Learning Techniques for Pseudo-Boolean Solving

José Santos
IST/UTL, INESC-ID
Lisbon, Portugal
jffs@sat.inesc-id.pt

Vasco Manquinho
IST/UTL, INESC-ID
Lisbon, Portugal
vmm@sat.inesc-id.pt

Abstract

The extension of conflict-based learning from Propositional Satisfiability (SAT) solvers to Pseudo-Boolean (PB) solvers comprises several different learning schemes. However, it is not commonly agreed among the research community which learning scheme should be used in PB solvers. Hence, this paper presents a contribution by providing an exhaustive comparative study between several different learning schemes in a common platform. Results for a large set of benchmarks are presented for the different learning schemes, which were implemented on `bsolo`, a state of the art PB solver.

1 Introduction

Pseudo-Boolean (PB) solvers have been the subject of recent research [4, 7, 10, 17], namely in applying techniques already successful in propositional satisfiability (SAT) solvers. In these algorithms, one of the most important techniques is the generation of no-goods during search [1, 4, 17]. This is typically done when a conflict arises, i.e, a given problem constraint is unsatisfiable. In that situation, a conflict analysis procedure is carried out and a no-good constraint is added to the constraint set so that the same conflicting assignment does not occur again.

In PB solvers, several conflict-based learning procedures have been proposed. However, there is no a consensus on which method is best for most set of instances [3]. In this paper, we start by defining the pseudo-Boolean solving problem and some operations that can be performed on PB constraints. Next, PB algorithms and conflict analysis procedures are briefly surveyed. Some implementation issues are described in section 5. Experimental results are presented for several learning procedures implemented on a state of the art pseudo-Boolean solver. Finally, conclusions are presented in section 7.

2 Preliminaries

In a propositional formula, a literal l_j denotes either a variable x_j or its complement $\bar{x_j}$. If a literal $l_j = x_j$ and x_j is assigned value 1 or $l_j = \bar{x_j}$ and x_j is assigned value 0, then the literal is said to be true. Otherwise, the literal is said to be false. A pseudo-Boolean (PB) constraint is defined as a linear inequality over a set of literals of the following normal form:

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

such that for each $j \in \{1, \cdots, n\}$, $a_j \in Z^+$ and l_j is a literal and $b \in Z^+$. It is well-known that any PB constraint (with negative coefficients, equalities or other inequalities) can be converted into the normal form in linear time [2]. In the remaining of the paper, it is assumed that all PB constraints are in normal form.

In a given constraint, if all a_j coefficients have the same value k, then it is called a cardinality constraint, since it requires that $\lceil b_i/k \rceil$ literals be true in order for the constraint to be satisfied. A PB constraint where any literal set to true is enough to satisfy the constraint, can be interpreted as a propositional clause. This occurs when the value of all a_j coefficients are greater than or equal to b.

An instance of the Pseudo-Boolean Satisfiability (PB-SAT) problem can be defined as finding an assignment to the problem variables such that all PB constraints are satisfied.

2.1 Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) [6] in PB algorithms is a generalization of the application of the *unit clause rule* [6] used in propositional satisfiability (SAT) algorithms. A constraint ω is said to be *unit* if given the current partial assignment, ω is not currently satisfied and there is at least one literal l_j that if assigned value 0, ω cannot be satisfied. Hence, l_j must be assigned value 1. Let $a^u_{\max}$ denote the largest coefficient of the unassigned literals in ω. Moreover, let the *slack* s of constraint ω to be defined as follows:

$$s = (\sum_{l_j \neq 0} a_j) - b \tag{2}$$

If $a^u_{\max} > s$ then ω is a unit constraint and literal l^u_{max} must be assigned value 1. Notice that given the same partial assignment, more than one literal can be implied by the same PB constraint.

During the search, let $x_j = v_x@k$ denote the assignment of v_x to variable x_j at decision level k. In the following sections we often need to associate *dependencies* (or an *explanation*) with each implied variable assignment. Dependencies represent sufficient conditions for variable assignments to be implied. For example, let $x_j = v_x$ be a truth assignment implied by applying the unit clause rule to ω. Then the explanation for this assignment is the set of assignments associated with the false literals of ω.

2.2 Constraint Operations

It is well-known that the fundamental operation used to infer new constraints using propositional clauses is *resolution* [14]. For PB constraints, instead of resolution, the technique of cutting planes [5, 9] can be used. This operation allows the linear combination of a set of constraints, in order to generate a new constraint that is a logical consequence of the original constraints. For any two pseudo-Boolean constraints and coefficients c and c' we can combine them as follows:

$$\frac{\begin{array}{c} c(\sum_j a_j x_j \geq b) \\ c'(\sum_j a'_j x_j \geq b') \end{array}}{c\sum_j a_j x_j + c'\sum_j a'_j x_j \geq cb + c'b'} \tag{3}$$

If c or c' is non-integer, a new constraint with non-integer coefficients may result after applying the cutting plane operation. In order to obtain a new constraint with integer coefficients, the *rounding* operation can be applied as follows:

$$\frac{\sum_j a_j x_j \geq b}{\sum_j \lceil a_j \rceil x_j \geq \lceil b \rceil} \tag{4}$$

It should be noted that the rounding operation might weaken the constraint such that the number of models of the resulting constraint is larger. For example, suppose we have the constraint $\omega : 2x_1 + x_2 + x_3 \geq 3$. Applying a coefficient $c = 0.5$ to ω we get a new constraint ω':

$$\omega' : x_1 + 0.5x_2 + 0.5x_3 \geq 1.5 \tag{5}$$

Applying the rounding operation to ω' results in a new constraint $\omega'_r : x_1 + x_2 + x_3 \geq 2$. Clearly, all models of ω' are also models of ω'_r, but not all models of ω'_r are models of ω'.

Another operation that can be used on PB constraints is *reduction* [2]. This operation allows the removal of literals by subtracting the value of the coefficient from the right hand side.

Consider the constraint $\omega : x_1 + 3x_2 + 3\bar{x_3} + 2\bar{x_4} + x_5 \geq 7$. If reduction is applied to ω in order to remove x_1 and $\bar{x_3}$, we would get $\omega' : 3x_2 + 2\bar{x_4} + x_5 \geq 3$. Note that if ω is a problem instance constraint, it is not possible to replace ω with ω'. Constraint ω' is a logical consequence from ω, but the converse is not true.

Algorithms to generate cardinality constraints from general pseudo-Boolean constraints can be found in [2, 4]. These algorithms find the minimum number of literals that must be set to 1 in order to satisfy the constraint. This is achieved by accumulating the literal coefficients of the constraint, in decreasing order, starting with the highest a_j. Let $m(\omega)$ denote the minimum number of literals to be set to true in order to satisfy constraint ω. Then, cardinality reduction can be defined as follows:

$$\frac{\omega : \sum_j a_j l_j \geq b}{\sum_j l_j \geq m(\omega)} \tag{6}$$

One should note that the resulting constraint is weaker than the original general pseudo-Boolean constraint. More details for cardinality reduction can be found in [4], which presents a stronger cardinality reduction.

3 Pseudo-Boolean Algorithms

Generally, pseudo-Boolean satisfiability (PB-SAT) algorithms follow the same structure as propositional satisfiability (SAT) backtrack search algorithms. The search for a satisfying assignment is organized by a decision tree (explored in depth first) where each node specifies an assignment (also known as *decision assignment*) to a previously unassigned problem variable. A *decision level* is associated with each decision assignment to denote its depth in the decision tree.

Algorithm 1 shows the basic structure of a PB-SAT algorithm. The `decide` procedure corresponds to the selection of the decision assignment thus extending the current partial assignment. If a complete assignment is reached, then a solution to the problem constraints has been found. Otherwise, the `deduce` procedure applies Boolean Constraint Propagation (and possibly other inference methods). If a conflict arises, i.e. a given constraint cannot be satisfied by extending the current partial assignment, then a conflict analysis [11] procedure is carried out to determine the level to which the search process can safely backtrack to. Moreover, a no-good constraint is also added to the set of problem constraints.

The main goal of the conflict analysis procedure is to be able to determine the correct explanation for a conflict situation and backtrack (in many cases non-chronologically) to a decision level such that the conflict does no longer occur. Moreover, a no-good constraint results from this process. However, the strategy for no-good generation that results from the conflict analysis differs between several state of the art PB-SAT solvers. These different strategies are reviewed and analysed in section 4.

Another approach to PB-SAT is to encode the problem constraints into a propositional satisfiability (SAT) problem [8] and then use a powerful SAT solver on the new formulation. Although this approach is successful for some problem instances [8], in other cases the resulting SAT formulation is much larger than the original PB formulation so that it provides a huge overhead to the SAT solver.

Algorithm 1 Generic Structure of Algorithms for PB-SAT Problem

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

4 Conflict Analysis

Consider that the assignment of a problem variable x_j is inferred by Boolean Constraint Propagation due to a PB constraint ω_i. In this case, ω_i is referred to as the *antecedent constraint* [11] of the assignment to x_j. The *antecedent assignment* of x_j, denoted as $A^{\omega}(x_j)$, is defined as the set of assignments to problem variables corresponding to false literals in ω_i. Similarly, when ω_i becomes unsatisfied, the antecedent assignment of its corresponding conflict, $A^{\omega}(k)$, will be the set of all assignments corresponding to false literals in ω.

The implication relationships of variable assignments during the PB-SAT solving process can be expressed as an *implication graph* [11]. In the implication graph, each vertex corresponds to a variable assignment or to a conflict vertex. The predecessors of a vertex are the other vertexes corresponding to the assignments in $A^{\omega_i}(x_j)$. Next, several learning schemes are presented that result from analyzing the implication graph in a conflict situation.

4.1 Propositional Clause Learning

When a logical conflict arises, the implication sequence leading to the conflict is analysed to determine the variable assignments that are responsible for the conflict. The conjunction of these assignments represents a sufficient condition for the conflict to arise and, as such, its negation must be consistent with the PB formula [11]. This new constraint (known as the *conflict constraint*), is then added to the PB formula in order to avoid the repetition of the same conflict situation and thus pruning the search space.

When an assignment to a problem variable x_j is implied by a PB constraint ω_i, one can see it as the conjunction of $A^{\omega_i}(x_j)$ implying the assignment to x_j. Moreover, when a constraint implies a given assignment, the coefficient reduction rule can be used to eliminate all positive and unassigned literals except for the implied literal. Hence, the conflict analysis used in SAT solvers by applying a sequence of resolution steps in a backward traversal of the implication graph can be directly applied to PB formulas [11]. Additionally, techniques such as the detection of Unique Implication Points (UIPs) [11, 18] can also be directly used in PB-SAT conflict analysis. As a result, a new propositional clause is generated and added to the original formula.

In the last pseudo-Boolean solver evaluation, some PB solvers used this approach, namely `bsolo` [10] and `PBS4` (an updated version of the original `PBS` solver [1]). This strategy is simple to implement in PB solvers, since it is a straightforward generalization of the one used in SAT

solvers. Moreover, considering the use of lazy data structures [12] for clause manipulation, the overhead of adding a large number of clauses during the search is smaller than with other types of constraints.

4.2 Pseudo-Boolean Constraint Learning

The use of PB constraint learning is motivated by the fact that PB constraints are more expressive. It is known that a single PB constraint can represent a large number of propositional clauses. Therefore, the potential pruning power of PB conflict constraints is much larger than that of propositional clauses.

The operation on PB constraints which corresponds to clause resolution is the *cutting plane* operation (section 2.2). As such, to learn a general PB constraint, the conflict analysis algorithm must perform a sequence of cutting plane steps instead of a sequence of resolution steps. In each cutting plane step one implied variable is eliminated. As with the clause learning conflict analysis, a backward traversal of the implication graph is performed and the implied variables are considered in reverse order. The procedure stops when the conflict constraint is unit at a previous decision level (1UIP cut) [18].

After the conflict analysis, the algorithm uses the learned constraint to determine to which level it must backtrack as well as implying a new assignment after backtracking. Therefore, the learned constraint must be unsatisfied under the current assignment. Moreover, it must also be an assertive constraint (must become unit after backtracking).

Consider the application of a cutting plane step to two arbitrary constraints ω_1 with slack s_1 and ω_2 with slack s_2. Moreover, consider that α and β are used as the multiplying factors. In this situation, the slack of the resulting constraint, here denoted by s_r, is given by linearly combining the slacks of ω_1 and ω_2: $s_r = (\alpha \cdot s_1) + (\beta \cdot s_2)$. As such, before the application of each cutting plane step, the learning algorithm verifies if the resulting constraint is still unsatisfied under the current assignment. If it is not, the implied constraint must be reduced to lower its slack [4, 17]. This process is guaranteed to work since the repeated reduction of constraints will eventually lead to a simple clause with slack 0.

Algorithm 2 presents the pseudo-code for computing the conflict-induced PB constraint $\omega(k)$. This algorithm performs a sequence of cutting plane steps, starting from the unsatisfied constraint $\omega(c)$. Notice that this algorithm can also implement a clause learning scheme. In this case, functions `reduce1` and `reduce2` remove all non-negative literals in $\omega(k)$ and ω_i, respectively, except for the implied literal. Next, these procedures can trivially reduce the obtained constraint to a clause. In order to implement a general PB learning scheme, function `reduce2` must eliminate only enough non-negative literals in ω_i to guarantee that after the cutting plane step, the resulting constraint remains unsatisfied at the current decision level.

4.3 Other Learning Schemes

Learning general PB constraints slows down the deduction procedure because the watched literal strategy is not as efficient with general PB constraints as it is with clauses or cardinality constraints [4, 16]. Note that in a clause, as well as in a cardinality constraint, it is only necessary to watch a fixed number of literals, whereas in a general PB constraint the number of watched literals varies during the execution of the algorithm.

The approach for cardinality constraint learning used in `Galena` [4] is based on the approach described for the general pseudo-Boolean learning scheme. The difference is that a

Algorithm 2 Generic Pseudo-Boolean Learning Algorithm

```
// W corresponds to the set of constraints in the PB formula and ω(c) to the conflicting constraint
V ← {x_i | x_j corresponds to a false literal in ω(c) at current decision level};
ω(k) ← reduce1(w(c))
while TRUE do
  x_j ← removeNext(V);
  ω_i ← implyingConstraint(x_j);
  if (w_i ≠ NULL ∧ |V| > 1) then
    ω'_i ← reduce2(ω_i, ω(k));
    ω(k) ← cutResolve(ω(k), ω'_i, x_j);
    V ← V \ {x_j} ∪ {x_k | x_k corresponds to a false literal in ω'_i at current decision level}
  else
    ω(k) ← reduce3(ω(k));
    Add ω(k) to W;
    btLevel ← assertingLevel(ω(k));
    if btLevel < 0 then
      return CONFLICT;
    else
      backtrack(btLevel);
      return NO_CONFLICT;
    end if
  end if
end while
```

post-reduction procedure is carried out so that the learned constraint is reduced into a weaker cardinality constraint. In Algorithm 2 this would be done in function `reduce3`.

Finally, a hybrid learning scheme was already proposed and used in `Pueblo` [17]. The authors noted that any solver which performs PB learning can be modified to additionally perform clause learning with no significant extra overhead. Moreover, despite the greater pruning power of PB learning, clause learning has its own advantages: it always produces an assertive constraint and it does not compromise as heavily the propagation procedure as general PB learning. As such, in their solver `Pueblo`, they implement a hybrid learning method [17].

5 Implementation Issues

In implementing a pseudo-Boolean solver, several technical issues must be addressed. In this section the focus is on generating the implication graph and on the use of cutting planes for PB constraint learning.

5.1 Generating the Implication Graph

It is widely known that lazy data structures [12, 17] for constraint manipulation are essential for the solver's performance. Nevertheless, the order of propagation of variable assignments in BCP has not been thoroughly studied. In the version of `bsolo` submitted to the last PB solver evaluation (PB'07) [15], the order of propagation in the implication graph was depth-first. In this paper it is shown that by changing it to a breadth-first propagation, `bsolo` was able to solve a larger number of instances (see section 6).

When generating the implication graph in a breadth-first way, one can guarantee that there is no other possible implication graph such that the length of the longest path between the

decision assignment vertex and the conflict vertex is lower than the one considered. Therefore, the learned constraint is probably determined using a smaller number of constraints. Moreover, considering that in PB formulations the same constraint can imply more than one variable assignment, the motivation for a breadth-first propagation is larger than in SAT formulations.

5.2 Dealing with Large Coefficients

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

One problem that may occur in the cutting plane operation is integer overflow. To avoid this problem, a maximum coefficient size was established (we used 10^6). Therefore, every time the solver performs a cutting plane step, all coefficients of the resulting constraint are checked if one of them is bigger than the established limit. If it is, the solver repeatedly divides the constraint by 2 (followed by rounding) until its largest coefficient is lower than a second maximum coefficient size (we used 10^5).

During the conflict analysis the accumulator constraint must always have negative slack. However the division rule does not preserve the slack of the resulting constraint, since it does not guarantee that the slack of the resulting constraint is equal to the slack of the original one, which can be verified in the next example where the constraint is divided by 2, followed by rounding:

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

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

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

5.3 An Initial Classification Step

After implementing different learning schemes, it was observed that each of the versions proved to be more effective than the others for some group of instances. One can easily conclude that different learning schemes behave better (or worse) depending on some characteristics of the instances given as input. For instance, a cardinality constraint learning scheme is more appropriate to deal with an instance with a large number of cardinality constraints than a clause learning scheme.

Our goal was then to try to define an initial classification step that could choose the best fitting learning scheme to solve a given problem instance. Therefore, in a very preliminary work, algorithm C4.5 [13] was used to generate a decision tree that given a problem instance,

determines which learning scheme is more appropriate to it. The classification of each instance was done according to structural attributes of the formula, namely number of variables, literals, types of constraints, among others.

6 Experimental Results

This section presents the experimental results of applying different learning schemes to the small integer non-optimization benchmarks from the PB'07 evaluation [15]. All the learning schemes were implemented on top of `bsolo`, a state of the art PB solver. Experimental results were obtained on a Intel Xeon 5160 server (3.0Ghz, 4GB memory) running Red Hat Enterprise Linux WS 4. The CPU time limit for each instance was set to 1800 seconds.

In Table 1, results for several learning schemes are presented. Each line of the table represents a set of instances depending on their origin or encoded problem. Each column represents a version of `bsolo` for a different learning scheme. Finally, each cell denotes the number of benchmark instances that were found to be satisfiable/unsatisfiable/unknown.

The basis for our work was the `bsolo` version submitted to PB'07 solver evaluation. This version implements a clause learning scheme and is identified with `CL1`. Version `CL2` corresponds to the previous version, but with a breadth-first approach for generating the implication graph. Next, results for the general PB learning scheme `PB` and cardinality learning scheme `CARD` are presented. One should note that both the `PB` and `CARD` learning schemes are hybrid (as in `Pueblo`) and always learn an assertive clause. Preliminary results on pure PB and cardinality learning schemes were not as effective as an hybrid learning scheme. In version `COMB`, an initial classification step was used in order to select the best fitting learning scheme for each instance (see section 5.3). The training set for the `COMB` version was composed of 100 instances (out of the total of 371 instances).

It can be observed from Table 1 that the original solver was greatly improved just by changing the propagation order. Version `CL2` was able to solve more 17 instances, most of them in the tsp benchmark set. Both the `PB` and `CARD` learning schemes improve on the original version of the solver. However, in the `PB` version, the overhead associated with maintaining the additional no-good PB constraints is a drawback, in particular on the FPGA and pigeon hole instances. Overall, the `CARD` learning scheme performs much better, proving to be a nice compromise between pruning power of the generated constraints, and the underlying overhead of constraint manipulation. Finally, the `COMB` version shows that a combination of all these learning schemes allows an even better performance. Finally, one should note that improvements are essentially on unsatisfiable instances. The gain on satisfiable instances is smaller.

In Table 2, the results of the best known solvers can be checked and compared with `bsolo`. `Pueblo` is able to solve 6 more instances than the current version of `bsolo`. By using a hybrid learning scheme, `Pueblo` seems to have found a nice balance between the additional overhead of new no-good constraints (mostly clauses) and the pruning power of new no-good PB constraints. Nevertheless, the work presented in this paper shows that `bsolo` can be competitive in a large set of problem instances.

In the Pseudo-Boolean Optimization (PBO) problem, the objective is to find an assignment such that all constraints are satisfied and a linear cost function is optimized. PB solvers can be easily modified [2] in order to also tackle this problem. Table 3 presents the results of the new version of `bsolo` in comparison with other solvers. Each cell contains the following information: the number of instances for which the optimum value was found, the number of satisfiable but non-optimal solutions, the number of unsatisfiable instances and the number of

Table 1: Results for all different learning schemes on non-optimization benchmarks

Benchmark	CL1	CL2	CARD	PB	COMB
armies	5/0/7	4/0/8	6/0/6	5/0/7	7/0/5
dbst	15/0/0	15/0/0	15/0/0	15/0/0	15/0/0
FPGA	36/2/19	36/2/19	36/21/0	35/1/21	36/21/0
pigeon	0/2/18	0/2/18	0/19/1	0/1/19	0/19/1
prog. party	4/0/2	3/0/3	4/0/2	4/0/2	3/0/3
robin	3/0/3	4/0/2	2/0/4	4/0/2	4/0/2
tsp	40/33/27	40/50/10	40/44/16	40/43/17	40/49/11
uclid	1/39/10	1/40/9	1/43/6	1/43/6	1/41/8
vdw	1/0/4	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0	32/68/0
Total	137/144/90	136/162/73	137/195/39	137/156/78	139/198/34

Table 2: Comparison with other solvers on non-optimization benchmarks

Benchmark	bsolo	Pueblo	minisat+	PBS4
armies	7/0/5	6/0/6	8/0/4	9/0/3
dbst	15/0/0	15/0/0	7/0/8	15/0/0
FPGA	36/21/0	36/21/0	33/3/21	26/21/10
pigeon	0/19/1	0/13/7	0/2/18	0/20/0
prog. party	3/0/3	6/0/0	5/0/1	3/0/3
robin	4/0/2	3/0/3	4/0/2	3/0/3
tsp	40/49/11	40/60/0	39/46/15	40/52/8
uclid	1/41/8	1/42/7	1/46/3	1/44/5
vdw	1/0/4	1/0/4	1/0/4	1/0/4
wnqueen	32/68/0	32/68/0	32/68/0	32/68/0
Total	139/198/34	140/203/28	130/165/76	130/205/36

unknown instances. Clearly, bsolo is able to prove optimality for a larger number of instances than other solvers. Additionally, due to the new learning schemes, bsolo is also able to find a larger number of non-optimal solutions.

7 Conclusions

Considering the disparity of results concerning the application of learning schemes in several state of the art PB solvers, the main goal of this work is to provide a contribution by implementing them in the same platform. Our results confirm that hybrid learning schemes perform better on a large set of instances. Moreover, our results show that cardinality constraint learning is more effective and robust learning scheme than others. It obtained much better results than the original clause learning scheme and also on our implementation of the PB hybrid learning scheme included in Pueblo. In our opinion, cardinality constraints are easier to propagate than PB constraints and are also more expressive than clauses. Therefore, this learning scheme seems a reasonable compromise between PB learning and pure clause learning.

References

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

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

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

Table 3: Comparison with other solvers on optimization benchmarks

Benchmark	bsolo	PBS4	minisat+	Pueblo
aksoy	26/52/0/1	11/63/0/5	25/45/0/9	15/64/0/0
course-ass	1/4/0/0	0/4/0/1	2/1/0/2	0/5/0/0
domset	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
garden	1/2/0/0	0/3/0/0	1/2/0/0	1/2/0/0
haplotype	0/8/0/0	0/8/0/0	8/0/0/0	0/8/0/0
kexu	0/40/0/0	0/40/0/0	11/29/0/0	0/40/0/0
logic-synthesis	51/23/0/0	19/55/0/0	31/41/0/2	32/42/0/0
market-split	4/16/4/16	0/20/0/20	0/20/0/20	4/16/4/16
mps-v2-20-10	12/18/1/1	7/21/1/3	10/15/1/6	10/18/1/3
numerical	12/18/0/4	12/11/0/11	10/9/0/15	14/17/0/3
primes-dimacs-cnf	69/35/8/18	69/36/8/17	79/26/8/17	75/30/8/17
radar	6/6/0/0	0/12/0/0	0/12/0/0	0/12/0/0
reduced	17/80/35/141	14/77/14/168	17/10/23/213	14/92/18/149
routing	10/0/0/0	4/6/0/0	10/0/0/0	10/0/0/0
synthesis-ptl-cmos	6/2/0/0	0/8/0/0	1/7/0/0	1/7/0/0
testset	6/0/0/0	4/2/0/0	5/1/0/0	6/0/0/0
ttp	2/6/0/0	2/6/0/0	2/6/0/0	2/6/0/0
vtxcov	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
wnq	0/15/0/0	0/15/0/0	0/15/0/0	0/15/0/0
Total	223/355/48/181	142/417/23/225	212/264/35/301	184/404/31/188

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

[5] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.

[7] H. Dixon, M. Ginsberg, E. Luks, and A. Parkes. Generalizing Boolean Satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.

[8] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

[9] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[10] V. Manquinho and J. P. Marques-Silva. Effective lower bounding techniques for pseudo-boolean optimization. In *Proceedings of the Design and Test in Europe Conference*, pages 660–665, March 2005.

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

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

[13] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[15] O. Roussel and V. Manquinho. Third pseudo-boolean evaluation 2007. http://www.cril.univ-artois.fr/PB07, 2007.

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

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

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