Constraints (2021) 26:26–55
https://doi.org/10.1007/s10601-020-09318-x

Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search

Jo Devriendt[1,2,3] · **Ambros Gleixner**[4,5] · **Jakob Nordström**[1,2]

Accepted: 12 November 2020 / Published online: 18 January 2021

Abstract

Conflict-driven pseudo-Boolean solvers optimize 0-1 integer linear programs by extending the conflict-driven clause learning (CDCL) paradigm from SAT solving. Though pseudo-Boolean solvers have the potential to be exponentially more efficient than CDCL solvers in theory, in practice they can sometimes get hopelessly stuck even when the linear programming (LP) relaxation is infeasible over the reals. Inspired by mixed integer programming (MIP), we address this problem by interleaving incremental LP solving with cut generation within the conflict-driven pseudo-Boolean search. This hybrid approach, which for the first time combines MIP techniques with full-blown conflict analysis operating directly on linear inequalities using the cutting planes method, significantly improves performance on a wide range of benchmarks, approaching a "best-of-both-worlds" scenario between SAT-style conflict-driven search and MIP-style branch-and-cut.

Keywords Pseudo-Boolean · Conflict-driven search · Cutting plane proofs · Integer Programming · Linear Programming · Cut generation · RoundingSat

This article belongs to the Topical Collection: *Special Issue on Constraint Programming, Artificial Intelligence, and Operations Research*
Guest Editors: Emmanuel Hebrard and Nysret Musliu

✉ Jo Devriendt
jo.devriendt@cs.lth.se

Ambros Gleixner
gleixner@zib.de

Jakob Nordström
jn@di.ku.dk

[1] University of Copenhagen, Copenhagen, Denmark

[2] Lund University, Lund, Sweden

[3] KU Leuven, Leuven, Belgium

[4] HTW Berlin, Berlin, Germany

[5] Zuse Institute Berlin Berlin, Germany

1 Introduction

The discovery of conflict-driven search in Boolean satisfiability (SAT) solving is one of the most impressive success stories in computer science over the last couple of decades. Although the SAT problem is NP-complete [18, 44], and is even widely believed to require exponential time in the worst case [16, 41], these days so-called *conflict-driven clause learning (CDCL)* solvers [47, 51][1] are routinely used to solve problems with hundreds of thousands or even millions of variables.

In view of this impressive performance, it is natural to try to export the conflict-driven paradigm to neighbouring areas such as *pseudo-Boolean (PB) solving*, where solvers deal with pseudo-Boolean decision or optimization problems. In this paper, we restrict our attention to *linear* pseudo-Boolean constraints, which are a straightforward generalization of the conjunctive normal form (CNF) used for SAT solvers; such pseudo-Boolean formulas are also known as *0-1 integer linear programs (ILPs)*. Some PB solvers essentially encode the input back to CNF and run CDCL [29, 35, 48, 59], but another approach, which is our focus in this work, is to extend the solvers from CNF to reason natively with linear constraints [17, 26, 31, 43, 62]. Such *conflict-driven PB solvers* have the potential to run exponentially faster than CDCL solvers, since the *cutting planes* method [19][2] they use is exponentially stronger than the *resolution* method [13, 22, 23, 56] underlying CDCL (in particular, there are formulas for which resolution needs an exponential number of reasoning steps while cutting planes only needs a linear number of steps [19, 37]).

It turns out that pseudo-Boolean solvers struggle to realize their full potential, however. This is perhaps most vividly illustrated by a particularly easy class of PB formulas, namely the *rationally infeasible* ones that have no solutions even with fractional assignments. Such formulas do not require Boolean reasoning, only *linear programming (LP)*, and the proof of infeasibility guaranteed to exist by Farkas' lemma [32] is always easy to derive with the cutting planes method. In spite of this, it is not hard to find small examples of rationally infeasible formulas that are completely beyond the reach of PB solvers in practice [30]. For some PB solvers, this is explained by their use of resolution, but solvers based on cutting planes have no such excuse [65].

This unsatisfactory state of affairs leads us to consider another modern breakthrough technology, namely *mixed integer programming (MIP)* [12]. MIP solvers, which, as a special case, handle 0-1 ILPs, repeatedly apply an LP solver to linear relaxations of the input problem. This means that they will detect right away if the input is rationally infeasible, and so such PB formulas do not pose much of a challenge. A downside of MIP solvers, though, is that they lack the sophisticated conflict analysis found in cutting-planes-based PB solvers. (Technically speaking, this is because MIP conflict analysis operates not on the linear constraints themselves but on disjunctive clauses extracted from such constraints, just as in CDCL).

Given the discussion so far, an attractive proposal that presents itself is to simply merge the two approaches. Is it possible to combine pseudo-Boolean solving and mixed integer linear programming to get the best of both worlds? This is the question we set out to address.

[1] A similar technique for constraint satisfaction problems was independently developed in [8].

[2] We emphasize that here the term *cutting planes* refers to a method for syntactically deriving new linear inequalities from a 0-1 integer linear program until it is shown to have no integer-valued solutions, and as such can be meaningfully applied even to rationally infeasible problems. In this sense, the terminology is both more general than and slightly different from what would commonly be understood by *cutting plane separation* in the context of MIP solving.

1.1 Our contribution

In this work, we investigate how to integrate linear programming within pseudo-Boolean conflict-driven solving, and present a thorough evaluation of a combined solver. Before summarizing our contributions, let us highlight some of the challenges that make this a nontrivial proposition (referring the reader to Section 2 for a more in-depth discussion of the technical terms involved).

We first note that simply running an LP solver as a preprocessing step before starting the PB search is not sufficient. Pseudo-Boolean formulas might be rationally feasible when viewed as linear programs, but quickly turn infeasible once just a couple of variable assignments have been made. In [30], some crafted benchmarks with this property are presented for which pseudo-Boolean solvers utterly fail to detect unsatisfiability and undo these assignments, although the residual problem has turned rationally infeasible. In such a scenario an LP preprocessing step will not help. Going to the other extreme, a naive full integration of PB and MIP solving technology would mean solving a linear program each time before the PB solver decides on a variable assignment during search. This is completely impractical, however, because LP solving operates on a time scale that is several orders of magnitude slower than the rapid alternation of variable decisions and propagations that lies at the core of PB search.

The next problem is what to do if an LP call deep inside the search shows that the formula is rationally infeasible under the current assignment explored by the solver. In such a case, the pseudo-Boolean solver should obviously backtrack, but in order to do so it needs to exhibit a violated linear constraint. Since the PB solver itself has not detected contradiction, no such constraint exists in the input formula or among previously learned constraints. A second, more subtle, issue is that efficient LP solvers use inexact floating-point arithmetic—how can this be incorporated into a Boolean solver that must maintain perfectly sound reasoning?

Going further, other interesting questions arise. When the LP relaxation is in fact feasible given the current assignment, and the LP solver finds a rational solution, MIP solvers generate *cuts*, i.e., implied linear constraints that invalidate this solution. Can it be helpful to enhance the pseudo-Boolean conflict-driven learning by adding also such constraints? In the other direction, the PB solver is constantly learning new constraints at a very high rate, and these constraints are also cuts in the sense that they eliminate rational points from the polytope described by the input formula. Should these learned constraints be communicated to the LP solver also, or is it better to run the LP solver on the original formula only?

Our approach is to interleave incremental LP solving within the pseudo-Boolean conflict-driven search. Our integration is fully dynamic, with the PB and LP solvers communicating continuously during execution. In order to balance resources and avoid that the LP solver starves the PB solver, LP calls are made with a strict time budget and are terminated as soon as this budget is exceeded.

If the LP solver detects rational infeasibility, we can use Farkas' lemma to obtain a linear combination of constraints that witnesses infeasibility under the current assignment. We will refer to this new constraint as a *Farkas constraint*. Since this Farkas constraint is violated, it can serve as the starting point of pseudo-Boolean conflict analysis. In this way we can also avoid any issues with rounding errors—although the LP solver uses floating-point arithmetic, the Farkas constraint is computed using exact arithmetic (and if it is not falsified by the current assignment, then the PB search will not backtrack but continue).

When the LP solver instead finds a rational solution, it generates a MIP-style *Gomory cut* that prunes away the rational solution and tightens the search space both on the PB and the LP side. The pseudo-Boolean solver can also use information from the rational solution to direct the search, e.g., by determining how to assign variables, and we have also explored

passing constraints learned during conflict analysis from the PB solver to the LP solver. To the best of our knowledge, this is the first time techniques from MIP solving such as LP relaxations and cut generation have been combined with pseudo-Boolean conflict analysis using the cutting planes method, which derives new learned constraints by operating directly on the linear constraints involved in the conflict rather than on disjunctive clauses obtained from these linear constraints.

We report on extensive experiments with a combined solver based on a pair of state-of-the-art PB and LP solvers. We find that just using an LP solver to detect rational infeasibility during search already significantly improves performance on a broad range of benchmarks. Letting rational solutions provided by the LP solver guide the search further boosts effectiveness. The effect of generating Gomory cuts is more uneven, but is beneficial for certain types of benchmarks. Passing constraints learned by the PB solver to the LP solver does not seem to pay off, though, but neither does it hurt performance much.

In experiments on benchmarks from the most recent pseudo-Boolean competition [55] our hybrid approach either outperforms other PB solvers or is close to the top solver. In addition, for some of the crafted benchmarks the hybrid solver is clearly better than the MIP solvers we tested. However, the MIP solvers are much better in general on 0-1 ILPs from MIPLIB [49], but here we can see that our PB solver with integrated LP solving outperforms pure PB solvers. Though we believe that there is ample room for further improvements, this already shows the potential benefits of combining the best from the two worlds of SAT-style conflict-driven learning and MIP-style branch-and-cut.

1.2 Related work

In terms of related work in pseudo-Boolean solving, Hooker [38, 39] observed that the resolution rule applied to disjunctive clauses can be viewed as an integer programming cut rule, and can be appropriately extended to general pseudo-Boolean constraints. In this way, techniques from operations research and logic programming can be integrated in a unified approach; see [5] for a monographic treatment, and [6, 7, 14] for some further developments. These methods have not shown to be efficient enough in general, but related ideas were used in [17] to provide the foundations for how conflict-driven pseudo-Boolean solvers work today. Another approach using *branch-and-bound* was proposed in *bsolo* [45]. Here the objective function is bounded by approximating a maximum independent set of constraints rather than by considering an LP relaxation. Also, *bsolo* uses the resolution method and learns only clauses.

In constraint programming (CP), many solvers have had LP solving integrated in their search routines, two early examples being *Eplex* [63] and *B-Prolog* [67]. A more recent addition is *LCG-glucose* [27], which uses a network flow propagator that internally solves a specialized LP and returns a Farkas constraint when this LP is infeasible. The Farkas constraint is used to backtrack and it is kept as a constraint in the form of a new linear propagator. *LCG-glucose* differs in three crucial aspects from our approach. Firstly, to balance the PB and LP solving routines, we do not run the LP solver until completion but terminate it early if needed. Secondly, we do not treat the learned Farkas constraint as a propagator providing clausal reasons for CDCL-style reasoning, but instead as a fully fledged linear constraint that takes part as-is in the much stronger conflict analysis based on the cutting planes method. Thirdly, we apply the LP solver to the relaxation of the complete input formula instead of only network flow constraints.

In the context of satisfiability modulo theories (SMT), solvers employ linear programming to support a theory of linear arithmetic [28], but two clear differences from our work are again that in SMT solvers the LP solver is run until completion and that the conflict analysis does not make use of the power of derivations over linear constraints.

In mixed integer programming a closely related solver is *SCIP* [2], which combines constraint propagation and SAT-style clause learning with classical MIP solving. (Similar approaches have also been adopted in closed-source solvers—see, e.g., [3].) Since it is important to understand the difference between *SCIP* and our approach, let us give a somewhat simplified high-level description of pseudo-Boolean search and conflict analysis phrased in MIP language.

During the search phase, the pseudo-Boolean solver chooses some unassigned variable and makes a *decision* to assign this variable 0 or 1, after which all further assignments are added that are immediately implied by some linear constraints until no more such *propagations* are possible. The solver then makes a new decision and propagates, and this cycle of decisions and propagations repeats until either a satisfying assignment is found or some linear inequality C is violated, in which case we say that C is *conflicting*. The latter scenario triggers *conflict analysis*, which works as follows:

1. The linear inequality R responsible for propagating the last variable x in C to the "wrong value" from the point of view of C is identified; this inequality R is referred to as the *reason constraint* for x.
2. One of the rules called *division* and *saturation*, respectively, is applied to R to generate a cut R_{cut} that propagates x to a $\{0, 1\}$-value when considered not only over Boolean values but even over the reals.
3. A new linear constraint D is computed as the smallest integer linear combination of R_{cut} and C for which the variable x cancels and is eliminated. It is not too hard to show that it follows from the description above that this constraint D is violated by the current partial assignment of the solver with the value of x removed.
4. Since the new constraint D is conflicting, we can set $C := D$ and go to step 1 again.

The conflict analysis continues until a termination criterion (which we do not describe in detail here, but which is the analogue of of the so-called *1-UIP* notion in SAT solving) declares the constraint D derived in step 3 to be the learned constraint. At this point, the solver undoes further assignments in reverse chronological order until D is no longer violated, and then switches back to the search phase. We refer the reader to [15] for a more detailed description of conflict-driven pseudo-Boolean solving.

In comparison, *SCIP* uses not only the simple but fast propagation in PB solvers, but in addition also the more powerful but more expensive and slower method of solving LP relaxations to guide the search. When an infeasible node is reached, the conflict graph analysis used in *SCIP* [1] does not operate on the reason constraints R as described above, but instead on disjunctive clauses extracted from these constraints. It is not hard to prove formally (appealing to [9, 19, 37]) that this incurs an exponential loss in reasoning power compared to performing derivations on the linear constraints themselves using the cutting planes method. Another important difference between *SCIP* and our hybrid solver is that we do not suffer from any issues with inexact floating-point arithmetic, since the actual reasoning steps use exact integer arithmetic. Thus, our approach is guaranteed to produce sound results, and could conceivably be used for, e.g., verification or theorem proving applications.

1.3 Paper outline

After an overview of preliminaries in Section 2 we describe our pseudo-Boolean solver with linear programming integration in Section 3. We present an experimental evaluation in Section 4 and conclude in Section 5.

2 Preliminaries

Throughout this paper, we use the term *pseudo-Boolean (PB) constraint* to refer to a $0-1$ linear inequality. We identify 1 with *true* and 0 with *false*. A *literal* ℓ_i denotes either a Boolean variable x_i or its negation $\overline{x}_i$, where $\overline{x}_i = 1 - x_i$. We assume without loss of generality that all PB constraints $\sum_i c_i \ell_i \geq w$ are written in *normalized form*, where literals ℓ_i are over pairwise distinct variables, coefficients c_i are non-negative integers, and w is a positive integer called the *degree of falsity* (or just *degree* for short). Clearly, any 0-1 integer linear constraint with positive or negative rational coefficients and degree of falsity can be rewritten as an equivalent constraint in normalized form.

A *(partial) Boolean assignment* ρ is a mapping of (a subset of) variables to $\{0, 1\}$, where we also identify ρ with the subset of literals it sets to true and write $\rho(x) = *$ if ρ does not assign x. The *slack*$(C\rho)$ of a constraint $C \doteq \sum_i c_i \ell_i \geq w$ under a partial assignment ρ is

$$slack(C, \rho) = -w + \sum_{i:\rho(\ell_i)\neq 0} c_i \ , \tag{1}$$

i.e., the maximal value the left-hand side of the constraint $\sum_i c_i \ell_i$ can attain under any partial assignment $\rho' \supseteq \rho$ extending ρ minus the degree of falsity w. We say that ρ *falsifies* C if $slack(C, \rho) < 0$ and *satisfies* C if for all partial assignments $\rho' \supseteq \rho$ extending ρ it holds that $slack(C, \rho') \geq 0$. If $slack(C, \rho) \geq 0$ but some literal ℓ_i in C has coefficient $c_i > slack(C, \rho)$, then C *propagates* ℓ_i to true under ρ (since setting ℓ_i false would falsify the constraint).

By a *pseudo-Boolean formula* φ we mean a set of PB constraints, or, in other words, a (decision version of a) *0-1 integer linear program (ILP)*. An assignment ρ is a *solution* to a formula φ if ρ satisfies all constraints in φ. A formula is *satisfiable*, or *feasible*, if it has a solution.

Pseudo-Boolean formulas and 0-1 integer linear programs are often used to encode optimization problems by also specifying an *objective function* f, which in our setting is a linear combination of literals $\sum_i c_i \ell_i$ that is also assumed to be in normalized form. A total assignment ρ is an *optimal (minimal) solution* for the formula φ with objective function f if it is a solution of φ and for all solutions ρ' of φ it holds that $\sum_i c_i \rho'(\ell_i) \geq \sum_i c_i \rho(\ell_i)$.

A *(partial) rational assignment* is a mapping of (a subset of) variables to the rational interval $[0, 1]$. All notions discussed above generalize to rational assignments in the straightforward way, and we will sometimes use them in this extended sense, but when not stated otherwise we are assuming the Boolean setting.

2.1 Conflict-driven pseudo-Boolean solving

We now present the bare essentials of conflict-driven pseudo-Boolean solving necessary for the discussions in this paper, again referring the reader to [15] for more details.

The current state of a pseudo-Boolean solver can be abstractly represented by a tuple (ψ, ρ), where ψ is a set of constraints called the *constraint database*, and ρ is the *current assignment* (often referred to as the *trail*, especially when the variable assignments are viewed as a list

sorted in chronological order). Initially, ψ is the input formula φ and ρ is the empty assignment. Given a solver state, the search loop starts with a *propagation* phase, which extends ρ exhaustively with all literals ℓ propagated by some constraint $C \in \psi$ until there are no further propagations. If C propagates ℓ, the solver adds ℓ to the trail ρ and stores the constraint C as the *reason* for ℓ. The solver reaches a *conflict* if the current assignment falsifies some constraint in the database. This triggers a *conflict analysis* phase where a *learned constraint* is derived (as briefly described in Section 1.2) that is a logical consequence of the current set of reason constraints and that will prevent the same conflict from happening in the future. This learned constraint is added to ψ, after which the solver backtracks as far as possible to a state where the learned constraint is propagating, and starts a new iteration of the search loop. Alternatively, if no conflict is detected, the solver extends ρ by making a *decision* to assign some currently unassigned variable. The dominant heuristic for which value to choose is to re-use the value to which the variable was last propagated, which is called *phase saving*. After a decision, a new iteration kicks off with a new propagation phase.

Every once in a while, the solver *restarts* by backtracking to a state where the solver has made no decisions and the assignment only consists of propagations that follow without any decisions. Less frequently, the solver *cleans* the constraint database by erasing learned constraints that are deemed to be less useful. In the solver we used for our experiments, the constraint database cleaning was coordinated with restarts. [3] The solver reports unsatisfiability whenever it learns a contradictory constraint $0 \geq w$ for some $w > 0$. If propagation does not lead to a conflict and all variables have been assigned, the solver reports that the input formula is satisfiable. These standard components of pseudo-Boolean conflict-driven solving are illustrated in the non-highlighted pseudo-code in Algorithm 1.

To optimize an objective function $f : \sum_i c_i \ell_i$, the solver can use a *linear bounding* strategy. Every time it finds a solution ρ, it adds the (normalized form of the) constraint $\sum_i c_i \ell_i < f(\rho)$. This *objective bound* forces the PB solver to find a better solution when the search loop is resumed. When the solver finally reports unsatisfiability, this is a proof that the solution last found is optimal. The non-highlighted part of Algorithm 2 illustrates how to iteratively call the conflict-driven pseudo-Boolean search routine to optimize an objective function.

2.2 Linear programming

LP solvers find optimal rational solutions to systems of linear inequalities with linear objective functions. A well-known LP algorithm is the *simplex* method [21], which *pivots* between rational solutions at the extremal points of the polytope defined by the inequalities. We use a simplex-based LP solver for the integration of linear programming and pseudo-Boolean solving in our hybrid solver.

The *primal* simplex method pivots along feasible solutions while monotonically improving the objective function. To find an initial feasible solution, the simplex algorithm can be applied to a *dual* problem, which is derived from the input problem. Modern LP solvers combine these techniques as a *primal and dual revised simplex* approach, deciding when to switch between the primal and dual problem on-the-fly. Additionally, the linear program can be solved under a partial assignment ρ, in which case the solution returned is a rational extension of ρ.

If the input program φ is rationally infeasible under an assignment ρ, an LP solver can be asked to return a list of *Farkas multipliers* $\lambda_k \geq 0$ such that taking the linear combination

[3] As a technical comment, this makes the code for constraint erasure slightly simpler to write, since there is no need to deal with constraints that cannot be removed because they are reasons for currently propagated literals, but in state-of-the-art SAT solvers restarts and database cleaning tend to be decoupled.

of the constraints $C_k \doteq \sum_i c_{k,i}\ell_{k,i} \geq w_k$ in φ with these multipliers yields a linear inequality $\sum_k \lambda_k C_k \doteq \sum_k \sum_i \lambda_k c_{k,i}\ell_{k,i} \geq \sum_k \lambda_k w_k$ that has negative slack under ρ, and thus certifies that φ does not have any rational solutions extending ρ. We refer to such an inequality as a *Farkas constraint*.

For further details on theory and algorithms of linear programming we refer the reader to [64].

3 Integrating linear programming in PB solving

In this section, we describe our hybrid solver combining pseudo-Boolean search and linear programming. Let us start by discussing the core of our approach: how to incorporate an LP solver in conflict-driven PB search to detect rationally infeasible subproblems.

The main change to the conflict-driven search loop described in Section 2.1 is that after a propagation phase that did not lead to a conflict, and before making the next decision, the pseudo-Boolean solver can make a call to the LP solver to check whether the linear relaxation of the input formula is rationally feasible under the current assignment. If the LP solver reports that the problem is feasible, the PB solver continues as usual by making a decision followed by a new round of propagations. If the LP relaxation is infeasible, however, the LP solver returns a set of Farkas multipliers from which a constraint C_{Farkas} is constructed as described in Section 2.2. This constraint is added to ψ, and since it is violated by the current assignment it can be used as the starting point of conflict analysis.

We abstract the search state of our hybrid solver as a triple $(\psi, \rho, \widehat{\psi})$, where ψ is the constraint database, ρ is the current assignment, and $\widehat{\psi}$ is the linear relaxation, which is the set of constraints passed to the LP solver. Initially, ψ and $\widehat{\psi}$ both consist of the input formula φ while ρ is the empty assignment. During the search, constraints will be added and removed from both ψ and $\widehat{\psi}$, but both sets of constraints will always contain the full input formula. The basic pseudo-Boolean search loop enhanced with linear programming is presented in Algorithm 1.

For a pseudo-Boolean optimization problem, the objective function f is also passed to the LP solver so that the linear relaxation under the current assignment is solved optimally with respect to f. When a (Boolean) solution to the PB problem has been found, we add a constraint to the linear relaxation enforcing that any new (rational) solution has to be better than this. See Algorithm 2 for a pseudo-code description.

The approach outlined in Algorithms 1 and 2 captures the main ideas of our work, but a naive implementation of these algorithms is not sufficient to achieve a well-performing hybrid solver. In the rest of this section we describe additional features that we have explored to improve the solver, and in Section 4 we report on how different settings affect performance. We want to emphasize right away that it is not possible to provide an exhaustive evaluation—if nothing else because the parameter space is infinite—but we believe it is fair to argue that we present a thorough investigation, and throughout Sections 3 and 4 we discuss additional ideas for further solver features and methods of analysis that we were not able to pursue within the framework of this project.

3.1 Ensuring the soundness of Farkas constraints

Most LP solvers use fast but imprecise floating-point arithmetic and return Farkas multipliers λ_k as floats. If we use floating-point multipliers to calculate the Farkas constraint, then

rounding errors can produce a constraint that is not implied by the original formula φ and is hence not not sound to add to the constraint database.

Algorithm 1 Simplified version of conflict-driven PB search with linear programming integration (LP-specific code highlighted in grey).

Data: pseudo-Boolean formula φ; constraint database ψ; linear relaxation $\widehat{\psi}$
Result: solution ρ to φ or empty assignment $\emptyset$ if φ is unsatisfiable
initialize ρ to the empty assignment $\emptyset$
initialize last propagated value to 0 for all variables
while *true* **do**
  while *some constraint* $C \in \psi$ *has an unassigned literal* ℓ_i *with coefficient* $c_i > \mathit{slack}(C, \rho)$ **do**
    $\rho \leftarrow \rho \cup \{\ell_i\}$
    record C as the reason for ℓ_i
    update last propagated value for variable corresponding to ℓ_i
  if *some* $C \in \psi$ *has* $\mathit{slack}(C, \rho) < 0$ **then**
    perform conflict analysis on C and the current reason constraints to derive a new constraint C_{learned}
    if C_{learned} *implies* $0 \geq 1$ **then return** $\emptyset$
    else
      add C_{learned} to ψ
      remove assigments from ρ in reverse order until $\mathit{slack}(C_{\text{learned}}, \rho) \geq 0$
  else if $\widehat{\psi}$ *is rationally infeasible under* ρ **then**
    extract Farkas multipliers to construct C_{Farkas}
    add C_{Farkas} to ψ
  else if ρ *assigns all variables* **then return** ρ
  else if *time to restart* **then** $\rho \leftarrow \emptyset$
  else if *time to clean* ψ **then** remove some learned constraints from $\psi \setminus \varphi$
  else decide to assign some unassigned variable to its last propagated value

Algorithm 2 Simplified version of pseudo-Boolean optimization with linear programming integration (LP-specific code highlighted in grey).

Data: pseudo-Boolean formula φ; objective function $f: \sum_i c_i \ell_i$
Result: solution ρ_{best} to φ minimizing f or empty assignment $\emptyset$ if φ is unsatisfiable
initialize ψ to φ
initialize $\widehat{\psi}$ to φ
initialize the LP solver with $\widehat{\psi}$ and f
call Algorithm 1 with $(\varphi, \psi, \widehat{\psi})$ and let ρ be the assignment returned
$\rho_{\text{best}} \leftarrow \rho$
while $\rho \neq \emptyset$ **do**
  $\rho_{\text{best}} \leftarrow \rho$
  add upper bound constraint $\sum_i c_i \ell_i \leq f(\rho_{\text{best}}) - 1$ to ψ and $\widehat{\psi}$
  call Algorithm 1 with $(\varphi, \psi, \widehat{\psi})$ and let ρ be the assignment returned
return ρ_{best}

It is possible to use specialized floating-point arithmetic to compute numerically safe linear combinations of constraints [20], but the pseudo-Boolean setting anyway requires constraints to have integral coefficients and degree. We therefore scale up and round the

multipliers λ_k to integers, and then use exact integer arithmetic to calculate a constraint that is guaranteed to be implied by φ. If the computed constraint C_{Farkas} is falsified by the current assignment even after rounding, then it will trigger pseudo-Boolean conflict analysis. Otherwise, PB search resumes as if no rational infeasibility was detected, and we reset the basis of the LP solver to avoid detecting rational infeasibility with the same Farkas multipliers, which would lead to the same problematic Farkas constraint again.

3.2 Limiting the frequency and duration of LP calls

Calling an LP solver to check for rational infeasibility can be on the order of a thousand times slower than performing a whole cycle of decisions and propagations up to conflict in a pseudo-Boolean solver. Moreover, during preliminary experiments we observed that sometimes this problem is even worse—there were PB formulas for which running a single LP call until completion took more than 5000 seconds, which was the overall time-out limit we had for our experiments. Therefore, in order to avoid starving the PB solver we limit both the total number of LP calls and the time for each call, where as a rough proxy for time we measure the number of simplex pivots performed by the LP solver (which also allows us to obtain reproducible results).

To limit the number of LP calls, the pseudo-Boolean solver only invokes the LP solver in the following situations during the search:

1. At the start of the search, to quickly spot a rationally infeasible input formula.
2. During restart before a constraint database cleanup, which allows to find a solution to the linear relaxation unconstrained by solver decisions.
3. Right before the first decision after backtracking, where the intuition is that since the solver recently reached an inconsistent search state, it might still be close to rational infeasibility.

Clearly, the situation 1 applies only once, and situations of type 2 are infrequent, arising roughly every ten thousand conflicts. Situations of type 3 occur quite frequently, however.

As a further limitation, before an LP call as above is made we compare the total number of pivots p performed by the LP solver in all calls so far to the total number of conflicts c encountered by the pseudo-Boolean solver. We fix a *pivot ratio* r, and the next LP call is only executed when $p/c \leq r$. We keep track of separate pivot counts p_r for *root LP calls* of types 1 and 2 on the one hand and p_d for *deep LP calls* of type 3 on the other hand, to decouple their impact: expensive root LP calls should not prevent deep LP calls from happening, and vice versa.

To limit the time for individual LP calls—in order to avoid the case when a single call causes a time-out—we also restrict the number of pivots made during a call by imposing a *pivot budget* b. If the LP solver reaches the pivot budget, it is terminated and the PB solver takes over again. This means that the effort spent on LP solving was wasted, however, and we want to avoid squandering time in this way in case our pivot budget was too stingy. When the LP solver is terminated during a call, we therefore double the pivot budget, so that a future LP call will run less risk of timing out without producing useful results.

Summarizing, LP calls are only made when $p/c \leq r$ in which case the pivot budget is determined as $b = b_{\text{init}} \cdot 2^t$, where b_{init} is some initial pivot budget and t is the number of forcibly terminated LP calls so far. Note that b never decreases, so eventually all LP calls will terminate. However, as b gets larger the LP calls will potentially become less frequent, as each single call can increase the pivot count significantly. In this way, the total time spent on LP solving is still limited.

We believe that this careful balancing of resources allocated to pseudo-Boolean search and LP solving is crucial to the success of our approach, and that it is a key explanation to how we can afford to run an LP solver on the whole input formula rather than only on specialized constraints.

3.3 Using LP solutions to guide variable assignments

An important component in conflict-driven solvers is the heuristic for variable decisions, i.e., which variable to assign next and what value (0 or 1) it should be assigned to. The dominant value heuristic in CDCL is *phase saving* [53], which assigns a variable the value to which it was last propagated, and this heuristic has also been inherited by PB solvers. In a hybrid PB-LP solver there is another potential source of information, however. If the LP solver has found a rational solution $\widehat{\rho}$ to the linear relaxation of the pseudo-Boolean formula, it is natural to ask whether it would be better to assign a variable x the value closest to $\widehat{\rho}(x)$ rather than its last propagated value.

When root LP calls made at the start of the search or before constraint database cleaning (calls of types 1 and 2 in Section 3.2) run until completion and produce a rational solution, we round the rational values and store these values as the current phase instead of the last propagated values. Phase saving is still applied as before, though, so later propagations will override these rounded rational assignments. Our hope is that in this way the adjusted phase saving heuristic will gently nudge the solver to search the neighbourhood of the optimal rational solution $\widehat{\rho}$.[4] Additionally, in case the rounded rational solution is an actual 0-1 solution, the PB search routine will immediately find it. We remark that similar solution-based phase-saving ideas have appeared in, e.g., [24].

3.4 Generating Gomory mixed integer cuts

A crucial technique in mixed integer linear programming is *branch-and-cut* (see, e.g., [50]), where one takes linear combinations of constraints and applies division with rounding to derive a new constraint—a *cut*—that removes rational solutions from the linear relaxation. We choose here to focus on *mixed integer rounding (MIR) cuts* [46], which in our setting can be applied to a (normalized) PB constraint $\sum_i c_i \ell_i \geq w$ with a divisor $d \in \mathbb{N}^+$ to yield

$$\sum_i \left(\min(rem(c_i, d), rem(w, d)) + \left\lfloor \frac{c_i}{d} \right\rfloor rem(w, d) \right) \ell_i \geq \left\lceil \frac{w}{d} \right\rceil rem(w, d) \tag{2}$$

(where $rem(c, d)$ denotes the remainder after integer division of c by d, and $\lceil a/b \rceil$ and $\lfloor a/b \rfloor$ denotes division rounded up and down, respectively). Just as a quick example—since the expression in (2) can feel a bit intimidating—applying the MIR inequality with divisor 3 to the PB constraint

$$x + 2y + 3z + 4w + 5u \geq 5 \tag{3}$$

yields

$$x + 2y + 2z + 3w + 4u \geq 4 \ . \tag{4}$$

When the simplex algorithm finds a rational solution $\widehat{\rho}$, it is possible to construct divisors d and sets of multipliers λ_k that define linear combinations of constraints $\sum_k \lambda_k C_k$, such that applying (2) yields a constraint falsified by $\widehat{\rho}$. This type of MIR cut is called a

[4] We remark that a less gentle nudge would be to fix the phase to the rounded rational values and switch off later updates of the phase during PB search, but we did not investigate this option.

Gomory mixed integer cut [46], but for simplicity we refer to such cuts simply as Gomory cuts in this paper. Given the important role such cuts play in MIP solving, it seems promising to transfer cut generation to conflict-driven PB solvers, especially in a hybrid solver that already makes use of linear programming.

3.5 Using learned constraints as cuts in the LP relaxation

Although we have not gone into too much details regarding how new constraints are derived during pseudo-Boolean conflict analysis, it can be shown that such constraints are also cuts in that they eliminate rational points in the polytope described by the linear relaxation of the input formula. (We hint at this in our brief sketch of PB conflict analysis in Section 1.2.)

The way we have described the linear programming integration above, the LP solver is only run on the original pseudo-Boolean formula. In a scenario when the LP solver finds a solution to the linear relaxation that turns out to falsify some constraint C learned by the PB solver, though, one could contemplate adding C as a cut to the LP relaxation to potentially improve the quality of future solutions produced by the LP solver.

3.6 Cut selection and management

In Sections 3.4 and 3.5 we discussed different kinds of cuts and how they can potentially be used in a hybrid PB-LP solver. We now describe how we select which cuts to include and how we manage the already added cuts, drawing inspiration from [20, 66].

At the start of the search or right before constraint database cleanups, we initiate a *cut generation phase* by running the LP solver to completion on the current set of constraints in the linear relaxation $\widehat{\psi}$ (which might include previously generated cuts) under a partial assignment consisting only of literals propagated before any variable has been assigned. Once the LP solver has found a rational solution $\widehat{\rho}$ (note that if the linear relaxation is infeasible, then we are done and the whole search is terminated), we collect learned constraints and Gomory cuts into a list of *candidate cuts* as described next.

We first iterate over all constraints in the PB solver constraints database and add to the candidate list those constraints that are violated by $\widehat{\rho}$. Second, we use the LP solver to generate Gomory cuts as discussed in Section 3.4, where we make sure to use exact integer arithmetic just as for the Farkas constraints in Section 3.1. (If the rounding of the multipliers λ_k results in a Gomory cut that is not violated by the LP solution $\widehat{\rho}$, then we discard the constraint.) For efficiency reasons, we limit the number of candidate Gomory cuts to some *Gomory cut limit* γ, using the criteria in [66] to select the most promising ones.

Once we have the list of candidate cuts, we *filter* the list to prioritize constraints that are clearly violated by $\widehat{\rho}$ and that in addition are not too similar to each other. We do so by normalizing all candidate cuts with respect to the Euclidean norm of their vector of coefficients and sorting them in increasing order of rational slack under $\widehat{\rho}$. We then go over the cuts in this order and add each candidate cut C_{cut} to the linear relaxation only if the pairwise cosines with all previously added candidate cuts is sufficiently low (a cosine of 1 would mean that C_{cut} is perfectly parallel to an already added cut).

Importantly, since Gomory cuts are obtained by applying linear combinations and division to previously derived constraints, each such cut is globally valid. When adding a Gomory cut to the linear relaxation, we also include it in the PB solver constraint database ψ analogously to how constraints learned during pseudo-Boolean conflict analysis are added.

Both for the pseudo-Boolean solver and the LP solver performance is negatively affected if the set of constraints grows too large. To avoid this, we prune $\widehat{\psi}$ at the end of a cut

generation phase by removing old cuts that no longer restrict the rational solution.[5] On the pseudo-Boolean solver side, Gomory cuts are dealt with in a similar fashion as regular learned constraints, and are erased according to the same heuristic.[6] We note here that it might be interesting to consider more refined strategies for erasures of cut constraints and study if they could improve performance.

4 Experimental evaluation

In this section, we report on experimental results. As our conflict-driven PB solver we use *RoundingSat* [31],[7] which we integrate with the LP solver *SoPlex* from the MIP solver *SCIP* [34]. In the tables and plots in this section we use the abbreviation *RS+SPX* to refer to this hybrid PB-LP solver. We refer the reader to the repository [25] for a Linux binary of the solver and full experimental data.

Our hybrid solver uses 128-bit precision to calculate Farkas constraints and Gomory cuts, but then scales down coefficients and degree to below the *RoundingSat* limit of 10^9. When scaling down we also adjust the constraints by eliminating literals with coefficients a million times smaller than the average coefficient. Based on preliminary experiments we set parameters as follows:

- To limit the resources used for LP solving as described in Section 3.2, the initial pivot budget is set to 1000, and the pivot ratio limit is fixed to 1 to ensure that the total number of LP pivots is roughly the same as the number of conflicts encountered during pseudo-Boolean search.
- When root LP calls performed in connection with restarts run to completion and produce a rational solution, we always use the rational solution to modify the phase saving heuristic as described in Section 3.3.
- When generating cuts as described in Section 3.6, the parallelism check allows cosines between candidate cuts of up to 0.1, a value taken from [66]. The Gomory cut limit γ was set to 100.

For pseudo-Boolean decision problems without an objective function, the LP solver *SoPlex* is given the (somewhat arbitrary) objective to minimize the sum of all variables—this was to avoid issues with cut generation for such problems. Limited experiments indicate that maximizing the sum of all variables instead does not significantly alter performance. A more refined approach might have been to set the objective function to maximize agreement with the phases of the variables (updating this information at suitably infrequent intervals), but we did not explore this. For inputs that consist only of clausal constraints (or, in other words, CNF formulas) without objective function, we choose to disable all LP solving features[8].

[5] Technically speaking, a constraint no longer restricts the solution if it has a zero dual multiplier.

[6] To be more precise, all constraints are sorted with respect to their so-called *literal block distance (LBD) score* inherited from CDCL solvers [4], and the constraints with the highest (worst) scores are removed. Since Gomory cuts are generated when the solver trail is empty, however, they do not have any initial LBD score, and so for such constraints we instead count the number of variables in the constraint (which is always an upper bound on the LBD score).

[7] We remark that an earlier version of *RoundingSat* participated in the Pseudo-Boolean Competition 2016 [55] under the name *cdcl-cuttingplanes*.

[8] There are 54 MIPLIBdec instances (out of a total of 556) that are only clausal, and 305 PB16dec instances (out of a total of 1783). All other instances in our benchmark sets have an objective functions and/or non-clausal input constraints.

In order to increase the confidence in the correctness of our implementation, we checked the output for 5536 decision instances and 2839 optimization instances (a superset of the benchmarks used for our experimental evaluation) and verified that the results produced were consistent with those from other solvers.

As hardware we used AMD Opteron 6238 nodes having 6 cores and 16 GiB of memory each. Each run was executed as a single thread on a node (leaving 5 cores unused to avoid possible timing issues due to competition for memory resources). All runs had a 5000-second time-out limit. We ran our experiments with the following solvers:

1. *SCIP* [34]: Version 7.0.0 using *SoPlex* version 5.0.0 as LP solver, with presolving support of *PaPILO* 1.0[9] but without symmetry detection. As *RoundingSat* currently does not support multiple CPU threads, we ran *SCIP* in single-thread mode.
2. *RoundingSat* [31, 57] (*RS* for short): Internal development version based on commit 69846924.
3. *RS+SPX*: Baseline integration of *RoundingSat* with *SoPlex* version 5.0.0, implementing the techniques from Section 3 up to and including Section 3.3.
4. *RS+SPX+GC*: Extension of *RS+SPX* with Gomory cut generation as described in Sections 3.4 and 3.6, but without adding learned constraints as cuts to the linear relaxation.
5. *RS+SPX+GC+LC*: Extension of *RS+SPX+GC* using Gomory cuts and also passing learned constraints from the PB solver to the LP solver as discussed in Sections 3.4, 3.5, and 3.6.
6. *Sat4j* [43]: The pseudo-Boolean solver *Sat4j* as per commit 94b8f653, using the *Both* strategy that essentially runs a CDCL solver and a cutting-planes-based PB solver in parallel[10].
7. *NaPS* [59]: The pseudo-Boolean solver *NaPS* as per commit 7aaa54f4, using the *bignum* version as suggested to us by the authors. Note that in constrast to *RoundingSat* and *Sat4j*, *NaPS* does *not* use cutting-planes-based reasoning but instead re-encodes the input to CNF and runs a CDCL solver.

4.1 Efficient detection of rational infeasibility

As a first sanity check, we investigate whether our hybrid solver can solve rationally infeasible *dominating set* formulas from [30], which are hard for the conflict-driven PB solvers *RoundingSat* and *Sat4j*. In Fig. 1 we plot the running time as a function of the size of the instance, so that a lower curve indicates better performance.

Just as in [30], we see that *Sat4j* and *RoundingSat* perform badly while *NaPS* does better (which is likely due to that the re-encoding to CNF introduces extra variables that happen to be helpful for the CDCL solver). For both *SCIP* and our hybrid solver all instances are shown to be infeasible during the initial LP call (we only plot results for the *RS+SPX* hybrid

[9] It is relevant to note here that the pseudo-Boolean solvers we use do not have any preprocessing, and so in this regard *SCIP* and the other MIP solvers considered in this section have a clear advantage over our hybrid solver integrating *RoundingSat* with *SoPlex*.

[10] This is a dual-threaded approach, and so the execution time for each instance is that of the best of the two subsolvers used. We let both subsolvers run until the time-out limit, and so *Sat4j* was in effect given twice the time of the other solvers in our experiments. We chose this setting since it is the standard way of running *Sat4j*, and since we wanted our different hybrid versions of *RoundingSat* to compete against the best version of *Sat4j* for each instance.

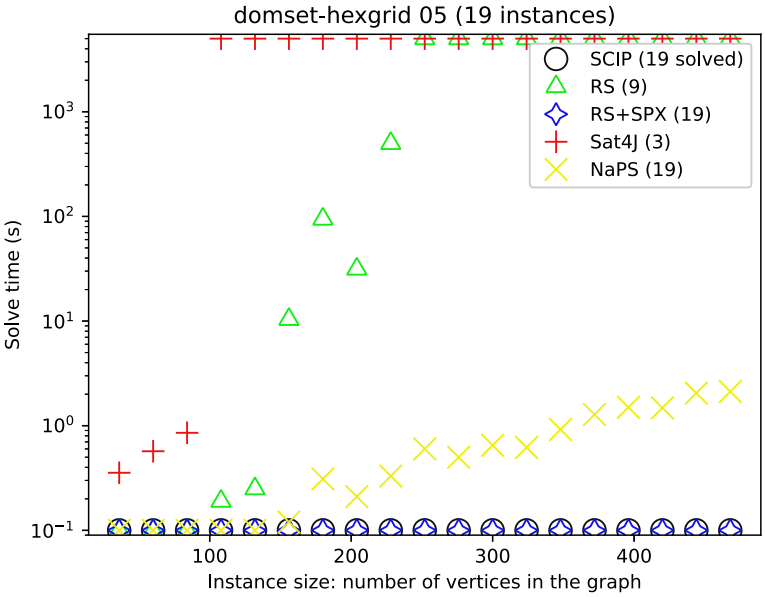

Fig. 1 Results for rationally infeasible dominating set formulas

solver configuration and not for *RS+SPX+GC* and *RS+SPX+GC+LC*, since the latter look very similar).

4.2 Knapsack problems and cut generation

We next consider a class of problems for which cut generation turns out to be useful: the *knapsack* optimization problems taken from [54][11], which we refer to as KNAP below. In Fig. 2 we plot the number of solved instances as a function of the time limit (so that a higher curve is better). The MIP solver *SCIP* solves almost all of the 783 instances, significantly outperforming the pseudo-Boolean solvers we evaluated. Among the PB solvers, *RoundingSat* leads the pack with almost two thirds of the instances solved.

Combining *RoundingSat* with LP solving yields a clear improvement, as shown in the curve for *RS+SPX*, and adding Gomory cut generation (as in the hybrid solver configuration *RS+SPX+GC*) yields another 70 solved instances. Passing learned constraints from the PB solver to the LP solver (*RS+SPX+GC+LC*) provides a further modest boost of 10 additional solved instances, but even this solver configuration gets nowhere close to the performance of *SCIP*. It is relevant to note in this context, however, that *SCIP*, as many other MIP solvers, identifies pure knapsack problems and tries to solve them directly by dynamic programming [33]. Whenever this is feasible with regard to the memory requirements of the dynamic program, this results in very low solving time measurements compared to other techniques based on tree search.

[11] Versions of these benchmarks in the *OPB* format [58] used by pseudo-Boolean solvers can be found at https://doi.org/10.5281/zenodo.3939055.

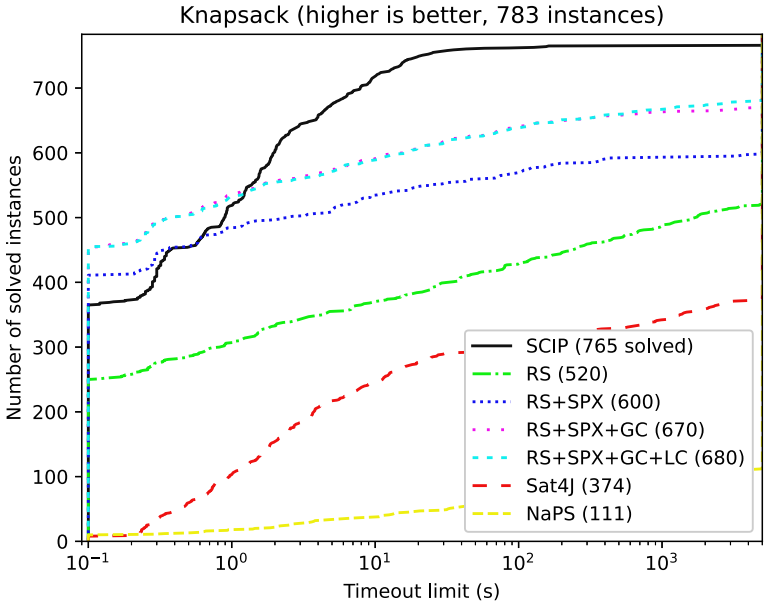

Fig. 2 Cumulative plot for knapsack problems solved to optimality

4.3 Evaluation on PB and MIP benchmark suites

To make a broader comparison to the state of the art in PB and MIP solving, we performed extensive experiments on problems obtained from two large public benchmark sets that each contain both optimization and decision problems.

Our first set consists of the linear, small-coefficient decision problems (DEC-SMALLINT-LIN) and optimization problems (OPT-SMALLINT-LIN) from the most recent pseudo-Boolean competition [55]. For brevity, we will refer to these benchmarks below as PB16dec and PB16opt, respectively.

Our second benchmark set collects 0-1 ILP instances from the mixed integer programming library MIPLIB 2017 [49]. Since this set contains fairly few decision instances, we also created decision versions of the 0-1 ILP optimization problems in the library. From each optimization problem we constructed one decision problem where the objective function f was replaced with a constraint stating that f should be at least the best known value, and a second problem where f was required to be strictly better than this value. This means that the first problem will most often be satisfiable and the second one unsatisfiable, but sometimes rounding makes the first type of problems unsatisfiable, and the second type of problems could also be satisfiable because the best value known is not optimal or because rounding changes the problem. Another issue that we had to deal with is that since *RoundingSat* can currently only deal with integers of magnitude at most 10^9, some of the instances had to be rescaled. We refer to these decision and optimization problems below as MIPLIBdec and MIPLIBopt, respectively[12]. Recall that for inputs that consist only of clausal constraints without objective function, all LP solving features are disabled in the hybrid

[12] Versions of these modified MIPLIB 2017 benchmarks in OPB format can be found at https://doi.org/10.5281/zenodo.3870965.

Table 1 Number of solved PB and 0-1 MIP instances (to optimality for optimization problems)

	SCIP	*RS*	*RS+SPX*	*RS+SPX +GC*	*RS+SPX +GC+LC*	*Sat4j*	*NaPS*
PB16dec (1783)	1123	**1472**	1453	1452	1451	1432	1400
PB16opt (1600)	**1057**	862	988	986	993	776	896
MIPLIBdec (556)	**264**	203	263	261	259	169	170
MIPLIBopt (291)	**125**	78	101	102	102	62	65

The bold entries are the best experimental results for each category

configurations. This is the case for 298 (out of 1783) PB16dec instances and 35 (out of 556) MIPLIBdec instances.

We expect *SCIP* to do very well on MIPLIBopt and MIPLIBdec, since MIP solvers are benchmarked against such instances, and we know from the Pseudo-Boolean Competition 2016 that *NaPS* should be best among the PB solvers on PB16opt whereas *RoundingSat* is expected to be best on PB16dec. In Table 1 we present the number of instances solved within the time-out limit for all solver configurations we evaluated. An optimization instance is considered solved if an optimal solution is found and optimality is proven. We remark that these results seem fairly robust in the sense that the relative ranking of solver configurations would look very similar if we replaced the number of instances solved by the so-called *PAR-2 scores* used in the SAT competitions [61], which take the sum of the running times for all benchmarks except that time-outs are punished by a factor of 2.

For the decision problems, we provide more detailed results in Figs. 3, 4, 5 and 6 by separating the benchmark sets into satisfiable and unsatisfiable instances and providing cumulative plots of the number of benchmarks solved as a function of the running time.

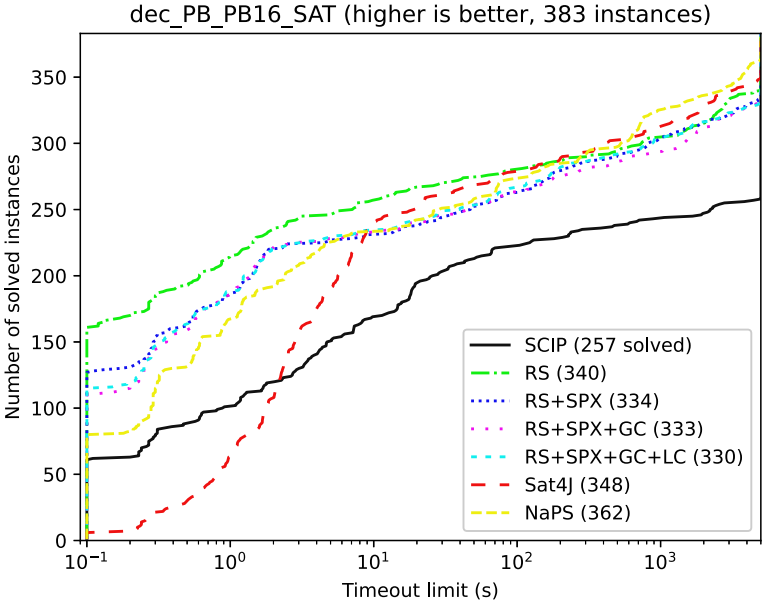

Fig. 3 Cumulative plot for satisfiable PB16 decision problems

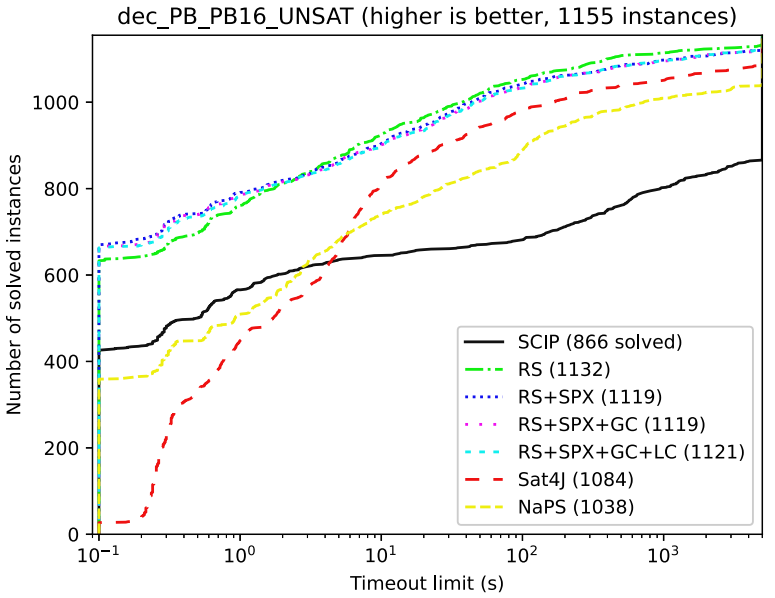

Fig. 4 Cumulative plot for unsatisfiable PB16 decision problems

A first observation is that *SCIP* is a powerful system, outperforming the PB solvers on MIPLIBopt and even PB16opt, as well as delivering top performance on MIPLIBdec. *RoundingSat* is the best solver overall for PB16dec, beating also the hybrid PB-LP solver

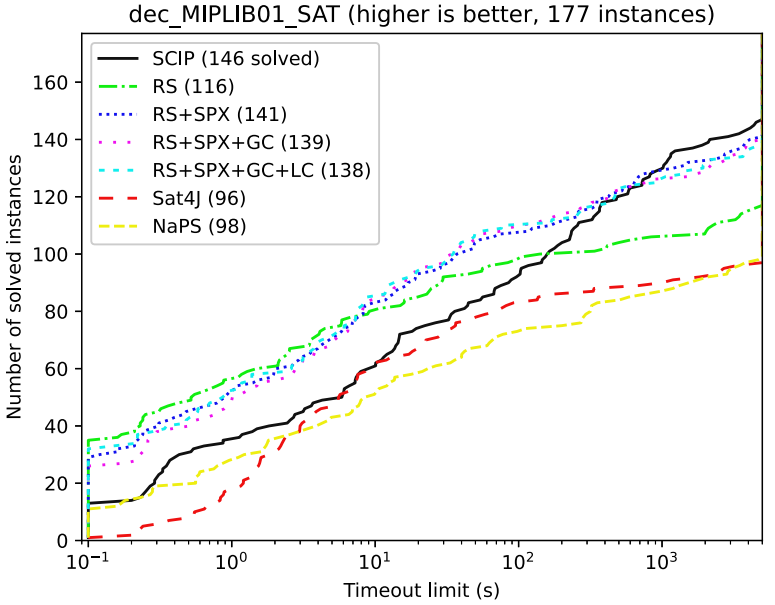

Fig. 5 Cumulative plot for satisfiable MIPLIB decision problems

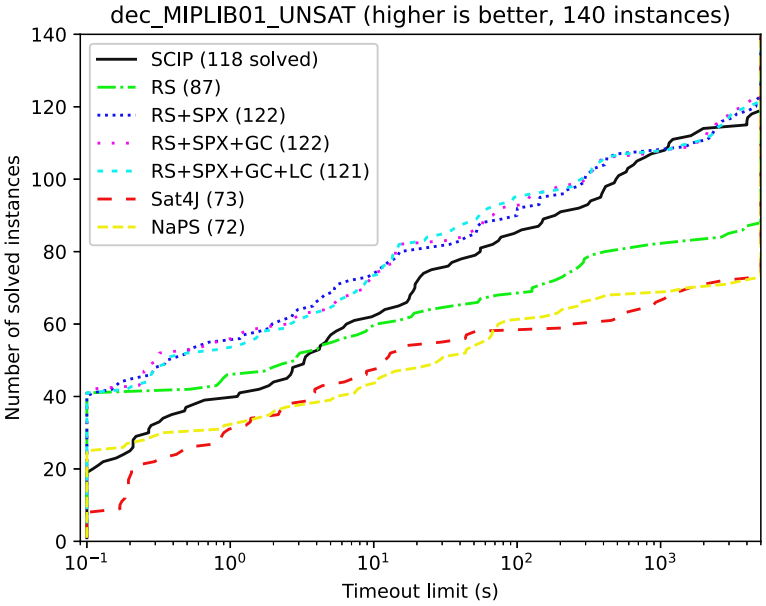

Fig. 6 Cumulative plot for unsatisfiable MIPLIB decision problems

configurations on these benchmarks, but is clearly behind *SCIP* for the other benchmark sets.

Turning to the hybrid PB-LP solvers, interleaving LP calls with the conflict-driven pseudo-Boolean search as in the baseline integration *RS+SPX* closes a significant part of the gap between *RoundingSat* and *SCIP*, up to the point where there is a difference of only 1 instance on MIPLIBdec. We see in Figs. 5 and 6 that *SCIP* is slightly better for the satisfiable instances here whereas the *RoundingSat*-based solvers are slightly better for unsatisfiable instances. Notably, compared to *RoundingSat*, *RS+SPX* has to pay a small price on PB16dec for this performance improvement, and as shown in Figs. 3 and 4 this loss comes from the satisfiable instances. If instead of running time we study the total number of conflicts until the formulas are solved, and use this measure as a proxy for search quality, then it seems that the search in the *RS+SPX* configuration is better that *RoundingSat* for the PB16dec benchmarks, since the conflict counts are lower, but it is not worth the extra time required to perform this more sophisticated search.

Another observation is that while *NaPS* is better than *RoundingSat* on the PB16opt benchmarks, all hybrid versions combining *RoundingSat* with *SoPlex* decisively beat *NaPS* on this benchmark set (where, as we recall, *NaPS* was best in the Pseudo-Boolean Competition 2016).

When it comes to cut generation, we see only marginal differences between the hybrid solver configurations *RS+SPX+GC* (with Gomory cut generation) and *RS+SPX+GC+LC* (where in addition learned constraints are added to the linear relaxation during cut generation). Hence, at least on the PB16 Competition and MIPLIB benchmarks it seems that having the PB solver pass learned constraints to the LP solver does not pay off. It would be interesting to investigate more in-depth why this is so. On the one hand, our results are in line with [1, 60], where it was found that adding clauses arising from conflict analysis as cuts to the LP relaxation was not worthwhile. On the other hand, these papers considered

only clausal constraints, whereas our experiment indicates that even non-clausal learned constraints may not form useful cuts.

The differences between the baseline hybrid solver *RS+SPX* and the version *RS+SPX+GC* with Gomory cuts are similarly small—the number of solved instances are within an additive constant 2 for all categories of benchmarks. We do not know whether this is because cut generation is not a beneficial approach for these problems, or whether it just indicates that our cut generation routine can be improved further. As a general comment, our policies for erasing cuts as described in Section 3.6 are fairly aggressive, and it is a natural question whether it would be possible to develop more effective policies.

4.4 Measuring the usefulness of constraints in the hybrid solver

To gain more insights into how the integration of LP solving affects the pseudo-Boolean solver, we have collected data on how the different constraints generated by the LP solver contribute compared to the regular learned constraints derived during pseudo-Boolean conflict analysis. We have measured the *usage* of constraints C, which we define as the number of times C appears as a conflict or reason constraint during PB conflict analysis, and thus contributes to the process of learning new constraints[13].

Below, we analyse our results for three different constraint types:

Learned constraints Constraints learned from regular pseudo-Boolean conflict analysis.

Farkas constraints Constraints constructed when the LP solver derives rational infeasibility, i.e., the constraints C_{Farkas} from Section 3.1.

Learned Farkas constraints Constraints produced from PB conflict analysis triggered by rational infeasibility, starting from C_{Farkas} as the conflict.

Although we do not discuss it here, we also collected data for other types of constraints, such as the original constraints in the PB formula and generated Gomory cuts. We again refer the reader to the online repository [25] for full data from our experiments and plots visualizing the results.

For a given type of constraints and a PB instance, we define the *average usage* of that constraint type for a PB solver running on the instance as the sum of the usage counts for all constraints of that type divided by the total number of constraints of that type generated during the solver run. This allows us to compare the average usage of, say, Farkas constraints and regular learned constraints. Our underlying assumption here—and it should be pointed out that this is an assumption, and not an iron-clad fact—is that the higher the usage of a constraint, the bigger its contribution to solving the problem, and similarly, the higher the average usage of a constraint type, the bigger the relative usefulness of the technique generating the constraint.

We compare the average usages of the three constraint types listed when running the hybrid solver configuration *RS+SPX+GC+LC* (i.e., with all cut features enabled) on all benchmarks in the KNAP, PB16dec, PB16opt, MIPLIBdec, and MIPLIBopt sets. The scatter plots in Figs. 7 and 8 show the average usages of Farkas constraints compared to regular learned constraints, and of learned Farkas constraints versus regular learned constraints, respectively. There is one data point in the scatter plot per benchmark formula, but we only

[13] We note, though, that this notion of usefulness is a fairly strict measure—a constraint C could also be useful by, e.g., propagating some literal that plays an important role in reaching a conflict without C appearing in the conflict analysis. It might therefore be interesting to follow up on the analysis presented here with a study also of other notions of usefulness.

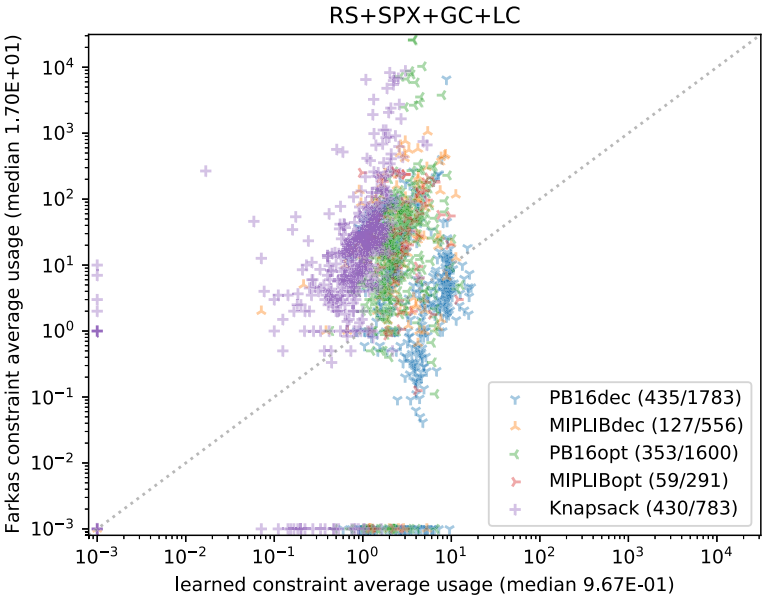

Fig. 7 Scatter plot of average constraint usage for Farkas and learned constraints

consider formulas that do not cause a time-out and for which the solver generates at least one constraint each of the types we are interested in comparing.

Figure 7 indicates that although there exist many instances where the Farkas constraints are virtually never used during conflict analysis, most of the time these constraints seem

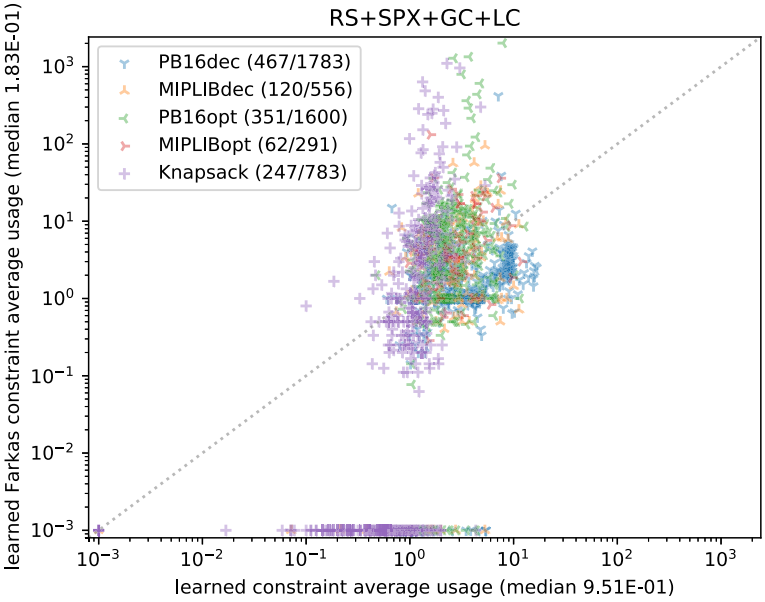

Fig. 8 Scatter plot of average constraint usage for learned Farkas and learned constraints

to be clearly more useful then regular learned constraints. Going by the median usage, the difference appears to be more than an order of magnitude. However, there are significant variations between benchmark sets. For KNAP, PB16opt, and MIPLIBopt, the data indicate that Farkas constraints are much more useful than regular learned constraints, whereas the opposite holds for most PB16dec problems. This agrees with our observations in Sections 4.2 and 4.3 that performance improved significantly on KNAP, PB16opt, and MIPLIBopt benchmarks when integrating LP solving, but deteriorated slightly on PB16dec benchmarks.

In contrast, for the learned Farkas constraints, i.e., the constraints resulting from PB conflict analysis triggered by a Farkas constraint provided by the LP solver, Fig. 8 paints a less favourable picture. Based on the median values, such learned Farkas constraints appear to be less useful than constraints learned from conflict analysis triggered during the regular PB search. However, there is a big spread in the usefulness measurements. On the one hand, there is a cluster of data points at the bottom of the plot resulting from formulas where learned Farkas constraints are essentially never used, but on the other hand there are other data points at the top of the plot corresponding to formulas where learned Farkas constraints are used heavily, up to a thousand times per constraint on average. We find these results intriguing, and believe that they merit further study. The reason to perform pseudo-Boolean conflict analysis in general is to learn new constraints that can restrict the search space, but for conflicts resulting from infeasibility of the linear relaxation it seems that the Farkas constraint provided by the LP solver tends to be more useful than anything the PB solver can derive by carrying out conflict analysis starting from this constraint.

4.5 A limited comparison with commercial MIP solvers

As outlined in the introduction, the grand goal that we set ourselves for this work was to integrate pseudo-Boolean solving and mixed integer programming in order to achieve "the best of both worlds." However, the results discussed in Section 4.3 show that the MIP solver *SCIP* outperforms our hybrid PB-LP solver even on its home turf of the PB competition. If this is already true for open-source solvers such as *SCIP*, what should be expected from commercial MIP solvers such as *Gurobi* [36] and *CPLEX* [40], which are generally acknowledged to be even faster? Is there any room for contributions from pseudo-Boolean solving techniques here?

We believe that the answer is "yes." What conflict-driven pseudo-Boolean solvers have to offer is sophisticated conflict analysis and learning. In contrast to the CDCL-style conflict analysis that has been ported to MIP previously, cutting-planes-based PB solvers operate directly on the linear constraints, rather than on disjunctive clauses extracted from these constraints, and this makes the reasoning exponentially more powerful.

Admittedly, we do not see much indication of such a benefit from pseudo-Boolean conflict analysis in our experiments—except perhaps for PB16dec, though this benchmark set contains a large number of crafted instances—but it is important to point out that this could well be a problem of selection bias. When choosing formulas for competitions or standardized benchmark sets, there is a natural tendency to pick problems that are interesting challenges, but that are still fundamentally within reach. If some formulas are completely impossible to solve without using some radically new and different approach—such as cutting-planes-based conflict analysis—then chances are that such instances will not make it into the competitions or benchmark sets.

One possible approach to address this issue is to construct crafted formulas, for which our theoretical understanding tells us that the conflict analysis used in pseudo-Boolean solvers should be essential, and then run experiments on such instances. For this purpose, we

have designed what we refer to as *composed pebbling-PHP* formulas, which combine the *pigeonhole principle (PHP) formulas* studied in [37] with the *pebbling formulas* from [11].

Before presenting our experimental results, we try to outline how these benchmarks are constructed. The pebbling formulas that serve as the starting point of our construction are defined in terms of directed acyclic graphs (DAGs), and express the simultaneous constraints that:

- all source vertices s in the DAG, i.e., vertices without incoming edges, are "true";
- if all immediate predecessors of a vertex v in the DAG are "true", then this vertex v itself must also be "true";
- however, the unique sink vertex z in the DAG, without any outgoing edges, must be "false".

It is a straightforward inductive argument over the vertices of the DAG in topological order that this collection of constraints is contradictory. By associating with every vertex v two Boolean variables v_1 and v_2 and defining "true" to mean that the disjunction $v_1 \vee v_2$ must hold, and by choosing the right kind of DAGs,[14] we obtain families of formulas that are exponentially hard for DPLL-style solvers [22] performing tree-like search without learning, but that are very easy for CDCL solvers [10, 42].

In a second step, we want to make these formulas exponentially hard for the resolution method employed in CDCL—the kind of reasoning that has been used before in MIP solvers—while maintaining the property that they are easy for pseudo-Boolean reasoning. We do so by composing the pebbling constraints above for every vertex v with a pigeonhole principle formula PHP_v encoding the (false) claim that $n + 1$ pigeons can be mapped to n holes in a one-to-one fashion. Informally speaking, we let the composed pebbling-PHP formulas consist of PB constraints encoding the following conditions:

- For every source s it holds that either s is "true" or PHP_s is satisfiable;
- if all immediate predecessors u of a vertex v are "true", then either v is also true or some formula PHP_u or PHP_v is satisfiable;
- Either the sink z is "false" or PHP_z is satisfiable.

Clearly, these PB formulas are unsatisfiable, but the point is that simple tree-like reasoning, corresponding to backtracking algorithms without re-use of derived constraints, is not sufficient to prove this (because of the pebbling constraints), and neither is resolution-based reasoning (because of the PHP formulas). However, conflict-driven PB solvers using the cutting planes method should in principle be able to prove unsatisfiability efficiently. As described above, these formulas encode decision problems, but they can be turned into optimization problems in a natural way by removing the constraint for the sink and instead setting the objective of minimizing the number of true variables (where the optimal value will be equal to the number of vertices in the DAG). This concludes the description of our crafted benchmark formulas.

We have run experiments on the decision versions of composed pebbling-PHP formulas[15] for all the solvers evaluated in this section, but in addition we have also included the commercial MIP solvers *Gurobi* 9.0.2 and *CPLEX* 12.10.0. Due to academic license

[14] Namely, DAGs with sufficiently high *black-white pebbling price*—in this case we use so-called *pyramid graphs*. See [15, 52] for more in-depth discussions of how pebbling formulas generated from different graphs have been used in proof complexity research.

[15] These instances of composed pebbling-PHP formulas in OPB format can be found at https://doi.org/10.5281/zenodo.3945406.

restrictions we could not use *Gurobi* on the supercomputing cluster where the rest of our experiments were run. Therefore, for this experiment we instead used an Intel Core i7-5600U machine with 16 GiB RAM. We have collected all time measurements in Fig. 9, where again running time is plotted as a function of the instance size, so that a lower curve is better.

All solvers based on *RoundingSat* perform well on these formulas, with a low variation in running times between different solver configurations. It is not immediately clear why *Sat4j* is so much worse, but this is consistent with results from [30] indicating that *RoundingSat* is better at cutting-planes-based reasoning (at least for the selection of PB benchmarks investigated in the literature so far). Intriguingly, *Gurobi* and *SCIP* perform very poorly, and even though *CPLEX* is better it is nowhere close to the *RoundingSat*-based solver configurations.

We cannot know for sure why the commercial solvers *Gurobi* and *CPLEX* exhibit such bad performance—not least because they are closed-source solvers—but, intuitively, the formulas have been constructed in such a way that cutting-planes-based reasoning with reuse of previously derived constraints should be essential to solve them efficiently. Conflict-driven PB solvers can achieve this thanks to the fact that they are operating directly on the linear constraints, but to the best of our knowledge no MIP solvers currently have this kind of conflict analysis implemented. And solvers using conflict analysis based on disjunctive clauses extracted from the constraints will need exponential time in view of the hardness results for the resolution method in [37].

As a final note, we find it surprising that adding LP solving, as in the hybrid configuration *RS+SPX*, does not improve performance. Since the pseudo-Boolean encoding of the PHP_v formulas yield rationally infeasible LPs, this should be an excellent opportunity for the LP solver to shine. But perhaps the explanation is that already the base PB solver *RoundingSat* easily eliminates the PHP_v formulas.

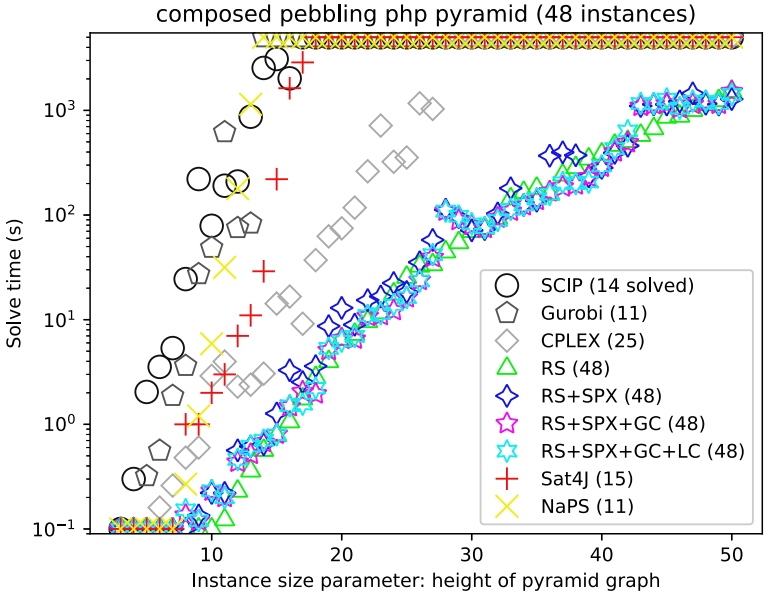

Fig. 9 Solve time on composed pebbling-PHP formulas

5 Concluding remarks

In this work, we propose a novel method to tightly integrate linear programming (LP) inside conflict-driven pseudo-Boolean (PB) solving. Our hybrid solver interleaves the conflict-driven PB search with calls to an LP solver. When the LP solver detects infeasibility, the Farkas certificate of unsatisfiability that it returns can be used as a starting point for pseudo-Boolean conflict analysis. If instead the LP solver finds a rational solution, then such solutions can be used to infer Gomory cuts that are passed to the PB solver, and in the other direction constraints learned during PB conflict analysis can be added to the LP relaxation. In contrast to previous LP integrations, our hybrid solver can exploit these derived constraints to the full extent, since our PB conflict analysis operates natively with linear inequalities as opposed to disjunctive clauses derived from these inequalities. To the best of our knowledge this is the first solver to combine MIP techniques such as solving LP relaxations and generating cuts with pseudo-Boolean conflict analysis over linear inequalities using the cutting planes reasoning method.

We identify several crucial aspects that require careful attention in order to achieve a successful integration of linear programming in the pseudo-Boolean solver. First, we carefully manage the time spent on LP solving (measured as the number of simplex pivots performed) in order to avoid starving the pseudo-Boolean solver, and we adaptively adjust the time limits to maximize the utility of the LP solver and avoid LP calls that have to be terminated before completion. Second, when the LP solver finds rational solutions to the linear relaxation of the pseudo-Boolean problem, we use such solutions to reinitialize the phase saving heuristic in the PB solver. Finally, when using information from the LP solver to compute Farkas witnesses of infeasibility or Gomory cuts eliminating rational solutions, we make sure to perform all calculations in exact integer arithmetic, meaning that even though the LP solver uses floating-point arithmetic the pseudo-Boolean solver will not be affected by any rounding errors in such calculations.

Although our proof-of-concept PB-LP hybrid solver integrating *RoundingSat* with *SoPlex* was mainly built for evaluation purposes, our experiments get close to a kind of "best-of-both-worlds" results, where for the pseudo-Boolean benchmarks from the latest PB competition [55] we retain excellent performance on decision problems and improve significantly on optimization problems, and for the 0-1 integer linear programming (ILP) benchmarks obtained from MIPLIB 2017 [49] we close a significant part of the gap between PB and MIP solvers. In a similar way, our hybrid solver gets much closer to MIP solver performance on the knapsack problems in [54].

As to future research directions, there are many questions raised by our work that we believe merit further study, concerning both design choices in the hybrid PB-LP solver and methods to analyse the impact of such choices. Several of these questions are discussed in detail in Sections 3 and 4, but we take the opportunity here to highlight again some problems that we consider to be particularly interesting.

For MIP solvers, the generation of new cutting planes inequalities, especially Gomory and MIR cuts, is a crucial component in the solving process [3]. When we tried to adapt this technique to the pseudo-Boolean setting we found it to be effective on some benchmarks, but we were not able to achieve any improvements in performance overall. In view of this, it seems important to investigate further how LP-based cut generation could be used in pseudo-Boolean solvers to yield the same kind of speed-ups as those seen in modern MIP solvers.

We find it similarly disappointing that sharing constraints learned during pseudo-Boolean conflict analysis with the LP solver does not lead to any significant gains in performance. It has been shown before that adding disjunctive clauses to an LP relaxation is not helpful, but we would like to understand if this is also true for stronger, general pseudo-Boolean constraints, and if so why. In order to gain more insights into how to share cut constraints between the PB solver and LP solver, it would be valuable to collect more statistics on how different types of constraints affect pseudo-Boolean solving, and whether different types of constraints could be managed in more refined ways to maximize their impact.

One quite unexpected, but positive, finding, is that the Farkas constraints generated from LP calls showing infeasibility turn out to be so useful for the pseudo-Boolean solver. We have no rigorous explanation why this is so. If we allow ourselves to speculate, perhaps one reason can be that Farkas constraints are derived in situations where the PB solvers believes that the current partial assignment might be extended to satisfy the formula, since no further propagations can be deduced from the individual constraints, whereas the LP solver can use more powerful propagation to see that the search has in fact reached a dead end. It could be that the linear combination of constraints applied to generate the Farkas witness is a type of derivation that the PB solver is less likely to perform during conflict analysis, since it only considers the constraints one by one during the search and never reasons on sets of constraints collectively.

But if Farkas constraints are so useful, how is it that the pseudo-Boolean conflict analysis triggered by such constraints seem to produce much less useful results than regular conflict analysis? Could it perhaps be that the Farkas constraints already have quite large coefficients, and that further steps in the conflict analysis quickly make the coefficients grow too large, so that the constraints have to be rounded down and lose in strengh? Could it even be the case that the PB solver would be better off by *not* performing conflict analysis on Farkas constraints and instead just backtrack immediately? These questions warrant further investigation.

A key concern if we wish to improve the hybrid PB-LP solver further is how to balance the time spent on PB solving and LP solving. Our current heuristic is static in the sense that it limits the total number of pivots performed by the LP solver by the PB solver conflict count. We have seen, however, that some families of problems benefit much more from the LP integration that others. If one could somehow measure at runtime how beneficial LP solving seems to be, then this could be used to adapt the time resources allocated to the LP solver dynamically to improve performance.

Another question is whether we can make more use of information provided by the LP solver to guide the pseudo-Boolean search. We have already experimented with rounding rational solutions to update the phase, i.e., to decide how variables are assigned to values in the PB solver, but it could be interesting to investigate whether later propagations should be allowed to overwrite the phase or whether the phase values obtained from the LP solver should be kept until the next time a rational solution is found. Also, perhaps the values in the rational solution, and how close to or far away from 0 or 1 they are, could be used to influence not only the values of variable assignments but also the choices of which variables to assign next.

In general, we believe that there should be ample room for technology transfer from mixed integer linear programming to pseudo-Boolean solving. As a concrete example, current state-of-the-art PB solvers have very limited preprocessing capabilities, which stands in striking contrast to both SAT solvers and MIP solvers. A very natural, and very attractive, idea would be to apply MIP presolving techniques for 0-1 ILPs to pseudo-Boolean formulas before starting the conflict-driven PB solver.

Finally, we want to mention that a potential extension of our work would be to leverage the LP solver to allow PB solvers to support continuous variables. In this setting, the PB solver would construct partial assignments to the Boolean variables that do not violate the formula, and the LP solver would then try to find extensions to total assignments that also include continuous variables. If no such satsfying assignment exists, exact integer arithmetic could be used to construct a Farkas constraint that would trigger a conflict in the PB solver, and this would guide the pseudo-Boolean search to find some other Boolean assignment that could serve as a candidate to be extended to solution encompassing also the continuous variables.

Acknowledgments We are most grateful to Jan Elffers for sharing results from his experiments combining *RoundingSat* and *SoPlex*, which served as the starting point for our hybrid solver. We are also thankful to him, as well as to Stephan Gocht and Janne Kokkala, for many interesting discussions throughout this project. We gratefully acknowledge useful feedback from participants of the *Theory and Practice of Satisfiability Solving* workshop in Oaxaca in August 2018, the *Pragmatics of SAT* workshop in Lisbon in July 2019, and the *Swedish Operations Research Conference (SOAK)* in Nyköping in October 2019. In particular, we thank Nikolaj Bjørner for suggesting to us that we should run experiments on knapsack problems.

Our computational experiments used resources provided by the Swedish National Infrastructure for Computing (SNIC) at the High Performance Computing Center North (HPC2N) at Umeå University. The first and third authors were supported by the Swedish Research Council grant 2016-00782, and the third author also received funding from the Independent Research Fund Denmark grant 9040-00389B. The second author was supported by the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant numbers 05M14ZAM and 05M20ZBM). Part of this work was carried out while the third author visited the Simons Institute for the Theory of Computing in association with the DIMACS/Simons Collaboration on Lower Bounds in Computational Complexity, which is conducted with support from the National Science Foundation.

References

1. Achterberg, T. (2007). Conflict analysis in mixed integer programming. *Discrete Optimization*, *4*(1), 4–20.
2. Achterberg, T., Berthold, T., Koch, T., Wolter, K. (2008). Constraint integer programming: a new approach to integrate CP and MIP. In *Proceedings of the 5th International conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR '08). Lecture notes in computer science*, (Vol. 5015 pp. 6–20): Springer.
3. Achterberg, T., & Wunderling, R. (2013). Mixed integer programming: Analyzing 12 years of progress. In Jünger, M., & Reinelt, G. (Eds.) *Facets of combinatorial optimization* (pp. 449–481): Springer.
4. Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International joint conference on artificial intelligence (IJCAI '09)* (pp. 399–404).
5. Barth, P. (1996). *Logic-based 0–1 constraint programming, operations research/computer science interfaces series* Vol. 5. Berlin: Springer.
6. Barth, P., & Bockmayr, A. (1995). Finite domain and cutting plane techniques in CLP(PB). In *Proceedings of the 12th international conference on logic programming* (pp. 133–147). Cambridge: MIT Press.
7. Barth, P., & Bockmayr, A. (1998). Modelling discrete optimisation problems in constraint logic programming. *Annals of Operations Research*, *81*, 467–496.
8. Bayardo Jr., R.J., & Schrag, R. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th national conference on artificial intelligence (AAAI '97)* (pp. 203–208).
9. Beame, P., Kautz, H., Sabharwal, A. (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, *22*, 319–351. preliminary version in IJCAI '03.
10. Ben-Sasson, E., Impagliazzo, R., Wigderson, A. (2004). Near optimal separation of tree-like and general resolution. *Combinatorica*, *24*(4), 585–603.

11. Ben-Sasson, E., & Wigderson, A. (2001). Short proofs are narrow—resolution made simple. *Journal of the ACM*, *48*(2), 149–169. preliminary version in STOC '99.
12. Bixby, R., & Rothberg, E. (2007). Progress in computational mixed integer programming— A look back from the other side of the tipping point. *Annals of Operations Research*, *149*(1), 37–41.
13. Blake, A. (1937). Canonical expressions in boolean algebra. Ph.D. thesis, University of Chicago.
14. Bockmayr, A. (1995). Solving pseudo-Boolean constraints. In *Constraint programming: Basics and trends. TCS school 1994. Lecture notes in computer science*, (Vol. 910 pp. 22–38): Springer.
15. Buss, S.R., & Nordström, J. Proof complexity and SAT solving. In Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.), *Handbook of Satisfiability, chap. 7 (pp. 233–350)*. Frontiers in Artificial Intelligence and Applications, IOS Press, 2nd edn. (2021), to appear. Currently available at http://www.csc.kth.se/jakobn/research/.
16. Calabro, C., Impagliazzo, R., Paturi, R. (2009). The complexity of satisfiability of small depth circuits. In *Revised selected papers from the 4th international workshop on parameterized and exact computation (IWPEC '09). Lecture notes in computer science*, (Vol. 5917 pp. 75–85): Springer.
17. Chai, D., & Kuehlmann, A. (2005). A fast pseudo-Boolean constraint solver. *IEEE transactions on computer-aided design of integrated circuits and systems*, *24*(3), 305–317. preliminary version in DAC '03.
18. Cook, S.A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the 3rd annual ACM symposium on theory of computing (STOC '71)* (pp. 151–158).
19. Cook, W., Coullard, C.R., Turán, G. (1987). On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, *18*(1), 25–38.
20. Cook, W., Dash, S., Fukasawa, R., Goycoolea, M. (2009). Numerically safe Gomory mixed-integer cuts. *INFORMS Journal on Computing*, *21*(4), 641–649.
21. Dantzig, G. (1963). *Linear programming and extensions*. Princeton: Princeton University Press.
22. Davis, M., Logemann, G., Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM*, *5*(7), 394–397.
23. Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM*, *7*(3), 201–215.
24. Demirović, E., Chu, G., Stuckey, P.J. (2018). Solution-based phase saving for CP: a value-selection heuristic to simulate local search behavior in complete solvers. In *Proceedings of the 24th international conference on principles and practice of constraint programming (CP '18). Lecture notes in computer science*, (Vol. 11008 pp. 99–108): Springer.
25. Devriendt, J., Gleixner, A., Nordström, J., Nordström J. (2020). Experimental Repository for "Learn to Relax: Integrating 0-1 Integer Linear Programming with Pseudo-Boolean Conflict-Driven Search". https://doi.org/10.5281/zenodo.3945412.
26. Dixon, H.E., & Ginsberg, M.L. (2002). Inference methods for a pseudo-Boolean satisfiability solver. In *Proceedings of the 18th national conference on artificial intelligence (AAAI '02)* (pp. 635–640).
27. Downing, N.R. (2016). Scheduling and rostering with learning constraint solvers. Ph.D. thesis, University of Melbourne, available at https://minerva-access.unimelb.edu.au/handle/11343/129704.
28. Dutertre, B., & de Moura, L. (2006). A fast linear-arithmetic solver for DPLL(t). In *Proceedings of the 18th international conference on computer aided verification (CAV '06). Lecture notes in computer science*, (Vol. 4144 pp. 81–94): Springer.
29. Eén, N., & Sörensson, N. (2006). Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, *2*(1-4), 1–26.
30. Elffers, J., Giráldez-cru, J., Nordström, J., Vinyals, M. (2018). Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. In *Proceedings of the 21st international conference on theory and applications of satisfiability testing (SAT '18). Lecture Notes in computer science*, (Vol. 10929 pp. 75–93): Springer.
31. Elffers, J., & Nordström, J. (2018). Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th international joint conference on artificial intelligence (IJCAI '18)* (pp. 1291–1299).
32. Farkas, J. (1902). ÜBer die Theorie der einfachen Ungleichungen. *Journal fur die reine und angewandte Mathematik*, *1902*, 1–27. https://doi.org/10.1515/crll.1902.124.1.
33. Furini, F., Ljubić, I., Sinnl, M. (2017). An effective dynamic programming algorithm for the minimum-cost maximal knapsack packing problem. *European Journal of Operational Research*, *262*(2), 438–448.
34. Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Bodic, P.L., Maher, S.J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., Witzig, J. (2020). The SCIP Optimization Suite 7.0. Technical report, Optimization Online. http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

35. Gebser, M., Kaufmann, B., Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, *187–188*, 52–89.
36. Gurobi optimizer. https://www.gurobi.com/.
37. Haken, A. (1985). The intractability of resolution. *Theoretical Computer Science*, *39*(2-3), 297–308.
38. Hooker, J.N. (1988). Generalized resolution and cutting planes. *Annals of Operations Research*, *12*(1), 217–239.
39. Hooker, J.N. (1992). Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, *6*(1), 271–286.
40. IBM ILOG CPLEX optimization studio. https://www.ibm.com/products/ilog-cplex-optimization-studio.
41. Impagliazzo, R., & Paturi, R. (2001). On the complexity of k-SAT. *Journal of Computer and System Sciences*, *62*(2), 367–375. preliminary version in CCC '99.
42. Järvisalo, M., Matsliah, A., Nordström, J., Živný, S. (2012). Relating proof complexity measures and practical hardness of SAT. In *Proceedings of the 18th international conference on principles and practice of constraint programming (CP '12). Lecture notes in computer science*, (Vol. 7514 pp. 316–331): Springer.
43. Le Berre, D., & Parrain, A. (2010). The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*, 59–64.
44. Levin, L.A. (1973). Universal sequential search problems. *Problemy peredachi informatsii*, *9*(3), 115–116. in Russian. Available at http://mi.mathnet.ru/ppi914.
45. Manquinho, V.M., & Marques-Silva, J.P. (2006). Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design*, *21*(5), 505–516.
46. Marchand, H., & Wolsey, L. (1998). Aggregation and mixed integer rounding to solve MIPs.
47. Marques-Silva, J.P., & Sakallah, K.A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, *48*(5), 506–521. preliminary version in ICCAD '96.
48. Martins, R., Manquinho, V.M., Lynce, I. (2014). Open-WBO: A modular maxSAT solver. In *Proceedings of the 17th international conference on theory and applications of satisfiability testing (SAT '14). Lecture notes in computer science*, (Vol. 8561 pp. 438–445): Springer.
49. MIPLIB 2017 (2018), http://miplib.zib.de.
50. Mitchell, J. (2002). Branch-and-cut algorithms for combinatorial optimization problems. Handbook of Applied Optimization, 65–77.
51. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th design automation conference (DAC '01), 530–535*.
52. Nordström, J. (2013). Pebble games, proof complexity and time-space trade-offs. *Logical Methods in Computer Science*, *9*(3), 15:1–15:63.
53. Pipatsrisawat, K., & Darwiche, A. (2007). A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th international conference on theory and applications of satisfiability testing (SAT '07). Lecture Notes in computer science*, (Vol. 4501 pp. 294–299): Springer.
54. Pisinger, D. (2005). Where are the hard knapsack problems? *Computers & Operations Research*, *32*(9), 2271–2284.
55. Pseudo-Boolean competition 2016. http://www.cril.univ-artois.fr/PB16/ (2016).
56. Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, *12*(1), 23–41.
57. RoundingSat. https://gitlab.com/miao_research/roundingsat.
58. Roussel, O., & Manquinho, V.M. (2016). Input/output format and solver requirements for the competitions of pseudo-Boolean solvers, revision 2324. Available at http://www.cril.univ-artois.fr/PB16/format.pdf.
59. Sakai, M., & Nabeshima, H. (2015). Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, *98-D*(6), 1121–1127.
60. Sandholm, T., & Shields, R. (2006). Nogood learning for mixed integer programming. In *Workshop on hybrid methods and branching rules in combinatorial optimization*.
61. The international SAT Competitions web page. http://www.satcompetition.org.
62. Sheini, H.M., & Sakallah, K.A. (2006). Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *2*(1-4), 165–189. preliminary version in DATE '05.
63. Shen, K., & Schimpf, J. (Oct 2005). Eplex: Harnessing mathematical programming solvers for constraint logic programming. In *Proceedings of the 11th international conference on principles and practice of constraint programming (CP '05). Lecture notes in computer science*, (Vol. 3709 pp. 622–636): Springer.
64. Vanderbei, R.J. (1996). *Linear programming: Foundations and extensions*. US: Springer.

65. Vinyals, M., Elffers, J., Giráldez-cru, J., Gocht, S., Nordström, J. (2018). In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving. In *Proceedings of the 21st international conference on theory and applications of satisfiability testing (SAT '18). Lecture notes in computer science*, (Vol. 10929 pp. 292–310): Springer.
66. Wesselmann, F., & Suhl, U.H. (2012). Implementing cutting plane management and selection techniques. Technical report, Optimization Online, http://www.optimization-online.org/DB_HTML/2012/12/3714.html.
67. Zhou, N., Tsuru, M., Nobuyama, E. (2012). A comparison of CP, IP, and SAT solvers through a common interface. In *Proceedings of the 24th IEEE international conference on tools with artificial intelligence (ICTAI '12)* (pp. 41–48).