The return of *xorro*

Flavio Everardo$^{2[0000-0002-6421-3158]}$, Tomi Janhunen$^{1[0000-0002-2029-7708]}$, Roland Kaminski$^{2[0000-0002-1361-6045]}$, and Torsten Schaub$^{2\star[0000-0002-7456-041X]}$

^{1} Aalto University and Tampere University, Finland
^{2} University of Potsdam, Germany

Abstract. Although parity constraints are at the heart of many relevant reasoning modes like sampling or model counting, little attention has so far been paid to their integration into ASP systems. We address this shortcoming and investigate a variety of alternative approaches to implementing parity constraints, ranging from rather basic ASP encodings to more sophisticated theory propagators (featuring Gauss-Jordan elimination). All of them are implemented in the *xorro* system by building on the theory reasoning capabilities of the ASP system *clingo*. Our comparative empirical study investigates the impact of the number and size of parity constraints on performance and indicates the merits of the respective implementation techniques. Finally, we benefit from parity constraints to equip *xorro* with means to sample answer sets, paving the way for new applications of ASP.

1 Introduction

Parity constraints constitute the basic building blocks of many relevant reasoning modes like sampling or (approximate) model counting [19], not to mention their pertinence to circuit verification and cryptography [18]. Although their application and computational treatment are very active research topics (cf. [12,11,3,4,23]) in the neighboring area of Satisfiability Testing (SAT [2]), almost no attention has so far been paid to their integration into ASP solving [17]. Modest approaches include the (discontinued) support of `#even` and `#odd` aggregates in *gringo* series 3[3] and their usage for sampling in the initial prototype of *xorro*[4] from 2009. In this earlier prototype, parity constraints were simply implemented via `#count` aggregates and a modulo-two operation (see Listing 1.1). An alternative idea was later used in *harvey* [13] (see Listing 1.2). Unlike these approaches, several SAT solvers feature rather sophisticated treatments of parity constraints, most popularly the award-winning solver *crypto-minisat* [24]. The difficulty lies in the inadequacy of CDCL-based solvers [9] (and more precisely their underlying resolution-based learning scheme) to effectively handle parity constraints. In fact, the translation of parity constraints into conjunctive normal form degrades search [14], although they could be directly solved with Gauss-Jordan elimination (GJE) in polynomial time [21]. Consequently, solvers like *crypto-minisat* pursue a hybrid approach, addressing parity constraints separately with GJE.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.
[3] This is achieved by uncompiling them during grounding using meta-encodings.
[4] https://sourceforge.net/p/potassco/code/HEAD/tree/branches/xorro

In what follows, we present the next generation of *xorro*, a full re-implementation, providing a wide spectrum of alternative ways for integrating parity constraints into ASP solving. On the one hand, this re-implementation draws upon the advanced interfaces of *clingo* for integrating foreign constraints and corresponding forms of inference. On the other hand, *xorro* takes advantage of the sophisticated solving techniques developed in SAT for handling parity constraints, such as GJE. More precisely, we propose two types of approaches in Section 2,[5] namely eager ones that rely on ASP encodings of parity constraints, and lazy ones using theory propagators within *clingo*'s Python interface. We then empirically evaluate the different approaches in view of their impact on solving performance, while varying the number and size of parity constraints.

2 Incorporating Parity Constraints into ASP

We expect the reader to be familiar with the basic syntax and semantics of logic programs as implemented by *clingo* (see [7,6] for details). In this section, we focus on the introduction of non-standard concepts needed in this paper.

Towards the definition of parity constraints, let $\top$ and $\bot$ stand for the Boolean constants *true* and *false*, respectively. Given atoms a_1 and a_2, the *exclusive or* (XOR for short) of a_1 and a_2 is denoted by $a_1 \oplus a_2$ and it is satisfied if *either* a_1 *or* a_2 is true (but not both). Generalizing the idea for n distinct atoms $a_1, \ldots, a_n$, we obtain an n-ary XOR constraint $(((a_1 \oplus a_2) \ldots) \oplus a_n)$ by multiple applications of $\oplus$. Since it is satisfied iff an odd number of atoms among $a_1, \ldots, a_n$ are true, it is called an *odd* XOR *constraint* and it can be written simply as $a_1 \oplus \ldots \oplus a_n$ due to associativity. Analogously, an *even* XOR *constraint* is defined by $a_1 \oplus \ldots \oplus a_n \oplus \top$ as it is satisfied iff an even number of atoms among $a_1, \ldots, a_n$ hold. Then, e.g., $a_1 \oplus a_2 \oplus \top$ is satisfied iff none or both of a_1 and a_2 hold. In the sequel, we also refer to even and odd XOR constraints as *parity constraints*. As shown in [18], any XOR constraint $a_1 \oplus \ldots \oplus a_n$ can be decomposed into two XOR constraints $a_1 \oplus a_2 \oplus aux$ and $aux \oplus a_3 \oplus \ldots \oplus a_n \oplus \top$ where aux is a new atom not used elsewhere. Finally, XOR constraints of forms $a \oplus \bot$ and $a \oplus \top$ are called *unary*.

To accommodate parity constraints in the input language, we rely on *clingo*'s theory language extension [8] that pertains to the common syntax of *aggregates*:

```
1   &odd{ 1 : p(1) }.
2  &even{ X : p(X), X>1 }.
```

More precisely, *xorro* extends the input language of *clingo* by aggregate names `&even` and `&odd` that are followed by a set, whose elements are *terms* conditioned by conjunctions of literals separated by commas.[6] The semantics of aggregates formed with keywords `&even` and `&odd` is defined by even and odd parity constraints, respectively. In the current implementation, they are interpreted as directives that select answer sets satisfying the parity constraint in question.[7] For now, parity constraints may not occur

[5] The distinction of eager and lazy approaches follows the methodology in Satisfiability modulo theories [1].

[6] In turn, multiple conditional terms within an aggregate are separated by semicolons.

[7] Our implementation of parity constraints fits perfectly with the parity constraints used in sampling and model counting.

in the bodies nor the heads of rules and the full integration of parity constraints into rules is left as future work. The parity constraints shown above yield two answer sets, viz. {`p(1)`} and {`p(1), p(2), p(3)`} in the context of a choice rule {`p(1..3)`}. Hence, the first constraint filters out answer sets not containing the atom `p(1)`, while the second requires that either none or both of the atoms `p(2)` and `p(3)` are included.

2.1 Eager Encodings of Parity Constraints

In the following, we present three different ways to encode parity constraints using primitives available in standard ASP. We refer to these encodings by nicknames *Counting*, *List*, and *Tree*, respectively. Each encoding leads to an *eager* evaluation of the corresponding parity constraint in terms of *nogoods*, which are used to invalidate answer sets as well as to explain reasons behind conflicts encountered by solvers. In the eager approach, nogoods resulting from parity constraints are generated in advance. As a consequence, the underlying answer-set solver may freely propagate truth values over (parts of) parity constraints during search.

Counting. Our first encoding is essentially the same as used in the previous generation of *xorro*. As shown in Listing 1.1, the idea is to introduce an analogous *counting aggregate* for the number of terms in the set and, in addition, to check that this number matches with the given parity. Recalling the preceding example in this section, the given parity constraints translate into integrity constraints embedding *#count* aggregates coupled with appropriate modulo `2` conditions. The net effect is that the first constraint enforces odd parity within {`p(1)`}, while the latter concerns even parity for the atoms in {`p(2),p(3)`}.

```
:- #count{ 1 : p(1)      } = N, N\2!=1.
:- #count{ X : p(X),X>1 } = N, N\2!=0.
```

Listing 1.1: Aggregate-based encoding parity constraints (`count.lp`).

List. The encoding presented in Listing 1.2 is based on an ordered list of terms expressed using predicates `term/1`, `first/1`, `last/1`, and `next/2`. The idea is to perform a sequence of tests for odd parity based on this list.[8] Line 1 sets the base case using the first term of the list. Then, Lines 2 and 3 recursively check for odd parity following the structure of the list. Note that `term(T)` holds iff the conditions related with the term `T` are satisfied. Line 4 determines if the parity of the entire term sequence is odd based on the status of last term in the list. Finally, the encoding should be combined with exactly one of the constraints in Lines 5 and 6. The first eliminates answer sets with odd parity, while the one commented away in the listing removes the even cases with respect to the parity constraint in question. The given encoding has been deployed, e.g., in the previous *gringo* versions (2 and 3) [10] as well as randomized testing [13].

```
1 odd(T) :- first(T), term(T).
2 odd(Y) :- next(X,Y),     term(Y), not odd(X).
3 odd(Y) :- next(X,Y), not term(Y),     odd(X).
4 odd    :- odd(T), last(T).
5 :- odd.
```

[8] This is analogous to parity evaluation using *binary decision diagrams* (BDDs).

```
6  % :- not odd.
```

Listing 1.2: List-based encoding of parity constraints (`list.lp`).

Tree. Our last eager representation resembles the previous encoding but the underlying topology for parity checks is different. A *balanced* binary tree is created for each parity constraint and the recursive evaluation proceeds in a bottom-up fashion. The terms are associated with the leaves of the tree while the root corresponds to the final outcome of the parity check. The structure of the tree is expressed using predicates `leaf/1`, `root/1`, and `edge/2`. Line 1 in Listing 1.3 sets the base case using the leaves of the tree. Lines 2 and 3 accumulate the result of the parity check towards the root of the tree, the value for each parent `P` is set based on the values of children `C1` and `C2` that need not be ordered by symmetry. The value observed for the root `R` (see Line 4) sets the result. In addition to the given rules, we have to include constraints for selecting the intended parity value as done in Lines 5–6 in Listing 1.2.

```
1  odd(T) :- leaf(T), term(T).
2  odd(P) :- odd(C1), not odd(C2), edge(P,C1), edge(P,C2)
3  odd(P) :- not odd(C1), odd(C2), edge(P,C1), edge(P,C2).
4  odd    :- odd(R), root(R).
```

Listing 1.3: Tree-based encoding of parity constraints (`tree.lp`).

2.2 Lazy Evaluation of Parity Constraints

Next, we switch our attention to the *lazy* evaluation techniques that generate nogoods related to parity constraints on demand only. This is in contrast with the eager approaches where such nogoods can be produced a priori. In practice, we have implemented parity reasoning modules in Python acting as *theory propagators* [8] for the *clingo* system. In what follows, we briefly explain how parity checking can be achieved in a more lazy fashion.

Lazy Counting. In this approach, the idea is to perform counting (modulo 2) in order to check parity constraints. However, such checks are performed only when a candidate answer set for the rest of the program has been found. Therefore, the evaluation of parity constraints does not interfere with the propagation of truth values while searching for answer sets. If a particular parity constraint is violated, then a corresponding nogood is generated.

Watched Literals. The propagation of truth values can be performed on demand by *watching* certain literals occurring in a constraint (such as 2 literals per clause [20]). The rough idea is to check the status of the constraint only if the truth values of the watched literals are changed. As a result, the constraint might be used for propagation or the literal(s) being watched is/are changed to some other literal(s). In case of parity constraints, however, all but one atom involved in a particular constraint must be assigned before the truth value of the final one is determined [18,24]. E.g., if $a_1, \ldots, a_{k-1}$ and $a_{k+1}, \ldots, a_n$ have been assigned false or, more generally, have an even parity in $a_1 \oplus \ldots \oplus a_n$, then a_k must be true. Therefore, we have to keep track of both parity values (even and odd) by watching 2×2 literals (two literals both phases) for each parity constraint. Is important to mention, that all atoms (or terms) contributing to parity constraints originate from the underlying logic program. Otherwise, they are removed by the *gringo* grounder due to closed world assumption (all occurrences of $\perp$ can be removed from parity constraints).

Figure 1 illustrates unit propagation over parity constraints $a \oplus b \oplus c$ and $c \oplus d \oplus \top$. Given the truth assignments a and $\neg b$ indicated in gray, the first constraint simplifies to $b \oplus c \oplus \top$. Furthermore, we get $\neg c$ and $\neg d$ through unit propagation. Had the assumptions originated from a candidate answer set $\{a\}$, no other answer sets would be feasible. The inferences made here can be recorded as learned nogoods $\{a, \neg b, c\}$ and $\{\neg c, d\}$ in order to perform similar (ordinary) unit propagation later on without consulting the parity propagator again.

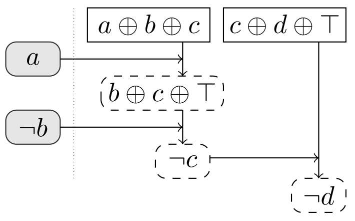

Fig. 1: Unit propagation involving parity constraints

Gauss-Jordan Elimination. Sets of parity constraints can also be cast as linear equation systems whose solutions can be determined using Gauss-Jordan elimination (GJE) [18,24,15]. The GJE method is complete for parity reasoning because it can be used to decide whether a conjunction of parity constraints is satisfiable as well as to find implied literals and equivalences. Plain Gaussian elimination can efficiently detect satisfiability, but not implied literals nor equivalences. This can be understood from the difference between the *row-echelon* and the *reduced row-echelon* forms for the matrix representations of parity equation systems.

$$\begin{array}{cccc|c} a & b & c & d & p \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{array} \qquad \begin{array}{ccc|c} a & c & d & p \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{array} \qquad \begin{array}{ccc|c} a & c & d & p \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{array} \qquad \begin{array}{ccc|c} a & c & d & p \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}$$

$$(a) \qquad\qquad (b) \qquad\qquad (c) \qquad\qquad (d)$$

Fig. 2: Deducing a, c, and d by GJE after b given

For the sake of illustrating GJE, let us use the constraints from Figure 1 along with $b \oplus c \oplus \top$. Figure 2(a) represents the respective equations as a matrix where the column "p" indicates parities for the equations. Figure 2(b) shows a column reduction when b is assigned true and reflected to the parity values. Figure 2(c) shows a row-echelon form for the matrix, already indicating satisfiability and the truth of d. By further simplification into reduced row-echelon form in Figure 2(d), we can clearly see how the values for other atoms are determined: a, c, and d must all be true. After the matrix is reduced, we need to find either a conflict or implications. If conflict, the nogood is the partial assignment. If implications, the nogood is the partial assignment coupled with each of the implication literals in negated form.

3 The *xorro* System

This re-implementation of *xorro* allows the user to solve parity constraints on top of an ASP program using a specific approach.[9] *xorro* is built on top of *clingo* 5.3, and

[9] https://github.com/potassco/xorro

the system architecture is shown in Figure 3. *xorro* follows the standard grounding-solving workflow of ASP, plus three additional blocks shown in solid lines which are preprocessing, transformation, and translation. The preprocessing module has two optional flags `--split` and `--pre-gje`. The split flag takes an integer to cut larger constraints into smaller ones using auxiliary variables, and the pre-gje flag enables XOR simplification for more than one constraint. If both flags are used together, GJE is performed first followed by splitting. The transformation block is performed before grounding to parse each parity constraint into facts and normal ASP rules. The translation block is called before solving and it is responsible for building additional features for each approach. Additionally, this implementation of *xorro* preserves the same functionality as its predecessor for near-uniform sampling.

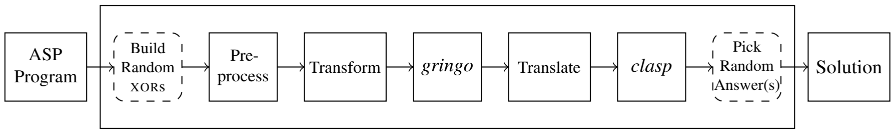

Fig. 3: Architecture of *xorro*

The workflow starts with an ASP program with parity constraints. Before grounding, we preprocess and transform each parity constraint using *clingo*'s Abstract Syntax Tree (AST) into auxiliary atoms of the form `__parity/2` and `__parity/3`. The atom `__parity/2` is added as a fact and it contains a numeric identifier and the parity as *odd* or *even*. The atom `__parity/3` is added in the head of a new rule where the body corresponds to its conditional literal. This atom contains the same information as the fact `__parity/2` plus the tuples of terms involved in the constraint.[10] For example, the transformation of the parity constraints from Figure 2a is shown in Listing 1.4.

```
__parity(0,odd). __parity(1,even). __parity(2,even).
__parity(0,odd,(a,)):-a.  __parity(0,odd,(b,)):-b.
__parity(0,odd,(c,)):-c.
__parity(1,even,(c,)):-c. __parity(1,even,(d,)):-d.
__parity(2,even,(b,)):-b. __parity(2,even,(c,)):-c.
```

Listing 1.4: Transformation of parity constraints to ASP.

The translation block depends on the given approach which is given by the flag `--approach` followed by a keyword indicating one of the approaches from Sections 2.1 and 2.2. In the case of an eager approach, we add additional structures to the program. For the eager count approach, we add the count aggregates with respect to the atom `__parity/3` for every parity constraint `__parity/2`. For the list and tree approaches, we benefit from *clingo*'s Python API by using *clingo*'s backend class to extend a logic program by adding statements directly in the intermediate format of ASP (*aspif* [16]).

For the lazy approaches, we benefit from the theory propagator interface of *clingo* which consists of four functions, namely *init*, *propagate*, *undo*, and *check*. We rely on them, so each lazy approach performs at a specific part of the search, being during

[10] The transformation process mimicks the use of a theory grammar for *clingo* [8].

propagation or fixpoints (partial or total assignments). All propagators keep the state of each parity constraint by its (solver) literals. From the three lazy approaches, the lazy count works on total assignment whereas the UP and GJE during propagation. The lazy count approach does not propagate and do not interfere with clasp propagation. On total assignment under the *check* method, we count the number of true literals. In case of conflict, add the nogood (whole assignment per constraint) and let *clasp* to propagate again. The Unit-Propagation (UP) propagator performs plain propagation over parity constraints. As its name suggests, it is performed in the *propagate* function. The *check* method is not implemented, and the state keeps 2x2-watched literals. For implementing Gauss-Jordan Elimination, two alternatives are proposed, called “gje-fp” and “gje-prop”. Their main difference is at which point of the search GJE is called. Both extend the UP approach. For “gje-fp” two propagators are registered, the UP followed by the “gje-fp” propagator, performing GJE on fixpoints. The “gje-prop” alternative registers only one propagator performing GJE when a unit clause is detected.

As mentioned, this implementation of *xorro* preserves the functionality of sampling following the concepts from the XORSample’ algorithm [12], by solving a program with `--s` random parity constraints with a density `--q`. The sampling components are enabled with `--sampling` and shown in Figure 3 in dotted rounded-corner squares. Unlike the algorithm from [12] which calls a model counter, *xorro* enumerates all remaining answer sets, and the last module randomly picks *n* (user-defined) answer sets. The sampling mode for *xorro* corresponds to the non-iterative algorithm from [12] recalling the possibility to get no answer set due to unsatisfiable parity constraints. Finally, the flag `--display` prints the random XOR constraints used in sampling.

4 Experiments

Different tests are proposed to measure the impact of solving a single or several parity constraints with *xorro*. We considered 301 satisfiable instances from 19 (9 tight, 10 non–tight) classes using all aforementioned approaches. However, we kept only 126 instances from eight classes (5 tight, 3 non-tight) for which *clingo*’s solving time surpasses one second. These benchmarks problems were taken from the second ASP competition [5], using encodings of the Potassco team. [11] No encoding or instance has been modified, just parity constraints are appended. The main objectives of our experiments are built around the following questions:

1. What is the impact of solving a single random parity constraint of different sizes ranging from 1, 10, 20, 30, 40, and 50 percent of unassigned variables?
2. Due to SAT solvers’ good performance on small size parity constraints, is there any benefit of splitting a single parity into small ones? On average from the literature, the size of a small XOR constraint is 4.
3. Due to sampling using high-density constraints (50 percent of unassigned variables), what is the impact of solving more than one high-density parity constraint with and without GJE preprocessing?

[11] http://flavioeverardo.com/research/benchmarks/xorro/

4. Do the eager approaches count and list used in the previous *xorro* version, *gringo* up to version 3, and in *harvey*, respectively, scale when solving high-density parity constraints?
5. Is there any approach outstanding from the rest?

To address these questions, we designed five experiments, each with different tests. Each test uses the set of instances from the benchmark classes in Table 1 coupled with a single or several XORs. We use *clingo*, and Python to randomly build each constraint excluding facts with the only condition that at least one answer set remains. Table 1 shows the range of atoms per class and the number of instances under column "#".

For the first experiment, we solved all the instances each with a single parity constraint of different sizes ranging from one, 10, 20, 30, 40, to 50 percent of unassigned variables. In the second experiment, we used the same parity constraints but we split them into smaller ones of size four. The auxiliary atoms were added into each instance as choice rules. For the third, fourth, and fifth experiments, we increased the number of high-density XORs to two, three, four, and five. We first solved the instances without preprocessing. Then, we reduced the length of the XORs by applying GJE, and lastly, GJE plus a split of size four. It is important to remark that the difference between small XORs and shorter ones due to split, is their shared variables. If independently drawn small XORs contain variables in common, they mean the same as a longer XOR with equivalence reasoning.

Class	#	MIN	AVG	MAX
Tight				
15Puzzle	16	9841	10514	11332
BlockedNQueens	15	7996	8008	8012
GraphColouring	9	2707	2837	3087
SchurNumbers	13	1291	1291	1291
Solitaire	22	7687	8702	9920
Non-Tight				
ConnectedDomSet	10	804	1463	2519
Labyrinth	29	56482	91733	120192
WireRouting	12	6085	15546	25330

Table 1: Range of atoms per class

Our tests follow the notation "cNxMperc" where N represents the number of parity constraints times M percentage or density per parity. For example the test "c03x50perc" solves an instance containing three XORs of 50% each. Our comparison considers solving each instance without parity constraints using *clingo* 5.3 in its default setting as benchmark reference. We ran *clingo* two times, once for the first two experiments and once more for the last three.

The experiments were run in parallel under Linux on an Intel Xeon E5-2650v4 high performance cluster equipped with 2.20 GHz processors. Each benchmark instance (in smodels output format, generated offline with *gringo* plus the parity constraints) was run three times per solver (*clingo* and *xorro*). Each run is restricted to 600 seconds time with 4 GB RAM. A run finished when the solver found an answer set or was aborted due to time or memory exhaustion.

4.1 Results

Our experiments' results are summarized in Tables 2a–3c giving average runtimes in seconds (using PAR–1 score) and the number of aborted runs is shown in parentheses.[12]

[12] For more detailed benchmarks results, including individual times per classes please go to http://flavioeverardo.com/research/benchmarks/xorro/

All the tables show the experiments and *clingo*'s runtimes without parity constraints on the left side followed by the corresponding lazy and eager approaches. The last row of each table shows the average runtimes followed by the total number of time or memory exhaustion. All tables show the best score from each test in bold. Tables 2a and 2b correspond to the results from the first two experiments addressing questions 1 and 2, respectively. Both experiments divide the search space into two parts. However, the results are completely contrasting. The counting approach with aggregates does not scale from the 10 percent size when solving the parity constraint as it is. However, this approach scales when the XORs are split. The grounding becomes the bottleneck on longer XORs causing timeouts or memory exhaustion on most of the instances and runs, whereas roles switch with shorter XORs. Despite the best performance of the lazy count approach on four out of six tests, the best PAR–1 score belongs to the tree approach. Both approaches perform better than *clingo* in five out of six tests in the first experiment. Contrary, both approaches perform worst when splitting XORs. A similar case occurs with the two worst approaches when solving non-split XORs (the count with aggregates and the UP). Both outperform the rest if the XORs are split. The best PAR–1 score from the second experiment belongs to the UP approach. We can see from both experiments the feasibility to reach an answer set faster than *clingo* when solving with a single XOR.

To resume, the length of the parity constraint matters depending on the approach to use. To solve a single parity constraint without a split, better use the tree or the lazy counting approach. On shorter XORs or a longer split XOR, better use UP or the eager counting.

		lazy		eager		
test	*clingo*	count	up	list	count	tree
1x01perc	883.19 (11)	1220.65 (15)	**644.72 (8)**	1035.84 (13)	1038.58 (13)	806.43 (10)
1x10perc	909.21 (11)	**802.27 (10)**	1026.35 (13)	951.96 (12)	2998.6 (39)	876.34 (11)
1x20perc	908.98 (11)	**802.52 (10)**	952.37 (12)	837 (10)	5025.73 (66)	882.33 (11)
1x30perc	884.17 (11)	827.38 (10)	1180.09 (15)	875.56 (11)	6885.11 (90)	**782.72 (10)**
1x40perc	884.07 (11)	**878.19 (11)**	1178.58 (15)	1103.47 (14)	7933.01 (104)	904.66 (11)
1x50perc	959.49 (12)	**877.13 (11)**	1178.37 (15)	959.15 (12)	8638.37 (113)	878.6 (11)
AVG (SUM)	904.85 (67)	901.36 (67)	1026.74 (78)	960.5 (72)	5419.9 (425)	**855.18 (64)**

(a) Average solving runtimes on a single XOR constraint ranging from 1 to 50 percent.

		lazy		eager		
test	*clingo*	count	up	list	count	tree
1x01perc	883.19 (11)	1196.01 (15)	**647.96 (8)**	804.07 (10)	1035.91 (13)	808.8 (10)
1x10perc	909.21 (11)	658.57 (8)	653.47 (8)	**644.19 (8)**	648.15 (8)	955.17 (12)
1x20perc	908.98 (11)	962.7 (12)	722.71 (9)	732.31 (9)	**568.47 (7)**	871.84 (11)
1x30perc	884.17 (11)	965.12 (12)	**571.27 (7)**	802.18 (10)	800.14 (10)	1029.28 (13)
1x40perc	884.07 (11)	1451.29 (18)	1032.9 (13)	1023.5 (13)	963.23 (12)	**880.07 (11)**
1x50perc	959.49 (12)	1512.16 (19)	725.18 (9)	1026.21 (13)	**508.78 (6)**	945.73 (12)
AVG (SUM)	904.85 (67)	1124.31 (84)	**725.58 (54)**	838.74 (63)	754.11 (56)	915.15 (69)

(b) Average solving runtimes on split XORs of size 4, ranging from 1 to 50 percent.

Table 2: Experiments solving a single parity constraint from different sizes.

For the last three experiments, the search space is divided into four, eight, 16 and 32 parts showing the solving performance over more than one parity constraint. From here on, we include the GJE elimination approaches and we exclude for now the eager counting approach which only performs on shorter (or split) XORs. Table 3a shows the results from experiment number three. The hardness of solving dense XORs starts to arise. lazy countinging outperforms the rest when solving with two XORs, opposed to *clingo* which performs the best in the remaining tests. Both eager approaches (tree and list) perform badly as the number of XORs increases. Lastly, there is a big difference between both GJE approaches. One (gje-prop) performing in between the eager and the other lazy approaches, whereas the other (gje-fp) does not scale at all. Both use the same routines to operate over columns, rows, detect conflicts, propagation and so on. Also, both perform exactly the same concerning the number of choices, restarts, conflicts, backjumps, etc. To recall, GJE outputs one of three results: variable(s) to propagate, conflict(s) or neither of both. The runtime difference occurs when performing GJE on fixpoints results in finding neither of both in most of the time. For example, "gje-prop" solves instances of the 15Puzzle class with an average of 10 calls to GJE without propagating or conflict. The same instances with "gje-fp" called on average over 5,000 times more GJE without finding a conflict or a literal to propagate.

		lazy				eager	
test	*clingo*	count	up	gje-fp	gje-prop	list	tree
2x50perc	1113.02 (14)	**902.33 (11)**	952.39 (12)	3806.57 (48)	1025.23 (13)	960.53 (12)	1144.23 (14)
3x50perc	**934.33 (12)**	976.59 (12)	1028.65 (13)	4038.56 (51)	1097.16 (14)	1567.05 (20)	1570.77 (20)
4x50perc	**1034.74 (13)**	1105.75 (14)	1182.47 (15)	4347.56 (55)	1421.77 (18)	1546 (20)	1622.86 (21)
5x50perc	**1009.25 (13)**	1189.58 (15)	1032.25 (13)	4655.35 (59)	1468.04 (19)	1770.75 (23)	1928.74 (25)
AVG (SUM)	**1022.83 (63)**	1043.56 (64)	1048.94 (68)	4212.01 (228)	1253.05 (79)	1461.08 (87)	1566.65 (91)

(a) Average solving runtimes from two to five high-density constraints.

		lazy			eager	
test	*clingo*	count	up	gje-prop	list	tree
2x50perc	1113.02 (14)	877.85 (11)	956.17 (12)	1026.36 (13)	1261.7 (16)	**736.64 (9)**
3x50perc	**934.33 (12)**	1027.76 (13)	1028.87 (13)	1095.95 (14)	981.04 (12)	1343.69 (17)
4x50perc	1034.74 (13)	**1006.4 (13)**	1112.8 (14)	1374.6 (18)	1729.47 (22)	1711.62 (22)
5x50perc	**1009.25 (13)**	1112.83 (14)	1187.97 (15)	1470.44 (19)	1852.3 (24)	2151.83 (28)
AVG (SUM)	1022.83 (63)	**1006.21 (62)**	1071.45 (69)	1241.84 (79)	1456.13 (90)	1485.94 (91)

(b) Average solving runtimes from two to five high-density with GJE preprocess.

		lazy			eager		
test	*clingo*	count	up	gje-prop	list	count	tree
2x50perc	1113.02 (14)	2333.18 (30)	**729.96 (9)**	8613.63 (111)	941.51 (12)	1060.47 (13)	1042.54 (13)
3x50perc	**934.33 (12)**	2413.15 (31)	947.77 (12)	9177.39 (118)	1917.18 (25)	1215.5 (15)	1798.67 (23)
4x50perc	**1034.74 (13)**	2321.53 (30)	1776.73 (23)	9579.91 (123)	2130.86 (28)	1668.8 (21)	2284.37 (30)
5x50perc	**1009.25 (13)**	2417.51 (31)	1927.46 (25)	9438.35 (122)	2367.49 (31)	1552 (19)	2278.92 (30)
AVG (SUM)	**1022.83 (51)**	2371.34 (122)	1345.48 (69)	9202.32 (474)	1839.26 (96)	1374.19 (68)	1851.13 (97)

(c) Average solving runtimes from two to five high-density with GJE and split preprocess.

Table 3: Experiments on multiple high-density parity constraints.

The fourth experiment solves the same parity constraints but with a GJE preprocessing step to reduce their length. The results are shown in Table 3b. From the five approaches, all except UP benefit from preprocessing. The tree and the lazy counting approaches got a speedup of 5.2 and 3.6% respectively. However, the tree approach remains worst, and the lazy counting now performs better than *clingo*.

The last experiment takes another notch by splitting the preprocessed XORs after GJE. We include the eager counting approach due to its performance on split constraints. The results from Table 3c show *clingo* outperforming the rest in three out of four tests, and also, in the overall score. The UP has the best score in only one test. Similarly to experiment number two, the lazy counting and the tree approaches perform poorly with shorter XORs. UP and the eager counting approach have the best scores but still quite far from *clingo*'s performance. None of the five approaches from the fourth experiment benefit from splitting in the fifth. It is the opposite. The tree, list, and UP downgrade their performance by 24-26%. The lazy counting by 135% and GJE by 600%. We confirm that splitting high-density parity constraints does not scale without further preprocessing. When split, we add more XORs and more variables (rows and columns respectively for GJE). We passed from five dense to 670 smaller constraints in the best case (Connected Dom Set class) against 100,160 shorter parity constraints in the worst (Labyrinth class). From a GJE perspective, we increased the size of the matrix from five rows and 402 columns in the best case, to 670 rows and 536 columns. The worst case passes from five rows and 64,596 columns to 100,160 rows and 86,128 columns. This makes our GJE implementation fail to scale without additional preprocessing. As stated in [18], so far, for larger matrices, the computational overhead of Gaussian elimination is significant. Also, [24,22] state that for efficient solving, the number of parity constraints and their density should be low. The tests show most of the approaches performed better with longer XORs. Additional preprocessing like equivalence reasoning reduces the number of constraints by creating longer XORs.

After running all experiments, we can see that the eager counting and list approaches stay behind when solving high-density constraints without any preprocessing (as in the previous *xorro* and *harvey* implementations). In contrary, they improved their performance especially when a split occurs. Also, it is difficult to identify an approach outstanding from the rest. From 24 tests, the lazy counting has the best score on six, followed by the UP and the tree approaches with four and three respectively. Plain *clingo* performs best on eight tests. On the other hand, depending on the number and the density of the parity constraints, some approaches perform best. From experiment number two, some *xorro* approaches perform better on XORs of lower densities than $n/2$ variables. This is sufficient for approximate model counting, but not for sampling [11].

5 Discussion

We presented different means of implementing parity constraints, ranging from ASP encodings to theory propagators. The fully re-implemented system *xorro* takes advantage of the hybrid reasoning capacities of the ASP system *clingo* for solving parity constraints on top of logic programs, providing an opening to develop new applications in ASP including sampling, (approximate) model counting, cryptography, and probabilistic reasoning.

Our experiments show that *xorro* scales depending on the combination of the number, density, and preprocessing of the parity constraints. For instance, cutting the search space by half helps *xorro* to reach an answer set faster than *clingo* even when using a high-density constraint. Some approaches scaled when splitting a single XOR as opposed to others who perform without preprocessing. On the other hand, solving high-density parity constraints hinders *xorro* performance compared to *clingo*'s, but there is evidence that preprocessing can lead to a significant speedup.

For future work, we plan to extend *clingo*'s input language with parity aggregates and investigate the performance of *clingo*'s multi-shot capabilities by incrementally solving parity constraints as an application for sampling. Finally, to improve our GJE approach, and inspired by [24,18], we want to explore an incremental GJE and strategies like ordered columns and turning GJE on/off automatically as well as cutoff values, rows and columns elimination with equivalence reasoning.

References

1. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere et al. [2], chap. 26, pp. 825–885
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
3. Chakraborty, S., Meel, K., Vardi, M.: A scalable and nearly uniform generator of SAT witnesses. In: Sharygina, N., Veith, H. (eds.) Proceedings of the Twenty-fifth International Conference on Computer Aided Verification (CAV'13). Lecture Notes in Computer Science, vol. 8044, pp. 608–623. Springer-Verlag (2013)
4. Chakraborty, S., Meel, K., Vardi, M.: A scalable approximate model counter. In: Schulte, C. (ed.) Proceedings of the Nineteenth International Conference on Principles and Practice of Constraint Programming (CP'13). Lecture Notes in Computer Science, vol. 8124, pp. 200–216. Springer-Verlag (2013)
5. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). Lecture Notes in Artificial Intelligence, vol. 5753, pp. 637–654. Springer-Verlag (2009)
6. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015)
7. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S.: Potassco User Guide. 2 edn. (2015), http://potassco.org
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Carro, M., King, A. (eds.) Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16). vol. 52, pp. 2:1–2:15. Open Access Series in Informatics (OASIcs) (2016)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
10. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In: Delgrande, J., Faber, W. (eds.) Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Lecture Notes in Artificial Intelligence, vol. 6645, pp. 345–351. Springer-Verlag (2011)

11. Gomes, C., Hoffmann, J., Sabharwal, A., Selman, B.: Short XORs for model counting: from theory to practice. In: Marques-Silva, J., Sakallah, K. (eds.) Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07). Lecture Notes in Computer Science, vol. 4501, pp. 100–106. Springer-Verlag (2007)
12. Gomes, C., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using XOR constraints. In: Schölkopf, B., Platt, J., Hofmann, T. (eds.) Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems (NIPS'06). pp. 481–488. MIT Press (2007)
13. Greßler, A., Oetsch, J., Tompits, H.: Harvey: A system for random testing in ASP. In: Balduccini, M., Janhunen, T. (eds.) Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17). Lecture Notes in Artificial Intelligence, vol. 10377, pp. 229–235. Springer-Verlag (2017)
14. Haanpää, H., Järvisalo, M., Kaski, P., Niemelä, I.: Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4), 27–46 (2006)
15. Han, C., Jiang, J.: When boolean satisfiability meets gaussian elimination in a simplex way. In: Madhusudan, P., Seshia, S.A. (eds.) Proceedings of the Twenty-fourth International Conference on Computer Aided Verification (CAV'12). Lecture Notes in Computer Science, vol. 7358, pp. 410–426. Springer-Verlag (2012)
16. Kaminski, R., Schaub, T., Wanko, P.: A tutorial on hybrid answer set solving with clingo. In: Ianni, G., Lembo, D., Bertossi, L., Faber, W., Glimm, B., Gottlob, G., Staab, S. (eds.) Proceedings of the Thirteenth International Summer School of the Reasoning Web. Lecture Notes in Computer Science, vol. 10370, pp. 167–203. Springer-Verlag (2017)
17. Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Magazine **37**(3), 25–32 (2016)
18. Laitinen, T.: Extending SAT Solver with Parity Reasoning. Dissertation, Aalto University (Nov 2014)
19. Meel, K.: Constrained Counting and Sampling: Bridging the Gap between Theory and Practice. Dissertation, Rice University (Aug 2018)
20. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01). pp. 530–535. ACM Press (2001)
21. Schaefer, T.: The complexity of satisfiability problems. In: Lipton, R., Burkhard, W., Savitch, W., Friedman, E., Aho, A. (eds.) Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOCS'78). pp. 216–226. ACM Press (1978)
22. Soos, M.: Enhanced gaussian elimination in DPLL-based SAT solvers. In: Le Berre, D. (ed.) Proceedings of the First Workshop on Pragmatics of SAT (PoS'10). EPiC Series in Computing, vol. 8, pp. 2–14. EasyChair (2012)
23. Soos, M., Meel, K.: Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In: Van Hentenryck, P., Zhou, Z. (eds.) Proceedings of the Thirty-third National Conference on Artificial Intelligence (AAAI'19). AAAI Press (2019), to appear
24. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09). Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer-Verlag (2009)