Divide and Conquer: Towards Faster Pseudo-Boolean Solving

Jan Elffers and Jakob Nordström
KTH Royal Institute of Technology
{elffers,jakobn}@kth.se

Abstract

The last 20 years have seen dramatic improvements in the performance of algorithms for Boolean satisfiability—so-called SAT solvers—and today conflict-driven clause learning (CDCL) solvers are routinely used in a wide range of application areas. One serious short-coming of CDCL, however, is that the underlying method of reasoning is quite weak. A tantalizing solution is to instead use stronger pseudo-Boolean (PB) reasoning, but so far the promise of exponential gains in performance has failed to materialize—the increased theoretical strength seems hard to harness algorithmically, and in many applications CDCL-based methods are still superior. We propose a modified approach to pseudo-Boolean solving based on division instead of the saturation rule used in [Chai and Kuehlmann ’05] and other PB solvers. In addition to resulting in a stronger conflict analysis, this also improves performance by keeping integer coefficient sizes down, and yields a very competitive solver as shown by the results in the Pseudo-Boolean Competitions 2015 and 2016.

1 Introduction

The Boolean satisfiability problem (SAT) is one of the most fascinating problems in computer science. Though deceptively easy to state—given a formula in propositional logic, is it possible to assign variables true and false so that it evaluates to true?—it has been the focus of extensive research ever since the dawn of computer science (and even before—cf. Gödel’s famous letter to von Neumann in 1956).

SAT was proven NP-complete in [Cook, 1971; Levin, 1973], laying the foundation for computational complexity theory, and a widely accepted hypothesis is that the problem requires exponential time in the worst case [Impagliazzo and Paturi, 2001]. Yet the last couple of decades have seen the development of SAT solvers based on the conflict-driven clause learning (CDCL) paradigm introduced in [Marques-Silva and Sakallah, 1999], with further improvements in [Moskewicz et al., 2001] and later papers, that are very efficient in practice. Today CDCL solvers are routinely used to solve large-scale real-world problems in, e.g., hardware and software verification, AI planning, automated theorem proving, and many other areas (see [Biere et al., 2009] for more details).

A drawback with CDCL is that from a mathematical point of view the method of reasoning is quite weak—it is based on the resolution proof system, for which exponential lower bounds are known even for simple combinatorial principles [Haken, 1985; Urquhart, 1987]. Another issue is that problems have to be encoded in conjunctive normal form (CNF) as a collection of disjunctive clauses, which loses the semantics of higher-level constraints and further hurts the reasoning power. This can be addressed by using preprocessing techniques for, e.g., Gaussian elimination and cardinality reasoning, but these techniques work only in limited cases and are again quite sensitive to the exact encoding of the problem.

An attractive option is to instead use (linear) pseudo-Boolean (PB) constraints i.e., integral linear inequalities over Boolean variables, which gives a succinct way of encoding problems in many different fields. PB constraints are more expressive than CNF, but are close enough that CNF-based techniques can be harnessed to attack pseudo-Boolean problems. The connection to integer linear programming (ILP) and, in particular 0-1 programming, makes it natural to also borrow insights from these areas.

Some pseudo-Boolean solvers are still based on resolution, in that they translate the input to CNF. This can be done eagerly, so that all linear inequalities are converted to CNF after which a CDCL solver is called (as in MiniSat+ [Eén and Sörensson, 2006], Open-WBO [Martins et al., 2014; Joshi et al., 2015], and NaPS [Sakai and Nabeshima, 2015]), or lazily in a modified CDCL solver that keeps the PB format of the input but derives new information only in the form of clauses (as in one of the methods in the Sat4j library [Le Berre and Parrain, 2010]). Another approach is to go beyond resolution and build solvers using the cutting planes method [Cook et al., 1987]. It should be noted that extending the conflict-driven framework to pseudo-Boolean constraints is non-trivial, but methods to do so were designed in the solvers PRS [Dixon and Ginsberg, 2002] and Galena [Chai and Kuehlmann, 2005], and were further developed in Pueblo [Shei and Sakallah, 2006] and Sat4j [Le Berre and Parrain, 2010] (related, but slightly different, ideas...
were also explored in bsolo [Manquinho and Marques-Silva, 2006]. Needless to say, this is far from a complete overview of pseudo-Boolean solving—see, e.g., the excellent survey in [Biere et al., 2009, Chapter 22] for more information.

From a theoretical standpoint, using cutting planes seems clearly preferable, since this method is never worse than resolution but can be exponentially stronger. In practice this is not so, however—often CDCL-based solvers outperform PB solvers based on cutting planes! One possible explanation for this state of affairs lies in the tension between the two desiderata that the method of reasoning utilized should (1) be as powerful as possible but also (2) allow for efficient search.

Regarding the second consideration, storing and manipulating linear constraints is much more costly than dealing with simple disjunctive clauses, and can slow down the solver prohibitively [Shaie and Sakallah, 2006]. Integer coefficients can also grow very large during search, requiring the usage of multi-precision arithmetic which decreases performance still further. In addition to these implementation concerns, a richer set of derivation rules makes the search space much larger and more challenging to explore.

Perhaps partly to address this issue, solvers based on [Chai and Kuehlmann, 2005] do not use the full power of cutting planes. In particular, instead of the division rule they employ the simpler to implement saturation rule. This again makes solvers quite sensitive to details in the input format, however, and causes them to degenerate to resolution in certain cases where cutting planes is very efficient [Hooker, 1992; Vinyals et al., 2018].

### Our Contribution

In this work, we further develop the theoretical foundations of conflict-driven pseudo-Boolean solving in [Chai and Kuehlmann, 2005] to incorporate the division rule in [Cook et al., 1987] as well as some other useful extensions, and report results from implementing these ideas in the solver RoundingSat.

The new method has several advantages. For problems where PB reasoning does not seem to afford much of an advantage compared to CDCL, the search speed measured in terms of number of conflicts per second is orders of magnitude faster than in other cutting-planes-based PB solvers, approaching the vicinity of standard CDCL. Thanks to the aggressive use of division, integer coefficients normally remain within the range of 32 bits (for inputs with coefficients of bounded size), which makes it possible to employ fixed-precision arithmetic. From a theoretical point of view, using division means that the solver has the potential to be exponentially stronger for problems where 0-1-integer reasoning rather than just CDCL or linear programming is crucial [Vinyals et al., 2018]. It seems fair to say that this is also borne out in practice, as shown by the Pseudo-Boolean Competitions in 2015 and 2016, as well as by further improvements obtained since then as reported in this paper.

### Organization of This Paper

In Section 2, we review the basics of conflict-driven pseudo-Boolean solving, after which our contributions are presented in Section 3. Section 4 contains the results from our empirical evaluations. We conclude in Section 5 with some directions for further research. Some supplemental material (including experimental data) can be found at www.csc.kth.se/~jakobn/RoundingSat.

### 2 Conflict-Driven Pseudo-Boolean Search

In this section we review how conflict-driven solving can be extended from CNF to pseudo-Boolean constraints, in order to present the framework on which our contribution is based.

Throughout this paper we identify 1 with true and 0 with false. A literal \( \ell \) is either a Boolean variable \( x \) or its negation \( \overline{x} \). We say that the sign of \( x \) is positive and that of \( \overline{x} \) is negative. By a pseudo-Boolean (PB) constraint we mean a linear inequality over the domain \( \{0,1\} \) with integer coefficients and integral constant term. In the context of pseudo-Boolean solving it is natural to represent the constraints in normalized form \( \sum c_i \ell_i \geq w \), where \( \ell_i \) are literals over pair-wise distinct variables, \( c_i \) are positive integers, and \( w \) is a positive integer called the degree of falsity (or just degree). In what follows, by “constraint” we mean a pseudo-Boolean constraint in normalized form unless otherwise stated. We think of a partial truth value assignment \( \rho \) as the set of literals set to true by \( \rho \). Thus, \( \rho(\ell) = 1 \) if \( \ell \in \rho \), \( \rho(\ell) = 0 \) if \( \ell \in \rho \), and if the variable is unassigned we write \( \rho(\ell) = * \).

We first recap how conflict-driven clause learning (CDCL) works for CNF inputs. Note that the clauses \( C \) in a CNF formula \( F \) are just PB constraints with all coefficients and degree of falsity equal to 1. The main loop of the CDCL algorithm is presented in Algorithm 1. The solver maintains a partial assignment \( \rho \) to the variables, which it is trying to extend to a total assignment satisfying the formula. We identify this \( \rho \) with an ordered set, referred to as the trail, containing the literals

```
Algorithm 1: CDCL main loop
1. \( \rho \leftarrow \) empty trail \( \triangledown \) \( D \leftarrow F \)
2. until solved do
3. \( (\rho,C_{conf}) \leftarrow \) propagate(\( \rho \), \( D \))
4. if \( C_{conf} \neq \) NONE then
5. if decision level of \( \rho = 0 \) then
6. \( \triangledown \) output UNSAT and terminate
7. \( C_{learnt} \leftarrow \) analyzeConflict(\( C_{conf} \), \( \rho \))
8. Revert \( \rho \) to backtrack level of \( C_{learnt} \)
9. \( D \leftarrow D \cup \{C_{learnt}\} \)
10. else
11. if \( \rho \) is a total assignment then
12. \( \triangledown \) output SAT and terminate
13. if \( \) time to restart then
14. \( \) Backtrack \( \rho \) to level 0
15. if \( \) time for clause database reduction then
16. \( \) Erase (roughly) half of learned clauses in \( D \)
17. \( \ell \leftarrow \) a literal unassigned by \( \rho \)
18. \( \rho \leftarrow \) append(\( \rho \), \( \ell/d \))
```
Algorithm 2: propagate($\rho, D$)

1. while there exists a clause $C \in D$ and an unassigned literal $\ell \in C$ implied by $C$ under $\rho$ do
2. $\rho \leftarrow \text{append}(\rho, \ell/C)$
3. if there exists a clause $C \in D$ falsified by $\rho$ then
4. $\rho \leftarrow \text{removeLast}(\rho)$
5. return ($\rho$, NONE)

Algorithm 3: analyzeConflict($C_{\text{confi}}, \rho$)

1. while $C_{\text{confi}}$ is not asserting do
2. $\ell \leftarrow$ literal assigned last on the trail $\rho$
3. if $\ell$ occurs in $C_{\text{confi}}$ then
4. $\ell_{\text{reason}} \leftarrow \text{reason}(\ell, \rho)$
5. $C_{\text{confi}} \leftarrow \text{Res}(C_{\text{confi}}, C_{\text{reason}})$
6. $\rho \leftarrow \text{removeLast}(\rho)$
7. return $C_{\text{confi}}$

Annotated with additional information why the assignments were made as follows. We write $\ell/C$ to denote that the literal $\ell$ on the trail was propagated by the constraint $C$ to avoid immediately falsifying the formula (as explained below), and also use the notation $C = \text{reason}(\ell, \rho)$ to show that $C$ was the reason for this propagation of $\ell$ when the trail was $\rho$. We write $\ell/d$ when the assignment to $\ell$ was a freely made decision by the solver, in which case $\text{reason}(\ell, \rho) = \text{NONE}$. The decision level of $\ell$ in $\rho$, denoted $\text{level}(\ell, \rho)$, is the number of decisions up to (and possibly including) $\ell$.

There are two subroutines, unit propagation (Algorithm 2) and conflict analysis (Algorithm 3), which we review next. In unit propagation, the solver repeatedly finds an unassigned literal which is implied by some clause and adds this literal to the trail. A literal $\ell$ in a clause $C$ is implied, or propagated, by $C$ under $\rho$ if all other literals in $C$ are falsified by $\rho$, meaning that $\rho$ has to be satisfied to avoid immediate contradiction. Unit propagation can terminate in two ways: either there are no further implications, or a clause is falsified. The unit propagation method returns the extended trail, plus potentially the falsified clause, which is called the conflict clause. The solver then calls conflict analysis if there was a falsified clause and makes a new decision otherwise.

The purpose of conflict analysis is to learn a new clause that explains why the current partial assignment failed, and then add this clause so that the solver can avoid exploring infeasible parts of the search space multiple times. Conflict analysis is typically done using the first unique implication point (1UIP) learning scheme. The solver starts with the conflict clause and the last reason clause in chronological order propagating a literal in the conflict clause to false, and resolves these two clauses using the resolution rule: given two clauses $C_1 \lor x$ and $D_1 \lor \neg x$ containing a unique variable $x$ with opposite signs, derive the resolvent $\text{Res}(C_1 \lor x, D_1 \lor \neg x) = C' \lor D$. Then this resolvent is in turn resolved with the reason clause propagating the last literal in the resolvent to false, etcetera. The key invariant in this algorithm is that at the end of each iteration of the loop, the new intermediate conflict clause $C_{\text{confi}}$ is falsified by what remains of the trail $\rho$ when the last literal is removed. 1UIP conflict analysis terminates the first time $C_{\text{confi}}$ contains a single literal $\ell'$ at the last decision level. This clause is then declared to be the learnt clause and is added to the clause database containing the formula $F$ as well as previously learnt clauses. Next the assertion level is calculated, which is the second highest decision level represented in the learnt clause. The trail will then be backtracked to this level, meaning that all assignments at higher levels are removed. It is not hard to see that at this point $C_{\text{confi}}$ will propagate $\ell'$ to true, i.e., flipping the assignment at the time of conflict. Clauses having the property that they cause propagation after backtracking are called asserting.

What we have presented above is a schematic, and somewhat simplified, description of CDCL search, with an emphasis on the conflict analysis that we want to extend to a pseudo-Boolean setting. We do not dwell on other important aspects, such as restart policy or clause database erase, since they are not the main focus of this work.

Let us now discuss how to generalize conflict-driven search from clauses to PB constraints. The main loop in Algorithm 1 remains the same. To generalize unit propagation, we need the notion of slack of a constraint $C \equiv \sum_i c_i \ell_i \geq w$, which is a measure of how close a partial assignment $\rho$ is to falsifying $C$. It is defined as the difference between the maximum value the sum $\sum_i c_i \ell_i$ may attain given the assignments already made in $\rho$ and the degree of falsity $w$, i.e.,

$$\text{slack}(\rho, C, \rho) = \sum_i \rho(\ell_i) \neq 0 c_i - w.$$ (1)

The constraint $C$ is falsified by $\rho$ if the slack is negative, and $C$ implies or propagates a literal $\ell_i$ under $\rho$ if $\rho(\ell_i) = \ast$ and $\text{slack}(\rho, C) < c_i$, meaning that unless $\ell_i$ is set to true the constraint $C$ will be falsified. Note that this generalizes the notion of propagating disjunctive clauses, which have slack 0 and coefficient 1 for the propagated literal. Unit propagation as in Algorithm 2 can now be done by calculating the slack of the constraints. One important problem is that when $C$ is a general constraint and not a disjunctive clause it seems harder to detect efficiently whether $C$ is propagating, but we will not discuss this further in this paper.

To generalize 1UIP conflict analysis, we would like to “resolve” a falsified constraint with a sequence of reason constraints in such a way that the final constraint is asserting. The natural extension of the resolution rule to general PB constraints is called generalized resolution [Hooker, 1992; Dixon et al., 2004] and is defined as follows. Given constraints $C \equiv a \ell + \sum_i c_i \ell_i \geq w$ and $C' \equiv b \ell' + \sum_i c_i' \ell'_i \geq w'$ containing some literal $\ell$ with opposite signs, let $g = \gcd(a, b)$. Then the (generalized) resolvent $\text{Res}(C, C', \ell)$ is

$$\frac{b}{g} \sum_i c_i \ell_i + \frac{a}{g} \sum_i c_i' \ell'_i \geq \frac{bw + aw' - ab}{g}$$ (2)

(where we implicitly assume that this constraint, as well as the result of any other operation performed below, is again reduced to normalized form). Note that in contrast to resolution applied on clauses we have to give a third argument specifying which literal should be cancelled, since there may be several literals with opposite signs. As an example, for the
Algorithm 4: Reduction by weakening and saturation.

1 while \( \text{slack}(\text{Res}(\text{Cconf}, \text{Creason}, \ell), \rho) \geq 0 \) do
2     \( \ell' \leftarrow \) a literal in \( C_{\text{reason}} \) not falsified by \( \rho \), \( \ell' \neq \ell \)
3     \( \text{Creason} \leftarrow \text{saturate}(\text{weaken}(\text{Creason}, \ell')) \)
4 return \( C_{\text{reason}} \)

Constraints \( C \triangleq 2x_1 + 2x_2 + x_3 \geq 3 \) and \( C' \triangleq x_1 + x_3 \geq 1 \) the resolvent over \( x_1 \) is \( \text{Res}(C, C', x_1) \triangleq 2x_2 + 4x_3 \geq 3 \).

However, when we try to perform conflict analysis as in Algorithm 3 we run into trouble. Recall that the key invariant during CDCL conflict analysis is that the currently derived constraint is falsified by what remains of the trail, meaning that at all times the derived constraint “explains” the conflict. This property no longer holds when we use generalized resolution, as shown in the next example.

Example 2.1 (Failed conflict analysis using generalized resolution). Consider a PB instance consisting of just \( C \triangleq 2x_1 + 3x_2 + x_3 \geq 3 \) and \( C' \triangleq 4x_2 + 4x_3 + 2x_4 + x_5 \geq 4 \). Suppose the solver makes decisions \( x_1 \), after which nothing propagates, and then \( x_2 \). At this point \( C \) propagates \( x_2 \), which in turn falsifies \( C' \), triggering conflict analysis. Using resolution, the solver would derive \( \text{Res}(C, C', x_2) \triangleq 8x_3 + 16x_4 + 5x_5 \geq 12 \). But this constraint is not falsified by the assignment \( \{x_3, x_5\} \) remaining on the trail.

In view of this example, generalized resolution on its own is not sufficient for conflict analysis. [Chai and Kuehlmann, 2005] solve this problem by designing a reduction algorithm that computes a constraint that is implied by the original reason constraint and for which generalized resolution maintains the key invariant when the reason is replaced by this new constraint. This algorithm uses two additional operations called weakening and saturation, which we describe next.

The weakening operation removes a literal from a constraint and subtracts its coefficient from the degree. For example, weakening the constraint \( C \triangleq x_1 + 2x_2 + 3x_3 \geq 4 \) on \( x_1 \) yields \( \text{weaken}(C, x_1) \triangleq 2x_2 + 3x_3 \geq 3 \). This is sound, because weakening on a literal \( \ell \) is equivalent to adding the constraint \( \ell \geq 0 \) multiplied by the appropriate coefficient. Saturation applied to a constraint \( C' \triangleq \sum_i c_i \ell_i \geq w \) returns \( \text{saturate}(C') \triangleq \sum_i \min(c_i, w) \cdot \ell_i \geq w \), i.e., it brings the magnitude of all coefficients down to the degree of falsity. As a concrete example, saturating the constraint \( x_1 + 2x_2 + 3x_3 \geq 2 \) gives \( x_1 + 2x_2 + 2x_3 \geq 2 \). This is sound because all literals take non-negative integral values, and if \( \sum_i c_i \ell_i \geq w \) is satisfied by setting some literal \( \ell_i \) to 1 for which \( c_i > w \), then clearly the contribution from \( w \cdot \ell_i \) is also sufficient to satisfy the constraint.

The reduction method in [Chai and Kuehlmann, 2005] is presented as Algorithm 4 and works by repeating the following steps. First, it tests whether the generalized resolvent of the reason constraint and the conflict constraint satisfies the key invariant. If this is the case, then the reason can be used as is and the algorithm terminates. Otherwise, it finds a non-falsified literal in the reason constraint different from the propagating literal, weakens the constraint on this literal, applies saturation, and continues with the next iteration.

Let us illustrate by a simple example how a single step in the conflict analysis of [Chai and Kuehlmann, 2005] works.

Example 2.2 (Illustration of Chai–Kuehlmann reduction). Suppose we have a pseudo-Boolean instance consisting of the two constraints \( C \triangleq 2x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 \geq 6 \) and \( C' \triangleq 2x_1 + 2x_2 + 2x_3 + 2x_4 \geq 4 \). Clearly, \( C \) and \( C' \) are just obfuscated encodings of the mutually contradictory cardinality constraints \( x_1 + x_2 + x_3 + x_4 + x_5 \geq 3 \) and \( x_1 + x_2 + x_3 + x_4 \geq 2 \), respectively. However, as we shall see reduction by saturation does not allow us to derive contradiction immediately.

Suppose the trail is \( \rho = \{x_1/d, x_2/C, x_3/C, x_4/C\} \). The constraint \( C' \) is falsified and \( C \) is the reason (\( C_{\text{conf}} \) and \( C_{\text{reason}} \) in Algorithm 4, respectively), and the variable to resolve over is the one that is last on the trail, i.e., \( x_4 \). We have \( \text{Res}(C', C, x_4) = x_5 \geq 1 \), which has non-negative slack 0, so the reason constraint must be reduced. The algorithm chooses a non-falsified literal different from \( x_4 \), say \( x_2 \), and weakens \( C \) to obtain \( 2x_1 + 2x_2 + x_4 + x_5 \geq 4 \). Applying saturation has no effect. The resolvent \( \text{Res}(C', 2x_1 + 2x_3 + x_4 + x_5 \geq 4, x_4) = 2x_2 + x_5 \geq 1 \) has slack 0, triggering further reduction. Say the next variable chosen is \( x_3 \). Weakening gives \( 2x_1 + 2x_3 + x_5 \geq 2 \). Saturation again changes nothing, and \( \text{Res}(C', 2x_1 + 2x_3 + x_5 \geq 2, x_4) \triangleq 2x_2 + x_5 \geq 1 \) again has slack 0. Finally, weakening on \( x_5 \) yields \( 2x_1 + x_4 \geq 1 \), after which saturation returns \( x_1 + x_4 \geq 1 \). Now the resolvent \( \text{Res}(C', x_1 + x_4 \geq 1, x_4) \triangleq 2x_2 + 2\geq 1 \) has negative slack -1, so we are done.

Because the slack of the resolvent is too expensive to compute manually after each weakening and saturation step of the reason constraint, an upper bound on the slack is used in practice. This upper bound is based on the subadditivity property of slack, which we state without proof.

Fact 2.3 (Slack is subadditive). For any pseudo-Boolean constraints \( C \) and \( C' \) and any partial assignment \( \rho \) it holds that \( \text{slack}(C + C', \rho) \leq \text{slack}(C, \rho) + \text{slack}(C', \rho) \).

In particular, this means that if for \( C \triangleq a \ell + \sum_i c_i \ell_i \geq w \) and \( C' \triangleq b \ell + \sum_i c_i' \ell_i \geq w' \) we let \( D = \text{Res}(C, C', \ell') \) and \( g = \gcd(a, b) \), then we have

\[
\text{slack}(D, \rho) \leq (b \cdot \text{slack}(C, \rho) + a \cdot \text{slack}(C', \rho))/g .
\]

(The advantage of using this estimate is that we do not need to compute the resolvent in each iteration, but instead it is sufficient to know the slacks of the conflict constraint (which remains constant) and the reduced reason constraint.

3 PB Conflict Analysis Using Division

We now proceed to describe our new method for conflict analysis, which we call rounding. The idea is to reduce the reason constraint into suitable form using division instead of saturation, where the result of dividing the constraint \( C \triangleq \sum_i c_i \ell_i \geq w \) by a positive integer \( d \) is

\[
\text{divide}(C, d) \triangleq \sum_i \lceil c_i/d \rceil \ell_i \geq \lceil w/d \rceil
\]

(which is a sound derivation since all numbers involved are non-negative integers). It is worth noting that this division rule (rather than saturation) is what was used to define the cutting planes method in [Cook et al., 1987].
As presented in Algorithm 5, our reduction method roundToOne\((C, \ell, \rho)\) takes a constraint \(C\), a literal \(\ell\) and a partial assignment \(\rho\) and outputs a constraint \(C'\) rounded on \(\ell\) that is a consequence of \(C\), has coefficient 1 for \(\ell\), and satisfies an additional property (to be discussed further) that makes it a suitable reason reduction algorithm.

To reduce the reason roundToOne is called with \(C_{\text{reason}} = \text{reason}(\ell, \rho)\) together with \(\rho\) and \(\ell\). The algorithm weakens away all literals not falsified by \(\rho\) and with coefficients not divisible by the coefficient \(c\) of \(\ell\) in \(C_{\text{reason}}\), and then divides by \(c\). In this case, the additional property of the output constraint is that it still implies \(\ell\). Equivalently, the slack is 0, and this implies that the resolvent in conflict analysis is guaranteed to be falsified, so that our desired invariant in conflict analysis holds. The reason for this is that slack is subadditive (see Fact 2.3) and we are resolving a reduced reason constraint with slack 0 and the previous intermediate conflict constraint, which has negative slack. Hence, the new intermediate conflict constraint also has negative slack as required. Let us formalize and prove these claims.

**Proposition 3.1.** For any trail \(\rho\) and any PB constraint \(C = \sum c_i \ell_i \geq w\) containing the literal \(\ell_i\) it holds that slack\((\text{roundToOne}(C, \ell_i, \rho), \rho)\) = \(\text{slack}(C, \rho) / c_i\).

**Proof.** It is straightforward to verify directly from Eq. (1) that weakening on non-falsified literals does not change the slack. After the weakening loop, the remaining non-falsified literals are all divisible by the coefficient \(c_i\) of the literal \(\ell_i\). Therefore, if we would not take the ceiling of the degree of \(C_{\text{reason}}\) but do exact division, then the (possibly rational) slack would be exactly \(1/c_i\) times the original slack. Because we round up the degree after division, the slack is \(\lfloor \text{slack}(C, \rho) / c_i \rfloor\).

**Corollary 3.2.** If \(C = \sum c_i \ell_i \geq w\) is a non-falsified constraint that propagates \(\ell_i\) under \(\rho\), i.e., \(0 \leq \text{slack}(C, \rho) / c_i\), then slack\((\text{roundToOne}(C, \ell_i, \rho), \rho)\) = \(0\). If instead \(\text{slack}(C, \rho) < 0\), then slack\((\text{roundToOne}(C, \ell_i, \rho), \rho)\) = \(0\), i.e., if \(C\) is falsified under \(\rho\), then so is roundToOne\((C, \ell_i, \rho)\).

**Example 3.3 (Illustration of roundToOne).** To see how our new reduction method works, consider the same PB instance and trail as in Example 2.2. Recall that the trail is \(\rho = \{x_1d, x_2/\ell, x_3/C, x_4/C, x_5/C\}\) and that the reason is \(C = 2x_1 + 2x_2 + 2x_3 + 2x_4 + 5 \geq 6\) to be resolved over \(x_4\) (the conflict constraint is irrelevant for this example). Our method first sets the divisor to be the coefficient of \(x_4\), which equals 2 (line 1 in Algorithm 5). Then it loops over all literals in the constraint, and weakens away those that are not falsified by \(\rho\) and have coefficient not divisible by 2 (lines 2–4). Literals \(x_2, \ldots, x_3\) are not falsified, and of these \(x_5\) has coefficient not divisible by 2, so \(x_5\) gets weakened away. When the loop ends, the constraint \(C\) equals \(2x_1 + 2x_2 + 2x_3 + 2x_4 + 5 \geq 5\). Finally, division by 2 (line 5) yields \(x_1 + x_2 + x_3 + x_4 \geq 3\). Thus, we see that we get a stronger constraint than in reduction by saturation, which gives \(x_1 + x_2 \geq 1\).

**Remark 3.4 (Variations on roundToOne).** Let us discuss some possible variants of the reduction algorithm. Consider the constraint \(C = 2x_1 + 3x_2 + 3x_3 + 3x_4 \geq 8\), to be rounded over \(x_1\), together with the assignment \(\rho = \{x_1, x_2, x_3, x_4\}\), for which roundToOne returns \(x_1 + x_3 \geq 1\). If instead of doing all literal weakenings in one go we would let the algorithm weaken on one literal at a time and terminate as soon as the slack after division is 0, then we could have obtained the stronger constraint \(x_1 + 2x_2 + 2x_4 \geq 3\). However, it can be shown that regardless of the order in which weakenings are carried out, this iterative approach will always weaken almost as many literals as roundToOne. More precisely, the difference is at most \(c - 1\) literals, where \(c\) is the divisor.

Another possibility would be to weaken partially on literals, e.g., deriving \(2x_1 + 2x_2 + 2x_3 + 3x_4 \geq 6\) from \(C\), after which division yields \(x_1 + x_2 + x_3 + 2x_4 \geq 3\). However, our empirical evaluations of this showed much worse results.

Besides rounding the reason, we employ roundToOne also in other places. First, we round the intermediate conflict constraint \(C_{\text{conflict}}\) before resolving it. By Corollary 3.2, this leaves \(C_{\text{conflict}}\) falsified, so the key invariant in conflict analysis still holds. The other point at which we call roundToOne is as a postprocessing step. Once we are done with conflict analysis, we find the literal \(\ell\) that was falsified last by \(\rho\) in the derived constraint, round \(\ell\), and declare the result to be the learnt constraint. The full conflict analysis is given in Algorithm 6.

There are some corner cases in PB conflict analysis that make it more complicated than CDCL. First, a resolution operation may cancel all falsified literals on the current decision level, in which case the analysis may continue to an earlier decision level. This is the reason for the condition “no or multiple falsified literals on the last level” on line 1 in Algorithm 6. It may even happen that conflict analysis is able to proceed to the topmost decision level, which means the instance is unsatisfiable. In this case, the solver terminates (lines 2–3). An-
other difference is that learnt constraints may be falsified at
their assertion level, in which case repeated conflict analysis
is performed before calling unit propagation.

**Remark 3.5 (Skipping resolution steps).** Another in-
teresting contrast to CDCL conflict analysis is that if
\(\text{slack}(C_{\text{conf}}, \rho)\) is very negative, then it might be possible
to ignore the resolution step with \(C_{\text{reason}}\) and just remove the
propagated literal \(\ell\) from the trail \(\rho\), still maintaining the
invariant \(\text{slack}(C_{\text{conf}}, \rho \setminus \{\ell\}) < 0\). Intuitively, it might seem
that this should lead to a more compact analysis involving
only the reasons truly needed for the conflict, and hence poten-
tially produce a better learnt constraint. When we evalu-
atied this empirically the results did not improve, however,
and so we ended up not using this idea, but it might still be an
interesting observation worthy of further exploration.

**Comparison to CutSat** The idea to use division during con-
flict analysis appears also in the general integer linear pro-
gramming (ILP) solver CutSat [Jovanovic and de Moura, 2013], so let us try to compare and contrast this solver to
RoundingSat. ILP constraints are not normalized but instead have
the form \(C = \sum c_i x_i \geq w\), where \(c_i\) can be negative. In
CutSat, the trail consists of bound refinements \(x \leq c \) or
\(x \geq c\). (In the PB case, the trail contains at the very start the
bounds \(x_i \geq 0\) and \(x_i \leq 1\) for all variables \(x_i\), and further refi-
nements \(x_i \geq 1\) or \(x_i \leq 0\) correspond to assignments.) Each
refinement has a reason constraint or is a decision. Decisions
always fix variable values, so if the currently best bound is
\(x \geq L (x \leq U)\), then the decision is \(x = L (x = U)\).

CutSat conflict analysis also works by resolving a conflict constraint with a sequence of reduced reason constraints. The reduction method outputs tightly propagating (or tight) con-
straints, which are on the form \(bx + \sum c_i x_i \geq w\), where \(x\) is the
refinement variable and \(b \in \{-1, 1\}\). The reduced reason
implies the same bound refinement on \(x\) as the original one with respect to the trail. (In the PB setting this means that the
slack is 0.) CutSat reduction starts with the reason, derives a
constraint \(C'\) with all coefficients divisible by the coeffi-
cent \(c\) of the propagated literal, and returns divide\((C', c)\). This is similar to how RoundingSat works: it also starts with
the reason, performs weakening, and then divides.

To discuss CutSat reduction in more detail, assume for sim-
plcity that the reason constraint is a lower bound refinement
\(C = cx + \sum c_i x_i \geq w\) with all coefficients positive (for
negative coefficients the role of upper and lower bounds are just reversed). CutSat iterates through the variables of \(C\) in reverse chronological order with respect to the trail. If a variable
\(x_i\) has a coefficient \(c_i > 0\) not divisible by \(c\), then CutSat
adds a constraint associated with this variable to \(C\) in order to make the coefficient divisible. To do so CutSat looks up
why \(x_i\) is bounded from above (which must be the case, since
otherwise the reason constraint would not be propagating).

If the latest upper bound refinement for \(x_i\) is due to a reason propagating, then this reason is made tight by calling the rea-
son reduction algorithm recursively. Suppose this call returns
the new constraint \(C' = -x_i + \sum c'_{ij} x_j \geq w'\) (with coeffi-
cient \(-1\) for \(x_i\) since we have an upper bound constraint).
Then generalized resolution is performed on \(C\) and \(C'\) with
respect to \(x_i\). This eliminates one coefficient not divisible by
c, but may introduce new variables with non-divisible coeffi-
cients. All such variables are dealt with in the same way in
reverse chronological order with respect to the trail.

Suppose instead that the latest upper bound refinement for
\(x_i\) is a decision \(x_i \leq U_i\). Because decisions always fix
variable values, this means that a matching lower bound re-
finement forcing \(x_i \geq U_i\) must exist earlier on the trail. Then
this matching constraint is made tight and added to \(C\) with
the smallest multiplier making the coefficient of \(x_i\) divisible.

For PB instances this works as follows. Assume again that
the reason is a lower bound refinement \(cx + \sum c_i x_i \geq w\)
with all coefficients positive, and let \(c_i > 0\) be not divisible
by \(c > 0\). We get a case analysis depending on whether
\(x_i \leq 0\) appears on the trail or not. If not, then we resolve with
the inequality \(x_i \leq 1\) (corresponding to weakening in
RoundingSat). If \(x_i \leq 0\) is on the trail, then we resolve by the tight
reduced constraint obtained from the reason that \(x_i \leq 0\) if
this reason exists. Otherwise, \(x_i \leq 0\) is a decision and we add the tight constraint \(x_i \geq 0\) with the smallest multiple so
that the coefficient becomes divisible by \(c\). This is equivalent
to rounding up the coefficient during a division step.

When comparing the two methods, we would argue that
RoundingSat seems clearly simpler to understand and imple-
ment (though this is also thanks to the fact that we need not worry about general ILP constraints but can focus on PB con-
straints). Our method is also more efficient in the worst case,
since no recursive calls are made and the conflict analysis
does not have to go as far back on the trail. While we be-
lieve that these aspects also make RoundingSat significantly
more efficient not only in the worst case but also in practice,
we should add the caveat that we have not performed a thor-
ough empirical evaluation of how CutSat and RoundingSat
compare on, e.g., PB competition benchmarks.

**4 Experimental Analysis**

In this section we report results from an experimental com-
parison of RoundingSat to Sat4j and Open-WBO. Recall that
Open-WBO is a resolution-based solver that does an eager
conversion to clauses and then runs a state-of-the-art CDCL
solver. From the Sat4j solver library we test two versions: a
resolution-based solver Sat4j Res, doing lazy conversion
to clauses, and a cutting planes-based solver Sat4j Res+CP,
which, however, also runs the resolution-based solver in par-
allel since the cutting planes-based solver on its own is not
considered competitive enough. In our experiments we gave
all solvers the same wall-clock time, which means that Sat4j
Res+CP got twice as much CPU time as the other solvers.

We evaluated the solvers on benchmarks submitted to the
Pseudo-Boolean Competitions over the past two decades in the
dec-SMALLINT-LIN category (decision problems, small integers, linear constraints) since RoundingSat is
currently designed to solve decision problems and repre-
seats constraints using 32-bit integers. Our choice of
Sat4j and Open-WBO was dictated by the results in the
dec-SMALLINT-LIN category from the Pseudo-Boolean
Competition 2016, where these two solvers together with our
solver were the top-three performers. We used the latest avail-
The results are presented in Table 1 (summary for all benchmarks) and Table 2 (detailed results per family in the large benchmark set PB16 elffers). On satisfiable benchmarks we see that RoundingSat is typically doing equally well as the other solvers, and the differences between solvers are relatively small. The unsatisfiable benchmarks differentiate between the solvers much more. We can divide the benchmark sets into two categories: those where pseudo-Boolean reasoning is crucial and those where resolution-based solvers are equally good. The benchmarks for which PB reasoning matters are PB05 aloul, PB06 roussel, and PB16 elffers. In the case where PB reasoning does not matter, we see no large differences between the solvers, except that on the PB16 quimper set Open-WBO solves 12% more unsatisfiable instances than the best of the other solvers. The cases where pseudo-Boolean reasoning matters are relatively rare. PB05 aloul and PB06 roussel are mainly pigeonhole principles encoded with cardinality constraints, which are easy to solve for solvers that can just add up linear inequalities but are exponentially hard for resolution when encoded into CNF, and so it is not surprising that RoundingSat and Sat4j Res+CP do much better than the two resolution-based solvers here.

The benchmark set PB16 elffers consists of various crafted instances. Subset cardinality formulas encode that we cannot have a majority of true and false variables simultaneously if the total number of variables is odd. For resolution these formulas are exponentially hard when generated from random matrices and easy in theory but challenging in practice when based on fixed bandwidth matrices, but are always easy for cardinality reasoning. This explains the difference in performance between RoundingSat and Open-WBO. Vertex cover formulas claim the existence of a too small vertex cover for a rectangular, toroidal grid graph. These instances have fractional solutions, however, meaning that proper pseudo-Boolean reasoning is needed (as opposed to for pigeonhole principle and subset cardinality formulas, where linear programming is enough). Perhaps somewhat surprisingly, Open-WBO performs very well although pseudo-Boolean reasoning should be crucial. The reason seems to be that since the input is presented in a helpful order, the CNF encoding of the cardinality constraint for the vertex cover size provides extension variables that allow Open-WBO to perform the required counting argument in resolution. RoundingSat also solves these instances ef-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RoundingSat</th>
<th>Sat4j Res+CP</th>
<th>Sat4j Res</th>
<th>Open-WBO</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB05 aloul</td>
<td>36 + 21</td>
<td>36 + 21</td>
<td>36 + 3</td>
<td>36 + 6</td>
<td>57</td>
</tr>
<tr>
<td>PB06 manquiho</td>
<td>14 + 0</td>
<td>14 + 0</td>
<td>14 + 0</td>
<td>3 + 0</td>
<td>14</td>
</tr>
<tr>
<td>PB06 ppp-problems</td>
<td>4 + 0</td>
<td>4 + 0</td>
<td>4 + 0</td>
<td>3 + 0</td>
<td>6</td>
</tr>
<tr>
<td>PB06 uclid</td>
<td>1 + 47</td>
<td>1 + 47</td>
<td>1 + 47</td>
<td>1 + 49</td>
<td>50</td>
</tr>
<tr>
<td>PB06 liu</td>
<td>16 + 0</td>
<td>16 + 0</td>
<td>16 + 0</td>
<td>17 + 0</td>
<td>20</td>
</tr>
<tr>
<td>PB06 namasivayam</td>
<td>72 + 128</td>
<td>72 + 128</td>
<td>72 + 128</td>
<td>72 + 128</td>
<td>200</td>
</tr>
<tr>
<td>PB06 prestwich</td>
<td>10 + 0</td>
<td>11 + 0</td>
<td>9 + 0</td>
<td>14 + 0</td>
<td>18</td>
</tr>
<tr>
<td>PB06 roussel</td>
<td>0 + 22</td>
<td>0 + 22</td>
<td>0 + 4</td>
<td>0 + 4</td>
<td>40</td>
</tr>
<tr>
<td>PB10 oliveras</td>
<td>34 + 32</td>
<td>34 + 32</td>
<td>34 + 33</td>
<td>34 + 33</td>
<td>80</td>
</tr>
<tr>
<td>PB11 heinz</td>
<td>2 + 0</td>
<td>2 + 0</td>
<td>2 + 0</td>
<td>2 + 0</td>
<td>4</td>
</tr>
<tr>
<td>PB11 lopes</td>
<td>42 + 26</td>
<td>37 + 25</td>
<td>37 + 25</td>
<td>33 + 28</td>
<td>193</td>
</tr>
<tr>
<td>PB12 sroussel</td>
<td>31 + 0</td>
<td>21 + 0</td>
<td>23 + 0</td>
<td>29 + 1</td>
<td>122</td>
</tr>
<tr>
<td>PB16 elffers</td>
<td>0 + 287</td>
<td>0 + 229</td>
<td>0 + 142</td>
<td>0 + 213</td>
<td>293</td>
</tr>
<tr>
<td>PB16 nossum</td>
<td>68 + 0</td>
<td>39 + 0</td>
<td>39 + 0</td>
<td>55 + 0</td>
<td>180</td>
</tr>
<tr>
<td>PB16 quimper</td>
<td>43 + 214</td>
<td>43 + 213</td>
<td>43 + 213</td>
<td>46 + 241</td>
<td>304</td>
</tr>
<tr>
<td>Sum</td>
<td>373 + 777</td>
<td>330 + 717</td>
<td>330 + 595</td>
<td>345 + 703</td>
<td>1581</td>
</tr>
</tbody>
</table>

Table 1: Results on benchmarks from past PB competitions (number of solved satisfiable instances + solved unsatisfiable instances).

<table>
<thead>
<tr>
<th>Family</th>
<th>RoundingSat</th>
<th>Sat4j Res+CP</th>
<th>Sat4j Res</th>
<th>Open-WBO</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex cover, rectangular grids</td>
<td>82</td>
<td>68</td>
<td>26</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>Subset cardinality, fixed bandwidth</td>
<td>28</td>
<td>28</td>
<td>4</td>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>Subset cardinality, random matrices</td>
<td>28</td>
<td>28</td>
<td>6</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>Perfect matching, random graphs</td>
<td>33</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>33</td>
</tr>
<tr>
<td>Even colouring, odd-dimension grids</td>
<td>25</td>
<td>17</td>
<td>20</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Even colouring, random graphs</td>
<td>22</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 2: Detailed results for some families in the benchmark set PB16 elffers (number of solved instances; all are unsatisfiable).
Table 3: Average number of conflicts per second by family. A dash denotes that all instances were solved within 1 second. An asterisk denotes that RoundingSat needs a much lower number of conflicts to refute the formula than resolution-based solvers.

<table>
<thead>
<tr>
<th>Benchmark Family</th>
<th>RoundingSat</th>
<th>Sat4j CP</th>
<th>Sat4j Res</th>
<th>Open-WBO</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB05 aloul*</td>
<td>–</td>
<td>79</td>
<td>2996</td>
<td>4538</td>
</tr>
<tr>
<td>PB06 manquiro</td>
<td>–</td>
<td>294</td>
<td>–</td>
<td>4</td>
</tr>
<tr>
<td>PB06 ppp-problems</td>
<td>3128</td>
<td>24</td>
<td>4409</td>
<td>4770</td>
</tr>
<tr>
<td>PB06 ulcid</td>
<td>2050</td>
<td>144</td>
<td>1211</td>
<td>948</td>
</tr>
<tr>
<td>PB06 liu</td>
<td>747</td>
<td>35</td>
<td>442</td>
<td>1951</td>
</tr>
<tr>
<td>PB06 namasivayam</td>
<td>14702</td>
<td>22</td>
<td>16239</td>
<td>13251</td>
</tr>
<tr>
<td>PB06 prestwich</td>
<td>1263</td>
<td>56</td>
<td>2830</td>
<td>6228</td>
</tr>
<tr>
<td>PB06 roussel</td>
<td>2076</td>
<td>237</td>
<td>1740</td>
<td>4762</td>
</tr>
<tr>
<td>PB10 oliveras</td>
<td>3069</td>
<td>34</td>
<td>1144</td>
<td>844</td>
</tr>
<tr>
<td>PB11 heinz</td>
<td>2191</td>
<td>85</td>
<td>2698</td>
<td>2882</td>
</tr>
<tr>
<td>PB11 lopes</td>
<td>3433</td>
<td>92</td>
<td>2452</td>
<td>1928</td>
</tr>
<tr>
<td>PB12 sroussel</td>
<td>1024</td>
<td>6</td>
<td>1581</td>
<td>1546</td>
</tr>
<tr>
<td>PB16 nossum</td>
<td>2832</td>
<td>14</td>
<td>3287</td>
<td>2223</td>
</tr>
<tr>
<td>PB16 quimper</td>
<td>4380</td>
<td>101</td>
<td>4206</td>
<td>9542</td>
</tr>
<tr>
<td>PB16 offers</td>
<td>Vertex cover, rectangular grids</td>
<td>6607</td>
<td>177</td>
<td>5412</td>
</tr>
<tr>
<td></td>
<td>Subset cardinality, fixed bandwidth*</td>
<td>–</td>
<td>19</td>
<td>3462</td>
</tr>
<tr>
<td></td>
<td>Subset cardinality, random matrices*</td>
<td>–</td>
<td>11</td>
<td>3623</td>
</tr>
<tr>
<td></td>
<td>Perfect matching, random graphs*</td>
<td>5806</td>
<td>98</td>
<td>2299</td>
</tr>
<tr>
<td></td>
<td>Even colouring, odd-dimension grids</td>
<td>21293</td>
<td>502</td>
<td>12835</td>
</tr>
<tr>
<td></td>
<td>Even colouring, random graphs*</td>
<td>10268</td>
<td>152</td>
<td>3391</td>
</tr>
</tbody>
</table>

Table 3: Average number of conflicts per second by family. A dash denotes that all instances were solved within 1 second. An asterisk denotes that RoundingSat needs a much lower number of conflicts to refute the formula than resolution-based solvers.

ficiently using native PB reasoning in terms of the original variables. Sat4j Res+CP solves most of the instances, though not the full set, whereas the approach in Sat4j Res with lazy clause generation plus resolution works very poorly. The last two benchmark sets, perfect matching and even colouring formulas, are both special cases of problems on the format $\sum_{i \in I} x_i = w_j, j \in [m]$, where every variable $x_i$ occurs in an even number of equations but $\sum_{j=1}^m w_j$ is odd. These formulas again require pseudo-Boolean reasoning, except when the underlying graph is simple enough to admit an efficient resolution-based approach (as for the instances generated from odd-dimension grids). These instances based on even-odd counting can be efficiently refuted in the cutting planes proof system provided that division is used, and indeed RoundingSat performs well. On random graphs these formulas seem likely to be exponentially hard for resolution, and so it is not surprising that Open-WBO is so bad. For Sat4j we believe that the poor performance is due to precisely the fact that this solver uses saturation rather than division.

Overall, we see that RoundingSat is almost always equally good as the best of the other solvers (with the possible exceptions of PB06 prestwich and PB16 quimper), and for several benchmark families it is markedly better. The differences are particularly stark for many of the crafted unsatisfiable instances in PB16 offers, which have been designed to be very easy for pseudo-Boolean reasoning but where RoundingSat often is the only solver able to prove unsatisfiability quickly.

Besides number of solved instances, we have also studied search speed measured as the number of conflicts per second\(^4\) (see Table 3, where we discard instances solved in less than 1 second). A couple of remarks are in order here. First, the number of conflicts per second depends on how many constraints the solver keeps in the database: more aggressive deletion gives a higher speed. We do not know how the solvers compare in this aspect. Second, RoundingSat solves certain formulas in just a few seconds while the resolution-based solvers time out. The conflict speed decreases over time as the constraint database size increases, so the comparison is then not entirely fair. The cases where we know that this happens are marked with an asterisk in the table.

For all benchmark families RoundingSat reaches a conflict speed that is one or several orders of magnitude higher than Sat4j CuttingPlanes, the other solver doing PB reasoning. Comparing RoundingSat to the CDCL-based solvers, for benchmarks where pseudo-Boolean reasoning does not help we see that RoundingSat is usually at most 3 times slower.

5 Concluding Remarks

In this paper we present a new approach to pseudo-Boolean solving, building on the conflict-driven search in [Chai and Kuehlmann, 2005] but using division instead of saturation. We report results from implementing these ideas in the solver RoundingSat and evaluating against other state-of-the-art pseudo-Boolean solvers. Our conclusions are that the

\(^4\)One issue here is that Open-WBO does not output this statistic, so for these particular measurements we had to hack the solver slightly. The values obtained are not perfect, but should be within a 1% error for all benchmarks except those in PB06 manquiro.
proof search is indeed stronger when pseudo-Boolean reasoning is essential, and that the aggressive use of division also helps to keep coefficient sizes down, speeding up the arithmetic operations. Although RoundingSat cannot compete with state-of-the-art solvers in terms of raw search speed, it is approaching CDCL-based methods measured in terms of number of conflicts per second. Our solver performs very well over all on the benchmarks tested. It remains competitive with resolution-based solvers when pseudo-Boolean reasoning does not really seem to pay off, and it can outperform CDCL on instances where more sophisticated reasoning is crucial. It also seems superior to the pseudo-Boolean solving approach with saturation instead of division.

An obvious direction for future research is to improve our solver further. As other pseudo-Boolean solvers, RoundingSat is sensitive to the input format in that it performs quite poorly on CNF formulas. A way to address this would be to implement rules that can rewrite CNF to more efficient linear constraints when possible. In particular, an important challenge is to do cardinality detection for sets of clauses encoding cardinality constraints. Cardinality detection was considered already in [Dixon and Ginsberg, 2002] (in the form of constraint strengthening) and was proposed again as a preprocessing step in [Biere et al., 2014]. It works well for certain crafted instances, but is too costly to implement in general when there is no guarantee that cardinality constraints will be found. Instead, we would like to implement lazy cardinality detection during proof search, where the price is paid only when there are strong indications that this will pay off.

Acknowledgements

We would like to thank Daniel Le Berre for many pleasant and enlightening discussions about pseudo-Boolean solving, and João Marques-Silva for helping us get an overview of relevant references. We are also grateful for the many detailed comments from the anonymous reviewers, which helped us to improve the manuscript substantially.

Our experiments were run on computational resources provided by the Swedish National Infrastructure for Computing (SNIC). The authors received support from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 279611. The second author was also funded by Swedish Research Council grants 621-2012-5645 and 2016-00782.

References


