Bipartite Perfect Matching Benchmarks

Cayden R. Codel, Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
{ccodel,jereeves,mheule,randy.bryant}@cs.cmu.edu

Abstract

The pigeonhole and mutilated chessboard problems are challenging benchmarks for most SAT solvers. Although some solvers employ special techniques that efficiently solve the canonical versions of these two problems, these techniques may fail with even slight problem variations. To evaluate and improve the robustness of SAT solvers, we designed a benchmark family of perfect matching problems on bipartite graphs that generalizes the pigeonhole and mutilated chessboard problems. Our benchmark generator supports various encodings and randomized constructions. Experimental results show that different variations degrade the performance of solvers in unexpected ways. As such, the benchmark family, taken as a whole, provides a good way to reveal the fragility of fine-tuned solving techniques. Tuning against it will encourage more robust solver implementations. We also studied the effect that adding symmetry-breaking clauses has on solver performance. We found that general solvers perform better with additional symmetry-breaking clauses, while some solvers that rely on special solving techniques perform worse.

1 Introduction

The pigeonhole and mutilated chessboard problems are challenging SAT benchmarks with exponentially-sized resolution proofs [1, 17]. Some SAT solvers can exploit the canonical structure of these two problems to efficiently solve them: for example, the hand-crafted schedules to construct binary decision diagrams (BDDs) in the solver PGBDD [7], the propagation redundancy proof system in the solver SaDiCaL [11], and the general cardinality resolution in the preprocessing stage of the solver Lingeling [3] and similarly Sat4j [?]. However, depending on the implementation these techniques may yield inconsistent performance under variations in the problems. A new suite of challenging benchmarks would serve to identify when these techniques fail and to guide the development of more robust solvers.

In this paper, we present a suite of benchmarks based on finding perfect matchings in bipartite graphs. The benchmark family leverages the general principle that randomness increases problem instance difficulty by mixing encodings and randomizing graph constructions. It also makes use of the perfect matching problem structure to incorporate redundant constraints. By varying multiple parameters, this benchmark family provides sets of formulas that are generally hard across solvers, as well as subsets particular to each solver that cause degraded performance. To further investigate the resilience of solving techniques, we added symmetry-breaking clauses to the benchmarks. In some instances, these clauses can make the problem easier by removing potential sub-matchings. However, we found that solvers with special solving techniques can have worse performance while other solvers improve.

In general, our experimental results show that solvers with powerful reasoning techniques that go beyond resolution are not robust—they only succeed when certain properties, such as cardinality constraints, are encoded in specific ways. Our generator of bipartite matching instances allows us to pinpoint the weaknesses of each tool. Ideally, the generator will help solver developers make their tools robust, both in terms of instance structure and the encoding. Already, our tool has revealed the lack of robustness in Kissat, the winner of the Main track of

the 2020 SAT Competition [5]. We did not expect this result, as KISSAT does not use techniques that go beyond resolution. Making KISSAT more robust would likely improve its performance on many real-world formulas.

2 Bipartite Problem Classes and Constraint Encodings

A *perfect matching* of an undirected graph is a subset of the edges such that every node is an endpoint of exactly one edge. If the nodes of a graph can be partitioned into two such that all edges of the graph have one endpoint in each partition, we call the graph a *bipartite graph.* A $K_{n,m}$ is a bipartite graph with node partitions of size n and m that is fully connected, meaning it has an edge between all possible pairs of nodes, one node from each partition.

The pigeonhole and mutilated chessboard problems can be generalized to the unsatisfiable problem of finding a perfect matching on a connected bipartite graph with unequal node partition sizes. We discuss how the two problems and their variants can be represented as bipartite graphs, how random bipartite graphs can be generated from random spanning trees, and how to extract a CNF formula from a graph using the direct, Sinz, and linear at-most-one encodings.

2.1 Problem classes

The **pigeonhole problem** is a classic combinatorial problem where $n+1$ pigeons must be uniquely placed into n holes. As a graph, each of the $n+1$ pigeons has edges to each of the n holes, forming a fully connected $K_{(n+1),n}$ bipartite graph. No perfect matching exists, because one partition has an extra node.

The **mutilated chessboard** is an $n \times n$ grid of alternating black and white squares with two opposite corners removed. The task is to cover the board with 2×1 dominoes. As a bipartite graph, nodes representing the squares are placed into two partitions according to their colors. Edges connect nodes representing squares that are adjacent horizontally or vertically. For the case where n is even, there will be $n^2/2$ nodes in one partition and $n^2/2-2$ in the other, precluding any perfect matching.

Random bipartite graphs are used to explore non-structured problem instances. The *density* of a bipartite graph with node partitions of size n and m is defined as the ratio of the number of edges to the number of possible edges $n \times m$. To generate a random connected bipartite graph, edges are added randomly to a random spanning tree until the desired density is reached. Note that a density of 1 for partition sizes $n+1$ and n is the n-pigeonhole problem.

Our benchmark generator can create encodings of perfect matching problems on random bipartite graphs for any n and m, but the graphs we used in our experiments only had partition sizes $n+1$ and n. We do note that a solver that implements parity reasoning could easily determine that formulas encoding perfect matchings for graphs with odd difference $|n-m|$ are unsatisfiable. However, none of the tested solvers employs such reasoning, so we only experimented with random graphs with partition difference of one.

2.2 Constraint Encodings

We can translate the problem of finding a perfect matching on a bipartite graph into a CNF formula by associating a Boolean variable with each edge in the graph. Assigning true to a variable means that the edge is in the matching. Constraints are added to ensure that each node has exactly one incident edge in any matching. A satisfying assignment is a perfect matching.

One way to encode the constraints of the perfect matching problem is to use at-most-one (AMO) and at-least-one (ALO) constraints on the nodes. An ALO constraint for a node is a clause containing one positive variable for each incident edge to that node. The variables are joined by disjunctions, indicating that at least one of the edges must be in the matching. An AMO constraint for a node is a collection of clauses that, together, encode that at most one of the edges incident to that node can be in the matching. These clauses can be encoded in several different ways, each with varying numbers of clauses and auxiliary variables. Three particular encodings are discussed below. Typically, ALO constraints are placed on the larger partition and AMO on the smaller, but placing Exactly-One constraints (combining AMO and ALO) on both partitions introduces redundant clauses that may help or hinder SAT solvers.

Direct Encoding for an AMO constraint is given by a set of binary clauses with negative literals and no auxiliary variables. $\text{AMO}(x_1, ..., x_n)$ is encoded as the conjunction of $(\overline{x}_i \vee \overline{x}_j)$ with $1 \leq i < j \leq n$, resulting in $n(n-1)/2$ binary clauses.

Sinz Encoding [14] adds signal variables that propagate values across variables in the set. There are three types of clauses: the first sets a signal for a variable to true if that variable is true, the second sets a signal for a variable to true if the preceding signal is true, and the third blocks both the variable and preceding signal from being true at the same time. This encoding method will create $n-1$ new variables and $3(n-1)+2$ binary clauses.

$$\overline{x}_i \vee s_i \quad \text{for } 1 \leq i \leq n \qquad\qquad \overline{s}_i \vee s_{i+1},\ \ \overline{s}_i \vee \overline{x}_{i+1} \quad \text{for } 1 \leq i < n$$

Linear Encoding uses a combination of new variables and the direct encoding in such a way that keeps the overall encoding compact. The linear encoding of $\text{AMO}(x_1, \ldots, x_n)$ uses the direct encoding for $n \leq 4$ and splits on $n > 4$ using auxiliary variables according to the following recursion:

$$\text{Direct}(x_1, x_2, x_3, y) \wedge \text{AMO}(\overline{y}, x_4, .., x_n)$$

where each y is a new auxiliary variable. The encoding uses $(n-3)/2$ auxiliary variables and $3n-6$ clauses. The cutoff of $n=4$ (commonly used in practice) was selected as the "optimal" value to minimize the sum of the number of variables and the number of clauses.

2.3 Solvers

We introduce each of the four solvers we used in our experiments and motivate why each solver was chosen. Three of these implement techniques that are well tuned for the canonical versions of the pigeonhole and mutilated chessboard problems.

Kissat [5] is an optimized implementation of CaDiCaL [4], which is a standard, state-of-the-art CDCL solver. Aside from some preprocessing techniques, Kissat is not especially attuned to solving pigeonhole or mutilated chessboard problem instances, making it a good baseline for comparison.

Lingeling [3] is a CDCL solver that specializes in pre-processing. Lingeling employs a pre-processing heuristic to perform cardinality resolution [6] in the same vein as cutting plane proofs [8]. Lingeling can often prove the unsatisfiability of pigeonhole and mutilated chessboard problems before starting CDCL search.

SaDiCaL [11] is a satisfaction-driven clause learner that extends CDCL solvers by learning PR clauses [12] based on "positive reducts." This enables SaDiCaL to efficiently solve hard problems that require exponentially-sized resolution proofs. SaDiCaL is more efficient than CDCL on pigeonhole and mutilated chessboard problems.

PGBDD [7] is a BDD-based solver that generates extended resolution proofs [13, 15]. Manually constructed schedules of BDD operations perform well on the mutilated chessboard

and pigeonhole problems (under the Sinz encoding). Without a hand-crafted schedule, PGBDD resorts to *bucket elimination*, processing BDDs in order of their variables, using conjunction and existential quantification operations for each bucket.

PGBDD-Sched is a modification of PGBDD that uses bucket permutations to approximate handcrafted schedules.[1] A (potentially sparse) grid can be induced for any bipartite graph with the Sinz encoding where nodes are variables and edges are constraints. The variable ordering is computed by counting row-wise, and the bucket permutation ordering is computed by counting column-wise. These orderings are extracted from the graph structure. Further implementation details are beyond the scope of this paper.

3 Experimental Results

We implemented a benchmark generator, BiPartGen, that generates CNF formulas for the problems and encodings discussed in the previous section. BiPartGen also supports the addition of clauses to break perfect matching symmetries. The repository can be found at https://github.com/jreeves3/BiPartGen-Artifact.

We ran our experiments on StarExec [16]. The specs for the compute nodes can be found online.[2] The compute nodes that ran our experiments were Intel Xeon E5 cores with 2.4 GHz, and all experiments ran with 8 GB of memory and a 900 second timeout. Solvers were evaluated using the PAR-2 scheme where a solver timeout is counted as twice the timeout in computing the average. We thank the community at StarExec for providing these computational resources.

3.1 Random Graph Experiments

Experiments were conducted on random graphs over a range of densities and encodings. Graphs were constructed by fixing the number of edges and altering the node count. In general, edge counts were tuned to the 1800 seconds time-out. While the focus is not on strict inter-solver comparison, larger instances (higher edge counts) with similar densities are more difficult. At each density, 60 random graphs were constructed, and average solving times are reported. Experiments with -A have AMO constraints on the nodes in the larger partition and ALO for the other, while -B experiments use Exactly-One constraints for all nodes. Mixed experiments randomly select one of the three AMO encodings for each node independently.

Figure 1 (top) shows the experimental results for Kissat. Kissat is relatively stable under constraint variation at lower densities, as seen by the tightness within -A and -B groups. On the other hand, redundant constraints degrade performance at higher densities. Specifically, the performance loss for high density Direct-B is surprising. This abnormality was duplicated on the other CDCL solvers CaDiCaL and Glucose_4.1 [2]. (We performed experiments with these additional solvers solely to show that other CDCL solvers suffer the same unexpected performance loss on these high-density graphs with redundant constraints.) For each of these solvers, performance is significantly improved by disabling bounded variable elimination [10]. For ten graphs at density 0.98, the execution time came down from all time outs to approximately 108 and 12 seconds for Glucose_4.1 and Kissat, respectively. This suggests that the problem is not with Kissat's default heuristics. This presents a novel problem set where these standard

[1] Bucket permutations were recently developed by the authors to generalize schedules over variables instead of clauses, requiring less bookkeeping and leading to a simpler implementation which make up for the slight performance loss.

[2] https://starexec.org/starexec/public/about.jsp

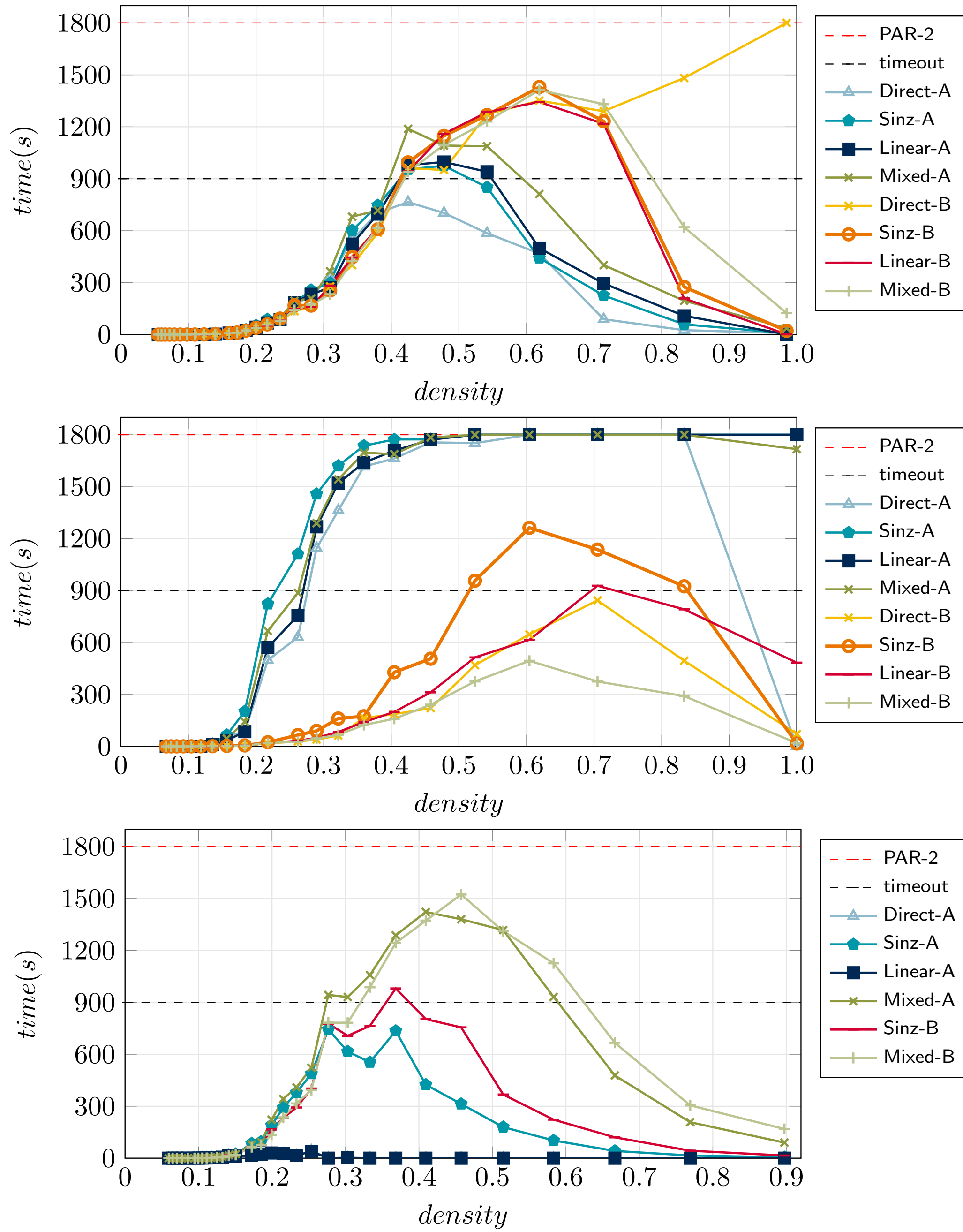

Figure 1: Average execution time over 60 seeds of random bipartite matchings with fixed edge count over different AMO encodings. Top: Kissat on random graphs with 130 edges; middle: SaDiCaL on random graphs with 110 edges; bottom: Lingeling on random graphs with 140 edges.

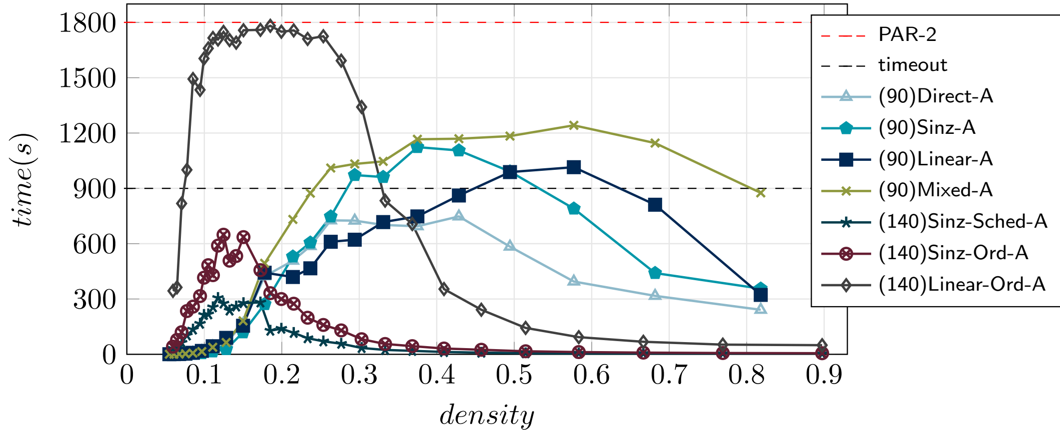

Figure 2: Average execution time over 60 seeds of random bipartite graph matchings with fixed edge counts at 90 or 140 over the different AMO encodings, on PGBDD with variable ordering (Ord) and bucket permutation (Sched) options.

solving techniques are detrimental. The breakdown of BVE should be further investigated in this context.

The results for SaDiCaL are given in Figure 1 (middle). In the case of pigeonhole, i.e., a density of 1.0, the tuned heuristics allow near instant solving in Direct-A. However, these heuristics do not benefit other -A experiments. Even for Direct-A, SaDiCaL times out on all 60 graphs at a density of 0.8. Following the opposite trend of Kissat, SaDiCaL performs better with additional constraints in the -B experiments and with the mixed encodings. This suggests SaDiCaL cannot consistently leverage structured formulas, except for the Direct-A pigeonhole benchmark pattern.

The results for Lingeling are given in Figure 1 (bottom). The cardinality reasoning in its preprocessor can handle the direct and linear encodings. The -B experiments for these encodings were omitted as they were also solved during preprocessing. This was not the case for the Sinz encoding, yielding stark differences in performance. Performance degrades even further with the mixed encodings, suggesting a higher preference for consistency. Regardless of encodings, Lingeling is resilient to adding redundant constraints.

The experiments in Figure 2 show PGBDD's poor performance on small graphs with 90 edges. In response, we considered two additional configurations that make use of scheduling techniques. While -Sched implements a bucket permutation and variable ordering strategy, -Ord simply places auxiliary variables next to adjacent problem variables in the variable ordering. Both -Ord and -Sched are run on graphs with 140 edges, showing that these configuration can outperform Kissat on the same sized problem instances. This shows that simple automatically generated schedules can solve hard random formulas. On the other hand, linear and direct encodings are problematic for PGBDD due to the connectedness of variables.

The benchmark family presented in this section provides enough variation that significant differences appear across solvers. For example, SaDiCaL does well on mixed encodings, Kissat suffers on the direct encoding with redundant constraints, and Lingeling cannot use cardinality reasoning on the Sinz encoding. In addition, the general hardness of these problems can be seen by the overlap of solvers plateauing with high execution times between densities 0.4 and 0.7. These middle density formulas have enough edges for the search space to be large, scattered across enough nodes for unit propagation to be less effective. For these problems,

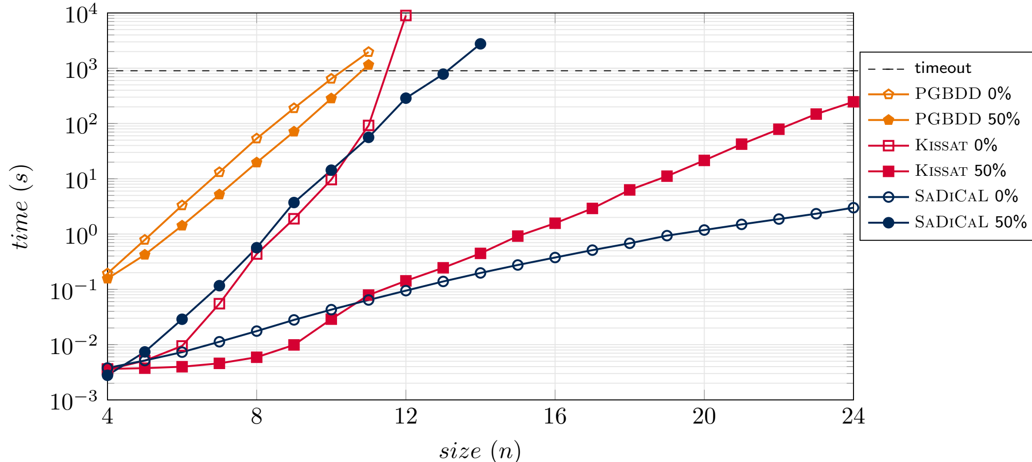

Figure 3: Execution time (in seconds) of solvers on canonical pigeonhole problems and on pigeonhole problems with 50% of the symmetries broken. Lingeling is not pictured because the addition of symmetry-breaking clauses had no effect.

PGBDD performs well with additional variable or bucket ordering information. Improvements to SaDiCaL's propagation redundant clause selection heuristics and Lingeling's cardinality constraint detection could bear similar results.

3.2 Symmetry-breaking Experiments

Many computational problems contain symmetries. For example, the presence of $K_{d,d}$ subgraphs in a perfect matching problem makes the search space larger by contributing many partial satisfying assignments which are permutations of the perfect matchings on the subgraphs. One way to remove these symmetries is to add symmetry-breaking clauses to the CNF formula. Previous work has shown that adding symmetry-breaking clauses to pigeonhole problem instances makes finding a proof of unsatisfiability an order of magnitude more efficient and allows solvers to find proofs for problem instances three times larger [9]. However, additional symmetry-breaking clauses may disrupt special solving techniques. The addition of these clauses to the formulas in our benchmark family provides another metric against which to test the robustness of solvers.

We configured BiPartGen to add symmetry-breaking clauses to disallow all but one of the possible perfect matchings on any $K_{2,2}$ or 6-cycle subgraph. Because in the worst case the number of symmetry-breaking clauses is exponential in the size of the graph, we also configured BiPartGen to optionally add a fraction of the symmetry-breaking clauses. We tested the solvers against formulas with no symmetries broken to establish a baseline and with half and with all possible symmetry-breaking clauses for certain subgraph types added.

For pigeonhole problems, we tested the solvers on problem instances of size $n \in [4, 24]$, of all three encoding types, and with symmetries due to $K_{2,2}$ subgraphs broken. For $n > 24$, the problem instances grew too large to run. Only $K_{2,2}$ symmetries were broken because symmetries of many larger subgraphs are automatically broken if their $K_{2,2}$ subgraph symmetries are. For mutilated chessboard problems, we tested the solvers on problem instances of even size $n \in [4, 30]$, of the direct and Sinz encodings, and with symmetries due to 6-cycles and $K_{2,2}$ subgraphs broken. For $n > 30$, all solvers time out on some encoding. Because each node in the mutilated chessboard graph has degree at most 4, the linear encoding is exactly the direct encoding, so

Table 1: Problem sizes at which timeout occurred on symmetry-broken pigeonhole (top four rows) and mutilated chessboard problems (bottom four rows). The columns indicate the fraction of symmetry-breaking clauses added. If a solver didn't time out on instances of size $n \leq 24$ for pigeonhole and $n \leq 30$ for mutilated chessboard, then runtime (in seconds) is reported in parens.

Encoding	Direct			Sinz			Linear		
Solver	0%	50%	100%	0%	50%	100%	0%	50%	100%
PGBDD	11	11	23	10	8	22	14	10	23
KISSAT	12	(246.87)	(0.03)	14	(3.93)	(0.04)	13	(3.12)	(0.048)
LINGELING	(0.01)	(0.03)	(0.08)	15	(4.75)	(0.20)	(0.01)	(0.01)	(0.04)
SADICAL	(3.00)	14	20	9	13	(10.11)	9	14	23
PGBDD	18	16	14	18	8	8	–	–	–
KISSAT	20	22	26	14	22	26	–	–	–
LINGELING	(0.02)	(0.02)	(0.03)	18	20	24	–	–	–
SADICAL	14	16	16	14	14	16	–	–	–

the linear encoding was not tested.

We ran our experiments with 0%, 50%, and 100% of the symmetry-breaking clauses mentioned above. The execution times were averaged over 60 seeds using the default constraints (-A from Section 3.1). Our findings are summarized in Table 1. Figure 3 shows plots of runtime curves.

For the pigeonhole problems, PGBDD and KISSAT benefit from the addition of symmetry-breaking clauses in all encodings. LINGELING is helped by their addition in the Sinz encoding. SADICAL had mixed results: on the direct encoding, the additional clauses cause timeouts on smaller instances, but on the other two, the additional clauses help performance. Figure 3 shows this behavior: the addition of 50% of the symmetry-breaking clauses help PGBDD, significantly help KISSAT, and cause SADICAL to time out on smaller problem instances.

For mutilated chessboard problems with symmetries of 6-cycles and $K_{2,2}$ subgraphs broken, KISSAT and SADICAL benefit from the addition of symmetry-breaking clauses. PGBDD does not benefit from the additional clauses at all, timing out on smaller problem instances. LINGELING has almost no change for the direct encoding and is helped for the Sinz encoding.

4 Conclusion and Future Work

In this work we presented a benchmark family generated from structured and random bipartite graph problems. We provided variation over several constraint and AMO encoding options with optional symmetry-breaking clauses. These formulas allowed us to evaluate and locate weaknesses in several solvers. Our CNF generator yielded a hard set of problems that existing and future solvers can use to improve their robustness.

The findings of this work have opened several avenues for future work. First, the problem space can be expanded with structured graph constructions considering node degree and connectedness, and extensions to k-partite graphs can be tested in search of harder problems. Second, symmetry-breaking clauses can be added in different systematic ways to determine which clauses are more important. And third, the special solving techniques of the presented solvers can be made more robust under this problem class.

References

[1] Michael Alekhnovich. Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science*, 310(1):513–525, 2004.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404, 2009.

[3] Armin Biere. Lingeling, Plingeling, PicoSAT, and PrecoSAT at SAT race 2010. unpublished, 2010.

[4] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, and YalSAT entering the SAT competition 2017. unpublished, 2017.

[5] Armin Biere, Katalin Fazekas, M. Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. unpublished, 2020.

[6] Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing*, volume 8561 of *LNCS*, pages 285–301. Springer, 2014.

[7] Randal E. Bryant and Marijn J. H. Heule. Generating extended resolution proofs with a BDD-based SAT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, volume 12651 of *LNCS*, pages 76–93, 2021.

[8] William Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[9] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proc. of the 5th Int. Conference on Principles of Knowledge Representation and Reasoning* (*KR 1996*), pages 148–159. Morgan Kaufmann, 1996.

[10] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of the 8th Int. Conference on Theory and Applications of Satisfiability Testing* (*SAT 2005*), volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[11] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 11427 of *LNCS*, pages 41–58. Springer, 2019.

[12] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. PRuning through satisfaction. In *Haifa Verification Conference (HVC)*, volume 10629 of *LNCS*, pages 179–194, 2017.

[13] Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *LNCS*, pages 54–60. Springer, 2006.

[14] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 3709 of *LNCS*, pages 827–831, 2005.

[15] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining BDDs. In *Computer Science Symposium in Russia (CSR)*, volume 3967 of *LNCS*, pages 600–611, 2006.

[16] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning*, volume 8562 of *LNCS*, pages 367–373. Springer International Publishing, 2014.

[17] Alasdair Urquhart. Hard examples for resolution. *J.ACM*, 34(1):209–219, 1987.