Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TIMO BERTHOLD STEFAN HEINZ MARC E. PFETSCH

Solving Pseudo-Boolean Problems with SCIP

Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

ZIB-Report 08-12 (July 2009)

Solving Pseudo-Boolean Problems with SCIP*

Timo Berthold, Stefan Heinz, and Marc E. Pfetsch

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{berthold,heinz,pfetsch}@zib.de

Abstract. Pseudo-Boolean problems generalize SAT problems by allowing linear constraints and a linear objective function. Different solvers, mainly having their roots in the SAT domain, have been proposed and compared, for instance, in Pseudo-Boolean evaluations. One can also formulate Pseudo-Boolean models as integer programming models. That is, Pseudo-Boolean problems lie on the border between the SAT domain and the integer programming field.
In this paper, we approach Pseudo-Boolean problems from the integer programming side. We introduce the framework SCIP that implements constraint integer programming techniques. It integrates methods from constraint programming, integer programming, and SAT-solving: the solution of linear programming relaxations, propagation of linear as well as nonlinear constraints, and conflict analysis. We argue that this approach is suitable for Pseudo-Boolean instances containing general linear constraints, while it is less efficient for pure SAT problems.
We present extensive computational experiments on the test set used for the Pseudo-Boolean evaluation 2007. We show that our approach is very efficient for optimization instances and competitive for feasibility problems. For the nonlinear parts, we also investigate the influence of linear programming relaxations and propagation methods on the performance. It turns out that both techniques are helpful for obtaining an efficient solution method.

1 Introduction

Over the past decade, SAT-solvers have grown increasingly more efficient. Since they allow to solve large SAT instances in a consistent and fast manner, also new fields of application have been sought. One such field are Pseudo-Boolean (PB) problems, in which SAT-models are extended by linear and nonlinear constraints. Several PB-solvers have been proposed and compared during the Pseudo-Boolean evaluations, see Manquinho and Roussel [13–16].

One way to solve PB-problems is by transformation to a SAT problem, see, e.g., Eén and Sörensson [9]; this approach is used, for instance, in the solver MINISAT+ [8, 9]. Another way is to handle PB-constraints directly in the solver, see, e.g., PBS [5]. Some solvers use a constraint programming approach, for example, ABSCONPSEUDO [11].

* Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

Pseudo-Boolean problems can also be formulated as an integer program (IP), in which the nonlinear constraints are linearized. This idea is used by the solver GLPPB, which applies GLPK [10] for solving the IPs. The solver BSOLO [12] combines integer programming techniques with SAT-solving. A comparison of the SAT versus integer programming approaches is given in Aloul et al. [4].

In this paper, we approach PB-problems from a *constraint integer programming* (CIP) point of view. CIP is a combination of integer and constraint programming (CP) methods. We use the framework SCIP that is based on a branch-and-cut method as commonly used for the solution of IPs; see Achterberg [2] for details. Hence, SCIP performs a branch-and-bound algorithm to decompose the problem into subproblems, solves a linear relaxation, and, in order to strengthen the relaxation, it possibly adds additional inequalities (cutting planes). It also incorporates methods from SAT-solving like conflict analysis and restarts. Furthermore, SCIP applies techniques from CP like constraint propagation.

Besides introducing a new PB-solver, the main contribution of this paper is the evaluation of extensive computations on the instances of the Pseudo-Boolean evaluation 2007 [16]. It turns out that SCIP is a very effective solver.

The structure of the paper is as follows. In Section 1.1, we define linear and nonlinear Pseudo-Boolean problems. In Section 2, we introduce the concept of constraint integer programming. Section 2.1 gives a brief account of the different techniques incorporated into SCIP. The computations are discussed in Section 3. We provide an overall summary, a comparison to the results of the PB evaluation 2007, and more details for different problem groups. We also investigate the behavior of SCIP on the problems including nonlinear constraints more thoroughly. We summarize the outcomes in Section 4 and discuss some future challenges.

One reason for the success of our approach is that SCIP is also a very fast CIP solver. It can be used free of charge for academic purposes [21].

1.1 Problem Definition

A *linear Pseudo-Boolean problem* is an optimization problem over n binary (Boolean) variables $x_1, \dots, x_n$ in the following form:

$$\begin{aligned} \min \quad & c^{\mathrm{T}} x \\ & Ax \geq b \\ & x \in \{0,1\}^n, \end{aligned} \tag{1}$$

where $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$. The term $c^{\mathrm{T}} x$ is called the *objective function*. The inequalities in $Ax \geq b$ are called *linear constraints*.

The above format is quite general. First, expressions that involve *literals* $\ell_j \in \{x_j, \overline{x}_j\}$ can be transformed into the above form by using the relation $x_j = 1 - \overline{x}_j$, i.e., if $\ell_j = \overline{x}_j$, we replace ℓ_j by $1 - x_j$, otherwise by x_j. Maximization problems can be transformed to minimization problems by multiplying the objective function coefficients by -1. Similarly, "$\leq$" constraints can be multiplied by -1 to obtain "$\geq$" constraints. Equations can be replaced by two opposite inequalities. Inequalities or objective functions involving rational coefficients can be multiplied with the least common multiple of the denominators of all coefficients to yield integer numbers.

Integer programs are extensions of linear Pseudo-Boolean instances, in which the variables may also assume arbitrary integer values. Integer programs may be further extended by allowing some variables to take continuous values, which yields *mixed integer programs* (MIPs).

SAT problems are special cases of Pseudo-Boolean problems: A clause of a SAT formula $\ell_1 \vee \cdots \vee \ell_k$ (with literals $\ell_1, \ldots, \ell_k$) can be expressed as

$$\sum_{j=1}^{k} \ell_j \geq 1. \tag{2}$$

Then the literals are transformed as explained above. Inequalities of the type (2) are also called OR-*constraints* or *set covering constraints*. In order to state feasibility problems we can set $c = 0$.

A *nonlinear Pseudo-Boolean constraint* over literals ℓ_{ij} is defined as follows

$$\sum_{i=1}^{t} d_i \prod_{j=1}^{k} \ell_{ij} \geq r, \tag{3}$$

where $d \in \mathbb{Z}^t$, $r \in \mathbb{Z}$. Each product $z_i = \prod_{j=1}^{k} \ell_{ij} \in \{0,1\}$ can be expressed as an AND-constraint

$$z_i = \bigwedge_{j=1}^{k} \ell_{ij}.$$

The resulting new variables z_i can be inserted into the linear constraint $d^{\mathrm{T}} z \geq r$, which is equivalent to (3). AND-constraints are nonlinear in the sense that they cannot be represented by a single linear constraint. They can either be treated directly by the solver or linearized by adding the following linear constraints.

$$\begin{aligned} z_i &\leq \ell_{ij} \qquad \text{for } j = 1, \ldots, k \\ \sum_{j=1}^{k} \ell_{ij} - z_i &\leq k - 1. \end{aligned} \tag{4}$$

These constraints suffice to describe an AND-constraint in the sense that 0/1-solutions of (4) are solutions of the corresponding AND-constraint and conversely. One can show that the polyhedron defined by (4), $z_{ij} \geq 0$, and $\ell_{ij} \leq 1$ for all i and j is integral, i.e., has only integral vertices.

In our approach we treat AND-constraints in varying levels of algorithmic effort; this is described in Section 3.5.

2 Constraint Integer Programming

The majority of the solvers for SAT, MIP, and CP work in the spirit of branch-and-bound, which means that they recursively subdivide the problem instance yielding a so-called branch-and-bound-tree, whose nodes represent subproblems of the original instance. Although this strategy implicitly enumerates all potential solutions, the hope is that due to effective processing and bounding of the subproblems, one may prune other parts of the tree.

SAT and MIP are special cases of the general idea of CP. The power of CP arises from the possibility to model the problem directly with a huge variety of different, expressive constraints. In contrast, SAT and MIP only allow for very specific constraints: Boolean clauses for SAT and linear and integrality constraints for MIP. Their advantage, however, lies in the sophisticated techniques available to exploit the structure provided by these constraint types.

An important point for the efficiency of solving algorithms is the interaction between constraints. For instance, in SAT-solving, this takes place via propagation of the variables' domains. In MIP solving there exists a second, more complex but very powerful communication interface: the LP-relaxation.

The goal of constraint integer programming is to combine the advantages and compensate the weaknesses of CP, MIP, and SAT. To support this aim, we slightly restrict the notion of a CP, in order to be able to apply MIP and SAT-solving techniques, and especially provide an LP-relaxation without losing the high degree of freedom in modeling.

Definition. A *constraint integer program* CIP $= (\mathfrak{C}, I, c)$ consists of solving

$$c^\star = \min\left\{c^{\mathrm{T}}x \;:\; \mathcal{C}_i(x) = 1 \text{ for all } i = 1, \dots, m,\; x \in \mathbb{R}^n,\; x_j \in \mathbb{Z} \text{ for all } j \in I\right\},$$

with a finite set $\mathfrak{C} = \{\mathcal{C}_1, \dots, \mathcal{C}_m\}$ of constraints $\mathcal{C}_i : \mathbb{R}^n \to \{0, 1\}$, $i = 1, \dots, m$, a subset $I \subseteq N := \{1, \dots, n\}$ of the variables, and an objective function $c \in \mathbb{Z}^n$. Defining $C := N \setminus I$, a CIP has to fulfill the following additional condition:

$$\forall\, \hat{x}_I \in \mathbb{Z}^I \;\exists\, (A', b') : \{x_C \in \mathbb{R}^C \;:\; \mathfrak{C}(\hat{x}_I, x_C)\} = \{x_C \in \mathbb{R}^C \;:\; A'x_C \le b'\} \quad (5)$$

where $A' \in \mathbb{Z}^{k \times C}$ and $b' \in \mathbb{Z}^k$ for some $k \in \mathbb{Z}_{\ge 0}$.

Restriction (5) ensures that the subproblem remaining after fixing all integer variables always is a linear program. Note that this does not forbid quadratic or more involved expressions – as long as the nonlinearity only refers to the integer variables.

Clearly, MIPs are special cases of CIPs. Hence, the same holds for PB-problems. One can also show that every CP with finite domains for all variables can be modeled as a CIP.

In the following, we will describe basic ideas for solving CIPs on the example of the CIP-framework SCIP (Solving Constraint Integer Programs). We keep the description quite brief and refer to Achterberg [2] for details.

Most MIP solvers are based on a *branch-and-cut* strategy, which is a combination of branch-and-bound and cutting plane algorithms. It is also a main component of SCIP. In every node of the tree the linear relaxation, i.e., a linear program (LP), of the current subproblem is solved. Then iteratively additional cutting inequalities are added to strengthen the relaxation. More information about this method can be found in Nemhauser and Wolsey [18], Caprara and Fischetti [7], and Padberg and Rinaldi [19].

The ideas to solve linear relaxations and to add cutting inequalities have also been used in PB-solvers like BSOLO [12] and PUEBLO [22, 23].

2.1 Main Components of SCIP

The central objects of SCIP are *constraint handlers*. There are constraint handlers for linear, integrality, AND, OR, and many other constraints. A constraint handler must be able to decide whether a given solution is feasible for all constraints of the respective type or not. Furthermore, it may provide supplementary algorithms like constraint specific presolving, or domain propagation and additional information such as a linear relaxation of the constraints.

The use of *constraint propagation* is an important part of state-of-the-art CP and SAT-solvers. The task is to analyze the set of constraints of the current subproblem and the local domains of the variables in order to infer additional valid constraints and domain reductions, thereby restricting the search space. In SCIP, every constraint handler may provide its own propagation method.

A main component of modern SAT-solvers is *conflict analysis*, which was introduced by Marques-Silva and Sakallah [17]. It enables SAT-solvers to learn from infeasible subproblems. The target is to deduce short, globally valid *conflict clauses* from a series of branching decisions which led to an infeasibility. These clauses enable the solver to prune other parts of the search tree and to apply non-chronological backtracking.

This approach has been transferred to CIPs/MIPs by Achterberg [1] and Sandholm and Shields [20]. The concept of a conflict graph is extended in such a way that the nodes represent bound changes instead of variable fixings in order to cope with general integer variables. Furthermore, in the (usual) case that the infeasibility arises from the linear relaxation, SCIP analyzes the LP via a greedy heuristic that tries to identify a small subset of the bound changes that suffices to render the LP infeasible. Note that it is NP-hard to find such a subset of minimal cardinality. After having analyzed the LP, SCIP proceeds in the same fashion as SAT-solvers: it constructs a conflict graph, chooses a cut therein, and produces a conflict constraint which consists of the bound changes along the frontier of this cut.

Tightening the problem via adding additional linear inequalities, which separate the current fractional LP optimum from the set of feasible integer solutions, is a basic concept in mixed integer programming. More precisely, the LP-relaxation is strengthened by adding linear constraints $a^{\mathrm{T}}x \geq \beta$ which are violated by the current LP-solution, but not by any of the feasible solutions. SCIP features a variety of MIP based cutting plane separators.

The task of primal heuristics is to produce feasible solutions of high quality in the early steps of the branch-and-bound process. For optimization problems, the knowledge of a good feasible solution helps to guide the remaining search, prune the tree, and apply propagation techniques. For pure feasibility instances, good primal heuristics are even more important, since finding any feasible solution suffices to solve the problem.

It is a crucial decision of the branch-and-bound process how a problem Q should be split into smaller subproblems. A popular branching strategy in MIP-solving is to split the domain of a binary variable x_j with fractional LP value into two parts, thus creating two subproblems $Q_1 = Q \cap \{x_j = 0\}$ and $Q_2 = Q \cap \{x_j = 1\}$. This technique is also used in most SAT-solvers. Moreover, SCIP supports branching schemes that create more than two subproblems or branch on constraints.

In SAT-solving, usually the branch-and-bound node to be investigated next is chosen in a depth first manner. The global lower bound (the minimum of the lower bounds of each open node) of a MIP can, however, usually be raised faster if the nodes are processed in an order that also takes these bounds into consideration. SCIP uses a combination of best estimate search (select a node where good feasible solutions are expected) and depth first search by default.

Presolving or preprocessing procedures aim to transform the original problem into an equivalent problem that is easier to solve. This can be done by removing irrelevant information such as redundant constraints or fixed variables and by strengthening the LP-relaxation of a problem via tightening the bounds of variables or the coefficients of linear constraints.

Finally, restarts are a widely used component of SAT-solvers. SCIP also incorporates restarts. The idea is to abort the search process if a certain amount of global problem reductions has been triggered in the early steps and restart the search from scratch. The motivation is to use the knowledge obtained in previous runs by reapplying other presolving mechanisms to the reduced problem and procedures which are only applied at the root node. Note, however, that restarts should be applied less frequently in MIP-solvers than in SAT-solvers, since the solution of individual nodes is more time consuming.

2.2 Pseudo-Boolean Presolving Techniques

We analyzed nonlinear Pseudo-Boolean models, which led us to incorporate two new presolving ideas into SCIP. The first presolving technique aims for strengthening constraints and can be described as follows: Let $k, \ell \in \mathbb{Z}_{>0}$. Furthermore, let $\sum_{i=1}^{k} a_i < \beta$ and $d_1, \dots, d_\ell \geq \beta$, where all a_i, d_j, and β are positive integers. Then the linear constraint

$$\sum_{i=1}^{k} a_i x_i + \sum_{j=1}^{\ell} d_j y_j \geq \beta,$$

in which all x_i and y_j are binary variables, can be replaced by the (stronger) OR-constraint

$$\sum_{y=1}^{\ell} y_j \geq 1.$$

The second presolving idea eliminates variables. Let $\sum_{i=1}^{k} a_i x_i = \beta$, where $a \in \mathbb{Z}^k, \beta \in \mathbb{Z}, x \in \{0,1\}^k$. If there exists exactly one i for which a_i is odd, then $x_i = 0$ if and only if β is even. Hence, we can fix $x_i = \beta \bmod 2$. If there exist exactly two distinct i, j for which a_i, a_j are odd, then $x_i = x_j$ if and only if β is even, and $x_i = 1 - x_j$ if and only if β is odd. Hence, one of the variables can be substituted.

Both preprocessing techniques can be applied, for instance, if general integer variables are decomposed into a sum of binary variables for which all coefficients are powers of 2 (also see Section 3.2). This often occurs in PB-models.

3 Computational Results

In this section we discuss the results of computations we performed for the test set of the Pseudo-Boolean evaluation 2007. These instances are split into the following seven groups:

- Opt-Bigint-Lin: linear PB optimization instances with "big" coefficients
- Opt-Smallint-Lin: linear PB optimization instances, "small" coef.
- Opt-Smallint-Nlc: nonlinear PB optimization instances, "small" coef.
- SatUnsat-Bigint-Lin: linear PB feasibility instances, "big" coef.
- SatUnsat-Smallint-Lin: linear PB feasibility instances, "small" coef.
- SatUnsat-Smallint-Nlc: nonlinear PB feasibility instances, "small" coef.
- Pure-Sat: SAT instances transformed into PB feasibility instances

For details we refer to the web page [16].

All computations reported in the following were obtained using version 1.00.4 of SCIP on Intel Quad Core 2.6 GHz computers (in 32 bit mode) with 4 MB cache, running Linux and 2.5 GB of memory. We used CPLEX 11 as an LP-solver. As in the PB evaluation we set a time limit of 1800 seconds.

When reporting results in the tables, we use the following notation. The tables first report the names of the (sub)groups, then in columns labeled "cnt" the number of instances in each (sub)group. For optimization problems, columns labeled "opt" give the number of instances that were solved to optimality or proved to be infeasible and columns labeled "feas" give the number of instances for which our code found a feasible solution, but could not prove that this solution is optimal. For feasibility problems, columns labeled "feas" give the number of instances that are shown to be feasible, while columns "infs" give the number of instances proven to be infeasible. In general, columns labeled "unkn" give the number of instances for which we could not find any feasible solution or could not prove infeasibility. For some of the instances optimal solution values or the feasibility status are known through the PB evaluation, and columns "fail" give the number of instances for which SCIP obtained a different result; see also the remark in Section 3.3. Columns labeled "Nodes" and "Time" report the number of search nodes and CPU time, respectively; the first subcolumn reports the total number of nodes (in thousands) and time in seconds, respectively, while the second subcolumn gives the geometric mean over the numbers used for the

Table 1. Results of SCIP for *all* instances in the Pseudo-Boolean evaluation 2007, split into the seven groups.

				Nodes		Time	
Type	cnt	solved	fail	total(k)	geom.	total	geom.
OPT-BIGINT-LIN	388	158	57	53649	35.5	307956.1	90.4
OPT-SMALLINT-LIN	807	447	2	229144	236.3	682208.8	94.5
OPT-SMALLINT-NLC	405	291	0	18035	664.8	215015.6	21.3
PURE-SAT	166	15	0	28358	222.4	274130.9	1200.1
SATUNSAT-BIGINT-LIN	14	5	0	14	53.5	1.9	1.0
SATUNSAT-SMALLINT-LIN	371	326	0	15142	48.2	127749.0	15.1
SATUNSAT-SMALLINT-NLC	100	73	0	1313	75.0	50638.3	18.7
Total	2251	1315	59	345658	148.1	1657700.6	57.8

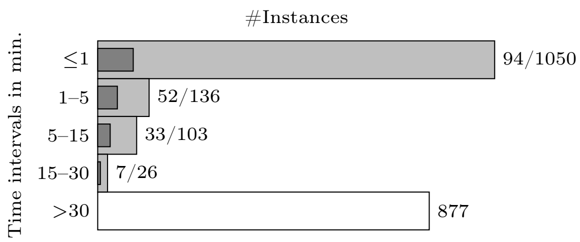

Fig. 1. Distribution over time

first subcolumn. Note that instances hitting the time limit are included with the time limit of 1800 seconds.

3.1 Overall Results and Comparison

Table 1 summarizes the results obtained by SCIP on all 2251 instances; more details about the results for the individual groups are reported in the next sections. The column labeled "solved" gives the number of instances for which optimality (or infeasibility) was proven for optimization instances, and the feasibility status was determined for feasibility instances. We used SCIP default settings for optimization instances and feasibility emphasis settings for the feasibility instances. In both cases we disabled expensive presolving methods.

Figure 1 gives a histogram of the solution times over all instances. Each bar corresponds to the number of instances that could be solved within a given time; failed instances are not reported. Light gray bar display the total number of solved instances. Among these instances dark gray bars list the number of instances that we could solve and could not be solved by any code participating in the Pseudo-Boolean evaluation 2007. The figure clearly shows that most of the instances that we could solve are solved in less than a minute. It also shows that we could solve 186 additional instances that could not be solved during the PB evaluation 2007.

In Table 2 we compare the number of instances solved by SCIP (in column labeled "SCIP") with the results of the solvers in the PB evaluation 2007; see below

Table 2. Comparison to Pseudo-Boolean evaluation 2007

Type	cnt	SCIP	MINISAT+	best solver test set	best solver instance
OPT-BIGINT-LIN	388	158	74	118	124
OPT-SMALLINT-LIN	807	447	243	270	396
OPT-SMALLINT-NLC	405	291	275	275	280
PURE-SAT	166	15	115	125	143
SATUNSAT-BIGINT-LIN	14	5	11	11	13
SATUNSAT-SMALLINT-LIN	371	326	305	341	367
SATUNSAT-SMALLINT-NLC	100	73	65	65	80
Total	2251	1315	1088	1205	1403

for a discussion of the different computing environments. In the competition, MINISAT+ turned out to be the best solver overall with respect to the number of solved instances. The number of instances solved by MINISAT+ is given in the corresponding column. We also determined the best solver in each group and report the number of solved instances by this solver in column "best solver/test set". If we choose the best solver instance-wise the numbers are given in column "best solver/instance".

It turns out that SCIP can solve about 20% more instances than MINISAT+. Moreover, even if we compare against the best solver in each group, SCIP is still better. Only if SCIP is compared against the best solver on each instance, it can solve less instances (which is mainly due to the PURE-SAT group). Since SCIP also solved 186 instances, not solved by any other PB-solver, we think that this is a very valuable contribution to the field of Pseudo-Boolean computation and shows the strength of the LP-based branch-and-cut approach.

Remark. It is important to note that Table 2 compares results obtained on different computers. The results of the PB evaluation 2007 were computed on bi-Xeon 3 GHz, 2MB cache computers and ours on Intel Quad Core 2.6 GHz, 4 MB cache, computers. We estimate (e.g. from the SPEC values [24]) that our computers are at most twice as fast. As shown in Figure 1, this difference would only effect the 26 instances for which SCIP needed between 15 and 30 minutes. Keeping this in mind, we think that the conclusions drawn above are fair.

3.2 BIGINT Instances

The Pseudo-Boolean evaluations contain instances with big coefficients ($> 2^{30}$). SCIP is not designed to handle such problems. In fact, in the PB evaluation, the solvers were classified as being able to handle these cases or not. We wanted to know, however, how well SCIP would perform on these instances as it is. Hence, we expect SCIP to produce some wrong answers. Table 3 and 4 give the results on the corresponding instances.

It turns out that for the OPT-BIGINT-LIN problems, SCIP had only 57 fails, and in 71 instances we agreed with the results of the other solvers. Note that we additionally check each found solution for feasibility and if it turns out to fail this check, the instance status is "unknown". Recall that there is no "easy" way

Table 3. SCIP results for OPT-BIGINT-LIN

	cnt	opt	feas	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
Handmade									
Course Assignment	1	1	0	0	0	0	1.0	1.1	1.1
Number Factorization	100	41	2	6	51	9	1.7	9480.5	23.6
MPS	276	108	55	107	6	53640	122.3	292979.6	159.6
Reduced MPS	3	0	1	2	0	0	1.0	5400.0	1800.0
Remaining	8	8	0	0	0	0	2.5	94.8	3.1
Total	388	158	58	115	57	53649	35.5	307956.1	90.4

Table 4. SCIP results for SATUNSAT-BIGINT-LIN

Type	cnt	feas	infs	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
Handmade									
Numerical Problems	14	5	0	9	0	14	53.5	1.9	1.0
Total	14	5	0	9	0	14	53.5	1.9	1.0

to check whether a claimed infeasible instance is really infeasible, i.e., there is in general no polynomial-time proof unless P = NP.

For SATUNSAT-BIGINT-LIN there is no “fail”, but the “unknowns” arise from solutions that fail the SCIP internal check. In total, the results for the instances with big coefficients are surprisingly good.

Instances with big coefficients are much less common in MIP, because there are hardly any efficient LP-solver that provides the needed higher accuracy for the linear relaxations. In fact, many instances with big coefficients in the PB evaluation test sets arise from the transformation of MIPs to PB-problems. For example, for some MIPLIB [6, 3] instances, continuous variables are discretized by the sum of binary variables $\sum_{i=-k}^{\ell} 2^i x_i$, which produces big coefficients depending on the values of k and ℓ. Since such variables can naturally be treated in IP-solvers, it usually makes no sense to perform a transformation. To support this conclusion, we performed the following experiment: We took the 38 original MIPLIB 2.0/3.0 instances whose transformed counterparts are contained in the PB evaluation test set (subgroup MPS). SCIP solved *all* of these instances in a total of 93.5 seconds. In contrast, SCIP took 27927.8 seconds in total to compute the PB counterparts; only 23 instances could be solved within the time limit. The main complications come from the discretization of continuous variables (all instances without continuous variables could be solved in a total of 200.5 seconds). We conclude that discretizing continuous variables is usually a bad idea. It remains to be seen, however, whether there are cases in which the transformation of general integer variables to 0/1-variables is beneficial.

3.3 SMALLINT Instances

Tables 5 and 6 report the results of SCIP for linear PB-problems with small coefficients. For the optimization problems in Table 5, our code could solve 447 of

Table 5. Results for OPT-SMALLINT-LIN

Type	cnt	opt	feas	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
Handmade									
Course Assignment	5	5	0	0	0	0	1.1	15.0	1.7
Weighted Domination Set	15	0	15	0	0	1440	88377.5	27000.0	1800.0
Graph Problems	15	0	15	0	0	430	27593.8	27000.0	1800.0
Haplotype Inference	8	0	5	3	0	8	9.9	14400.0	1800.0
Minimum-Size Prime Implicant	130	96	13	19	2	2314	819.4	68973.1	76.2
Misc	3	2	1	0	0	149	265.9	2161.3	86.6
MPS	32	23	5	4	0	9019	126.9	19316.9	33.1
Numerical Problems	34	10	4	20	0	3995	7485.3	45166.5	389.5
Synthesis PTL/CMOS Circuits	8	8	0	0	0	0	2.1	10.5	1.4
Queens Problems	15	0	0	15	0	0	1.0	27000.0	1800.0
Radar Surveillance	12	12	0	0	0	0	5.9	1022.6	10.9
Reduced MPS	273	127	65	81	0	33204	236.0	272007.6	141.3
Routing	10	10	0	0	0	0	1.1	1.4	1.0
Travelling Tournament Problem	8	2	3	3	0	21	89.8	10808.2	390.7
Industrial									
Logic Synthesis	74	70	4	0	0	327	12.3	8509.1	4.0
Random									
Kexu Benchmarks	40	0	40	0	0	1253	13676.1	72000.0	1800.0
Market Split Problem	40	9	15	16	0	176976	402890.6	58168.9	1151.6
Remaining	85	73	7	5	0	3	2.6	28647.5	22.5
Total	807	447	192	166	2	229144	236.3	682208.8	94.5

807 instances to optimality and found feasible solutions for another 192. The best solver (BSOLO) of the PB evaluation 2007 was able to solve at most 270 instances to optimality and even if we would choose the best solver for each instance, only 396 instances could be solved. This indicates that SCIP is extremely efficient on this class of instances. Table 5 shows that many subgroups of instances could be solved with relatively little effort, while in other subgroups no instance could be solved to optimality.

Remark. For the two failed instances in Table 5, SCIP claims infeasibility although they are feasible. The reasons are numerical instabilities, and cutting

Table 6. Results for SATUNSAT-SMALLINT-LIN

Type	cnt	feas	infs	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
Handmade									
Graph Problems	15	5	0	10	0	0	3.4	22767.1	1436.3
Numerical Problems	5	0	0	5	0	10	8.4	9000.0	1800.0
Pigeon Hole	20	0	20	0	0	0	1.0	0.6	1.0
Progressive Party Problem	6	2	0	4	0	1	203.5	8571.2	1301.8
Queens Problems	112	38	68	6	0	9207	6.1	11327.4	1.8
Traveling Salesperson Problem	100	40	60	0	0	5780	8718.4	32041.9	105.3
Travelling Tournament Problem	6	3	0	3	0	47	1797.4	5506.1	188.9
Industrial									
FPGA	57	36	21	0	0	0	1.7	94.1	1.5
UCLID Benchmarks	50	0	33	17	0	94	47.4	38440.6	101.1
Total	371	124	202	45	0	15142	48.2	127749.0	15.1

Table 7. SCIP results for PURE-SAT

Type	cnt	feas	infs	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
Handmade									
Pigeon Hole	20	0	2	17	0	2618	6515.1	32401.1	850.6
Remaining	146	1	12	133	0	25740	140.0	241729.8	1258.0
Total	166	1	14	150	0	28358	222.4	274130.9	1200.1

Table 8. Results for the 405 OPT-SMALLINT-NLC instances

Setting	opt	feas	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
default	291	65	49	0	18035	664.8	215015.6	21.3
no propagation	291	58	56	0	14062	1117.8	219104.5	35.7
no LP-relaxation	265	48	92	0	95880	3625.5	254690.6	27.3
linearized	283	96	26	0	65820	3018.0	231388.4	22.2

planes cut off all feasible solutions. Since SCIP and the underlying LP-solver work with finite floating point arithmetic, it seems impossible to completely avoid such outcomes. Nevertheless, in the past PB evaluations a solver was excluded from the results of a group, if it produced a fail in any of the instances of this group.

3.4 Pure Satisfiability Problems

Table 7 shows the results of SCIP on the pure SAT instances. As expected SCIP performs much worse than the other solvers, e.g., MINISAT+ (see Table 2). The reason is that the LP-relaxation provides very little information about SAT problems (for instance setting all variables to a value of $\frac{1}{2}$ gives a feasible (fractional) solution for nontrivial formulae). Furthermore, the overhead of solving an LP and handling a more complex tree structure is not compensated by their benefits.

3.5 Nonlinear Problems

For the nonlinear instances, i.e., the ones including AND-constraints, we were especially interested in different ways to handle the nonlinearities. Tables 8 and 9 show the results of SCIP on these optimization and feasibility instances, respectively. For each case we ran four variants of SCIP. The results of the default

Table 9. Results for the 100 SATUNSAT-SMALLINT-NLC instances

Setting	feas	infs	unkn	fail	Nodes total(k)	Nodes geom.	Time total	Time geom.
default	53	20	27	0	1313	75.0	50638.3	18.7
no propagation	52	20	28	0	1386	95.3	54375.5	22.2
no LP-relaxation	51	15	34	0	28222	1193.9	64728.7	25.4
linearized	51	15	34	0	15806	334.0	62258.6	26.6

setting are included in the overall comparison of Tables 1 and 2; note that this version dynamically deals with the linear relaxation (as in (4)) of the AND-constraints. In one variant ("no propagation") we turned off the propagation of AND-constraints, but kept the dynamic handling of the linear relaxation. In the next variant ("no LP-relaxation"), we kept propagation, but turned off the linear relaxation of AND-constraints. In the final version ("linearized") we replaced each AND-constraint by the full set of linear constraints (4).

It turns out that the default setting is superior to the other variants: it was the fastest and solved the most instances. Interestingly, the linearized version found much more feasible solutions in Table 8. One explanation is that primal heuristics are much more efficient if they have the whole information of the linear relaxation available. We conclude that it usually pays off to use all of the mentioned techniques to handle AND-constraints and to deal with the relaxation dynamically.

4 Conclusions and Outlook

In this paper, we presented a very efficient solver to treat Pseudo-Boolean problems. We see the following components to be crucial for the success of the approach: a fast LP solver, adding cutting inequalities, and the interaction of many different ideas incorporated in the solver. Moreover, it pays off to treat AND-constraints with a combination of LP-based and CP methods.

Pseudo-Boolean problems are at the border between the SAT and IP world. For instances bearing more similarity with SAT structures, e.g., feasibility problems with many constraints that have 0/1 coefficients only, it seems best to ignore the LP-relaxation and rather rely on fast combinatorial SAT-techniques. For instances having more similarity to general integer programs, e.g., optimization instances with many inequalities with arbitrary coefficients, it seems better to rely on IP-techniques based on a linear relaxation. This view is supported by the fact that SCIP is best on the linear PB optimization instances with small coefficients and worst on pure SAT instances, while for SAT-based solvers like MINISAT+ the outcome is reverse. Nonlinear PB-problems currently seem to take a middle position between the two worlds. It will be interesting to see whether more advanced techniques from one or the other side can help.

References

1. T. Achterberg, *Conflict analysis in mixed integer programming*, Discrete Opt., 4 (2007), pp. 4–20.
2. ———, *Constraint Integer Programming*, PhD thesis, TU Berlin, 2007.
3. T. Achterberg, T. Koch, and A. Martin, *MIPLIB 2003*, Oper. Res. Lett., 34 (2006), pp. 1–12. See `http://miplib.zib.de`.
4. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, *Generic ILP versus specialized 0-1 ILP: an update*, in Proc. IEEE/ACM International Conference on Computer-aided Design, San Jose, California, USA, L. T. Pileggi and A. Kuehlmann, eds., ACM, 2002, pp. 450–457.

5. ———, *PBS: A backtrack-search pseudo-boolean solver and optimizer*, in Proceedings of Fifth International Symposium on Theory and Applications of Satisfiability Testing (SAT 2002), Cincinnati, Ohio, 2002, pp. 346–353.
6. R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh, *An updated mixed integer programming library: MIPLIB 3.0*, Optima, 54 (1998), pp. 12–15.
7. A. Caprara and M. Fischetti, *Branch-and-cut algorithms*, in Annotated Bibliographies in Combinatorial Optimization, M. Dell'Amico, F. Maffioli, and S. Martello, eds., John Wiley & Sons, Chichester, UK, 1997, ch. 4, pp. 45–63.
8. N. Eén and N. Sörensson, *Minisat+*. http://minisat.se/MiniSat+.html.
9. ———, *Translating pseudo-boolean constraints into SAT*, J. Satisf. Boolean Model. Comput., 2 (2006), pp. 1–26.
10. GLPK, *GNU linear programming kit*. http://www.gnu.org/software/glpk/.
11. F. Hemery and C. Lecoutre, *AbsconPseudo 2006*. http://www.cril.univ-artois.fr/PB06/papers/abscon2006V2.pdf, 2006.
12. V. M. Manquinho and J. Marques-Silva, *On using cutting planes in pseudo-boolean optimization*, J. Satisf. Boolean Model. Comput., 2 (2006), pp. 209–219.
13. V. M. Manquinho and O. Roussel, *Pseudo Boolean evaluation 2005*. http://www.cril.univ-artois.fr/PB05/, 2005.
14. ———, *The first evaluation of pseudo-Boolean solvers (PB'05)*, J. Satisf. Boolean Model. Comput., 2 (2006), pp. 103–143.
15. ———, *Pseudo Boolean evaluation 2006*. http://www.cril.univ-artois.fr/PB06/, 2006.
16. ———, *Pseudo Boolean evaluation 2007*. http://www.cril.univ-artois.fr/PB07/, 2007.
17. J. P. Marques-Silva and K. A. Sakallah, *GRASP: A search algorithm for propositional satisfiability*, IEEE Trans. Comput., 48 (1999), pp. 506–521.
18. G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, New York, 1988.
19. M. Padberg and G. Rinaldi, *A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems*, SIAM Rev., 33 (1991), pp. 60–100.
20. T. Sandholm and R. Shields, *Nogood learning for mixed integer programming*, Tech. Rep. CMU-CS-06-155, Carnegie Mellon University, Computer Science Department, 2006.
21. SCIP, *Solving constraint integer programs*. http://scip.zib.de, 2007. Zuse Institute Berlin.
22. H. M. Sheini and K. A. Sakallah, *Pueblo: A modern pseudo-boolean SAT solver*, in Proc. Conference on Design, Automation and Test in Europe (DATE '05), IEEE Computer Society, 2005, pp. 684–685.
23. ———, *Pueblo: A hybrid pseudo-boolean SAT solver*, J. Satisf. Boolean Model. Comput., 2 (2006), pp. 165–189.
24. SPEC – Standard Performance Evaluation Corporation, *CPU2006 results*. http://www.spec.org, 2008.