On PB Encodings for Constraint Problems

Thibault Falque, Romain Wallon

On PB Encodings for Constraint Problems

Thibault Falque
Exakis Nelite
CRIL, Univ Artois & CNRS

Romain Wallon
CRIL, Univ Artois & CNRS

Abstract

One of the possible approaches for solving a CSP is to encode the input problem into a CNF formula, and then use a SAT solver to solve it. The main advantage of this technique is that it allows to benefit from the practical efficiency of modern SAT solvers, based on the CDCL architecture. However, the reasoning power of SAT solvers is somehow "weak", as it is limited by that of the resolution proof system they use internally. This observation led to the development of so called pseudo-Boolean (PB) solvers, that implement the stronger cutting planes proof system, along with many of the solving techniques inherited from SAT solvers. Additionally, PB solvers can natively reason on PB constraints, i.e., linear equalities or inequalities over Boolean variables. These constraints are more succinct than clauses, so that a single PB constraint can represent exponentially many clauses. In this paper, we leverage both this succinctness and the reasoning power of PB solvers to solve CSPs by designing PB encodings for different common constraints, and feeding them into PB solvers to compare their performance with that of existing CP solvers.

2012 ACM Subject Classification Artificial intelligence

Keywords and phrases constraint programming, PB solving, SAT encodings

Digital Object Identifier 10.4230/LIPIcs.DPCP.2022.

1 Introduction

The *Constraint Satisfaction Problem* (CSP) aims at determining whether a set of constraints is consistent and, when it is, finding a solution that satisfies all the constraints. To solve such problems, several approaches have been proposed [17]. One of these is to natively deal with the constraints of the problem, using efficient data structures to represent both the constraints and the structure of the problem, as it is done for instance in `Choco` [22], `Nacre` [10] and `ACE`[1]. Another possible approach is to leverage the practical efficiency of modern SAT solvers, based on the CDCL architecture [19, 20, 7], to solve CSPs. As these solvers take as input a propositional formula in *Conjunctive Normal Form* (CNF), one needs to *encode* the variables (especially their domains) and the constraints of the original problem into clauses to use them. In this context, many encodings have been proposed, as those described in [24, 9, 5, 2, 23], which often provide good empirical results.

However, the main weakness of SAT solvers is the resolution proof system they use during conflict analysis. This proof system has a weak inference power, and some apparently simple problems cannot be solved efficiently. In particular, SAT solvers are known to perform poorly on instances requiring the ability to "count". This is for example the case for the well-known *pigeonhole principle* problem, which requires an exponential number of resolution steps to derive the unsatifiability of the input [12]. This observation led to the development of a different kind of solvers, called *pseudo-Boolean* (PB) solvers [6, 15, 8]. These solvers inherit many features of modern SAT solvers, and implement a proof system that is stronger than the

[1] `https://github.com/xcsp3team/ace`

resolution proof system, and which is known as the *cutting planes* proof system [11, 13, 21]. These solvers can natively deal with PB constraints, i.e., linear equations or inequations over Boolean variables. This is an interesting observation, as among existing SAT encodings of CSPs, many of them actually use intermediate PB representations of the constraints before encoding them into clauses. This step is necessary to use a SAT solver, but it requires both to introduce additional variables and to increase the number of constraints to give to the solver (a single PB constraints can represent exponentially many clauses [1]).

In this paper, we introduce new encodings for CSPs leveraging both the succinctness of PB constraints and the inference power of PB solvers. The proposed encodings are based on well-known representations of variable domains using Boolean variables, such as the *direct-encoding*, the *log-encoding* or the *order-encoding*, to encode popular CSP constraints into PB constraints. To this end, we first introduce some preliminaries about CSP and PB solving. We then formally describe the proposed encodings for different constraints, and empirically evaluate them on different sets of instances.

2 Preliminaries

2.1 Constraint Programming

A *constraint network* (CN) is composed of a finite set of variables and a finite set of constraints. Each variable X takes its value in a finite set called *domain* of X, denoted $\texttt{dom}(X)$. Each constraint defines a relation on a set of variables. A *solution* of a CN is an assignment of values to all its variables such that all the constraints of the CN are satisfied. A CN is said to be *consistent* if it has at least one solution, and the corresponding decision problem, called *Constraint Satisfaction Problem* (CSP), is to determine whether a CN is consistent.

2.2 SAT Solvers and CSPs

A variable x is *Boolean* when $\texttt{dom}(X) = \{0, 1\}$. We call a *literal* ℓ a Boolean variable x or its negation $\bar{x} = 1 - x$. A literal ℓ is *satisfied* when ℓ is assigned to 1, and *falsified* otherwise. A clause is a disjunction of literals, requiring at least one of its literals to be satisified. A problem is in *Conjunctive Normal Form* (CNF) when it is a conjunction of clauses. The *SATisfiability problem* (SAT) is to determine whether such a conjunction is consistent.

The SAT problem is the first problem that has been proven to be NP-complete [4]. It is thus possible to use SAT solvers to solve CSPs, using different encodings. In particular, to represent the domain of a CSP variable X, one can use the so-called *direct-encoding* (see, e.g., [24]). It defines a Boolean variable x_v for each value $v \in \texttt{dom}(X)$. In this case, the value assigned to X may be retrieved by identifying the (only) Boolean variable x_v to be satisfied.

This representation is particulary useful in the case where the domain of a variable is an enumerated set of values. When it is a range of values, another option is to represent the variable using the *log-encoding* (see, e.g., [24]), which uses the binary representation of the variable X using Boolean variables b_i representing the bits of X, so that $X = \min(\texttt{dom}(X)) + \sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i$. Observe here the use of $\min(\texttt{dom}(X))$, which guarantees that the binary decomposition always starts at 0.

Finally, another approach is that based on the *order-encoding* [23], which defines a Boolean variable $x_{\geq v}$ for each value $v \in \texttt{dom}(X) \setminus \{\min(\texttt{dom}(X))\}$. This variable is satisfied if and only if $X \geq v$. In this case, the value assigned to X can be retrieved by identifying two variables $x_{\geq v}$ and $x_{\geq v+1}$ such that the former is satisfied and the latter is falsified, in which case X is assigned to v.

2.3 Pseudo-Boolean (PB) Constraints

A *pseudo-Boolean* (PB) constraint is a constraint of the form $\sum_{i=1}^{n} \alpha_i \ell_i \triangle \delta$, where n is a positive integer, the *weights* (or *coefficients*) α_i and the *degree* δ are integers, ℓ_i are literals and $\triangle \in \{<, \leq, =, \geq, >\}$. A PB constraint is said to be *normalized* when all the coefficients and the degree of this constraint are positive, and $\triangle$ is $\geq$. It is well known that any PB constraint may be rewritten as a conjunction of normalized PB constraints, which is particularly useful for the encodings we present later on. A *PB cardinality constraint* is a normalized PB constraint in which all the coefficients are equal to 1, and a *clause* is a PB cardinality constraint with its degree equal to 1. This definition is equivalent to the definition of clauses as disjunctions of literals, and shows that PB solvers generalize SAT solvers.

3 Purely PB Encodings

In this section, we use the *direct-encoding*, *log-encoding* and *order-encoding* to represent the domains of the variables from a CN, and use these representations to encode different common CSP constraints into PB constraints. In particular, we focus on counting constraints and on constraints allowed for the *mini-solvers* track of the XCSP competition.

3.1 (De)activating PB constraints

To encode both the domain of the variables and the constraints of a CSP into PB constraints, it is often needed to activate (or deactivate) a constraint. To do so, a common practice is to introduce *selectors*, i.e., a fresh variables s such that its satisfaction entails that of the considered constraint. In the case of PB constraints, such a selector s could have the following semantics, using $\Rightarrow$ to denote material implication: $s \Rightarrow \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$. The particular form of PB constraints allows to concisely represent such an implication with the (single) PB constraint $\delta \bar{s} + \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$.

Recall that, in this case, the satisfaction of the constraint does not guarantee the satisfaction of s. If such a guarantee is needed, we must add the reciprocal implication, i.e., $s \Leftarrow \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$, which can be represented using single PB constraint $(\sum_{i=1}^{n} \alpha_i - \delta + 1)\, s + \sum_{i=1}^{n} \alpha_i \bar{\ell_i} \geq \sum_{i=1}^{n} \alpha_i - \delta + 1$.

From now on, we denote by s a selector for which only one implication is defined, and by $\mathcal{S}$ a selector for which both implications are defined (when needed, indices may be added to these selectors).

To illustrate the use of selectors, let us consider the constraint $\sum_{i=1}^{n} \alpha_i \ell_i \neq \delta$. This constraint cannot be normalized directly, as $\neq$ is not an allowed operator for a PB constraint. However, we can oberve that this constraint is equivalent to the disjunction of the two constraints $(\sum_{i=1}^{n} \alpha_i \ell_i \leq \delta - 1)$ and $(\sum_{i=1}^{n} \alpha_i \ell_i \geq \delta + 1)$. Let us define two selectors $s_{\leq}$ and $s_{\geq}$, such that $s_{\leq} \Rightarrow \sum_{i=1}^{n} \alpha_i \ell_i \leq \delta - 1$ and $s_{\geq} \Rightarrow \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta + 1$. These two constraints, combined with the disjunction $s_{\leq} \vee s_{\geq}$, allow to represent the constraint above using PB constraints.

3.2 Variables and Domains

In order to make sure that the encodings of the variables presented in the previous section effectively encode the variables of the original problems, some constraints must be added. In the case of the *direct-encoding*, one simply needs to add the constraint $\sum_{v \in \mathtt{dom}(X)} x_v = 1$ to make sure that the variable X is assigned exactly one value. It is then possible to represent the variable X using the equality $X = \sum_{v \in \mathtt{dom}(X)} v x_v$.

In the case of the *log-encoding*, the value of X is given by the equality $X = \min(\texttt{dom}(X)) + \sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i$. To make sure that the domain of X is correct, one can use the constraint $\sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i \leq \max(\texttt{dom}(X)) - \min(\texttt{dom}(X))$ (recall that we only use this encoding to encode domains that are intervals).

Finally, in the case of the *order-encoding*, the constraints to add are exactly the same clauses as those used in [23]. Thus, for each value $v \in \texttt{dom}(X) \backslash \{\min(\texttt{dom}(X))\}$, the implication $x_{\geq v} \Rightarrow x_{\geq v-1}$ is added to the solver. When the domain of X is not an interval, it is possible to forbid a value v by adding the implication $x_{\geq v} \Rightarrow x_{\geq v+1}$. Additionally, it is possible to represent the value of X using the equality $X = \min(\texttt{dom}(X)) + \sum_{v \in \texttt{dom}(X) \backslash \{\min(\texttt{dom}(X))\}} x_{\geq v}$. As for the *log-encoding*, let us remark the use of $\min(\texttt{dom}(X))$ to make sure that the encoding starts at 0. Without loss of generality, we can thus always represent a CSP variable X using a weighted sum of literals, to which a constant μ may be added, giving $X = \mu + \sum_{i=1}^{n} \alpha_i \ell_i$.

Additionally, encoding CSP constraints often requires to determine whether a variable X is assigned to a given value v. In the case of the *direct-encoding*, one only needs to check the value of x_v. To obtain such a value with the *order-encoding*, note first that X is assigned to v if and only if the conjunction $x_{\geq v} \wedge \overline{x_{\geq v+1}}$ is satisfied. To obtain a variable x_v equivalent to that used in the *direct-encoding*, one just needs to use the constraint $x_v \Leftrightarrow x_{\geq v} \wedge \overline{x_{\geq v+1}}$, which can be encoded using PB constraints, as shown in the previous section. Obtaining such a value is a little bit harder with the *log-encoding*, as one needs to check the assignment of *all* the Boolean variables used in the representation of the variable to know the value of X. In this case, we thus propose to use a lazy form of the *direct-encoding*, where X is first encoded using the *log-encoding*, and the *direct-encoding* is used only when x_v is required. The constraint $\sum_{v \in \texttt{dom}(X)} v x_v = \min(\texttt{dom}(X)) + \sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i$ may be used to make sure that both encodings encode the same value. It is thus possible to obtain, for each variable X and for each value $v \in \texttt{dom}(X)$, a unique Boolean variable x_v representing the assignement $X = v$.

3.3 Constraint `cardinality`

As mentioned before, one of the main advantages of PB solvers compared to SAT solvers is their ability to count efficiently. To benefit from this advantage, we first propose to encode `cardinality` constraints.

A first variant of this constraint is to force bounds on the number of variables among $\{X_1, \ldots, X_N\}$ that are assigned a given value v. Let m and M be the minimum and maximum number of variables X_i that can be assigned to v, respectively, and let x_v^i be the Boolean variable representing the assignment $X_i = v$. Clearly, the number of satisfied x_v^i must be between m and M. Thus, this `cardinality` constraint can be represented with $m \leq \sum_{i=1}^{N} x_v^i \leq M$, which can easily be decomposed into two PB constraints. Let us remark that, depending on the values of m and M, these two constraints may represent an exponential number of clauses without requiring the use of auxiliary variables [1], as SAT solvers would.

Another variant of the `cardinality` constraint is to make sure that a variable C is assigned to the number of variables X_i that are assigned to a given value v. Let $\mu + \sum_{i=1}^{n} \alpha_i \ell_i$ be the representation of the variable C. This variable must be equal to the number of satisfied x_v^i. So, if we use the same notations as before, this can be encoded using the equality $\sum_{i=1}^{N} x_v^i = \mu + \sum_{i=1}^{n} \alpha_i \ell_i$, which is equivalent to the PB constraint $\sum_{i=1}^{N} x_v^i - \mu - \sum_{i=1}^{n} \alpha_i \ell_i = 0$.

There exists other variants of the constraints described here, where the values are variables Z instead of constants. To encode them, one simply needs to replace the variables x_v^i by

variables x^i_Z, such that $x^i_Z \Leftrightarrow (X_i - Z = 0)$. By representing X_i and Z into weighted sums of literals, this constraint may be normalized following the procedure described in Section 3.1.

Let us note that, as for `cardinality` constraints, PB encodings may also be used to represent other counting constraints, such as `count` or `nValues`. They are omitted in this paper for space reasons, and also because we did not find any XCSP instance that uses these constraints, and we thus cannot make any conclusion on these encodings for the moment.

3.4 Constraints from the *Mini-Solvers* Track

There exist many different types of constraints, but not all of them are supported by our approach, as encodings remain to be found. In order to still solve a wide variety of problems, we now focus on the constraints allowed for the *mini-solvers* track of the XCSP'19 competition, which are often sufficient for that purpose.

3.4.1 Constraint `sum`

A `sum` constraint is a constraint of the form $\sum_{i=1}^{N} A_i X_i \odot k$, where $\odot \in \{<, \leq, =, \neq, \geq, <\}$. It is clear that such a constraint can easily be represented using PB constraints. Indeed, if each X_i may be written as $X_i = \mu^i + \sum_{j=1}^{n_i} \alpha^i_j x^i_j$, where x^i_j are Boolean variables, then the constraint above is equivalent to $\sum_{i=1}^{N} A_i \left(\mu^i + \sum_{j=1}^{n_i} \alpha^i_j x^i_j\right) \odot k$. which can be developed as a PB constraint (the case of the operator $\neq$ is treated as in Section 3.1).

3.4.2 Constraint `allDifferent`

The global constraint `allDifferent` enforces that a set of variables are all assigned to different values We illustrate its encoding following the example `allDifferent`$(X_1, \ldots, X_n)$.

Let $\mathcal{D} = \bigcup_{i=1}^{N} \texttt{dom}(X_i)$. The semantics of `allDifferent` enforces that each value $v \in \mathcal{D}$ is used at most once among all variables X_i. Let $v \in \mathcal{D}$, and let us note x^i_v the Boolean variable representing $X_i = v$ ($i \in 1..N$). We can represent this constraint on v using the PB constraint $\sum^{N}_{\substack{i=1 \\ v \in \texttt{dom}(X_i)}} x^i_v \leq 1$. This constraint has to be applied on all values $v \in \mathcal{D}$. The operation must thus be repeated on each possible value.

Let us observe that this encoding is similar to that of the *pigeonhole-principle*, which is hard for SAT solvers based on resolution [12]. This encoding allows thus to benefit from the reasoning power of PB solvers, at least on the subset of constraints encoding `allDifferent`.

3.4.3 Constraint `extension` with Supports

A *support* is a constraint that explicitly lists all the possible solutions for a constraint. To encode supports, let us first remark that a PB constraint of the form $\sum_{i=1}^{n} \ell_i \geq n$ represents the conjunction of the literals ℓ_i, i.e., $\bigwedge_{i=1}^{n} \ell_i$. Based on this observation, we can encode a (unique) support of the form $(X_i \mid 1 \leq i \leq N) = (v_i \mid 1 \leq i \leq N)$ using the PB constraint $\sum_{i=1}^{N} x^i_{v_i} \geq N$, where $x^i_{v_i}$ is the Boolean variable representing the assignment $X_i = v_i$ ($i \in 1..N$). When several tuples t are allowed in a support, we need to add a selector s_t to each constraint associated to a tuple, giving the PB constraint $s_t \Rightarrow \sum_{i=1}^{N} x^i_{v_i} \geq N$. Indeed, the set of allowed tuples is a disjunction, so one of the tuples must be assigned to the associated variables: we thus add the clause $\bigvee s_t$ to encode the full support.

Finally, let us remark that, if the symbol $*$ is used in one of the allowed tuples instead of a value v_i (to represent that the variable X_i can take any value), one just need to ignore the literal corresponding to the variable X_i in the constraint above.

4 Experimental results

This section presents some experimental results of the PB encodings presented in the previous section on different sets of (decision) instances. To evaluate the performance of our approach, we implemented the encodings in the solver `Sat4j`[2] [15], and we executed different variants of this solver, denoted `Sat4j` + $\mathcal{S}$ in the rest of this section, where $\mathcal{S}$ is the name of the considered variant. Unless otherwise specified, the combination of the *direct-encoding* and *log-encoding* is used to represent the domain of the variables. The solver `Sat4j` + `OrderEncodingBothPOS2020` uses the *order-encoding* for this purpose, while `Sat4j` + `OrderEncodingPrimitiveBothPOS2020` also exploits this encoding for representing more efficiently primitive constraints (as in [23]). We also executed the PB solver `RoundingSat` [8] on the PB encoding built by our implementation in `Sat4j`.

We compare these solvers with different state-of-the-art CP solvers, namely `ACE`[3] – the new name of `AbsCon` – `Choco` [22] and `sCOP` [23]. We also executed the previous implementation of a CP solver provided with the `Sat4j` library, denoted `Sat4j` + `SAT` [5].

All solvers have been run on a cluster of computers equipped with 32 GB of RAM and two quadcore Intel Xeon X5550 (2.66 GHz). The time limit was set to 1200 seconds.

In the context of our experiments, we used two benchmarks composed of decision problems from the XCSP library [3, 18]. The first one, denoted $\mathcal{I}_{\texttt{card}}$, is composed of 5 families of problems, and contains 145 instances. These instances are those with `cardinality` constraints taken from the XCSP website[4]. To obtain this set, we had to remove some instances containing constraints that are not yet encoded in our implementation (in particular, `lex` constraints). The second benchmark, denoted $\mathcal{I}_{\texttt{XCSP18-19}}$, is the set of instances that were used for the *mini-solvers* tracks of the XCSP18 and XCSP19 competitions. It is made of 32 families of problems, and contains 371 instances.

Below is presented the experimental analysis of our approach on these two benchmarks. These analyses have been made using Metrics[5], a tool for analyzing experiments which guarantees the reproducibility of the results.

4.1 Benchmark $\mathcal{I}_{\texttt{card}}$

Figure 1 shows an overview of the performance of the different solvers executed on $\mathcal{I}_{\texttt{card}}$. This figure is a so-called *cactus-plot*. Each line corresponds to a solver, and shows the number of instances solved within a given time limit by this solver. Here, we can see that the solver `RoundingSat` solves more instances than the other solvers. For instance, it solves 16 more instances than `ACE` within 1200 seconds. We can also observe that the solvers based on SAT, in this case `Sat4j` + `SAT` and `sCOP`, which respectively implement the *direct-encoding* and the *order-encoding*, do not perform well on these instances requiring the ability to count (recall that they contain `cardinality` constraints).

To compare more precisely the two first solvers of this *cactus-plot*, Figure 2 shows a *scatter-plot* comparing `ACE` and `RoundingSat`. In this plot, each point represents an instance, and its color the family of problems to which it belongs. The x coordinate shows the runtime of `RoundingSat` on this instance, while its y coordinate shows that of `ACE` on this same instance. This figure clearly shows that `RoundingSat` solves very efficiently instances from

[2] `https://gitlab.ow2.org/sat4j/sat4j-csp-pb`
[3] `https://github.com/xcsp3team/ace`
[4] `https://xcsp.org`
[5] `https://github.com/crillab/metrics`

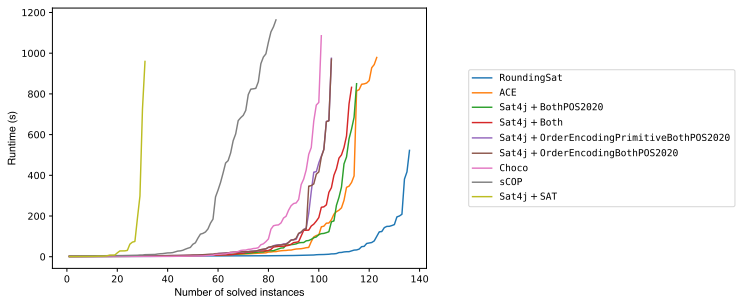

Figure 1 Cactus-plot of different solvers on $\mathcal{I}_{\texttt{card}}$.

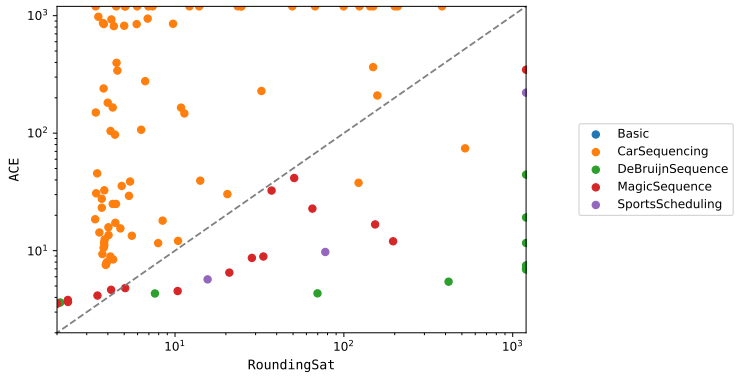

Figure 2 *Scatter-plot* comparing the runtime (in seconds) of `ACE` and `RoundingSat` on $\mathcal{I}_{\texttt{card}}$.

the `CarSequencing` family. In this family, there are mostly `cardinality` and `sum` constraints, which can be encoded into "pure" PB constraints, while constraints in the other families often require to use clauses, which limit the inference power of PB solvers. We observed the same behaviour for the different PB solvers we considered in this study, even though the different variants `Sat4j` that use PB constraints are not as efficient as `RoundingSat`.

4.2 Benchmark $\mathcal{I}_{\texttt{XCSP18-19}}$

In order to see whether our approach is able to solve more general problems, we now evaluate it on the instances of the benchmark $\mathcal{I}_{\texttt{XCSP18-19}}$, which contains a wide variety of problems and constraints. For the constraints of the *mini-solvers* track, we use a combination of the

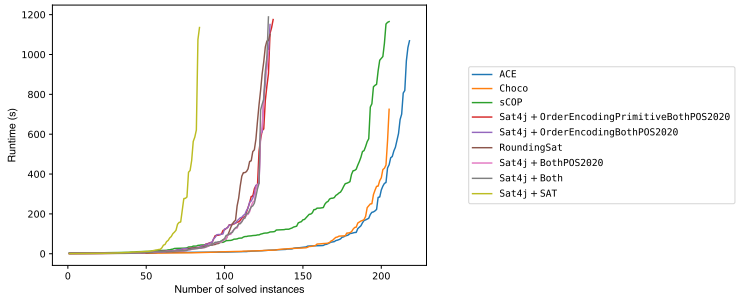

Figure 3 Cactus-plot of different solvers on $\mathcal{I}_{\texttt{XCSP18-19}}$.

encodings presented in Section 3 and the classical representations of these constraints using clauses when we do not provide a PB encoding for the constraint to encode. This allows our implementation to deal with all the instances of the set $\mathcal{I}_{\texttt{XCSP18-19}}$. Figure 3 shows an overview of the solvers we ran on this benchmark.

The faster solver is `ACE`, which solves much more instances than the others. Then follow `Choco` and `sCOP`, respectively in second and third position. The first `Sat4j` solver is at the fourth position. It is `Sat4j` + `Resolution`: this PB solver actually implements a conflict analysis as classical SAT solvers do, by lazily infering a clause each time a PB constraint is encountered during the analysis. The main advantage of this approach is that it allows to combine the succinctness of PB constraints and the use of the efficient data structures of SAT solvers. Yet, the efficiency of this solver is not enough to generalize the good performance observed on the set $\mathcal{I}_{\texttt{card}}$. This may be explained by the fact that (too) many constraints have to be encoded using clauses in the considered problems, which does not allow to exploit the full power of the proof system implemented in PB solvers.

Moreover, we can observe that the order of the PB solvers is not the same between the two considered benchmarks. This may be explained by the complementarity of the approaches implemented by the different solvers, as described in, e.g., [14, 16].

5 Conclusion

In this paper, we proposed to use different Boolean encodings for the domain of CSP variables to define new encodings based on PB constraints. In particular, we considered constraints recognized by the *mini-solvers* of the XCSP competition, as well as different variants of the `cardinality` constraint. The main advantage of the proposed encodings is that they allow to exploit the inference power of PB solvers, and especially their ability to count. The experimental analysis showed that our encodings, combined with the use of PB solvers, indeed allow to solve efficiently problems containing mainly `sum` and `cardinality` constraints. However, this good performance does not generalize to problems containing other types of constraints, in which native CP solvers and solvers based on SAT encodings remain faster.

Currently, our approach only allows to solve problems containing only a subset of all existing constraints. Our first perspective is to define encodings for other types of constraints, so as to submit our solver to the next XCSP competition. We also would like to investigate the use of different encodings for the constraints we already have, so has to improve the performance of PB solvers when solving CSPs, especially by favoring the use of pure PB constraints rather than clauses. Another perspective is to exploit the complementarity between the different solving paradigms for CSPs (either native, based on SAT or based on PB) to leverage the best of all approaches.

References

1 Belaid Benhamou, Lakhdar Sais, and Pierre Siegel. Two proof procedures for a cardinality based language in propositional calculus. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *Proceedings of STACS 1994*, pages 71–82. Springer, 1994. `doi:10.1007/3-540-57785-8\_132`.

2 Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of all different, global cardinality and related constraints. In Craig Boutilier, editor, *Proceedings of IJCAI 2009*, pages 419–424, 2009.

3 Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3-core: A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020. `arXiv:2009.00514`.

4 Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi:10.1145/800157.805047.

5 Ines Lynce Daniel Le Berre. Csp2sat4j: A simple csp to sat translator. In *Proceedings of the second CSP Competition*, 2008.

6 Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *AAAI'02*, pages 635–640, 2002.

7 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

8 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *Proceedings of IJCAI 2018*, pages 1291–1299, 2018.

9 Marco Gavanelli. The log-support encoding of CSP into SAT. In Christian Bessiere, editor, *CP 2007*, pages 815–822. Springer, 2007. doi:10.1007/978-3-540-74970-7_59.

10 Gaël Glorian. Nacre. In *Solver Descriptions of XCSP3 Competition 2018*, 2018.

11 Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, pages 275–278, 1958.

12 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297 – 308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science. doi:http://dx.doi.org/10.1016/0304-3975(85)90144-6.

13 J. N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12(1):217–239, 1988. doi:10.1007/BF02186368.

14 Daniel Le Berre, Pierre Marquis, and Romain Wallon. On weakening strategies for PB solvers. In Luca Pulina and Martina Seidl, editors, *SAT 2020*, pages 322–331. Springer, 2020. doi:10.1007/978-3-030-51825-7_23.

15 Daniel Le Berre and Anne Parrain. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

16 Daniel Le Berre and Romain Wallon. On dedicated cdcl strategies for pb solvers. In *Proceedings of SAT 2021*, pages 315–331, 2021.

17 C. Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE/Wiley, 2009.

18 Christophe Lecoutre and Nicolas Szczepanski. PYCSP3: modeling combinatorial constrained problems in python. *CoRR*, abs/2009.00326, 2020. arXiv:2009.00326.

19 Joao Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, pages 220–227, 1999.

20 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. doi:10.1145/378239.379017.

21 Jakob Nordström. On the Interplay Between Proof Complexity and SAT Solving. *ACM SIGLOG News*, 2(3):19–44, August 2015. doi:10.1145/2815493.2815497.

22 Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco solver documentation. *TASC, INRIA Rennes, LINA CNRS UMR*, 6241, 2016.

23 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009. doi:10.1007/s10601-008-9061-0.

24 Toby Walsh. SAT v CSP. In Rina Dechter, editor, *Proceedings of CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000. doi:10.1007/3-540-45349-0_32.