


Speeding up Pseudo-Boolean Propagation

Robert Nieuwenhuis 

Barcelogic.com, Barcelona, Spain

Albert Oliveras 

Technical University of Catalonia, Barcelona, Spain

Enric Rodríguez-Carbonell 

Technical University of Catalonia, Barcelona, Spain

Rui Zhao

Technical University of Catalonia, Barcelona, Spain

Abstract

Unit propagation is known to be one of the most time-consuming procedures inside CDCL-based SAT solvers. Not surprisingly, it has been studied in depth and the two-watched-literal scheme, enhanced with implementation details boosting its performance, has emerged as the dominant method.

In pseudo-Boolean solvers, the importance of unit propagation is similar, but no dominant method exists: counter propagation and watched-based extensions are efficient for different types of constraints, which has opened the door to hybrid methods. However, probably due to the higher complexity of implementing pseudo-Boolean solvers, research efforts have not focused much on concrete implementation details for unit propagation but rather on higher-level aspects of other procedures, such as conflict analysis.

In this paper, we present (i) a novel methodology to precisely assess the performance of propagation mechanisms, (ii) an evaluation of implementation variants of the propagation methods present in ROUNDINGSAT and (iii) a detailed analysis showing that hybrid methods outperform the ones based on a single technique. Our final contribution is to show that a carefully implemented hybrid propagation method is considerably faster than the preferred propagation mechanism in ROUNDINGSAT, and that this improvement leads to a better overall performance of the solver.

2012 ACM Subject Classification Theory of computation

Keywords and phrases SAT, Pseudo-Boolean Solving, Implementation-level Details

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.22

Funding All authors are supported by grant PID2021-122830OB-C43, funded by MCIN/AEI/10.13039/501100011033 and by “ERDF: A way of making Europe”. Second and third authors are supported by Barcelogic through research grants C-11423 and C-11422, respectively.

1 Introduction

CDCL-based SAT solvers [25] have become the method of choice to solve a variety of problems coming from diverse areas such as system verification [10], security [17, 32], cryptography [31] and even mathematics [22]. Nevertheless, theoretical results [4, 27] have identified problems for which no polynomial CDCL execution exists. The reason for this is that resolution, the proof system on which SAT solvers are based, does not provide polynomial proofs for them [21]. This is particularly troublesome for problems, like the pigeon hole principle, which frequently appear in real-world problems [2]. Another well-known limitation of SAT regards the poor expressivity of its input language, which does not allow, for example, encoding numerical constraints in a natural way.

Pseudo-Boolean (PB) solving, also known as 0-1 Integer Linear Programming, has emerged as a remarkable alternative to SAT. The input language to PB solvers, consisting of 0-1 linear constraints, is an extension of propositional clauses, and allows for more compact



© Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao; licensed under Creative Commons License CC-BY 4.0

27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024).

Editors: Supratik Chakraborty and Jie-Hong Roland Jiang; Article No. 22; pp. 22:1–22:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

encodings. In addition, CDCL-based PB solvers [28] are, at least from the theoretical point of view, exponentially more powerful than SAT solvers because they can reason using the cutting-planes proof system [13], which is exponentially stronger than resolution.

CDCL-based PB solvers incorporate many of the techniques that turned out to be essential in SAT solvers. Among them, a crucial procedure is unit propagation, which, given a clause where all literals are false except one that is unassigned, extends the assignment by adding this literal. The efficiency of this procedure is paramount for the overall performance of solvers, and hence efficient ways to implement it have been developed. Out of all these methods, the two-watched literal scheme [26] has established itself as the dominant method in SAT. It is based on a simple fact: if two non-false literals exist in a clause, no unit propagation is possible. Hence, two non-false literals are watched, and only when one of them becomes false, the clause is checked for propagation.

This idea is not so simple in PB constraints because the number of literals to be watched depends on its coefficients. Due to this reason, PB watched-literal propagation and counter propagation, the simpler version where all literals in every constraint are watched, have co-existed in PB solvers with no strong evidence of one method clearly outperforming the other: the PBS solver [3] used counter-based propagation without mentioning the possibility of using watches. The GALENA [11] authors mentioned that watches are effective when the sum of the coefficients is large compared with the independent term. However, they opted to always use counters. Initially, the PUEBLO solver [29] used counters but later moved to a version with watch propagation [30]. In the SAT4J [5] solver, both procedures are implemented and, according to the authors, not much difference was observed among them, with a slight preference for watches. Finally, ROUNDINGSAT [14] concluded that watched propagation was more efficient but, when specialized procedures for clauses and cardinality constraints are implemented, which is a standard technique used in all solvers, the difference is fairly small.

Surprisingly, with the remarkable exception of ROUNDINGSAT, it is not possible to find in the literature rigorous experimental evaluations that support any of these design decisions. The situation is dramatically different when it comes to SAT-based unit propagation. As an example, the cache behavior of watched-literal propagation has been thoroughly studied in a number of papers [33, 12, 23, 24], where important propagation speedups are reported. This caused solvers like MINISAT [18], PICOSAT [6] or LINGELING [7] to incorporate carefully designed implementations of watched propagation [8]. A very pedagogical reference is the source code of CADICAL [9], where design decisions are discussed with pointers to the papers on which they rely.

This type of research efforts are very delicate because minimal changes on the implementation of propagation procedures can cause solvers to have dramatically different runtimes that are not directly related with the propagation scheme. This makes improvements that are not groundbreaking impossible to be perceived, because they are mostly blurred by the chaotic behavior of the solver.

In this paper we leverage SAT-solver implementers' knowledge and use that to improve the performance of PB solvers. For that purpose, we considered ROUNDINGSAT and improved the implementation of their propagation routines. We chose this solver because it is the most successful and actively maintained PB solver: its team has recently devoted considerable efforts to develop smart conflict analysis procedures [19], add core-guided techniques [16] and integrate MIP solvers [15], which has led ROUNDINGSAT to be "the world's fastest pseudo-Boolean solver" [1]. This speed is particularly striking because, in our opinion, its authors have focused more on the development of techniques that can give exponential improvements than on lower-level implementation aspects.

In order to evaluate the impact of the improvements we introduce, we also present a novel methodology that, by generating log files with the information of a solver execution, allows us to guarantee that the solver always traverses the search space in the same way. With this certainty, we can assess the precise impact of different unit propagation variants.

This paper is organized as follows. After some preliminaries in Section 2, we explain our novel methodology to compare propagation procedures in Section 3. Sections 4 and 5 present our improvements on counter and watched-based propagation, respectively. Section 6 reports on the development of a hybrid method for propagation and we conclude in Section 7.

Contributions

- (I) First, we present a novel methodology for experimentally evaluating different propagation schemes. This has allowed us to detect small improvements that otherwise would have been unnoticed. It is well known in the solver developer community that by joining the forces of many small improvements, significant speed-ups may be achieved.
- (II) Second, we describe and evaluate improvements on the implementation of counter and watch-based propagation in ROUNDINGSAT, giving concrete reasons for their impact.
- (III) Third, we report on the use of hybrid approaches, where the decision to use counter or watched propagation is done per constraint. Although this was already mentioned in GALENA and ROUNDINGSAT, where it was said to be not superior to counters in the first solver, and left as future work in the second, here we show that a hybrid approach clearly outperforms the virtual best solver that always chooses the best option between using counters or watches for *all* constraints.
- (IV) Finally, we show that improvements in unit propagation do not negatively affect other components of the solver. That is, the improved solver is indeed faster and solves more instances than the original one.

2 Preliminaries

Pseudo-Boolean Constraints. Let \mathcal{X} be a set of propositional variables. A *literal* is either a variable (x) or the negation of one (\bar{x}). We will assume that $\bar{\bar{x}} = x$. A (*pseudo-Boolean* or *PB*) *constraint* is a 0-1 linear inequality $\sum_i c_i l_i \geq d$ where the l_i 's are literals and, without loss of generality, the c_i 's (*coefficients*) and d (*degree*) are positive integers. When all coefficients are 1 we say that the constraint is a *cardinality constraint* and if, in addition, d is also 1 we say that it is a *clause*. A *formula* is a set of constraints.

Satisfiability. An *assignment* ρ is a set of non-contradictory literals. It is *total* if for any $x \in \mathcal{X}$ either $x \in \rho$ or $\bar{x} \in \rho$, and *partial* otherwise. A literal l is *true* in ρ if $l \in \rho$, is *false* if $\bar{l} \in \rho$ and is *undefined* otherwise. Given a constraint C of the form $\sum_i c_i l_i \geq d$, an assignment ρ *satisfies* it if $\sum_{i:l_i \in \rho} c_i \geq d$, and *falsifies* it if no extension of ρ can satisfy it. If we define $slack(C, \rho) = (\sum_{i:\bar{l}_i \notin \rho} c_i) - d$, it can be seen that ρ falsifies C if and only if $slack(C, \rho) < 0$. Note the slack expression sums the coefficients of all non-false literals, and that is the maximum value that the left hand side of the constraint can reach. If even that does not exceed d , no extension of ρ will do. An assignment that satisfies all constraints of a formula is called a *model*. Sometimes, instead of looking for any model, we are interested in finding one that minimizes a certain linear objective function $\sum_i a_i l_i$.

Unit Propagation. Given a constraint $C = \sum_i c_i l_i \geq d$ and an assignment ρ , we say that C *unit propagates* l_i under ρ if l_i is undefined in ρ , but l_i is true in any total assignment extending ρ that satisfies C . The latter is equivalent to checking whether $slack(C, \rho) < c_i$,

i.e., if we do not set l_i to true, the constraint becomes falsified. Given a formula F and an assignment ρ , *unit propagation* of F under ρ is the outcome of applying the following two rules until a fixpoint is reached: (i) if ρ falsifies a constraint $C \in F$, a *conflict* is found with *conflicting constraint* C and we stop, (ii) if ρ unit propagates some literal l due to constraint C , extend $\rho := \rho \cup \{l\}$ with *reason* C .

Conflict-Driven Pseudo-Boolean Solving. A generalization of the well-known CDCL [25] algorithm for SAT can be applied to the pseudo-Boolean case [28]. The algorithm starts with an empty assignment ρ and proceeds as follows: (1) Apply unit propagation, possibly extending ρ . (2) If a conflict is found, a conflict analysis procedure derives a constraint C (called *lemma*) that can be safely added to the formula. If C is the constraint $0 \geq 1$, the formula is unsatisfiable, otherwise it is guaranteed that after removing some literals from ρ in a last-in first-out way (*backjumping*), C allows some literal to be unit propagated. Hence, we go to step 1. (3) If no conflict is found, and ρ is total, it is a model of the formula. Otherwise, an undefined literal l (*decision literal*) is added to ρ and we go to step 1. The choice of l is determined by sophisticated heuristics.

Since the set of constraints might grow too much, periodically a *cleanup* process that deletes some lemmas is performed. Also periodically, the process *restarts*: roughly speaking, ρ is reset to the empty set. Since all lemmas in the formula are still kept, the behavior of the algorithm will change.

3 Design of a Fair Evaluation of Different Propagation Mechanisms

Developing a state-of-the-art PB/SAT solver not only consists in implementing and combining the appropriate techniques. Optimizing its implementation is a critical task, which turns out to be very complex due to the “chaotic” behavior of these solvers. On a single instance, seemingly innocuous changes may have a strong impact on the overall solver runtime, but this does not necessarily mean that this change is worth incorporating to the solver. One prominent example of this phenomenon takes place when optimizing unit propagation. A simple change in the order in which clauses are processed during this procedure drives the solver towards finding one conflicting clause or another, which in turn, leads to different learned lemmas and to completely different search space traversal. Hence, evaluating the impact of a new implementation technique on a small set of benchmarks is hopeless. This is usually fixed by performing exhaustive experiments on very large sets of benchmarks, hoping that, by evaluating some measure over all the collected data, one will be able to conclude whether this implementation method pays off or not. In our opinion, this has at least two problems. Firstly, it makes the evaluation of these improvements a very time-consuming task. Secondly, modest improvements are not observable with this methodology, because the noise produced by the changes in search space exploration end up hiding their real impact.

In order to avoid this troublesome scenario, we want to force the solver to explore the search space in the same way, independently of the unit propagation mechanism we implement. The good news is that the outcome of the unit propagation procedure is almost unique. If there is no conflict, any complete propagation procedure will add exactly the same literals to the assignment. The only difference is that they may be added in a different order, or due to different reasons, but we will see that this is not a problem for our method. Otherwise, if unit propagation derives a conflict, any complete procedure will find it. However, since propagation stops as soon as it finds the first conflicting clause, different propagation variants may find different conflicting clauses, which will lead to the generation of different lemmas by conflict analysis and subsequent different search space exploration.

Intuitively our methodology relies on an *oracle* that, after a conflicting clause is found, tells the solver which lemma should be learned. In principle, that would allow the solver to always explore the search space in the same way¹. However, there are other ingredients in the CDCL procedure that have a strong effect on the search behavior: decision heuristics, cleanups and restarts. For that reason, our oracle will in addition have to inform the solver about (i) which literal should be the next decision, (ii) when the next cleanup should be applied and which constraints should be removed, and (iii) at which point a restart should be applied.

In this paper, the role of the oracle will be played by a *log*, which is a file obtained from the execution of a PB solver where all the previously mentioned information is written. It is not difficult to modify a PB solver so that it reads these logs. The solver then mostly consists of only the unit propagation procedure, since the outcome of all other computationally-expensive tasks is provided by the oracle. With this infrastructure setup, we can change the unit propagation procedure with the guarantee that the search behavior of the solver (i.e. number of decisions, number of conflicts, etc.) will be the same.

Since ROUNDINGSAT² is probably the fastest existing CDCL-based PB solver, we have chosen it in order to conduct our work. In the rest of the paper, except in the very last experiment, ROUNDINGSAT is always executed reading the logs we previously generated. This experimentation³ was done on 3.3Ghz 16GB Intel Xeon E-2124 machines over a small (about 100) set of benchmarks we selected from the OPT-SMALLINT-LIN (optimization problems with small integers and linear constraints) category of the 2016 Pseudo-Boolean Competition⁴, the last edition that took place. Benchmarks were chosen so as to contain a variety of runtimes, ranging from easier (a few seconds) to more difficult ones (more than one hour).

4 Counter-Based Propagation

ROUNDINGSAT follows the standard way to implement propagation in Conflict-Driven Learning solvers: for each literal l , a vector $wlist(l)$ (the *watch list* of l) contains a superset of the constraints⁵ that might have become false or propagating due to adding \bar{l} to the assignment. Hence, even by visiting only these constraints, all propagations and conflicts are detected. Let us see in detail how this is implemented in ROUNDINGSAT.

In counter-based propagation, every constraint C of the form $\sum_i c_i l_i \geq d$ has an element of type *Watch* in $wlist(l_i)$ for all i . This element is a pair $\langle ctrPtr, idx \rangle$, where $ctrPtr$ can be seen as a pointer to C and idx is the position of l_i in C . At some point after a literal l is added to the assignment, l is *checked for propagation*: all elements of $wlist(\bar{l})$ are traversed and the function in Algorithm 1 is called, which assumes that constraints are always sorted from largest coefficient to smallest.

¹ In conflicting decision levels, the number of propagated literals may still be different due to the possibility of finding different conflicting clauses. Fortunately, this had a negligible impact on propagation time.

² In order to reproduce the results in [14] we have worked on the version used in that paper. However, we want to remark that the newest version of ROUNDINGSAT has not changed its propagation routines.

³ Additional material can be found in <https://github.com/dearzaorui/speedup-roundingSat>.

⁴ <https://www.cril.univ-artois.fr/PB16/>

⁵ Note that watch lists also contain clauses and cardinality constraints, which have specialized propagation procedures, but we mostly ignore them in this paper. We added binary clauses as another particular type of constraint, but this did not change the performance significantly.

■ **Algorithm 1** Counter-based propagation procedure.

```

1 Function Propagate-Counter(Watch w):
2   Constraint ctr := w.ctrPtr
3   if isDeleted(ctr) then return
4   if not isPBCounter(ctr) then return
5   coef := ctr[w.idx].coef
6   maxCoef := ctr[0].coef
7   ctr.slack := ctr.slack - coef
8   if ctr.slack < 0 then return CONFLICT
9   if ctr.slack < maxCoef then // possible propagation
10  |   i := 0
11  |   while i < ctr.size and ctr.slack < ctr[i].coef do
12  |   |   if isUndef(ctr[i].lit) then propagate(ctr[i].lit)
13  |   |   i := i + 1
14  return OK

```

Let us first remark that lines 3 and 4 are not part of a standard counter propagation routine. However, for every element in the watch list, ROUNDINGSAT first checks whether it is a deleted constraint, and then checks the type of the constraint in order to decide which propagation mechanism is invoked. We added them to Algorithm 1 to make it evident that in ROUNDINGSAT these two lines are always executed.

If we consider $\hat{\rho}$ to be the set of literals in the current assignment ρ that have been checked for propagation, the *slack* field in a constraint stores the sum of the coefficients of the non-false literals w.r.t $\hat{\rho}$ minus the degree, i.e. $slack(C, \hat{\rho})$. Since $\hat{\rho} \subseteq \rho$, it holds that $slack(C, \rho) \leq slack(C, \hat{\rho})$. If this is a negative value (line 8), the constraint is falsified by $\hat{\rho}$ and hence by ρ .

In order to check for propagation, if *maxCoef* is the maximum coefficient in the constraint we know that if $slack(C, \hat{\rho}) \geq maxCoef$, then $slack(C, \hat{\rho}) \geq c_i$ for all i and hence $\hat{\rho}$ does not propagate any literal. This is checked in line 9. Otherwise, the constraint is checked for propagation w.r.t $\hat{\rho}$. Note that since the constraint is ordered by coefficient, once a literal is not large enough to be propagated, none of the subsequent literals will be (second condition in line 11). A final remark is that eventually all literals are checked for propagation ($\hat{\rho} = \rho$) and hence all conflicts and propagations w.r.t. ρ are finally computed.

Algorithm 1 is in fact a simplified version of the one in ROUNDINGSAT [14], which contains an important optimization that allows the loop at line 11 to not always start with $i = 0$. More concretely, if one execution of the loop stops at some position $k < ctr.size$ and no backjump is applied, the next traversal of the constraint can start at position k . This improvement will be present in all our variations of counter-based propagation because we independently confirmed that it does indeed improve the performance.

Minimizing the Number of Constraint Loads. As we have mentioned, when a literal l becomes false, the watch list of \bar{l} is traversed, checking constraints in that list for propagation. This has no remarkable difference with what happens in SAT solvers. Since constraints are consecutively accessed in the order in which they appear on the watch list, one would like to keep them as close in memory as possible, so as to minimize the number of cache faults. This is usually done with a class (ClauseAllocator in systems deriving from MINISAT or Arena in CADICAL or KISSAT [9]) that encapsulates the clause memory management. However, due to the huge number of clauses, this is still a bottleneck in SAT solvers. As the CADICAL source code literally says: “the cache line with the clause data is forced to be loaded here and thus this first memory access below is the real hot-spot of the solver”.

Hence, in a PB solver, when we visit constraints we should try to avoid loading the constraint into memory as much as possible. This is something that, as we will see, was not considered in ROUNDINGSAT, where priority was given to developing an extremely modular and flexible system where all information is kept in their expected data type. This is a very natural decision design that facilitates extending the system with more sophisticated reasoning techniques, but has some impact on efficiency.

More concretely, a *Constraint* object in ROUNDINGSAT contains, among others, the following fields: a C-like array of 32-bit integers with its coefficients and literals, a 64-bit integer storing the degree, a 64-bit integer for the slack (where the 3 smallest possible values are reserved to express that this constraint is a clause, a cardinality constraint or a PB constraint), a 64-bit integer that uses 1 bit to express whether the constraint is deleted, 1 bit to indicate whether counter or watch propagation is to be used for this constraint, 30 bits for the size, and some others bits for information that is not relevant for our purposes.

A quick analysis of Algorithm 1 reveals that the solver always executes line 3, which forces the constraint to be loaded into memory, because the deletion information is inside the constraint. In order to minimize the number of times that the constraint is loaded (the hot-spot of the solver) we suggest to store all necessary information about a constraint in the watch list by modifying the structure *Watch*. More concretely:

- The type of constraint (binary clause, clause, cardinality constraint, counter PB constraint, watched PB constraint), needed in line 4 of Algorithm 1, will be stored in *Watch*.
- The coefficient of the watched literal (line 5) will also be stored in *Watch*.
- The slack of the constraint cannot be stored in *Watch*, because modifying it (line 7) would require traversing all other watch lists where this constraint occurs. Hence, we create a vector of 64-bit integers, indexed by a constraint identifier. This is much better from the memory point of view, because all slacks are in contiguous memory, unlike what happens with constraints.
- The maximum coefficient of the constraint will not be stored anywhere. What we suggest is that the slack vector contains an integer corresponding to the definition of $slack(C, \hat{\rho})$ minus the maximum coefficient. We call this vector *slackMM* and will be enough to detect conflicts and propagations.
- The information about whether a constraint is deleted or not could be stored in *Watch*. However, that would make cleanup slightly slower since whenever a constraint is marked for deletion, the corresponding watch lists should be traversed. What we suggest is to keep this information in the *slackMM* vector, where 1 bit corresponds to deletion information and the rest to the value we have described before.

Summing up, for PB constraints the *Watch* structure needs three pieces of information: the identifier of the constraint, the index of the literal in the constraint, as well as the coefficient⁶. The new structure will contain 3 integers in such a way that they also suffice to also store all the additional information we need (type of constraint, the other literal in binary clauses, a cached literal⁷ for long clauses, and the index of the literal in a cardinality constraint):

- *internal_id*: stores the identifier of the constraint, only needed for PB constraints.

⁶ The index is indeed not necessary for counter-based propagation. However, it will be needed for watch-based propagation. Since our goal is to develop a hybrid method, we store the index here as well.

⁷ Sometimes called blocking literal, it is a well-known technique in SAT-based unit propagation that prevents loading the clause in some situations.

■ **Algorithm 2** Improved counter-based propagation procedure.

```

1 Function Improved-Propagate-Counter(Watch w):
2   id := w.identifier()
3   if slackMM[id].isDeleted then return
4   if w.type() ≠ PB-counter then return
5   coef := w.coefficient()
6   slackMM[id].slack := slackMM[id].slack − coef
7   if slackMM[id].slack < 0 then // possible prop., but check conflict
8     Constraint ctr := constraints[id]
9     slack := slackMM[id].slack + ctr[0].coef // ctr sorted by coef.
10    if slack < 0 then return CONFLICT
11    i := 0
12    while i < ctr.size and slack < ctr[i].coef do
13      if isUndef(ctr[i].lit) then propagate(ctr[i].lit)
14      i := i + 1
15  return OK

```

- *idx_type*: this is used for storing the index of the literal and partially, the constraint type. For binary clauses this integer is equal to -1 ; for longer clauses to an integer smaller than -1 ; for cardinality constraints, the last bit of this integer is 1, and the rest of the bits store the index of the literal in the constraint; finally, for PB constraints its last bit is 0, and the rest of the bits contain the index. Note that this yet does not allow us to distinguish between counter or watch-based constraints. This will be done using the sign of the next field.
- *coef_lit*: for bin clauses, it contains the other literal in the clause. For PB constraints, it contains the coefficient of the literal for counter-based constraints, which is always positive, and minus the coefficient for watch-based constraints. This field is not used for long clauses or cardinality constraints.

The Constraint structure is the same except for the slack information, that has been removed. All these changes allow us to implement the more efficient version of counter-based propagation of Algorithm 2.

Note that lines 3, 4 and 5, that check for deletion, constraint type and retrieve the coefficient, do not load the constraint yet and only require access to information in the Watch element. If line 6 is executed, *slackMM* will be loaded into memory. Only if the check in line 7, that corresponds to line 9 in Algorithm 1, succeeds we load the constraint into memory in line 8. In this case, we recompute the actual slack of the constraint in line 9 and check for conflict afterwards. The rest of the algorithm is unchanged.

We computed, for every benchmark in our suite, the percentage of Watch elements of type PB for which we loaded the constraint, i.e, for which line 8 was executed and obtained an average of 6.29% and a median of 1.26%. This is a remarkably low number that shows that the number of constraint loads is reduced dramatically by Algorithm 2.

In addition, in [14] it is explained how to restore the slack information upon backjumping: whenever a literal is removed from ρ , the watch list of \bar{l} is visited and for every PB constraint in that list, its slack is increased by the coefficient of l in the constraint. This again requires accessing all constraints, whereas in our implementation, only the *slackMM* vector will be loaded into memory.

An analysis of the impact of this improvement can be seen in the top-left scatter plot of Figure 1. Plots of this type will always compare a baseline system with an enhancement, with a caption of the form “Enhancement vs. Baseline”. A point $(100, 2)$ represents a benchmark

for which the baseline version took 100 seconds and the enhancement took $100/2 = 50$ seconds, whereas a point $(100, -2)$ corresponds to a benchmark for which the baseline version took 100 seconds and the enhanced one took $100 \cdot 2 = 200$ seconds. Hence, points with positive y 's always represent benchmarks for which the enhancement was indeed faster. Note the logarithmic scale on the x axis and that, for each plot, the scale on the y may be different. For the top-left scatter plot in Figure 1, the baseline system (*original counter*) corresponds to a variant of ROUNDINGSAT that uses counter-based propagation for all constraints except for cardinality constraints and clauses.

Revisiting Garbage Collection Frequency. In order to evaluate the impact of each of the improvements we present, we will always consider as a baseline the system that implements all previous modifications. Hence, we now consider as the reference a system where most information has been moved out of the Constraint class and we assess what is the gain obtained by adding the new improvement we present next.

Whenever a cleanup is performed, ROUNDINGSAT does the following: the set of constraints is traversed and some of them are marked as deleted, changing the bit allocated for this inside *slackMM*. In this case, a variable that contains the amount of memory that is occupied by deleted constraints is increased. If this variable exceeds a certain threshold, garbage collection is applied: all surviving constraints are reallocated contiguously in the database and watch lists are rebuilt with no reference to the deleted constraints.

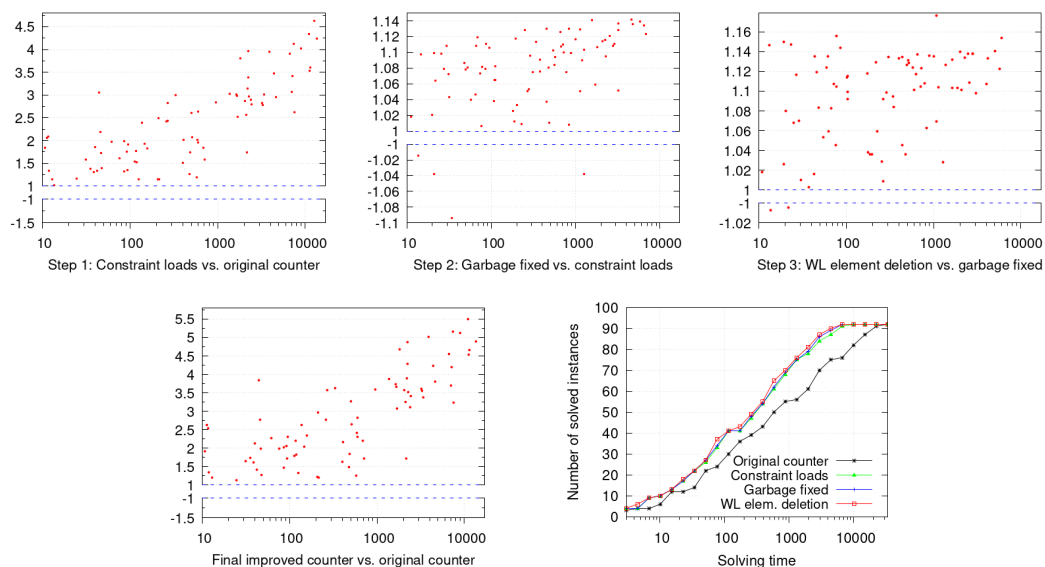
In our opinion, this has two unwanted effects. First of all, watch lists are larger than strictly necessary, and this can slow down the system. Secondly, the constraints database is not as compact as possible and accesses to constraints will be more costly. We suggest to apply garbage collection in every cleanup. This would probably not pay off in a system that applies cleanups extremely often, because the extra cost of applying the garbage collection procedure at every cleanup would exceed the speedups obtained during propagation. Since after this change there will be no deleted constraints in any watch list, it is no longer necessary to use one bit of *slackMM* to store this information. This will slightly speed up the access to the slack information, that will now occupy all 64 bits of *slackMM*.

As the middle scatter plot in the first row of Figure 1 shows, this modification has beneficial effects in almost all benchmarks. This is obviously a less remarkable improvement than the first one, but our novel methodology has allowed us to detect that this indeed enhances the runtime and, moreover, is extremely easy to implement.

Deletion of Elements in Watch Lists During Propagation. When performing propagation in a counter-based system, no PB constraint is ever deleted from any watch list. This is because all literals in every constraint are always “watched”. However, let us remember that watch lists also contain clauses, for which the two-watched literal scheme is used. In this method, literals are sometimes unwatched, and this forces the corresponding element in the watch list to be deleted. The same phenomenon happens with watched-literal extensions for propagating cardinality constraints.

In order to use contiguous memory, watch lists in ROUNDINGSAT, and in state-of-the-art SAT solvers, are implemented as vectors. This makes it difficult to delete an element which is not the last one. In ROUNDINGSAT, this is done by copying the last element of the vector to the position of the element to be deleted, and then removing the last element. However, in SAT solvers like MINISAT, CADICAL and KISSAT, another solution is proposed.

22:10 Speeding up Pseudo-Boolean Propagation



■ **Figure 1** Impact of several improvements on counter-based propagation.

The idea is to read the watch list and, at the same time, write in the same vector the list that will result after the deletions have been performed. This is possible by keeping two pointers, one pointing to the position we are reading, and the other one to the position where we are writing. Since the first will always be to the right (or at the same position) of the second one, both things can be done at the same time without the two processes interfering with each other.

In terms of memory access this latter mechanism is better than the solution used in ROUNDINGSAT which, in the presence of long lists, often accesses positions in the vector that are very far away. This is what happens in counter-based propagation, where the fact that all literals are watched creates watch lists that are very long. On the other hand, one could argue that the solution used in SAT solvers incurs in some overhead due to performing a larger number of write operations. This, for example, would occur in a list where only the first element has to be deleted: ROUNDINGSAT would only perform one write operation whereas the other solution has as many writes as elements in the vector.

The rightmost plot in the first row of Figure 1 shows that implementing the SAT approach to deleting elements on top of the system that incorporates the two previous adjustments results in improvements in almost all benchmarks. Again, gains are limited. However, the addition of the last two small enhancements has some positive impact. This can be seen by comparing the leftmost scatter plot in the second row (a comparison between the original ROUNDINGSAT system and the one with all improvements) and the leftmost plot in the first row (a comparison between the original ROUNDINGSAT versus the system that only incorporates the first major improvement). Note that the distribution of the points is very similar, but the y scale is larger.

All this is summarized in the cactus plot of Figure 1. Note that there was no time limit in this experiment: we let all systems run until they processed the entire log. This is why the cactus plot shows that all systems end up processing all benchmarks. However, if a more strict time limit was given, the performance of the system with all improvements outperforms the system with only the first one, which is of course the major contributor to the overall speedup of the system.

■ **Algorithm 3** Watch-based propagation procedure.

```

1 Function Propagate-Watch(Watch w):
2   Constraint ctr := w.ctrPr
3   if isDeleted(ctr) then return
4   if not isPBWatched(ctr) then return
5   coef := ctr[w.idx].coef
6   maxCoef := ctr[0].coef
7   ctr.wslack := ctr.wslack - coef
8   i := 0
9   while i < ctr.size and ctr.wslack < maxCoef do
10    Lit l := ctr[i].lit
11    if not isFalse(l) and not ctr[i].watched then
12      ctr[i].watched := true
13      wlist(l) := wlist(l) ∪ {ctr, i}
14      ctr.wslack := ctr.wslack + ctr[i].coef
15    i := i + 1
16  if ctr.wslack ≥ maxCoef then
17    ctr[w.idx].watched := false
18    wlist(ctr[w.idx].lit) := wlist(ctr[idx].lit) \ {ctr, w.idx}
19    return OK
20  if ctr.wslack < 0 then return CONFLICT
21  j := 0
22  while j < ctr.size and ctr.wslack < ctr[j].coef do
23    if isUndef(ctr[j].lit) then propagate(ctr[j].lit)
24    j := j + 1
25  return OK

```

5 Watch-Based Propagation

Watch-based propagation can be seen as a refinement of counter-based propagation. The idea is to associate to each constraint a set of watched literals $watches(C)$. In ROUNDINGSAT, for each literal $l \in watches(C)$, an entry of type $Watch = \langle ctrPtr, idx \rangle$ is added to $wlist(l)$, and only when l becomes false we will traverse $wlist(C)$, and hence C , checking for a propagation or a conflict. In counter-based propagation, $watches(C)$ consists of all literals in C and hence, whenever any literal in C becomes false a visit to the constraint is triggered. In watch-based propagation, we will do so only when some literal in $watches(C)$ becomes false. Hence, a reduction in the size of $watches(C)$ will reduce the amount of work to be done.

Following the notation in [14], if we define $watchslack(C, \rho) = (\sum_{i: \bar{l}_i \notin \rho} c_i) - d$ we can easily see that $watchslack(C, \rho) \leq slack(C, \rho)$ and hence if $watchslack(C, \rho) \geq maxCoef(C)$ we can guarantee that C is neither conflicting nor propagating. Hence our goal is to watch a small set of literals for which the previous inequality holds. Note also that $watchslack(C, \rho)$ only changes if some $l \in watches(C)$ becomes false. All these remarks are the core of watch-based propagation routine in ROUNDINGSAT that we show in Algorithm 3.

It starts by decrementing the $watchslack$ in line 7. If, after that, it is still larger than the maximum coefficient the algorithm skips the loop in lines 9–14 and executes lines 16–17: the literal that became false is removed from the $watches$ set by marking it as unwatched in ctr and removing it from the corresponding watch list. Otherwise, the loop in lines 9–14 tries to extend the set $watches$ so that the $watchslack$ is larger than the maximum coefficient. If this is possible, lines 16–17 are executed. Otherwise, if the $watchslack$ is negative, a conflict is declared in line 19. Finally, if $watchslack$ is not larger than the maximum coefficient, but it

22:12 Speeding up Pseudo-Boolean Propagation

is not negative, a propagation might be possible. This is checked in lines 20–23 by traversing the constraint and looking for literals to be propagated. For a more detailed description and theoretical justification, we refer the reader to [14].

Again, this is a slightly simplified version of what is done in ROUNDINGSAT. In particular, the same optimization that we mentioned in counter-based propagation that allows one to not always start the loop in lines 20–23 at position 0 is applied. Also, if the function is called on a certain constraint and the loop in lines 9–14 cannot extend *watches* so that *watchslack* is larger than the maximum coefficient, we know that in any subsequent call to this routine on the same constraint that is made with no backjump in between, we can skip this loop because it will not succeed either. These two optimizations are present in all our modifications of the algorithm.

Minimizing the Number of Constraint Loads. As we mentioned in the previous section, our aim is to reduce the number of times when a constraint is loaded into memory, because it is a very time-consuming task. If we analyze the code, we can see that, if after decreasing the *watchslack* it is still large enough, we have to unmark the coefficient, which is information stored in the constraint. In particular, ROUNDINGSAT uses the sign of the coefficient (which we know are always positive) to indicate whether the literal is watched or not. Otherwise, if we have to extend the set of *watches* we obviously have to load the constraint into memory as well. Hence, there does not seem to be too much room for reducing the number of constraint loads.

One could try to develop a sophisticated way to store outside of the constraint object the information about who is watched in every constraint. That would allow us to avoid loading the constraint when the *watchslack* is large enough even after we decrease it. Our experimental analysis showed that this situation happens in around 20% of the cases. Hence, we believe there is still some space for improvement by a major modification of the propagation scheme.

Despite our initial analysis of this improvement was not very optimistic, our results show that, by applying the same modification we described for counter-based propagation we still obtain propagation speed-ups, as the leftmost plot in the first row of Figure 2 shows. This is due to at least three reasons. First of all, for deleted constraints we do not have to load the constraint into memory. Secondly, checking the type of constraint that the *Watch* element contains, in order to decide which propagation routine should be called, does not require loading the constraint. This is beneficial, for example, in constraints like clauses where the propagation mechanism does not require to always load the constraint into memory. Finally, backjumping traverses watch lists and updates the *watchslack* of the constraints appearing in them. Having moved this information outside of the constraint prevents the system from loading all these constraints into memory.

Revisiting Garbage Collection Frequency. Note that, as we did with counter-based propagation, we evaluate improvements by comparing them with a system that already incorporates all previously proposed adjustments. As it happened with counter-based propagation, incrementing the garbage collection frequency only produces some moderate improvements. The center plot in the first row of Figure 2 shows that the improvements are even more limited than with counter-based propagation. The most plausible reason is that, since deleted constraints only appear in the watch lists of its watched literals, which might be a very small subset of them, the number of elements marked for deletion in these lists is not large enough to be a problematic issue.

Deletion of Elements in Watch Lists During Propagation. It was unexpected to us that, as the rightmost plot in the first row of Figure 2 indicates, modifying the deletion of elements in watch lists to mimic the way it is done in SAT solvers slowed down the propagation procedure. However, a more careful analysis allowed us to realize that this should not have been such a big surprise.

The reason for this behavior is the length of the watch lists. As we mentioned, the original way of ROUNDINGSAT to delete elements in watch lists accesses the last element of the list, and this can be harmful due to memory access issues if the list is very long. For counters, the average watch list length over the benchmarks we are considering was 2365, and the median 564. For watch-based propagation, this figure was much lower: 203 on average and a median of 71. Hence, our concerns about having too long lists do not apply in watch-based propagation. However, since our goal is to develop a hybrid approach where some constraints will be watched and some others will be counter-based, we will keep this modification.

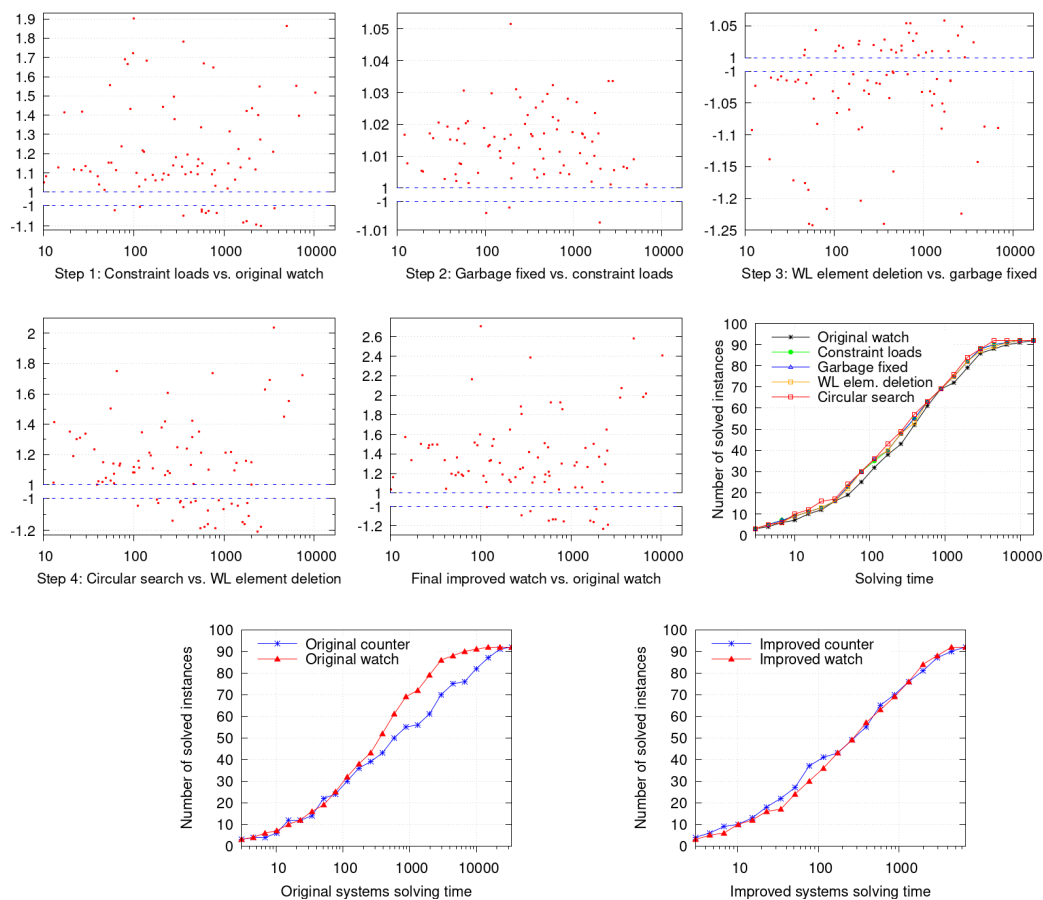
Circular Search for Watched Literals. In [20], a simple but effective improvement on the two-watched literal scheme for propagation in SAT solvers was described. The idea is that, whenever the clause is traversed in order to find another literal to watch, instead of starting from the beginning of the clause, as it was usually done, we search for it in a *circular* way. More precisely, the novel procedure stores with every clause the position in which the last search for new literals to be watched stopped. Next time the same operation is performed, the search starts from that position instead of from the beginning.

The reason why that is beneficial is that it is known that one prefers to watch literals that are inactive, i.e., that are rarely added to the assignment, because they cause almost no work in propagation. By searching in a circular way, all literals have the same chances to be watched and hence, if inactive literals exist, it will be more likely that the procedure ends up watching them.

As the leftmost plot in the second row of Figure 2 shows, this also pays off in the case of watch-based propagation for PB constraints. Note that, since we only wanted to evaluate the impact of this modification in the propagation of PB constraints, it has not been incorporated to the specialized propagation procedures for clauses or cardinality constraints.

A summary on the impact of all these improvements can be seen in the center scatter plot in the second row of Figure 2, where quite a consistent improvement can be observed. We want to remark that these are important speedups. We have to consider that we did not change the algorithmic nature of the propagation algorithms, but rather modified implementation details of them. Since we execute all systems on the same log, the search space is the same and we cannot expect exponential improvements in time.

Once we have developed our improved version of counter and watch-based propagation, it is worth comparing them and analyze whether one of them dominates the other. This is done in the last row of Figure 2 where, in the leftmost plot, we compare the performance of the two original methods in ROUNDINGSAT. It is clear that watch-based propagation outperformed counters. This was already mentioned in [14], but our more precise evaluation methodology allows us to conclude that the difference is probably larger than what it was reported in that paper. The analysis for the improved versions, in the rightmost plot of the same figure, reveals that they are quite similar. Since we now have two methods that are comparable, it makes even more sense to try to combine them in a hybrid procedure that determines which propagation method to use for each constraint.



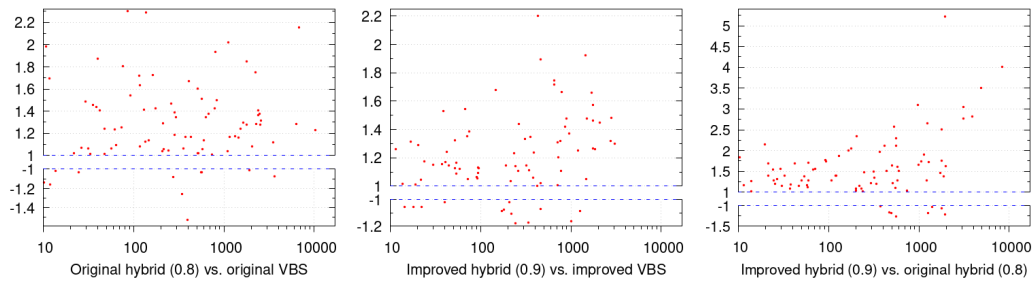
■ **Figure 2** First two rows show the impact of several improvements on watch-based propagation. Last row compares, for the original and the improved versions, the performance of counter with respect to watched-based propagation.

6 A Hybrid Pseudo-Boolean Propagation Approach

The idea of using a *hybrid* approach where the propagation method is decided per constraint is not a new one. Already in [11] it was mentioned that “Another experiment assessed the performance of a hybrid BCP scheme. [...] The hybrid scheme was not found to be superior to the simple counter scheme.” Unfortunately, the results of that experiment are not reported in the paper. In [14] this possibility is left as future work.

However, ROUNDINGSAT indeed incorporates a hybrid propagation mechanism: whenever a constraint is added to the database, it is first sorted from largest to smallest coefficient. After that, the smallest prefix of the constraint whose *watchslack* is larger than the maximum coefficient is computed. If the percentage of literals not in this prefix (the literals that will not be watched) is larger than a predefined threshold, watches are used. Hence, the larger the threshold, the larger the number of constraints for which counter-based propagation will be used. The rationale is that watches are preferred over counters when a sufficiently small percentage of the literals are watched.

The default version in ROUNDINGSAT uses watch-based propagation. However, one can instruct the solver to use the hybrid propagation mechanism, but a concrete percentage threshold has to be given. An extensive evaluation allowed us to conclude that 0.8 was the best threshold when using this hybrid method on top of the original ROUNDINGSAT



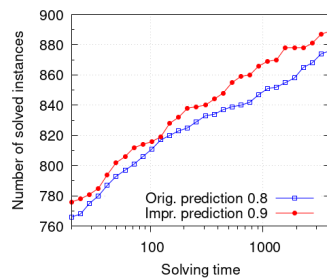
■ **Figure 3** Comparison between different hybrid methods. Leftmost plot uses the original counter and watch-based propagation routines, whereas the center plot uses the improved ones. Rightmost plot compares a hybrid method on top of them.

propagation routines. The leftmost plot in Figure 3 compares this method with a Virtual Best Solver (VBS) that, for each benchmark, selects the best possibility between using counters or watches for all constraints. When using the hybrid approach on top of our improved propagation routines, 0.9 was the best threshold and a comparison with the improved VBS is in the center plot. We want to remark that improving upon a VBS is always a challenging task.

It might seem that 0.8 and 0.9 are very large values that leave no possibility for constraints to be watched. This is definitely not the case. In the original system with threshold 0.8 the median of the percentage of constraints that are watched was 72%, whereas it was 56% in the improved version with threshold 0.9. Since our improvements had a larger impact on counter-based propagation, it is not a surprise that the best improved hybrid version uses counter-based propagation more often than the best original hybrid version.

Finally, we compare in the rightmost plot of Figure 3 the best hybrid methods that use the original and the improved propagation mechanisms, respectively. We can see that, not surprisingly, our improvements on these procedures also result in an improvement on the corresponding hybrid methods.

The last contribution of this paper is to determine whether the improvements we have shown in propagation speed translate into improvements on the overall runtime of ROUNDINGSAT. That is, we now run ROUNDINGSAT without reading the logs, but rather as a complete PB solver that performs conflict analysis, cleanups, restarts, etc., in an autonomous way, without relying on an external oracle. In Figure 4 we compare the best system using the improved propagation routines, which is the hybrid approach with threshold 0.9 with the best system using the original propagation routines, which is the hybrid approach with threshold 0.8. Results are on all 1600 benchmarks in the category OPT-SMALLINT-LIN with a time limit of 3600 seconds. We believe that the plot is very clear and leaves no doubt that remarkable speedups are obtained thanks to the research presented in this paper.



■ **Figure 4** Hybrid method on original and improved propagation. No logs are used.

7 Conclusions and Future Work

We have presented a novel methodology to evaluate propagation procedures and applied it to improve the implementation of these routines inside ROUNDINGSAT, currently the fastest CDCL-based PB solver. This has resulted in important speedups in performance.

As future work, we plan to precisely analyze the impact of maintaining (an upper bound on) the maximum coefficient of *undefined* literals in order to have a more precise filter for propagation. Also, computing slacks with respect to the whole assignment, instead of only considering the propagated literals is part of our future work. Additionally, we want to develop hybrid methods that choose between counter or watched-based propagation by dynamically evaluating how active the literals in a constraint are, as opposed to only focusing on their coefficients.

References

- 1 RoundingSAT web page. <https://gitlab.com/MIAOresearch/software/roundingsat>. Accessed: 2024-03-11.
- 2 Roberto Javier Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and maxsat. *Ann. Oper. Res.*, 218(1):71–91, 2014.
- 3 Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Kareem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ICCAD '02*, pages 450–457, New York, NY, USA, 2002. ACM. doi:10.1145/774572.774638.
- 4 Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011. doi:10.1613/jair.3152.
- 5 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010. URL: http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_4_LeBerre.pdf, doi:10.3233/SAT190075.
- 6 Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008. URL: http://jsat.ewi.tudelft.nl/content/volume4/JSAT4_5_Biere.pdf.
- 7 Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2010. Technical Report 10/1, August 2010, FMV Reports Series.
- 8 Armin Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 of *EPiC Series in Computing*, page 88. EasyChair, 2014. doi:10.29007/JHD7.
- 9 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froylyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
- 10 Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018. doi:10.1007/978-3-319-10575-8_10.
- 11 Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(3):305–317, 2005.
- 12 Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *J. Satisf. Boolean Model. Comput.*, 6(1-3):99–120, 2009. doi:10.3233/SAT190064.

- 13 W. Cook, C. Coullard, and Gy. Turan. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18:25–38, 1987.
- 14 Jo Devriendt. Watched propagation of 0-1 integer linear constraints. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2020. doi:10.1007/978-3-030-58475-7_10.
- 15 Jo Devriendt, Ambros M. Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-boolean conflict-driven search. *Constraints An Int. J.*, 26(1):26–55, 2021. doi:10.1007/S10601-020-09318-X.
- 16 Jo Devriendt, Stephan Gocht, Emir Demirovic, Jakob Nordström, and Peter J. Stuckey. Cutting to the core of pseudo-boolean optimization: Combining core-guided search with cutting planes reasoning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3750–3758. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16492.
- 17 Julian Dolby, Mandana Vaziri, and Frank Tip. Finding Bugs Efficiently With a SAT Solver. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 195–204, 2007. doi:10.1145/1287624.1287653.
- 18 Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference on Theory and Applications of Satisfiability Testing, SAT '03*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.
- 19 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/ijcai.2018/180.
- 20 Ian P. Gent. Optimal implementation of watched literals and more general techniques. *J. Artif. Intell. Res.*, 48:231–251, 2013. doi:10.1613/JAIR.4016.
- 21 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2 & 3):297–308, 1985.
- 22 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.
- 23 Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 519–534. Springer, 2010. doi:10.1007/978-3-642-16242-8_37.
- 24 Norbert Manthey and Ari Saptawijaya. Towards improving the resource usage of sat-solvers. In Daniel Le Berre, editor, *POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010*, volume 8 of *EPiC Series in Computing*, pages 28–40. EasyChair, 2010. doi:10.29007/3VWV.
- 25 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.
- 26 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference, DAC '01*, pages 530–535, New York, NY, USA, 2001. ACM.
- 27 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011. doi:10.1016/j.artint.2010.10.002.

- 28 Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in AI and Applications*, pages 695–733. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-695.
- 29 Hossein M. Sheini and Karem A. Sakallah. Pueblo: A modern pseudo-boolean SAT solver. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 684–685. IEEE Computer Society, 2005. doi:10.1109/DATE.2005.246.
- 30 Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *JSAT*, 2(1-4):165–189, 2006.
- 31 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.
- 32 Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005*, pages 139–143, 2005. doi:10.1007/11513988_13.
- 33 L. Zhang and S. Malik. Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference on Theory and Applications of Satisfiability Testing, SAT '03*, volume 2919 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2004.