A Cardinal Improvement to Pseudo-Boolean Solving

Jan Elffers, Jakob Nordström
1Lund University, 2University of Copenhagen, 3KTH Royal Institute of Technology
jan.elffers@cs.lth.se, jn@di.ku.dk

Abstract

Pseudo-Boolean solvers hold out the theoretical potential of exponential improvements over conflict-driven clause learning (CDCL) SAT solvers, but in practice perform very poorly if the input is given in the standard conjunctive normal form (CNF) format. We present a technique to remedy this problem by recovering cardinality constraints from CNF on the fly during search. This is done by collecting potential building blocks of cardinality constraints during propagation and combining these blocks during conflict analysis. Our implementation has a non-negligible but manageable overhead when detection is not successful, and yields significant gains for some SAT competition and crafted benchmarks for which pseudo-Boolean reasoning is stronger than CDCL. It also boosts performance for some native pseudo-Boolean formulas where this approach helps to improve learned constraints.

1 Introduction

The Boolean satisfiability problem (SAT) is one of the most formidable challenges—and impressive success stories—of computer science. Although SAT is an NP-complete problem (Cook 1971; Levin 1973), and is widely believed to be exponentially hard in the worst case, the last few decades have witnessed the emergence of highly efficient SAT solvers based on conflict-driven clause learning (CDCL) (Marques-Silva and Sakallah 1999), with further improvements in (Moskewicz et al. 2001) and later papers. Today, these solvers are routinely used to solve large-scale real-world problems in a wide range of application areas (see (Biere et al. 2009) for a comprehensive reference).

One shortcoming of CDCL is that it uses a fairly weak method of reasoning—namely resolution, for which exponential lower bounds are known even for simple combinatorial principles (Haken 1985; Urquhart 1987). Another weakness is that the input must be encoded in conjunctive normal form (CNF), which deprives the solver of the possibility to reason with more expressive higher-level constraints.

A natural strengthening of CNF is to allow pseudo-Boolean (PB) constraints, i.e., integral linear inequalities over Boolean variables, which gives a succinct way of encoding problems in many domains. CDCL solvers can then be applied to such problems by translating the input to CNF (as in, e.g., MiniSat+ (Eén and Sörensson 2006), Open-WBO (Martins, Manquinho, and Lynce 2014), and NaPS (Sakai and Nabeshima 2015)) or by keeping the pseudo-Boolean input but deriving new constraints in the form of disjunctive clauses (as in methods implemented in clasp (Gebser, Kaufmann, and Schaub 2012) and the Sat4j library (Le Berre and Parrain 2010)). Another approach, which is quite attractive from a theoretical point of view, is to go beyond resolution and try to harness the power of cutting planes (Cook, Couillard, and Turán 1987). The conflict-driven paradigm has been extended to this setting in pseudo-Boolean solvers like PRS (Dixon and Ginsberg 2002), Galena (Chai and Kuehlmann 2005), Pueblo (Sheini and Sakallah 2006), Sat4j (Le Berre and Parrain 2010), and RoundingSat (Elffers and Nordström 2018).

A critical flaw of current cutting-planes-based solvers, however, is that while this method of reasoning is exponentially more powerful than resolution in theory, and does not care much about whether the input is presented in CNF or pseudo-Boolean form, the algorithms actually used in implementations of such solvers make them collapse to CDCL as soon as the input is given in CNF (Hooker 1988; 1992; Vinyals et al. 2018). This is clearly a highly unsatisfactory state of affairs—all the more so since a user of the solver might not be aware of that a problem can be rewritten more efficiently using pseudo-Boolean constraints, or that such low-level details of the syntactic representation can make an exponential difference in performance.

Our goal is to address this problem in a principled way, by rewriting CNF when a more efficient representation is possible, while maintaining good overall performance when no obvious improvement can be made. More concretely, in this work we propose a way of recovering cardinality constraints, saying that at most \( k \) variables (or at least \( k \) variables) in some subset can be true, from CNF. Cardinality constraints are already enough to make pseudo-Boolean reasoning exponentially stronger than resolution, and have been argued to capture most of the improvement over CDCL in practice (Chai and Kuehlmann 2005).
Earlier work on recovering cardinality constraints, which we will refer to as cardinality detection, was done in (Ansótegui 2004; Ansótegui et al. 2007). Some SAT solvers in the SAT competition also include recovery of cardinality constraints in the preprocessing phase (Biere 2013), though in these solvers the cardinality constraints are only used in a variable elimination algorithm to check for unsatisfiability during preprocessing. To the best of our knowledge, the only previous systematic investigation of cardinality detection in the context of pseudo-Boolean reasoning is (Biere et al. 2014). There, two types of detection were proposed: syntactic and semantic. They are both performed during a preprocessing step, and the recovered cardinality constraints are added to the formula before starting the pseudo-Boolean search. The general idea is to discover building blocks in the form of short clauses, and then extend them as far as possible to larger cardinality constraints. As an example, consider the at-most-1 constraint \( x_1 + x_2 + x_3 + x_4 \leq 1 \), which can be encoded in CNF as \( S = \{ x_1 \lor x_2 \lor x_3 \lor x_4 \land x_1 \lor x_2 \lor x_3 \land x_2 \lor x_4 \land x_3 \land x_4 \} \). Cardinality detection starts from a single clause, say, \( x_1 + x_2 \leq 1 \) (which is the PB representation of \( x_1 \lor x_2 \)), extends this constraint by adding \( x_3 \) after detecting the clauses \( \{ x_1 \lor x_3, x_2 \lor x_3 \} \), and then also adds \( x_4 \) after finding the remaining clauses in \( S \). In syntactic cardinality detection, the building blocks are found by syntactic checks of the input formula. In the semantic approach they are found by probing, i.e., assigning a variable and studying what other assignments are propagated. For example, to detect all binary clauses containing \( x_1 \), one sets \( x_1 \) to true and collects literals \( \ell \) propagated to false—for all such literals we have a building block \( x_1 \lor \ell \) that could be part of an at-most-1 constraint. For \( k > 1 \) one can set \( k \)-tuples of literals \( \{ \ell_1, \ldots, \ell_k \} \) to true and gather building blocks \( \ell_1 \lor \cdots \lor \ell_k \lor \ell' \) for every literal \( \ell' \) propagated to false.

1.1 Our Contribution

Our approach differs from (Biere et al. 2014) in several key aspects. We do not perform cardinality detection during a separate preprocessing step, but continuously during the solver execution. We have no syntactic detection, which is the main focus of (Biere et al. 2014), but instead use a semantic technique. However, we do not use dedicated probing, which quickly becomes too inefficient as \( k \) grows. Instead, we analyze the directed acyclic graph of unit propagations constructed during the “forward phase” of the solver search algorithm, and find clauses by considering cuts in this graph. During the “backward phase” of conflict analysis, we check on-the-fly if the short clauses involved in the conflict can be strengthened to cardinality constraints using previously detected building blocks. Importantly, these modifications allow us to recover at-most-\( k \) constraints not only for \( k \in \{1, 2\} \) as in (Biere et al. 2014) but also for larger \( k \).

Furthermore, the fact that cardinality detection is performed throughout the search means that the solver can also discover and exploit cardinality constraints that were not there in the original input, but have arisen as a result of the constraints learned during search. Recovery of short clauses during CDCL search has been considered before (Han, Jin, and Someni 2011; Heule, Järvisalo, and Biere 2013). However, our work differs from these papers in that they detect only binary clauses whereas we find also longer (though still comparatively short) clauses, and in that we use the found clauses to form syntactic cardinality constraints.

We have implemented our cardinality detection algorithm inside the pseudo-Boolean solver RoundingSat (Elffers and Nordström 2018). Our main conclusion is that the overhead of our approach is not too high on average, while at the same time we are able to solve a number of formulas faster than other CDCL and PB solvers. In more detail, in the relevant decision track in the latest pseudo-Boolean competition our solver performs similarly to though somewhat worse than the baseline RoundingSat solver. On CNF formulas from the SAT competitions, CDCL solvers run much faster in general, as is to be expected. Nevertheless, there are a number of benchmarks where the extra effort of our approach pays off, and where our solver is the only solver that finishes within the time limit. Besides that, there are many examples of crafted benchmarks where the new cardinality detection solver is better than existing CDCL and pseudo-Boolean approaches, including the earlier mentioned PB solver with cardinality detection from (Biere et al. 2014). Interestingly, we have also found that cardinality detection improves performance dramatically for some pseudo-Boolean formulas which are very hard in practice for PB solvers such as RoundingSat although the encoding seems to be the most natural one and the solvers should be able to prove unsatisfiability very efficiently in principle. While our work certainly leaves room for further improvement, this suggests that on-the-fly cardinality detection could be a useful technique in improving pseudo-Boolean reasoning in general, in contrast to just recovering cardinality constraints encoded in an unhelpful manner as a set of clauses.

1.2 Organization of This Paper

After giving some very brief preliminaries in Section 2, we describe our method for cardinality detection in more detail in Section 3 and report results from experimental evaluations in Section 4. We end in Section 5 with a discussion of possible directions for future work.

2 Preliminaries

Throughout this paper we use the term pseudo-Boolean (PB) constraint to refer to a linear inequality \( \sum a_i \ell_i \geq A \) or \( \sum a_i \ell_i \leq A \), where \( a_i \) and \( A \) are integers and \( \ell_i \in \{ x_i, \bar{x}_i \} \) are positive or negative literals over Boolean variables related by \( x_i + \bar{x}_i = 1 \). This last equality can always be used to rewrite any PB constraint in so-called normalized form as a greater-than-or-equal constraint with all coefficients \( a_i \) non-negative, and so we can freely assume that this is the case in what follows. A cardinality constraint is a PB constraint with all \( a_i \in \{0,1\} \), where an at-most-\( k \)-constraint is of the form \( \sum \ell_i \leq k \) and an at-least-\( k \)-constraint looks like \( \sum \ell_i \geq k \). Semantically speaking, a disjunctive clause \( D = \ell_1 \lor \cdots \lor \ell_w \) is exactly the same as an at-least-1 constraint \( \ell_1 + \cdots + \ell_w \geq 1 \), and so we identify the two. We can also view \( D \) as a set of literals, and say that \( D \) is a sub-clause of \( D' \) if \( D \subseteq D' \) viewed as sets. We note that any
pseudo-Boolean constraint \( \sum a_i \ell_i \geq A \) with \( a_i, A \) positive can be rounded to a clause \( \sum \ell_i \geq 1 \) that is implied by the constraint.

Due to limited space we cannot go into too much detail regarding conflict-driven pseudo-Boolean solving, but here follows a condensed description of the notions that we will need. At all times, the solver maintains an ordered trail of literals \( \ell \) assigned to true, where every assignment is either a free decision or a unit propagation caused by some PB constraint \( C \) that would be violated if \( \ell \) were to be set to false; such a \( C \) is referred to as a reason for \( \ell \). The solver alternates between making decisions and exhaustively inferring all unit propagations that follow from each such decision until either a satisfying assignment is found or a conflict is reached in the form of a violated constraint. In the latter case, the solver switches to conflict analysis. Here the solver combines the conflicting constraint with the reason constraints responsible for the propagations falsifying it one by one in reverse chronological order using the so-called resolution rule, after first reducing every reason constraint to suitable form (this reduction step is the main difference between PB solving and CDCL). The conflict analysis ends when an assertive constraint has been derived that will automatically flip some literal assignment when the solver backtracks.

Typically, the solver spends the vast majority of its time on unit propagation, and so achieving fast propagation is crucial for good performance. Another interesting measure is the number of conflicts reached before the solver terminates, which can be viewed as an indicator of the quality of the search. We refer the reader to the survey chapter (Buss and Nordström 2020) for a more in-depth comparison of CDCL and pseudo-Boolean solving.

### 3 Cardinality Detection Method

As explained above, our cardinality detection algorithm consists of two phases. During the solver “forward phase” of decisions and unit propagation we derive short clauses, which we call building blocks, that could potentially be used to form cardinality constraints. During the “backward phase” of conflict analysis we try to obtain cardinality constraints from these building blocks. The second phase is fairly standard, but the first phase, and how everything fits together, is novel. In this section, we describe all of this in more detail and give some illustrating examples.

#### 3.1 Finding Cardinality Building BlockClauses

As a motivating example, consider again the at-most-1 constraint \( x_1 + x_2 + x_3 + x_4 \leq 1 \), encoded in clausal form as \( \{ x_i \lor \neg x_j \mid 1 \leq i < j \leq 4 \} \). In order to recover this cardinality constraint, we need to detect the clausal building blocks. If, say, \( x_1 \lor \neg x_4 \) is not part of the original formula syntactically, then we would hope that the solver would detect the implication \( x_1 \rightarrow \neg x_4 \lor x_2 \rightarrow x_1 \) during unit propagation. However, such an implication might be indirect: it could be that \( x_1 \) implies \( y \) and \( z \) through the clauses \( \neg y \lor x_1 \) and \( \neg z \lor x_1 \), which in turn imply \( \neg y \lor x_2 \lor x_1 \lor x_2 \). The solver needs to be able to discover also such indirect implications.

The sequence of decisions and unit propagations a solver makes during search can be described by a directed implication graph, where there is a vertex for every literal assigned to true, and for a propagated literal this vertex has incoming edges from the negations of all other literals in the clause that propagated it (these negated literals have also been set to true, since the clause is propagating, and so there are vertices for them in the graph). In our example, we could read off from the implication graph that it is sound to derive \( \overline{x_1} \lor x_2 \). This is so since every path to \( x_2 \) in the implication graph goes through \( x_1 \), and hence setting \( x_1 \) to true is sufficient to propagate \( x_2 \) to false. In general, for every set of literals \( L \) such that all paths from decisions to a literal \( p \) go through a vertex in \( L \), we have the implication \( (\bigwedge_{\ell \in L} \ell) \rightarrow p \), or in clausal form \( \lor_{\ell \in L} \overline{\ell} \lor p \). We refer to such a set \( L \) as a cut, and the building blocks that our algorithm derives are clauses corresponding to such cuts.

Importantly, we do not collect all cuts and clauses that can be obtained from them, but only one cut and clause for each propagated literal. The intuition is that this clause should essentially contain the decisions directly responsible for propagating the literal plus the literal itself.

In more detail, our cut generation algorithm for deriving clausal building blocks takes a parameter \( w_{\text{max}} \) specifying the maximum width of clauses to detect. For example, if \( w_{\text{max}} = 2 \), then only binary clauses are derived, which only suffices to form at-most-1 constraints. The idea is that during the search the solver should find such clauses for propagated literals by computing a cut containing the decisions that have a path to this literal in the implication graph and adding the clause encoding that this cut implies the propagated literal as described above. We need a slight adjustment of this, however, in that in addition to decisions literals we also include in the cut propagated literals for which the already computed cuts hit the \( w_{\text{max}} \) size bound. See Algorithm 1 for the pseudocode, where \( \text{cut}[p] \), if defined, contains a (recursively computed) cut of size less than \( w_{\text{max}} \) for the literal \( p \) in the implication graph. In Algorithm 1, the roundToClause method reduces the pseudo-Boolean reason constraint \( \sum a_i \ell_i \geq A \) for literal \( p \) to a clause \( \sum \ell_i \geq 1 \) (possibly after first having weakened the constraint by removing some literals, so that the rounded clause is still propagating). Next, the decision cut for \( p \) is constructed. For each literal \( \ell \) in the reason clause, if \( \ell \) has a decision cut already calculated, we add the set of decisions responsible for propagating it to the cut for \( p \). Otherwise, we add \( \ell \) itself.

#### 3.2 Filtering Building BlockClauses

A first naive approach would be to add all found building block clauses to the constraint database and let the solver use all of them for propagation and conflict analysis. This turns out not to be a good idea, however: it slows down the search speed, and also significantly worsens search quality measured in total number of conflicts. One way to decrease the number of clauses is to compute cuts for all literals during propagation, but only add the corresponding clauses for literals that actually take part in the conflict analysis during the backward phase. Furthermore, for such clauses we...
also apply a filtering step, and send clauses not passing this filter to a separate storage of inactive constraints that do not take part in the search (i.e., the solver does not consider such clauses when performing unit propagation).

Our filter accepts only building blocks \( C \) that are part of an at-least-2 constraint, by which we mean that there is some literal \( \ell \notin C \) such that for all literals \( \ell_0 \in C \) the clause \( \langle C \setminus \{\ell_0\} \cup \{\ell\} \rangle \) can be found in storage. This is perhaps best illustrated by an example. Suppose that building blocks \( C_1 = x_1 \lor x_2 \lor x_3 \), \( C_2 = x_1 \lor x_3 \lor x_4 \), and \( C_3 = x_2 \lor x_3 \lor x_4 \) have previously been detected. Then if the clause \( C_4 = x_1 \lor x_2 \lor x_3 \) is added, we see that all clauses \( \langle C_1 \setminus \{x_i\} \cup \{x_4\} \rangle \) for \( i = 1, 2, 3 \) are in storage, and these clauses together with \( C_4 \) imply the constraint \( x_1 + x_2 + x_3 + x_4 \geq 2 \). Hence, the filter accepts \( C_4 \) (and also retroactively \( C_1, C_2, \) and \( C_3 \)).

A more generous criterion for being part of an at-least-2 constraint could have been that all clauses \( \langle C \setminus \{\ell_0\} \cup \{\ell\} \rangle \) or subclauses thereof could be found (in our example, that \( x_1 \lor x_2 \lor x_4 \) and \( x_3 \lor x_4 \) were present, say). However, we have found that using such a filter degrades performance substantially on certain benchmarks.

### 3.3 Forming Cardinality Constraints

The actual construction of cardinality constraints happens during the backward phase while the solver is analysing the conflict it just reached. When a short clause appears as the reason in the conflict analysis, the solver attempts on the fly to extend this clause to a largest possible cardinality constraint. As described in the introduction, this is done by a greedy algorithm that considers variables in decreasing order of activity in recent conflicts (in more technical terms, variables with higher scores as measured by the variable state independent decaying sum (VSIDS) (Moskewicz et al. 2001) are tried first). We have also attempted to use heuristics based on the activity of building block clauses in conflict analysis, as well as other non-adaptive heuristics, but have found that the approach based on variable activity worked best.

This recovery of cardinality constraints is integrated in the pseudo-Boolean conflict analysis as described in Algorithm 2. In this algorithm, we resolve the conflicting constraints with reason constraints for the propagations falsifying it until the derived constraint is such that it will flip the value assigned to some literal before the last decision was made. As mentioned in Section 2, an extra step in PB conflict analysis compared to CDCL is that the reason constraints need to be reduced in order for the algorithm to work. What we add further to do on-the-fly cardinality detection is the \( \text{cardDetect} \) function, which recovers cardinality constraints from building block clauses as has been described above.

One important difference from the limited cardinality constraint detection during the filtering step, however, is that during cardinality constraint recovery we do not require exact syntactic matches of clauses taking part in cardinality constraints, but also look for subclauses of such clauses. The reason for this is that it might well happen that a cardinality constraint is formed by stronger clauses than those in its canonical encoding to CNF. All short clauses participate in forming cardinality constraints, including building block clauses in inactive storage that have not yet passed our filtering test.

### 3.4 A Worked-out Toy Example

Let us now show how our PB solver with cardinality detection can efficiently solve a pseudo-Boolean formula with some clausal constraints. We consider a toy formula encoding the contradictory constraints \( x_1 + x_2 + x_3 + x_4 \geq 3 \) and \( x_1 + x_2 + x_3 + x_4 \leq 2 \), but with the second constraint obfuscated as a set of clauses with extra variables added as in Figure 1. Note that the last two constraints \( x_1 + x_3 + x_4 \geq 1 \) and \( x_2 + x_3 + x_4 \geq 1 \) are just the pseudo-Boolean representations of the clauses \( x_1 \lor x_3 \lor x_4 \) and \( x_2 \lor x_3 \lor x_4 \), which are the building blocks we are expecting for the at-most-2 constraint.

Suppose that the solver decides to set \( x_2 \) true and then \( x_1 \) true. Then the two top right constraints propagate \( y_1 \) to true and then \( x_3 \) to false as shown in the implication graph in Figure 2 (where we also show the propagating constraints in clausal form to make the illustration easier to follow). Also, the three bottom left constraints in Figure 1 propagate \( z_1 \) and \( z_2 \) to true, which in turn propagates \( x_4 \) to false as shown in the implication graph in Figure 3.
with. This is hard to see from the results in the competitions because the benchmarks in the competitions were selected to be challenging but not completely out of reach for existing solvers. This means that if some benchmarks critically required on-the-fly cardinality detection to be solved fast, then by design such formulas would have likely been to be omitted from the competition sets.

On CNF formulas, we compared our solver to Sat4j Card from (Biere et al. 2014) as well as to the state-of-the-art SAT solvers CaDiCaL (CaDiCaL 2019) and Glucose (Au- demand and Simon 2009). On pseudo-Boolean (OPB) formulas, we again compared our solver to Sat4j Card and also to the state-of-the-art PB solvers Sat4j (Le Berre and Porrain 2010) and clasp (Gebser, Kaufmann, and Schaub 2012). Sat4j comes in one version Sat4j Res that only reasons with clauses, and another Sat4j Res+CP that runs in parallel the clause-based solver and a solver that reasons with general PB constraints. The solver Sat4j CP is too slow to run on its own in a competition setting. However, Sat4j Card uses only this solver plus cardinality detection. The experiments in (Biere et al. 2014) used also a different tool for cardinality detection, namely Riss (Manthey 2012). However, the solvers using Riss for preprocessing were found to do worse than what was reported in the paper, and therefore we discarded them.

For our experiments on SAT competition benchmarks we used a timeout of 5000 seconds. The results are as given in Figure 4. Here we see that RoundingSat-Card is not able to keep up with the fastest CNF SAT solvers, but that there exist benchmark families for which our cardinality detection turns out to be essential to solve the formula quickly. These families are, among others, synthesis of shortest straight-line programs (Fuhs and Schneider-Kamp 2010) and the GrandTour5abs puzzle (Chowdhury, Müller, and You 2018).

4 Experimental Evaluation

We have implemented our approach within the pseudo-Boolean solver RoundingSat (Ellfers and Nordström 2018). We refer to the solver with cardinality detection as RoundingSat-Card. We set \( w_{\text{max}} = 5 \) in all experiments. In (Biere et al. 2014), the bound is set to \( w_{\text{max}} = 3 \) (binary and ternary clauses), and this low value is necessary for their approach to run fast enough. We report results from two types of benchmarks: SAT and PB competitions from 2016–2018, and a number of crafted benchmarks. The motivation for including these crafted benchmarks is to make clear that the approach put forward in this paper has potential to solve formulas which earlier solvers could not deal
solvers exhibit similar performance on these benchmarks.

Our second PHP-like formula is the two-pigeons-per-hole principle formula, which encodes the claim that \(2n - 1\) pigeons do not fit into \(n - 1\) holes if each hole has room for at most 2 pigeons. We use this benchmark to test detection of at-most-2 constraints. We consider four different encodings: the binomial encoding, which lists all building blocks in the input, and the three encodings of \textit{Minisat+} (Eén and Sörensson 2006): sort networks, BDDs and adder networks. The reason that we use different encodings than in (Biere et al. 2014) is that the encodings used there are specifically for at-most-1 constraints. See Figure 6 for the results (except that the binomial encoding was so much easier than the other encodings for all solvers that we omit them). Here we see that \textit{RoundingSat-Card} scales better than \textit{Sat4j Card}, and also better than the best CDCL solvers.

We consider one more PHP-inspired formula. Here the needed building blocks are not already implied by unit propagation but have to be found through clause learning. These benchmarks are 3-colouring formulas generated by a reduction from the so-called graph functional pigeonhole principle. This reduction was proposed in the paper (Lauria and Nordström 2017). The results are presented in Figure 7. Observe that \textit{RSCard} scales reasonably well and much better than \textit{RoundingSat} without cardinality detection, for which these formulas are exponentially hard. We wish to highlight two technical challenges when solving this formula. Firstly, typically multiple cardinality constraints can be detected, but only one choice works. It turns out that extending with variables in the order of VSIDS scores selects the right constraint. Secondly, adding all ternary clause building blocks degrades performance (in terms of number of reasoning steps, not only running time), but adding a filtering step for building blocks resolves this issue.

When comparing \textit{RSCard} and \textit{Sat4j Card} on pigeonhole principle-like problems, we note that probing can only recover clauses which were implied by unit propagation from the initial set of clauses. An encoding of a cardinality constraint into CNF is called arc-consistent if unit propagation on the set of clauses does not miss any propagations that one gets when running unit propagation on the cardinality constraint. For probing to find all building blocks of a cardinality constraint, the CNF encoding must be arc-consistent. This holds for all encodings of the pigeonhole principle in Figure 5. However, the sorter network and BDD encoding of the two pigeons per hole principle are also arc-consistent, but \textit{Sat4j Card} does not perform well on these formulas. This indicates that a necessary condition for \textit{Sat4j Card} to solve a formula efficiently is that all cardinality constraints the solver needs to run fast consist of clauses that are already implied by unit propagation by the initial set of clauses. From the results on the two pigeons per hole principle, we see that this is not a sufficient condition for \textit{Sat4j Card} to recover the cardinality constraints.

Next, let us investigate a family of unsatisfiable benchmarks where the difficulty is not in finding the building blocks, but in combining them in the right way, because there are many possible overlapping cardinality constraints to find. These benchmarks encode the claim that a \((k - 1)\)-colourable graph cannot contain a \(k\)-clique. This formula is easy for cutting planes, because a \((k - 1)\)-colouring can be mapped to \((k - 1)\) at-most-1 constraints which when added to the clique constraint give a contradiction. However, every independent set in the graph is a possible at-most-1
constraint, so it is not clear how the solver would find 
\((k-1)\) such constraints that partition the graph. In our
experiments, we set \(k = 25\) and let the graph be a random
\((k-1)\)-colourable graph with each edge between differently
coloured vertices present with probability \(n^{-2/(k-1)}\). This
is the probability where the transition between probably not
containing a clique and probably containing a clique hap-
pens, if we ignore that the graph is \((k-1)\)-colourable. From
Figure 8 we can see that cardinality detection significantly
improves the solving power of \(\text{RoundingSat}\), though it still
seems that it does not work well when \(n\) gets large.

We also would like to highlight one formula where car-
dinality detection pays off even though the formula as pre-
sented to the pseudo-Boolean solver is already easy in the-
ory. The even colouring formula (Markström 2006) is an unsat-
sifiable benchmark where it seems that PB solvers have
to combine addition and division in the right way to run fast. This turns out to be very challenging for PB solvers

| Encoding         | \(\text{Sat4j Card}\) | \(\text{RSCard}\) | \(\text{RS clasp}\) |
|------------------|-----------------------|-------------------|---------------------|
| \(\text{CNF}\)   | 10                    | 10                | 2                   |
| \(\text{OPB}\)   | 10                    | 10                | 2                   |
| \(\text{recoverTree}\) | 1                   | 10                | 2                   |
| \(\text{recoverTreeReuse}\) | 1                  | 10                | 2                   |
| \(\text{recoverInput}\) | 1                  | 6                 | 2                   |

Figure 10: Results for division-friendly formulas (10 bench-
marks per family).

in practice, however. In slightly more details, the formula
encodes that for a graph with all vertices having even de-
gree but with an odd number of edges, it is not possible to
colour edges black and white so that each vertex has the
same number of black and white adjacent edges. See Fig-
ure 9 for experimental results on 4-regular random graphs
with a split edge (to make the edge count odd). Quite in-
terestingly, cardinality detection significantly improves per-
formance of \(\text{RoundingSat}\) not only for the \(\text{CNF}\) encoding
(where original \(\text{RoundingSat}\) has no chance) but also for the
PB encoding.

Finally, let us discuss a set of benchmarks called divi-
sion-friendly formulas (Gocht, Nordström, and Yehudayoff
2019), which are obfuscated versions of (otherwise ex-
remely easy) subset cardinality formulas (Mikša and
Nordström 2014). The encodings allow a PB solver imple-
menting the division rule, such as \(\text{RoundingSat}\), to solve the
formulas efficiently in theory, but in practice these formulas
are very hard for all pseudo-Boolean solvers. We see in Fig-
ure 10 that our solver with cardinality detection deals much
more successfully with these obfuscated formulas, although
it has difficulties with one specific encoding.

5 Conclusion and Future Work

In this paper we present a new approach to cardinality detec-
tion in pseudo-Boolean solvers that, in contrast to previous
works such as (Biere et al. 2014), is not applied during pre-
processing but throughout the execution of the solver, and is
not based on probing or syntactic checks but extracts infor-
mation semantically during search. Our experiments show
that our approach incurs only a moderate overhead in the
worst case, and sometimes yields dramatic improvements
in efficiency. In particular, for formulas in \(\text{CNF}\) that en-
code cardinality constraints this technique allows pseudo-
Boolean solvers to go beyond \(\text{CDCL}\) and harness the power
of cutting-planes-based reasoning.

When running on \(\text{CNF}\) formulas, pseudo-Boolean reason-
ing is at a disadvantage to \(\text{CDCL}\), because the data structures
and algorithms required are more involved, and we can-
not match the overall performance of state-of-the-art \(\text{CDCL}\)
\(\text{SAT}\) solvers in a competition setting. However, our pseudo-
Boolean solver enhanced with cardinality detection manages
to solve some challenging benchmarks that are out of reach
for existing \(\text{CDCL}\) and \(\text{PB}\) solvers. This shows that already
our proof-of-concept implementation could play a useful
role as a part of a portfolio \(\text{SAT}\) solver. Furthermore, even
for some native pseudo-Boolean formulas, where there are
no hidden PB constraints to detect, our experiments show
that adding on-the-fly cardinality detection can improve the pseudo-Boolean search and make the solver run faster.

As to future work, we believe that there is substantial room for improvements of our technique for cardinality detection. We would like to develop better heuristics for how to generate building block clauses from cuts in the implication graph and for which building blocks to keep or throw away. Another possible direction is to perform the cardinality detection in an adaptive fashion, where the solver would spend more or less time on detection depending on how much this work has paid off so far during search (provided that a meaningful way of measuring this could be found).

We already know of crafted CNF formulas for which our approach does not work, although the formulas contain cardinality constraints (e.g., the so-called relativized pigeonhole principle (RPHP) formulas in (Atserias, Lauria, and Nordström 2016)), and as a first step would want to be able to solve also such formulas.

Another highly relevant challenge would be to detect cardinality constraints without going via the binomial encoding in our building block clauses. For example, an at-most-50-out-of-100 constraint is completely infeasible to detect with our current method, since the intermediate clausal representation becomes prohibitively large. However, if such a constraint were provided in the BDD encoding of MiniSat+ (Eén and Sörensson 2006), then a procedure recognizing this encoding might be able to recover the constraint. Thus, we are suggesting to go back to a mix of semantic and syntactic detection techniques.

We would also like to extend the pseudo-Boolean detection to recover not only cardinalities but also more general linear inequalities where some of the coefficients are greater than 1, such as, e.g., $2x_1 + x_2 + x_3 + x_4 \geq 2$. So far we have not been able to implement detection and recovery of such constraints, since it would seem to require a redesign of the filtering method that we use to retain or throw away building blocks of cardinality constraints. Dropping the filter completely does not seem to be an option, as this ruins the effectiveness of the approach.

A final remark is that we have observed that probing sometimes beats our implication-graph-cut technique in finding useful building block clauses. However, if the decision heuristic of the solver is modified by setting the VSIDS decay factor to the fastest possible decay—at least that when prioritizing variables to decide on the solver has very little memory and quickly forgets everything except the variables involved in the very latest conflicts—then our solver seems to be equally good as probing at finding the right building blocks. We would like to understand better the relationship between probing on the one hand and our implication-graph-cut technique combined with with fast VSIDS decay factor on the other.

Acknowledgments
We thank Daniel Le Berre for providing us with the solvers and benchmarks used in (Biere et al. 2014), and Romain Wallon for helping us figure out the correct versions of the solvers to use. We are grateful to Massimo Lauria for sharing the generator scripts for the benchmarks described in (Lauria and Nordström 2017). We gratefully acknowledge useful feedback from the participants of the workshop Pragmatics of SAT 2019, where a preliminary version of this work was presented. Throughout this project, we have benefited greatly from many interesting and enjoyable discussions with Jo Devriendt, Stephan Gocht, and Janne Kokkala. Last but not least, we are thankful to the anonymous reviewers for many detailed comments and useful suggestions, which helped improve this manuscript considerably.

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 authors were supported by the Swedish Research Council grants 621-2012-5645 and 2016-00782, and the second author also received funding from the Knut and Alice Wallenberg grant KAW 2016.0066.

References
Ansôtégui, C.; Larribia, J.; Li, C. M.; and Manyà, F. 2007. Exploiting multivalued knowledge in variable selection heuristics for SAT solvers. *Annals of Mathematics and Artificial Intelligence* 49(1-4):191–205.

Ansôtégui, C. 2004. Complete SAT solvers for Many-Valued CNF Formulas. Ph.D. Dissertation, University of Lleida.

Atserias, A.; Lauria, M.; and Nordström, J. 2016. Narrow proofs may be maximally long. *ACM Transactions on Computational Logic* 17(3):19:1–19:30. Preliminary version in CCC ’14.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI ’09)*, 399–404.

Bayardo Jr., R. J., and 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)*, 203–208.

Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Biere, A.; Le Berre, D.; Lonca, E.; and Manthey, N. 2014. Detecting cardinality constraints in CNF. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT ’14)*, volume 8561 of *Lecture Notes in Computer Science*, 285–301. Springer.

Biere, A. 2013. Lingeling, plingeling and treengeling entering the SAT competition 2013. In *Proceedings of SAT Competition 2013*, 51–52.

Buss, S., and Nordström, J. 2020. Proof complexity and SAT solving. Chapter to appear in the 2nd edition of (Biere et al. 2009). Draft version available at https://www.math.ucsd.edu/~sbuss/ResearchWeb/ProofComplexitySAT/.

2019. CaDiCaL. http://fmv.jku.at/cadical/.

Chai, D., and 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.

Chowdhury, M. S.; Müller, M.; and You, J.-H. 2018. GrandTouche puzzle as a SAT benchmark. In *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, 59–60.

Cook, W.; Couillard, C. R.; and Turán, G. 1987. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics* 18(1):25–38.
Cook, S. A. 1971. The complexity of theorem-proving procedures. In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC ’71), 151–158.

Dixon, H. E., and Ginsberg, M. L. 2002. Inference methods for a pseudo-Boolean satisfiability solver. In Proceedings of the 18th National Conference on Artificial Intelligence (AAAI ’02), 635–640.

Een, N., and Sörensson, N. 2006. Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2(1-4):1–26.

Elffers, J., and Nordström, J. 2018. Divide and conquer: Towards faster pseudo-Boolean solving. In Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI ’18), 1291–1299.

Fuhls, C., and Schneider-Kamp, P. 2010. Synthesizing shortest linear straight-line programs over GF(2) using SAT. In Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT ’10), volume 6175 of Lecture Notes in Computer Science, 71–84. Springer.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. Artificial Intelligence 187–188:52–89.

Gocht, S.; Nordström, J.; and Yehudayoff, A. 2019. On division versus saturation in pseudo-Boolean solving. In Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI ’19), 1711–1718.

Haken, A. 1985. The intractability of resolution. Theoretical Computer Science 39(2-3):297–308.

Han, H.; Jin, H.; and Somenzi, F. 2011. Clause simplification through dominator analysis. In Proceedings of the Design, Automation & Test in Europe Conference (DATE ’11), 143–148.

Heule, M.; Järvisalo, M.; and Biere, A. 2013. Revisiting hyper binary resolution. In Proceedings of the 10th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR ’13), volume 7874 of Lecture Notes in Computer Science, 77–93. Springer.

Hooker, J. N. 1988. Generalized resolution and cutting planes. Annals of Operations Research 12(1):217–239.

Hooker, J. N. 1992. Generalized resolution for 0-1 linear inequalities. Annals of Mathematics and Artificial Intelligence 6(1):271–286.

Lauria, M., and Nordström, J. 2017. Graph colouring is hard for algorithms based on Hilbert’s Nullstellensatz and Gröbner bases. In Proceedings of the 32nd Annual Computational Complexity Conference (CCC ’17), volume 79 of Leibniz International Proceedings in Informatics (LIPIcs), 2:1–2:20.

Le Berre, D., and Parrain, A. 2010. The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation 7:59–64.

Levin, L. A. 1973. Universal sequential search problems. Problemy peredachi informatsii 9(3):115–116. In Russian. Available at http://mi.mathnet.ru/ppi914.

Manthey, N. 2012. Solver description of RISS 2.0 and PRISS 2.0. Technical Report KRR Report 12-02, Technische Universität Dresden. Available at http://www.wv.inf.tu-dresden.de/Publications/2012/report12-02.pdf.

Markström, K. 2006. Locality and hard SAT-instances. Journal on Satisfiability, Boolean Modeling and Computation 2(1-4):221–227.

Marques-Silva, J. P., and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5):506–521. Preliminary version in ICCAD ’96.

Martins, R.; Manquinho, V. M.; and 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), volume 8561 of Lecture Notes in Computer Science, 438–445. Springer.

Mikša, M., and Nordström, J. 2014. Long proofs of (seemingly) simple formulas. In Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT ’14), volume 8561 of Lecture Notes in Computer Science, 121–137. Springer.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In Proceedings of the 38th Design Automation Conference (DAC ’01), 530–535.

Sakai, M., and 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.

Sheini, H. M., and 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.

Urquhart, A. 1987. Hard examples for resolution. Journal of the ACM 34(1):209–219.

Vinyals, M.; Elffers, J.; Giráldez-Cru, J.; Gocht, S.; and 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), volume 10929 of Lecture Notes in Computer Science, 292–310. Springer.