x'_4 x'_7 x'_{10} x_5 x_8 x_{11}$ |
| 2 | $+ C_1$ | $x'_2 + x'_5 + x'_8 + x_9 + x_{10} \geq 3$ | $x_1 x'_4 x'_7 x'_{10} x_5 x_8$ |
| 3 | $+ C_2$ | $x'_2 + x'_5 + x_6 + x_7 + x_9 + x_{10} \geq 3$ | $x_1 x'_4 x'_7 x'_{10} x_5$ |
| 4 | $+ C_3$ | $x'_2 + x_3 + x_4 + x_6 + x_7 + x_9 + x_{10} \geq 3$ | $x_1 x'_4 x'_7 x'_{10}$ |
| 5 | $+ C_5$ | $x'_1 + x'_2 + x_3 + x_6 + x_9 \geq 3$ | $x_1 x'_4 x'_7$ |

The resulting PB constraint is $x'_1(0@4) + x'_2 + x_3(0@1) + x_6(0@2) + x_9(0@3) \geq 3$. The learned CNF clause, comprising of the false literals in the learned PB constraint is, therefore, $x'_1(0@4) \vee x_3(0@1) \vee x_6(0@2) \vee x_9(0@3)$. The learned PB constraint is unit at decision level 2 ($l_p = 2$) implying $x'_1$, $x'_2$ and $x_9$ to true. On the other hand, the learned CNF clause is unit at decision level 3 ($l_c = 3$), implying $x'_1$ to true. Note that at decision level 3, the learned PB constraint is still violated and therefore, our algorithm learns both constraints, backtracks to level 2, implies $x'_1$, $x'_2$ and $x_9$ to true and records the PB constraint as the reason for the implications.

**Scenario B: The learned PB constraint does not become unit.** In this case, the solver checks the status of $P$ at decision level $l_c$:

1. If $P$ is satisfiable (not violated), then the implication due to $C$ alone is propagated. This is similar to case 2 of scenario A.
2. If $P$ is violated, one possibility is to invoke the learning procedure to analyze the causes of this violation yielding two new learned constraints (CNF and PB). Another possibility is to backtrack to the highest decision level before $l_c$ at which

$P$ becomes satisfiable (but not necessarily unit). At that point, there are no implications due to the learned constraints and the solver must record both $C$ and $P$ and make a new decision assignment. This is the approach implemented in the Pueblo solver. We should note that this backtracking strategy is complete because each pruning step eliminates a new portion of the search space and provides an explanation (a CNF or PB constraint) for why a solution cannot be found in that portion. In this case, the learned CNF and PB constraints are never removed to maintain the completeness of the procedure.

As an example, consider the set of constraints of (9), but now assume that due to other parts of the problem, $x_3$, $x_6$ and $x_9$ are all implied to false at the decision level 3 and $x_1$ is assigned to true at the $4^{th}$ decision level, resulting in the violation of $C_4$. Thus, the resulting PB and CNF constraints are as follows:

$$\begin{aligned} P: &\quad x_1'(0@4) + x_2' + x_3(0@3) + x_6(0@3) + x_9(0@3) \geq 3 \\ C: &\quad x_1'(0@4) \vee x_3(0@3) \vee x_6(0@3) \vee x_9(0@3) \end{aligned}$$

At decision level 3, $C$ is unit but $P$ is still violated. On the other hand, at decision level 2, none of the two learned constraints are unit. In this case, the solver can either backtrack to decision level 3 and invoke the learning process on $P$ or backtrack to decision level 2 and continue the search.

### 3.3.3 Efficient Managements of PB Constraints

As noted in [6, 33], the major disadvantage of learning PB constraints is that the high overhead of propagating through those constraints might offset the speed-up due to their pruning power. On the other hand, unlike unrestricted PB constraints, unit propagation on cardinality constraints is more efficient, mainly due to the fact that finding the unvalued literal with maximum coefficient is avoided. As mentioned earlier, this is addressed in [6] by weakening learned PB constraints to cardinality constraints. In contrast, our strategy for managing the overhead of unrestricted PB constraints is based on a) watching more than the minimum number of (but not all) literals to simplify the propagation process and b) controlling the number of learned PB constraints as search progresses. Some details of these two strategies are explained below:

**Efficient Propagation through PB Constraints** Our proposed watching scheme for PB constraints is to watch enough literals in the PB constraints to make $S_W \geq b + a_{\max}$. By using $a_{\max}$, rather than $a^U_{\max}$ as in (5), more than the minimum number of literals ends up being watched, but propagation becomes considerably faster since it is no longer necessary to find $a^U_{\max}$. Note that this technique was suggested in [6] but was not found "beneficial" and therefore was not implemented for unrestricted PB constraints.

**Bounded Learning Scheme** In this approach, the number of times that each learned PB constraint has participated in a conflict (referred to as that constraint's *activity*) is stored. In order to keep the number of PB constraints in the problem under control, those learned PB constraints whose activity has dropped below a set threshold are periodically removed. Following this procedure guarantees that at all times, only

those PB constraints relating to the current status of the assignment sequence are propagated. This would limit the time spent in propagating PB constraints and delegates the role of the dropped PB constraints to their companion learned CNF clauses. The maximum number of learned PB constraints is increased after each restart; this heuristic was found to be useful experimentally.

## 4. Implementation

We implemented the procedures and schemes described in this paper in our PB solver, Pueblo, as an extension to MiniSAT 1.12 [16]. The implementation details are covered in this section and should be read together with the MiniSAT description presented in [16]. Note that Pueblo can solve both PB satisfiability and optimization problems and has the capability to handle integer coefficients that can be represented with at most 32 bits.

### 4.1 Constraints

Exploiting the capability of MiniSAT to handle arbitrary constraints over Boolean variables through its *Constr* abstract base class, we added a *PseudoBool* constraint class in addition to its existing *Clause* constraint class. Unique procedures for propagating and calculating reasons for this class of constraints are presented in Figure 5. Using the base class enables Pueblo to use all of MiniSAT's solving procedures independent of the type of constraint being handled. These *PseudoBool* constraints are either created at the beginning of the search or learned through PB learning procedure. Each *PseudoBool* constraint is created or learned using the *PB_new* procedure, presented in Figure 6.

### 4.2 Accumulator

The accumulator is the PB constraint that contains the resolvent of the cutting plane based PB learning procedure. The PB learning process starts with the violated constraint considered as the accumulator and continues by adding it to the implying constraints and saving it back in itself. In order to avoid searching for variables, the accumulator is implemented as an array whose size is equal to the number of problem variables. In this scenario, CNF learning consists of detecting and separately storing the false literals of the accumulator. Details of the accumulator class are presented in Figure 7.

### 4.3 Pueblo Solver

Pueblo major modifications in MiniSAT solving procedure are listed below:

#### 4.3.1 Learning

Pueblo adopts the same learning flow as in MiniSAT augmenting it with cutting plane generation (PB learning) at each step. At each step in the backward traversal of the implication graph, the CNF or PB constraint involved in that implication is added to the accumulator while performing MiniSAT's *calcReason()* routine to eliminate the implied literal. The cutting plane is saved in the accumulator while the learned CNF clause is stored in the *out_reason* following the learning procedure of MiniSAT (refer to Figure 5).

```
class PseudoBool : public Constr
    int rhs
    int watchsum
    int amax
    float activity
    bool learnt
    Vec<PBTerm> terms       - PBTerm comprises of a literal, a coefficient and watched bool.
                            - terms is sorted based on coefficients
    bool propagate(Solver S, lit p)
        int p_idx = terms.index(p)
        terms[p_idx].unwatch(this)
        - update watchsum
        for(int i = 0; i < size() && watchsum < amax + rhs; i++)
            Lit lit = terms[i].lit
            if(S.value(lit)≠ l_False && !terms[i].watched()) terms[i].watch(S, this)
        - check for conflict
        if(watchsum < rhs_goal)
            terms[p_idx].watch(S, this)
            return False
        - check for satisfiability
        if(watchsum ≥ amax + rhs) return True
        for(int i = 0; i < size(); i++)
            Lit lit = terms[i].lit
            int coeff = PBTerms[i].coeff
            if(watchsum ≥ coeff + rhs) break
            if(S.value(lit) == l_Undef)
                if(!S.enqueue(lit, this))
                    terms[p_idx].watch(S, this)
                    return False - conflict in the Solver
        return True

    void calcReason(Solver S, lit p, vec<lit> out_reason)
        - all learned constraints are initially active
        if (learnt) S.pbBumpActivity(this)
        - calculate the multiplier to eliminate p from accumulator
        int mul = (p == lit_Undef) ? 1 : S.accumulator.coeff(var(p))
        S.accumulator.goal += mul * rhs
        for (int i = 0; i < size(); i++)
            lit = terms[i].lit
            if(lit == p)
                S.accumulator.goal − = mul*terms[i].coeff
                continue
            - adds this literal to the accumulator
            UpdateAccumulator(S, mul, terms[i].lit, terms[i].coeff)
            - saves the false literals for CNF learning
            if (S.value(lit) == l_False) out_reason.push(¬lit)
```

**Figure 5.** Implementation of *PseudoBool* constraint class in Pueblo

If an over-satisfaction is detected in the accumulator, the step resulting in over-satisfaction is undone and replaced by adding the weakened CNF clause to the accumulator. The learning stops when the first UIP is reached as detected by MiniSAT *analyze* procedure.

```
bool PB_new(Solver S, int goal, Vec<lit_coef> pbs, PseudoBool out, Clause c, bool learnt)
    out = new PseudoBool
    out.rhs = goal
    for (int i = 0; i < pbs.size(); i++)
        out.terms[i].lit = Lit(pbs[i].lit)
        if(pbs[i].coef < 0)
            out.rhs += abs(pbs[i].coef)
            out.terms[i].lit = ¬ out.terms[i].lit
        out.terms[i].coeff = abs(pbs[i].coef)
    sort(out.terms, size())
    if( (out.terms[0].coeff == out.rhs && out.terms[size()].coeff == out.rhs) ||
      out.rhs == 1 ) - checking if PB constraint is equal to a CNF clause
        bool ret = convertPBtoCNF(S, out, c, learnt) - creates clause with literals in terms of out
        xfree(out)
        return ret
    out.amax = out.terms[0].coeff
    out.learnt = learnt
    if(out.rhs == 0) return True
    for (int i = 0; i < size(); i++) - setting up the watch list
        Lit lit = out.terms[i].lit
        if (out.watchsum < out.rhs + out.amax) out.terms[i].watch(S, this)
        if (out.watchsum ≥ out.rhs + out.amax) break
    if(out.watchsum < out.rhs) return False
    if(out.watchsum < out.rhs + out.amax)
        for(int i = 0; i < size(); i++)
            Lit lit = out.terms[i].lit
            if(S.value(lit) == l_Undef)
                if(out.watchsum ≥ out.terms[i].coeff + out.rhs) break
                if(!S.enqueue(lit)) return False
    if(learnt) S.varBumpActivity(lit, coeff/out.rhs)
    return True
```

**Figure 6.** Implementation for creating and adding new *PseudoBool* constraints in Pueblo

The details of conflict analysis procedure of Pueblo based on the *analyze* method of MiniSAT is presented in Figure 8.

### 4.3.2 Backtracking and Constraint Recording

In Pueblo, the lowest decision level at which the learned PB constraint is unit is determined by checking the invariant of (5) at each decision level. If such decision level was found, the solver backtracks to the minimum level between this level and the *backtrack_level* computed in MiniSAT's *analyze* routine. Otherwise, the highest decision level at which the learned PB constraint is not violated is determined and the solver backtracks to that level or the *backtrack_level*, whichever lower. This procedure is demonstrated in Figure 9. Both learned PB and CNF constraints are recorded and their watched literals are properly setup, as demonstrated in Figure 10.

```
class accumulator
    Vec<lit_coef> terms - lit_coef is a structure with a literal and an integer coefficient
    int goal
    - set all literals in terms to undefined
    void reset()
    - check if satisfied, subtracts pb and adds CNF weakened of pb
    void checkOverSatisfaction(PseudoBool pb)
    - multiplies all coefficients by mul
    void multiplyBy(int mul)
    - adds to the coefficient of var(lit)
    void updateCoef(Lit lit, int coef)
    - returns the coef of var
    int coeff(int var)
    - return literal having var
    Lit lit(int var)
    - returns sum of coefficients of false literal at level i
    int sumAssignedFalseAtLevel(int i)
    - returns the largest coefficient among unvalued literals
    int amaxAtLevel(int i)
    - constructs a vector of literals with non-zero coefficients
    void getLiterals(Vec<Lit> lits)
    - converts all coefficients to positive by changing the sign of literals
    void normalize()
```

**Figure 7.** Implementation of the *accumulator* class in Pueblo

#### 4.3.3 Activity Heuristics

The variable activity heuristic of MiniSAT is extended to PB constraints in such a way that it recognizes the coefficient of each variable in the learned constraint. Therefore, the activity of each variable that is present in a newly learned PB constraint is increased by the ratio of its coefficient to the right-hand side of that PB constraint.

#### 4.3.4 Constraint Removal

In Pueblo, the number of active PB constraints is periodically reduced to a fixed number of constraints. This procedure basically removes all PB constraints that are not *locked* (to an implication) and are less active than a pre-set threshold limit. This limit is increased at each restart. Through our experiments on benchmarks used in the PB'05 evaluation [31], we found that an initial threshold of 50 and a growth rate of 10% produces the best results.

## 5. Experimental Analysis

We present comprehensive experimental analysis of the methods described in this paper using the benchmarks in [31]. These benchmarks include instances of logic synthesis [38], prime DIMACS [30], FPGA and global routing [1], the progressive party problem [35] and model RB [37]. Detailed results and analysis of the performance of various solvers on these benchmarks are reported in [25].

More specifically, we first compare our hybrid method against other solution methods, namely ILP and conversion-to-SAT. Next, we compare Pueblo against other PB SAT meth-

```
void analyze(Constr confl, Vec<Lit> out_learnt, int out_btlevel)
    Vec<char> seen = analyze_seen, seen.growTo(nVars(), 0)
    int pathC = 0
    Lit p = lit_Undef
    Vec<Lit> p_reason
    bool inPB = FALSE
    out_learnt.push()
    out_btlevel = 0
    do
        if(inPB)
            - finding the multipliers to generate the cutting plane such that p is removed
            int mul = confl.terms.getCoeff[var(p)]
            accumulator.multiplyBy(mul)
        p_reason.clear()
        confl.calcReason(this, p, p_reason)
        - check if accumulator is satisfied and if so undo adding and replace with weakened confl
        accumulator.checkOverSatisfaction(confl)
        for (int j = 0; j < p_reason.size(); j++)
            Lit q = p_reason[j]
            if (!seen[var(q)])
                seen[var(q)] = 1
                if (level[var(q)] == decisionLevel())
                    pathC++
                else if (level[var(q)] > 0)
                    out_learnt.push(¬q)
                    out_btlevel = ::max(out_btlevel, level[var(q)])
        - Select next constraint to look at:
        do
            p = trail.last()
            confl = reason[var(p)]
            - check if this literal should be removed from the accumulator
            inPB = accumulator.coeff(var(p)) ≠ 0 && (value(accumulator.lit(var(p))) == l_False
            undoOne()
        while(!seen[var(p)])
        seen[var(p)] = 0
        pathC–
    while (pathC > 0)
    out_learnt[0] = ¬p
```

**Figure 8.** Implementation for conflict analysis method in Pueblo

ods adopting different learning and propagation strategies. For this, we use the results of the PB'05 evaluation [25] and present a thorough analysis.

### 5.1 Comparison of Solution Methods

The results of applying three different methods for solving PB constraints, namely ILP (using XPRESS-MP [10]), pre-processing to SAT (using MiniSAT+ [17]) and PB-SAT (using Pueblo) on a representative set of the benchmarks from [31], are presented in Table 1 and Table 2. All these experiments were conducted on an AMD Opteron 2.2GHz machine with 8GB of RAM running Linux - Kernel 2.6.5.

```
void undoPB()
    accumulator.normalize() - convert all coefficients to positive
    int tmp_lhs = accumulator.goal
    for(int i = root_level; i < decisionLevel(); i++)
        - subtracts the sum of coefficients in the accumulator became false at level i
        tmp_lhs − = accumulator.sumAssignedFalseAtLevel(i)
        - check if accumulator is unit/conflict at this level
        if( tmp_lhs < accumulator.goal + accumulator.amaxAtLevel(i) )
            if(tmp_lhs < accumulator.goal) - no unit level exists
                bt_level = i-1
            else bt_level = i
            break
    cancelUntil(::max(bt_level, root_level))
```

**Figure 9.** Implementation of PB backtracking in Pueblo

```
bool recordPB(Vec<Lit> clause, int backtrack_level)
    PseudoBool pb
    Clause c
    Vec<Lit> PBLits - literals in the learned PB constraint
    accumulator.getLiterals(PBLits)
    undoPB()
    if(!PBnew(this, PBLits, pb, c)) return True - learn the PB constraint
    if(pb ≠ NULL )
        pb_learnts.push(pb)
        pbDecayActivity()
    if(c ≠ NULL)
        learnts.push(c)
        claDecayActivity()
    Clause CNFlearnt
    bool CNFunit = False
    if(decisionLevel() > backtrack_level)
        cancelUntil(::max(bt, root_level)) - backtrack to earlier level
        CNFunit = True
    check(Clause_add(this, clause, CNFlearnt)) - learn the CNF clause
    if(CNFunit) check(enqueue(clause[0], CNFlearnt))
    if(CNFlearnt ≠ NULL)
        learnts.push(CNFlearnt)
        claDecayActivity()
    return False
```

**Figure 10.** Implementation of conflict-induced constraint recording method in Pueblo

As expected, the performance of the ILP solver on problems that are under-constrained is superior to the other methods. On the other hand, SAT-based methods perform exceedingly well on problems with a high number of constraints as well as problems with a larger proportion of CNF clauses (refer to Table 3 for distributions of constraint types).

The PB SAT method of Pueblo outperforms the pure SAT method by taking advantage of cutting planes and the PB structure of the problems. Our learning methods based on cutting plane theory were able to reduce the performance gap between the ILP and SAT-based methods on those problems that were solved more efficiently in the ILP framework

**Table 1.** Experimenting using different Solving Methods for *optimization* benchmarks. The best times are bold.

| benchmark | | vars/constraints | Time (sec.) | | |
|---|---|---|---|---|---|
| | | | XPRESS-MP | MiniSAT+ | Pueblo |
| [3] | lseu | 89/29 | **0.191** | 99.021 | 331.37 |
| | p0282 | 282/222 | **0.308** | 488.40 | 4.95 |
| | mod008 | 319/7 | **0.021** | 63.133 | 0.27 |
| | nw04 | 87482/73 | 14.92 | 54.957 | **3.08** |
| | l152lav | 1989/194 | 0.043 | 232.43 | **0.03** |
| [38] | 5xp1.b | 465/860 | **2.241** | >1000 | 132.96 |
| | 9sym.b | 310/977 | 0.325 | **0.175** | 11.67 |
| | alu4.b | 808/1839 | **65.925** | >1000 | >1000 |
| | clip.b | 350/716 | **0.309** | 3.211 | 29.15 |
| | count.b | 467/695 | **0.883** | >1000 | 53.72 |
| | f51m.b | 407/539 | **0.351** | 782.61 | 33.99 |
| [37] | frb30-15-4 | 450/17832 | >1000 | **95.304** | >1000 |
| | frb30-15-5 | 450/17795 | >1000 | **31.072** | >1000 |
| [30] | aim100,3,4y1,1 | 200/441 | 51.926 | 0.014 | **0.01** |
| | aim200,1,6y1,2 | 400/521 | >1000 | 0.026 | **0.01** |
| | bf0432-007 | 2080/4709 | 806.84 | **0.033** | 0.11 |
| | ii32d3 | 1648/20303 | >1000 | **57.24** | 953.04 |
| | par16-1-c | 634/1582 | 223.59 | 0.813 | **0.59** |
| | par16-1 | 2030/4326 | 573.64 | **4.95** | 7.15 |

**Table 2.** Experimenting using different Solving Methods for *decision* benchmarks. The best times are bold.

| benchmark | | vars/constraints | Time (sec.) | | |
|---|---|---|---|---|---|
| | | | XPRESS-MP | MiniSAT+ | Pueblo |
| [1] | fpga35_33sat | 1733/1256 | 0.557 | >1000 | **0.3** |
| | fpga35_35sat | 1838/1330 | 0.941 | 52.452 | **0.35** |
| | fpga40_39sat | 2340/1678 | 0.955 | 203.43 | **0.84** |
| | fpga40_40sat | 2400/1720 | 1.252 | 468.60 | **0.34** |
| | fpga45_44sat | 2970/2113 | 5.33 | >1000 | **2.71** |
| [35] | ppp:1-12,16 | 4662/5691 | 204.50 | 6.723 | **1.97** |
| | ppp:1-13 | 4632/35770 | >1000 | 574.83 | **4.32** |
| | ppp:1,3-13,19 | 4608/35130 | >1000 | 125.73 | **2.81** |
| | ppp:1-9,16-19 | 4626/5649 | >1000 | >1000 | **62.97** |

without compromising the advantages of SAT-based techniques for over-constrained problems. Specifically, in the representative decision benchmarks of Table 2, the hybrid PB SAT approach performs considerably better on more benchmarks than the other two methods.

Note that these benchmarks are especially good for a hybrid solver such as Pueblo, because the input is already hybrid.

It is worth mentioning that the overall performance of our SAT-based method on optimization problems can be further improved by applying the lower bound computation in [26]; such local estimates are routine in ILP solvers and contributes to their robust performance.

### 5.2 Comparison of PB SAT Methods

Figure 11 compares the performance of Pueblo against `galena` and PBS on the representative benchmark set of Table 3. In Table 3 the distribution of each type of constraint, namely unrestricted PB, cardinality and CNF, in each benchmark suite is presented. The data of Figure 11 are from the PB'05 evaluation [31]. For details of the settings, benchmarks and results, the reader is referred to [25].

The first two rows in this figure show the comparison between Pueblo and, respectively, `galena` and PBS for different *optimization* problems, whereas the second two rows correspond to the comparison for different *satisfiability* problems. The X and Y axes show, respectively, Pueblo and each of the competitors' execution times. A dot above the diagonal line indicates that Pueblo's performance is better than the corresponding competitor, and vice versa. The two uppermost horizontal lines and the two rightmost vertical lines represent, respectively, benchmarks that ended in UNKNOWN or SAT instead of OPT. According to the notations used in the PB'05 evaluation [31, 25], in these benchmarks, `galena` and Pueblo reported SAT for optimization problems when "solver found a solution (*s SATISFIABLE* was output)" and PBS did not provide SAT results to any of the optimization problems. In these benchmarks, UNKNOWN results reported for `galena` corresponded to cases when "solver couldn't decide (*s UNKNOWN* was output)" or "solver was terminated by a signal (SIGSEGV for example) and didn't output a solution line" and UNKNOWN results for PBS and Pueblo corresponded to the cases when "solver exceeded the time limit and gave no answer". Figure 12 demonstrates the comparison of Pueblo against `galena` and PBS on all benchmarks of Table 3 collectively as well as on SMALLINT MIPLIB optimization benchmarks of [31] where the majority (more than 90% in average) of the constraints are of PB and cardinality types. For more details on these benchmarks, the reader is referred to [25].

`galena` is based on the work in [6] and utilizes cutting plane PB learning and post-reduction of learned PB constraints to cardinality constraints. PBS, on the other hand, is based on CNF learning [1]. These results show that general PB learning in many applications performs comparably or better than learning strategies producing CNF or cardinality constraints. For instance in routing benchmark suites (right most column in Figure 11), learning unrestricted PB constraints enables the solver to considerably prune the search space and consequently reduces the solve time in such a way that Pueblo could quickly (in a few seconds) solve instances that `galena` and PBS were unable to solve.

The comparison of Pueblo and `galena` demonstrates the effectiveness of each solver's strategy to control the overhead of learning PB constraints in different benchmarks. As discussed earlier, Pueblo adopts simultaneous CNF and PB learning and periodic discarding

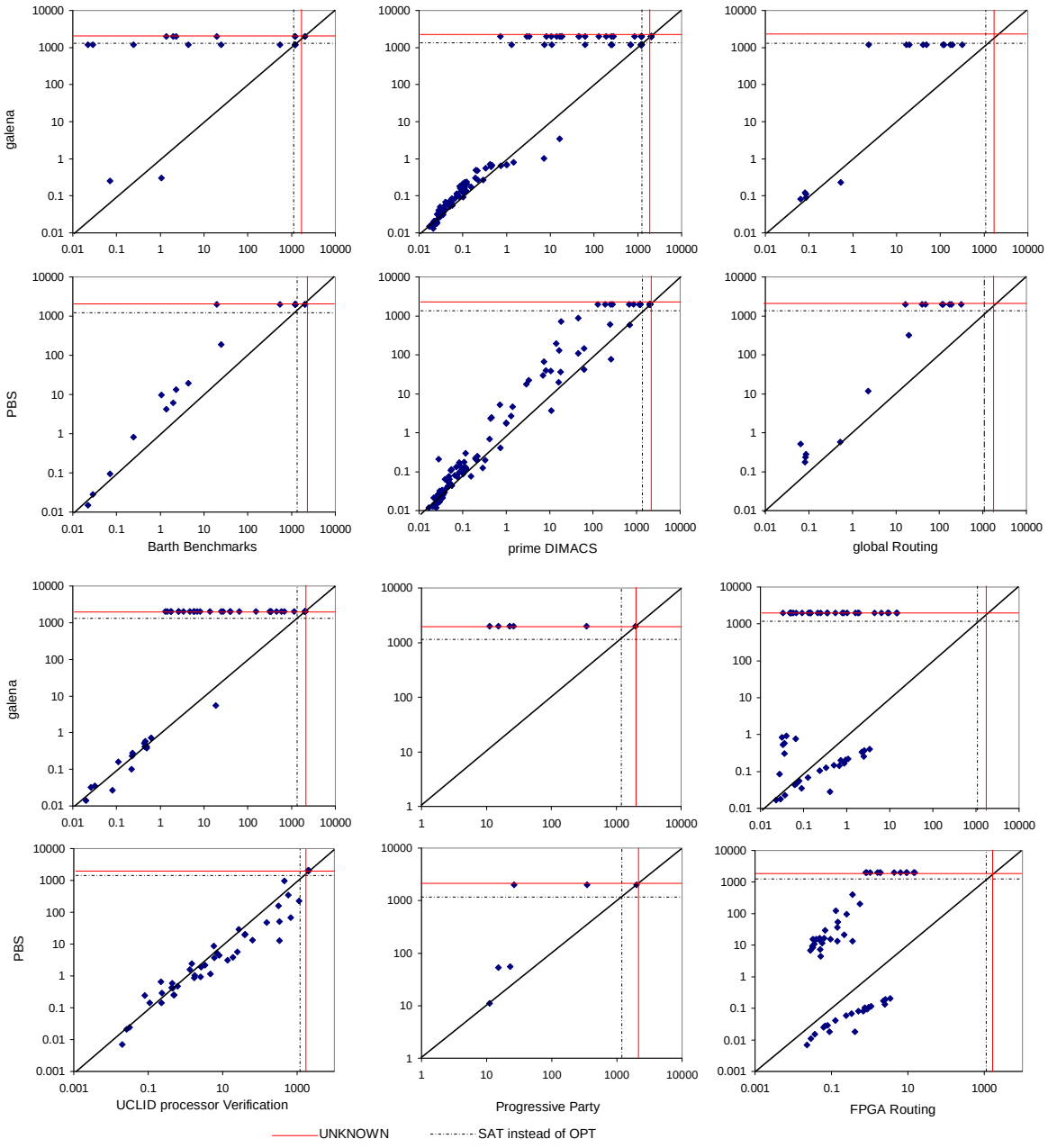


**Figure 11.** Results of PB'05 evaluations for different benchmark suites, comparing Pueblo (X-axis) to galena/PBS (Y-axis).

of learned PB constraints, while `galena` weakens the learned PB constraints to cardinality constraints.

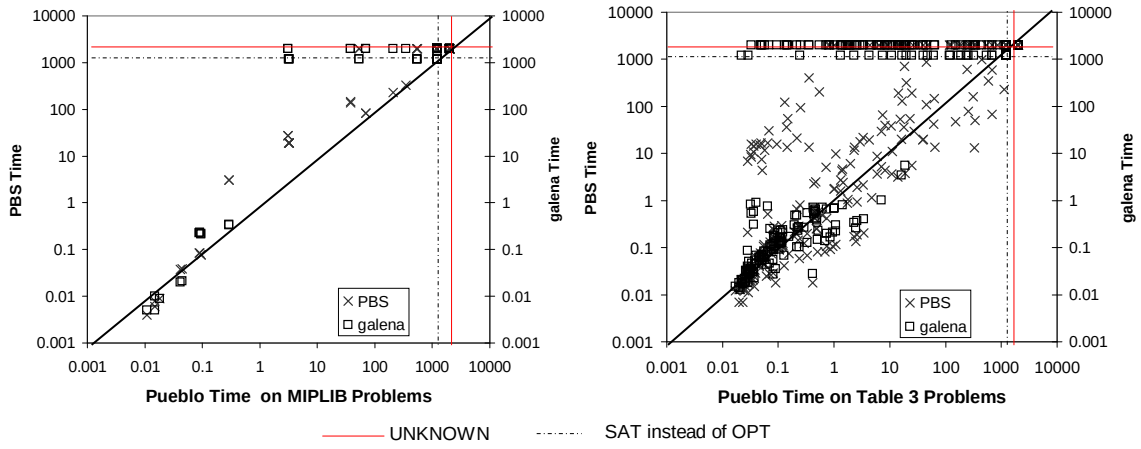


**Figure 12.** Results of PB'05 evaluations for MIPLIB problems as well as all benchmarks of Table 3, comparing Pueblo to galena/PBS

**Table 3.** Benchmarks distributions of constraint types per instance in each suite

| Benchmark Suite | # of Inst's | avg # of Constraints | avg % PB | avg % CARD | avg % CNF |
|---|---|---|---|---|---|
| Barth Suite [3] | 20 | 727 | 21.7% | 30.5% | 47.8% |
| prime DIMACS [30] | 156 | 9707 | 0% | 0.2% | 99.8% |
| global Routing [1] | 15 | 1579 | 0% | 1.5% | 98.5% |
| UCLID proc. verification [5] | 50 | 79522 | 8.8% | 0% | 91.2% |
| Progressive Party [35] | 6 | 35672 | 0.2% | 15.6% | 84.2% |
| FPGA Routing [1] | 57 | 480 | 0% | 22.7% | 77.3% |

## 6. Conclusions and Future Work

In this paper, we covered learning methods based on cutting planes for solving PB constraints and compared our method to pure SAT and ILP methods as well as to other PB SAT methods. On the learning front, we proposed a hybrid procedure that produces both a PB and a CNF constraint after each conflict. We described the major challenges for handling learned PB constraints and experimentally demonstrated the advantages of our hybrid method with periodic PB constraint removal to overcome those barriers.

The integration of logic-based reasoning and integer programming methods promises to be a vibrant area of research for the next several years. Inference-based learning as discussed in this paper has proven to be very successful when combined with SAT solver algorithms for solving a wide range of problems including hardware verification and synthesis. In addition to adopting new methods, better integration strategies that could control the high overhead of PB constraints could be a viable path for further research.

## Acknowledgement

This work was funded by the National Science Foundation under ITR grant No. 0205288. We are grateful to Mr Daniel Le Berre for his helpful suggestions and also would like to thank the reviewers for their valuable comments.

## References

[1] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the 2002 IEEE/ACM international Conference on Computer-Aided Design ICCAD'02*, pages 450–457, 2002.

[2] P. Barth. *Logic-based 0-1 Constraint Programming*. Kluwer, Norwell MA, 1995.

[3] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Saarbrücken, 1995.

[4] R. E. Bixby, E. A. Boyd, and R. R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, **25**:16, 1992.

[5] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU logic formulas via boolean and pseudo-boolean encodings, 2002.

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

[7] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Math.*, **4**(3):305–337, 1973.

[8] Nadia Creignou, Sanjeev Khanna, and Madhu Sudan. *Complexity classifications of boolean constraint satisfaction problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[9] H. Crowder, E. Johnson, and M. W. Padberg. Solving large-scale zero-one linear programming problems. In *Operations Research*, volume **31**, pages 803–834, 1983.

[10] Dash Inc. *XPRESS-MP version 15.25.03*. http://www.dashoptimization.com/.

[11] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, **5**(7):394–397, July 1962.

[12] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, **41**(3):273–312, 1990.

[13] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Eighteenth national conference on Artificial intelligence*, pages 635–640, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[14] Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing boolean satisfiability i: Background and survey of existing work. *Journal of Artificial Intelligence Research*, **21**:193–243, 2004.

[15] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[16] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Conference on Theory and Applications of Satisfiability Testing SAT*, pages 502–518, 2003.

[17] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, volume **2**, pages 1–26, 2006.

[18] Daniel Frost and Rina Dechter. Dead-end driven learning. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 294–300, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[19] Matthew L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res. (JAIR)*, **1**:25–46, 1993.

[20] R. E. Gomory. An algorithm for integer solutions to linear programs. In *Recent Advences in Mathematical Programming*, pages 269–302. McGraw-Hill, Inc., 1963.

[21] M. Grotschel, M. Junger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. In *Operations Research*, volume **32**, pages 1195–1220, 1984.

[22] J. N. Hooker. Generalized resolution and cutting planes. *Ann. Oper. Res.*, **12**(1-4):217–239, 1988.

[23] ILOG Inc. *CPLEX version 9*. http://ilog.com/products/cplex/.

[24] Roberto Bayardo Jr. and Daniel Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI '96), Volume* **1**, pages 298–304. AAAI Press / The MIT Press, August 4-8 1996.

[25] V. M. Manquinho and O. Roussel. The first evaluation of pseudo-boolean solvers (PB05). *Journal on Satisfiability, Boolean Modeling and Computation*, volume **2**, pages 103-143, 2006.

[26] Vasco M. Manquinho and João P. Marques-Silva. Effective lower bounding techniques for pseudo-boolean optimization. In *Proceedings of the Design, Automation and Test in Europe DATE'05*, pages 660–665, 2005.

[27] Vasco M. Manquinho and João P. Marques-Silva. On applying cutting planes in DLL-based algorithms for pseudo-boolean optimization. In *Eighth International Conference on Theory and Applications of Satisfiability Testing SAT'05*, pages 451–458, 2005.

[28] P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, **48**(5):506–521, 1999.

[29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *38th Design Automation Conference*, pages 530–535, 2001.

[30] Clara Pizzuti. Computing prime implicants by integer programming. In *ICTAI*, pages 332–336, 1996.

[31] Pseudo-Boolean Evaluation PB'05. http://www.cril.univ-artois.fr/PB05/.

[32] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *International Journal on Artificial Intelligence Tools (IJAIT)*, **3**(2):187–207, 1994.

[33] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A modern pseudo-boolean sat solver. In *Proceedings of the Design, Automation and Test in Europe DATE'05*, pages 684–685, 2005.

[34] R. Stallman and G. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, **9**:135–196, 1977.

[35] Joachim P. Walser. Solving linear pseudo-boolean constraint problems with local search. In *AAAI/IAAI*, pages 269–274, 1997.

[36] Laurence A. Wolsey. *Integer Programming*. John Wiley and Sons, 1998.

[37] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 337–342, 2005.

[38] S. Yang. Logic synthesis and optimization benchmarks user guide version, 1991.