# Clausal Proofs for Pseudo-Boolean Reasoning

Randal E. Bryant[1], Armin Biere[2], and Marijn J. H. Heule[1]

[1] Carnegie Mellon University, Pittsburgh, PA, United States
{Randy.Bryant, mheule}@cs.cmu.edu
[2] Albert-Ludwigs University, Freiburg, Germany
biere@cs.uni-freiburg.de

**Abstract.** Pseudo-Boolean (PB) solvers can apply powerful reasoning methods to determine whether a set of parity or cardinality constraints is unsolvable. By converting the intermediate constraints generated by the PB solver into ordered binary decision diagrams (BDDs), a proof-generating, BDD-based Boolean satisfiability (SAT) solver can generate a clausal proof that the original conjunctive normal form formula encoding the constraints is unsatisfiable. Working together, the two solvers can generate proofs of unsatisfiability for problems that are intractable for other proof-generating SAT solvers. The PB solver can, at times, detect that the proof can exploit modular arithmetic to give smaller BDD representations and therefore shorter proofs.

## 1 Introduction

Like all complex software, modern satisfiability (SAT) solvers for propositional formulas in conjunctive normal form (CNF) are prone to bugs. In seeking to maximize their performance, developers may attempt optimizations that are either unsound or incorrectly implemented. Requiring a solver to be formally verified, so that all of its results can be trusted, is not feasible for current solvers. On the other hand, ensuring that each execution of the solver yields the correct result has become a standard requirement. For a satisfiable formula, the solver can generate a purported solution, and this can be checked directly. For an unsatisfiable formula, the solver can produce a proof of unsatisfiability in a logical framework that enables checking by an efficient and trusted proof checker. Proof generation is a vital capability when SAT solvers are used for formal correctness and security verification, and for mathematical theorem proving.

Most high-performance, proof-generating SAT solvers are based on conflict-driven, clause-learning (CDCL) algorithms [39]. Although the methods used by earlier solvers were limited to steps that could be justified within a resolution framework [40, 48], modern solvers employ a variety of optimizations that require a more expressive proof framework, with the most common being Deletion Resolution Asymmetric Tautology (DRAT) [29,46]. Like resolution proofs, a DRAT proof is a *clausal proof* consisting of a sequence of clauses, each of which preserves the satisfiability of the preceding clauses. An unsatisfiability proof starts with the clauses of the input formula and ends with an empty clause, indicating logical falsehood. The fact that this clause can be derived from the original formula proves that the original formula cannot be satisfied.

Even with the capabilities of the DRAT framework, some solvers employ reasoning techniques for which they cannot generate unsatisfiability proofs. For example, the

LINGELING solver can automatically detect exclusive-or operations and cardinality constraints encoded in the input file and apply more powerful reasoning methods to detect that the formula is unsatisfiable [6]. However, the program cannot perform proof generation when these techniques are enabled. To overcome the proof-generating limitations of current solvers, some have suggested more powerful proof frameworks should be developed and deployed, for example, based on pseudo-Boolean constraints [26] or Binary Decision Diagrams [5]. Staying with DRAT would avoid the need to develop and certify new proof systems, formats, and checkers.

Current CDCL solvers do not use the full power of the DRAT framework. In particular, DRAT supports adding *extension variables* to a clausal proof, in the style of extended resolution [44]. These variables serve as abbreviations for formulas over existing input and extension variables. Compared to standard resolution, allowing extension variables can yield proofs that are exponentially more compact [18], and the same holds for the extension rule in DRAT. In general, however, CDCL solvers have been unable to exploit this capability, with the exception that some of their preprocessing and inprocessing techniques [8, 32] require extension variables [36]. One solver attempted to introduce extension variables as it operated [3], but it achieved only modest success.

In 2006, Biere, Jussila, and Sinz made the key observation that the underlying logic behind algorithms for constructing Reduced, Ordered Binary Decision Diagrams (BDDs) [10] can be encoded as steps in an extended resolution framework [33, 43]. By introducing an extension variable for each BDD node generated, the logic for each recursive step of standard BDD operations can be expressed with a short sequence of proof steps. BDDs provide a systematic way to exploit the power of extension variables. The recently developed solver PGBDD [11, 12] builds on this work with a more general capability for existentially quantifying variables. It can generate unsatisfiability proofs for classic challenge problems for which the shortest possible standard resolution proofs are of exponential size.

We show that BDDs can provide a bridge between *pseudo-Boolean reasoning* and clausal proofs. Pseudo-Boolean (PB) constraints have the form $\sum_{j=1,n} a_j\, x_j \rhd b$, where each variable $x_j$ can be assigned value 0 or 1, the coefficients $a_j$ and constant $b$ are integers, and the relation symbol $\rhd$ is either $=$, $\geq$, or $\equiv \mod r$ for some modulus $r$. A PB solver can employ Gaussian elimination or Fourier-Motzkin elimination [20, 47] to determine when a set of these constraints is unsatisfiable. Augmenting PGBDD with a pseudo-Boolean solver enables the use of these powerful reasoning methods while generating DRAT proofs of unsatisfiability.

To enable proof generation, the PB solver generates BDD representations of its intermediate constraints and has proof-generating BDD operations construct proofs that each of these constraints is logically implied by previous constraints. When the PB solver reaches a constraint that cannot be satisfied, e.g., the equation $0 = 2$, the constraint will be represented by the *false* BDD leaf $\perp$, which yields a proof step consisting of the empty clause. The resulting proof is checkable within the DRAT framework without any reference to pseudo-Boolean constraints or BDDs. Barnett and Biere [5] also proposed using BDDs when proving that the constraints generated by a PB solver were logically implied by their predecessors, but they proposed doing so in a separate proof framework rather than as the solver operates.

As an optimization, we show that the PB solver can automatically detect cases where the unsatisfiability proof can be generated using modular arithmetic. This leads to more compact BDD representations, and therefore shorter proofs.

We demonstrate the power of augmenting PGBDD with pseudo-Boolean reasoning by showing that this combination can achieve polynomial scaling on two classes of problems for which CDCL solvers have exponential performance. These include parity constraints involving exclusive-or operations [16, 45] and cardinality constraints, including the mutilated chessboard [2] and pigeonhole problems [28]. Although PGBDD on its own can also achieve polynomial scaling for both classes of problems, incorporating pseudo-Boolean reasoning makes the solver much more robust. It can handle wider variations in the problem definition, the method of encoding the problem into CNF, and the BDD variable ordering. It also operates with a greater degree of automation, requiring no guidance from the user. These capabilities eliminate major shortcomings of PGBDD. As another perspective, we enable the reasoning capabilities used by LINGELING to be part of a DRAT proof-generating SAT solver.

## 2 Pseudo-Boolean Constraints

Let $x_j$, for $1 \leq j \leq n$, be a set of variables, each of which may be assigned value 0 or 1, and $a_j$, for $1 \leq j \leq n$, be a set of integer coefficients. Constant $b$ is also an integer. A *pseudo-Boolean* constraint is of the form $\sum_{j=1,n} a_j\, x_j \rhd b$, with $\rhd$ defining the relation between the left-hand weighted sum and the right-hand constant. For an *integer equation*, $\rhd$ is $=$, i.e., the two sides must be equal. For an *ordering constraint*, $\rhd$ is $\geq$. For a *modular equation*, $\rhd$ is $\equiv \bmod r$, where $r$ is the chosen modulus.

Three constraint types are of special importance for solving cardinality problems. An *at-least-one* (ALO) constraint is an ordering constraint with $a_j = +1$ for all $j$, and $b = +1$. An *at-most-one* (AMO) constraint is an ordering constraint with $a_j = -1$ for all $j$, and $b = -1$. An *exactly-one* constraint is an integer equation with $a_j = +1$ for all $j$ and $b = +1$.

### 2.1 BDD Representations

Many researchers have investigated the use of BDDs to represent pseudo-Boolean constraints [1,23,31]. As examples, Figure 1 shows BDD representations of the three forms of constraints for $n = 10$ and $b = 0$, with $a_j = +1$ for odd values of $j$ and $-1$ for even values. The modular equation has $r = 3$. As can be seen, the BDDs for both the integer equation (A) and ordering constraint (B) have an increasing number of nodes at each level for the first $n/2$ levels, with a node at level $k$ for each possible value of the *prefix sum* $\sum_{j=1,k-1} a_j\, x_j$. As the level $j$ approaches $n$, however, the number of nodes at each level decreases. If a prefix sum becomes too extreme on the negative side, it becomes impossible for the remaining values of the sum to reach $b = 0$. For the integer equation, a similar phenomenon happens if a prefix sum becomes too extreme on the positive side. For an ordering constraint, a sufficiently positive prefix sum will guarantee that the total sum will be at least 0. For the modular sum (C), the number of nodes at any level cannot exceed $r$—one for each possible value of the prefix sum modulo $r$.

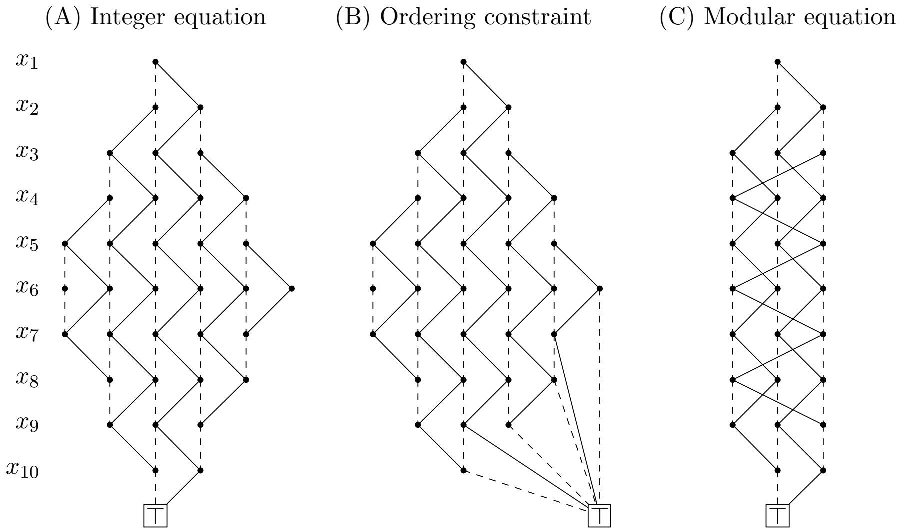


**Fig. 1.** Example BDD representations of pseudo-Boolean equations and ordering constraints. Solid (respectively, dashed) lines indicate the branch when the variable is assigned 1 (resp., 0). The leaf representing the *false* Boolean constant $\perp$ and its incoming edges are omitted.

Letting $a_{\max} = \max_{1 \leq j \leq n} |a_i|$, the BDD representation of an integer equation or ordering constraint will have at most $2\, a_{\max} \cdot n$ nodes at any level, while the representation of a modular equation will have at most $r$ nodes at any level. Although large values of $a_{\max}$ ($a_{\max} \gg n$), can cause the BDDs to be of exponential size [1, 31], our use of them will assume that both $a_{\max}$ and $r$ are small constants. The BDD representations will then be $O(n^2)$ for integer equations and ordering constraints, and $O(n)$ for modular equations. These bounds are independent of the BDD variable ordering.

Most BDD operations are implemented via the *Apply* algorithm [10], recursively traversing a set of argument BDDs to either construct a new BDD or to test some property of existing ones. The BDDs representing pseudo-Boolean constraints are *levelized*: every branch from a node at level $i$ goes to a leaf node or to a node at level $i + 1$. We can therefore derive a bound on the maximum number of recursive steps to perform an operation on $k$ argument BDDs, assuming both $a_{\max}$ and $r$ are small constants. Due to the caching of intermediate results, the maximum number of steps at each level will be bounded by the product of the number of argument nodes at this level. The operation will therefore have worst-case complexity $O(n^{k+1})$ for integer equations and ordering constraints, while it will have complexity $O(k \cdot n)$ for modular equations.

### 2.2 Solving Systems of Equations with Gaussian Elimination

We use a formulation of Gaussian elimination that scales each derived equation, rather than dividing by the pivot value [4, 41]. Performing the steps therefore requires only addition and multiplication. This allows maintaining integer coefficients and automatically detecting a minimum, possibly non-prime, modulus for equation solving.

Consider a system of $m$ integer or modular equations $E$, where each equation $\mathbf{e}_i$, for $1 \leq i \leq m$ is of the form $\sum_{j=1,n} a_{i,j}\, x_j = b_i$. Applying one step of Gaussian elimination involves selecting a *pivot*, consisting of an equation $\mathbf{e}_s$ and a variable $x_t$ such that $a_{s,t} \neq 0$. Then an equation $\mathbf{e}'_i$ is generated for each value of $i$:

$$\mathbf{e}'_i, = \begin{cases} \mathbf{e}_i & a_{i,t} = 0 \\ -a_{i,t} \cdot \mathbf{e}_s + a_{s,t} \cdot \mathbf{e}_i, & a_{i,t} \neq 0 \end{cases} \tag{1}$$

where operations $+$ and $\cdot$ denote addition and scalar multiplication of equations. Observe that $a'_{i,t} = 0$ for all equations $\mathbf{e}'_i$. Letting $E' = \{\mathbf{e}'_i | i \neq s\}$, this step has reduced both the number of equations and the number of variables in the system by one. By renumbering the equations in $E'$ to range from 1 to $m-1$, this process can be repeated.

Repeated applications of the elimination step will terminate when either (1) all equations have been eliminated, or (2) an unsolvable equation is encountered. In the former case, the system may have solutions, but these may assign values other than 0 and 1 to the variables, except for the special case of modular equations with $r = 2$. If some elimination step generates an equation of the form $0 = b$ with $b \neq 0$, then this equation has no solution in any case, and therefore neither did the original system. Our proofs of unsatisfiability rely on reaching this condition.

For the modular case, all coefficients and the constants are kept within the range $0$ to $r-1$. For integer equations, the coefficients can grow exponentially in $m$. Fortunately, the cardinality problems we consider only require coefficient values $-1$, $0$, and $+1$.

As we have seen, the BDD representations of modular equations have bounded width, making them both more compact and making the algorithms that operate on them more efficient than for integer equations. As we will see, the unsatisfiability proof generated by applying Gaussian elimination to a system of modular equations can be significantly more compact than for the same equations over integers. This gives rise to an optimization we call *modulus auto-detection.* The idea is to apply Gaussian elimination to a set of integer equations, recording the dependencies between the equations generated, but without performing any proof generation. Once the solver reaches an equation of the form $0 = b$ where $b \neq 0$, it chooses the smallest $r \geq 2$ such that $b \bmod r \neq 0$. It then generates a proof, reinterpreting the Gaussian elimination steps using modulo-$r$ arithmetic. Since the only operations of (1) are multiplication and addition, the final equation will be $0 \equiv b \pmod{r}$, which has no solution. Here we can see that allowing $r$ to be composite is both valid and may be optimal. For example, the smallest choice for $b = 30$ would be $r = 4$, rather than the prime $r = 7$. Auto-detection can be applied whenever Gaussian elimination encounters an unsolvable equation.

### 2.3 Solving Systems of Ordering Constraints with Fourier-Motzkin Elimination

Consider a system of $m$ constraints, with constraint $\mathbf{c}_i$, for $1 \leq i \leq m$, of the form $\sum_{j=1,n} a_{i,j}\, x_j \geq b_i$. Applying one step of Fourier-Motzin elimination [20, 47] to this system involves identifying a *pivot*, consisting of a variable $x_t$ such that $a_{i,t} \neq 0$ for at least one value of $i$. The constraints are partitioned into sets $C^+$, $C^-$, and $C^0$, depending on whether each $a_{i,t}$ is positive, negative, or zero, respectively. For each pair $i$ and $i'$ such that $\mathbf{c}_i \in C^+$ and $\mathbf{c}_{i'} \in C^-$, a new constraint $\mathbf{c}_{i,i'}$ is generated as:

$$\mathbf{c}_{i,i'} = -a_{i',t} \cdot \mathbf{c}_i + a_{i,t} \cdot \mathbf{c}_{i'} \tag{2}$$

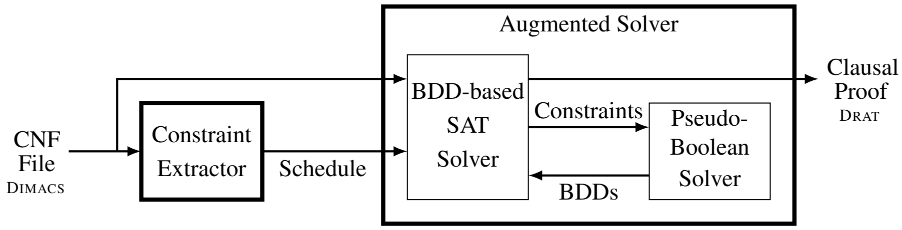


**Fig. 2.** Overall Structure of Solver. It augments the BDD-based SAT solver PGBDD with inferences from a pseudo-Boolean constraint solver. The constraint extractor is a separate program.

Then a new system of constraints is formed, consisting of constraints $C^0$ along with all constraints of the form $\mathbf{c}_{i,i'}$. Observe that the generated constraints will all have $a_{i,i',t} = 0$, and therefore this step has reduced the number of variables in the constraints by one, but it has possibly increased the number of constraints.

As with Gaussian elimination, repeated application of the elimination step will terminate when either (1) all variables have been eliminated or (2) an unsolvable constraint is encountered. In the former case, the constraints can be satisfied, although possibly assigning values other than 0 or 1 to some of the variables. An unsolvable constraint is one where the sum of the positive coefficients is less than the constant term. If such a constraint is encountered, then the original system of constraints has no solution.

Fourier-Motzkin elimination would appear to be hopelessly inefficient. The number of constraints can grow exponentially as the elimination proceeds, and the coefficients can grow doubly exponentially. Fortunately, the cardinality problems we consider have the property that for any variable $x_t$, there is at most one constraint $\mathbf{c}_i$ having $a_{i,t} = +1$, at most constraint $\mathbf{c}_{i'}$ having $a_{i',t} = -1$, and no other constraint with a non-zero coefficient at position $t$. This property is maintained by each elimination step, and so the number of constraints will decrease with each step, and the coefficients will be restricted to the values $-1$, $0$, and $+1$.

## 3 Overall Solver Operation

Figure 2 illustrates the structure of our solver. As is shown, a combination of two programs follows the standard use model for proof-generating SAT solvers, reading the input CNF formula expressed in the standard DIMACS format and generating proofs in the standard DRAT format. No other guidance or hint is provided. The constraint extractor identifies pseudo-Boolean constraints encoded as clauses in the input file and generates a *schedule* indicating how clauses should be combined and quantified to derive BDD representations of the constraints. The BDD-based SAT solver PGBDD provides the capability to generate DRAT proof steps while performing BDD operations. PGBDD supplies the constraints to a PB solver, which applies either Gaussian elimination or Fourier-Motzkin elimination. The PB solver generates BDD representations of the constraints it generates, and the SAT solver generates a proof that each new constraint is logically implied by previous constraints. When the PB solver encounters an unsolvable constraint, the SAT solver generates an empty clause, completing the proof.

(A)
Exclusive-Or/Nor

CLAUSES
```
-1  2  6 0
 1 -2  6 0
 1  2 -6 0
-1 -2 -6 0
 3  6  7 0
-3 -6  7 0
-3  6 -7 0
 3 -6 -7 0
```

SCHEDULE
```
c 1 2 3 4
a 3
=2 0 1.1 1.2 1.6
c 5 6 7 8
a 3
=2 1 1.3 1.6 1.7
```

(B)
Exactly-one, direct encoding

CLAUSES
```
1 2 3 4 0
-1 -2 0
-1 -3 0
-1 -4 0
-2 -3 0
-2 -4 0
-3 -4 0
```

SCHEDULE
```
c 1 2 3 4 5 6 7
a 6
= 1 1.1 1.2 1.3 1.4
```

(C)
At-most-one, Sinz encoding [42]

CLAUSES
```
-1  5 0
-2  5 0
-1 -2 0
-5 -3 0
-5  6 0
-3  6 0
-6 -4 0
```

SCHEDULE
```
c 1 2 4 5
a 3
q 5
c 6 7
a 2
q 6
c 3
a 1
>= -1 -1.1 -1.2 -1.3 -1.4
```

**Fig. 3.** Examples of pseudo-Boolean constraints extracted from CNF representations. Schedules use a stack notation indicating clauses, conjunction and quantification operations, and constraints.

### 3.1 Constraint Extraction

The constraint extractor uses heuristic methods to identify how the clauses match standard patterns for exclusive-or/nor, at-least-one (ALO), and at-most-one (AMO) constraints. The heuristics are independent of any ordering of the clauses or variables, although they do depend on the polarities of the literals. The generated schedule indicates how to combine clauses and to quantify variables to give the different constraints. The schedule uses a stack notation, having the following commands:

| | |
|---|---|
| `c` $c_1, \ldots, c_k$ | Generate and push the BDDs for the specified clauses. |
| `a` $m$ | Pop the top $m+1$ elements. Combine with $m$ AND operations. Push the result. |
| `q` $v_1, \ldots, v_k$ | Quantify the top element by the specified variables. |
| $C$ $b\ a_1 \,.\, v_1, \ldots, a_k \,.\, v_k$ | Confirm that the top stack element implies the constraint |

The different constraint types $C$ are '`=`' for integer equations, '`=2`' for mod-2 equations, and '`>=`' for integer orderings. Each constraint line lists the constant $b$ and then indicates the non-zero terms as a combination of coefficient and variable, separated by '`.`'.

The operation of the extractor is illustrated via a series of examples in Figure 3. As shown in (A), a $k$-way exclusive-or or exclusive-nor is encoded with $2^{k-1}$ clauses, listing all possible combinations of the variables having odd (XOR) or even (XNOR) parity. The schedule lists the clause numbers, forms their conjunction, and indicates a mod-2 equation. The constant $b$ is 1 for exclusive-or and 0 for exclusive-nor.

As shown in (B), an exactly-one constraint can be expressed as a combination of an ALO constraint and an AMO constraint. The extractor assumes that any clause with all literals having positive polarity encodes an ALO constraint. In this example, a $k$-way AMO constraint is encoded directly as a set of $k\,(k-1)/2$ binary clauses.

Example (C) shows a 4-way AMO constraint encoded with auxiliary variables according to the method proposed by Sinz [42]. The extractor examines how variables occur in binary clauses. Those that occur only with negative polarity are assumed to be constraint variables, while those that have mixed polarity are assumed to be auxiliary variables. As is shown, the generated schedule for an AMO constraint encoded with auxiliary variables employs *early quantification* [13] to linearize the conjuncting of clauses and the quantification of auxiliary variables.

The heuristics used for identifying auxiliary variables and partitioning the clauses into distinct constraints apply to a wide range of AMO constraints, including those using hierarchical encodings [15, 34] and those considered in other constraint extraction programs [9]. Our method can be overly optimistic, labeling some subsets of clauses incorrectly. Fortunately, any such error will be quickly identified when the solver attempts to prove that the BDD generated by conjuncting the clauses and quantifying the auxiliary variables implies the BDD generated for the constraint.

### 3.2 Solver Operation

PGBDD starts by generating the BDD representation of each input clause, and then it performs a series of conjunction and existential quantification operations on BDDs [11, 12]. As the solver manipulates BDDs to track the solution state, it also generates clauses according to resolution and extension proof rules. The state of the solver at any time is captured by a set of *terms* $T_1, T_2, \ldots, T_n$, where each term $T_i$ consists of:

- A root node $u_i$ in the BDD.
- The extension variable associated with this node, also written as $u_i$.
- A unit clause, included in the generated clauses, consisting of extension variable $u_i$, asserting that BDD node $u_i$ has a path to the *true* leaf node $\top$ for any variable assignment that satisfies the input clauses.
- Implicitly, the set of defining clauses $\theta(u_i)$ for the nodes in the subgraph having root $u_i$. These are included in the generated clauses and provide the semantic model for the BDD within the proof framework.

The initial terms are generated from the clauses in the input formula. Successive terms are created by performing conjunction and existential quantification operations.

PGBDD can perform the proof-generating BDD operations APPLYAND, used to perform conjunction, and PROVEIMPLICATION, used to generate proofs of implication. The APPLYAND operation takes as arguments BDD roots $u$ and $v$, and it generates a BDD representation with root $w$ of their conjunction. It also generates a proof of the clause $\overline{u} \vee \overline{v} \vee w$, proving the implication $u \wedge v \rightarrow w$. The PROVEIMPLICATION operation performs implication testing without generating any new BDD nodes. It takes as arguments BDD roots $u$ and $v$, and it generates a proof of the clause $\overline{u} \vee v$, proving that $u \rightarrow v$. If the implication does not hold, it signals an error.

Several methods can be used to control the operation of PGBDD. When driven by commands from a schedule file, as it encounters a set of clause identifiers, it generates a term $T_i$ for each of the specified input clauses $C_i$ and pushes the term onto a stack. The prover generates the proof $\theta(u_i), C_i \vdash u_i$, i.e., that there will be a path from BDD root $u_i$ to leaf node $\top$ for any variable assignment that satisfies the clause.

When the solver encounters a conjunction or quantification command, it adds successive terms by applying the specified conjunction and existential quantification operation. Given newly generated BDD root $u_{n+1}$, it must prove that $u_{n+1}$ is *implication redundant* with respect to the existing terms. That is, if $u_{n+1}$ was generated by applying some operation to terms $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$, then it must generate a proof of the clause $\overline{u}_{i_1} \vee \overline{u}_{i_2} \vee \cdots \vee \overline{u}_{i_k} \vee u_{n+1}$. This clause can then be resolved with the unit clauses associated with the existing terms to yield the unit clause $u_{n+1}$, allowing a new term $T_{n+1}$ to be added. If some step generates a term $T_{n+1}$ with BDD representation $u_{n+1} = \bot$, then the prover will generate the empty clause, completing a proof of unsatisfiability.

To add pseudo-Boolean reasoning, we augment the program with a PB solver that can generate BDD representations of the intermediate constraints it creates. The SAT solver then generates a new term for each of these BDDs. The proof generator need not have any understanding of the operation of the PB solver, and vice-versa.

Suppose some set of input clauses encodes a pseudo-Boolean constraint, possibly using auxiliary variables, as illustrated in Figure 3. The SAT solver performs the series of conjunction and quantification operations specified by the schedule to reduce the clauses to a single term $T_n$ consisting of BDD root $u_n$ and unit clause $u_n$. The auxiliary variables have been quantified away, and so $u_n$ depends only on the constraint variables. It then passes the constraint to the PB solver, which generates a BDD representation with root $u_{n+1}$. The SAT solver then uses the PROVEIMPLICATION operation to generate the clause $\overline{u}_n \vee u_{n+1}$. This can be resolved with unit clause $u_n$ to generate the unit clause $u_{n+1}$, and so the BDD representation of the constraint becomes term $T_{n+1}$. This process is repeated to convert the input formulas into a set of pseudo-Boolean constraints, each represented as a term in the SAT solver.

Once the SAT solver has converted all of the input clauses into constraints, it passes control to the PB solver. From that point on, the SAT solver serves in a support role, generating proofs to justify each step of the PB solver. As the PB solver operates, it generates a BDD representation of each new constraint: for each equation $\mathbf{e}'_i$ generated by Gaussian elimination (1) or each ordering constraint $\mathbf{c}_{i,i'}$ generated by Fourier-Motzkin elimination (2). For a new BDD with root $u_{n+1}$ generated from constraints represented by terms $T_i$ and $T_j$, it uses the APPLYAND operation to generate the conjunction $w$ of the BDDs with roots $u_i$ and $u_j$, as well as a proof of the clause $\overline{u}_i \vee \overline{u}_j \vee w$. It then uses the PROVEIMPLICATION operation with arguments $w$ and $u_{n+1}$ to generate a proof of the clause $\overline{w} \vee u_{n+1}$. It can then resolve the unit clauses for terms $T_i$ and $T_j$ with the generated clauses to generate a proof of the unit clause $u_{n+1}$, and so the BDD representation of the constraint becomes term $T_{n+1}$. When some step of the PB solver generates an unsolvable equation or ordering constraint, it encodes the constraint as the *false* BDD leaf $\bot$, and the SAT solver will generate a proof of the empty clause.

As an optimization, we implemented an operation APPLYANDPROVEIMPLICATION combining the functions of APPLYAND and PROVEIMPLICATION. It takes as arguments

BDD roots $u$, $v$, and $w$ and generates a proof that $u \wedge v \rightarrow w$ without constructing the BDD representation of $u \wedge v$. We found this reduced the total proof lengths by over $2\times$.

# 4 Experimental Results

We implemented the solver PGPBS (for Proof-Generating Pseudo-Boolean Solver) by combining our BDD-based SAT solver PGBDD with a PB solver that can generate BDD representations of the constraints it creates.[3] The Gaussian elimination solver employs a standard greedy pivot selection heuristic, attributed to Markowitz [22, 38], that seeks to minimize the number of non-zero coefficients created. The Fourier-Motzin solver uses a similar heuristic for selecting pivot variables.

The executions of PGPBS follow the flow illustrated in Figure 2, with constraints extracted directly from the input CNF file, and with the generated schedule driving the operation of the solver. Some measurements were taken using a BDD variable ordering according to their numbering in the input file, while others used a random BDD variable ordering to assess the sensitivity to the variable ordering. All generated proofs were checked with an LRAT proof checker [19]. We used KISSAT, winner of the 2020 SAT competition [7], as a representative CDCL solver. All measurements labeled "PGBDD" are for the earlier version of the solver, without pseudo-Boolean reasoning [11, 12].

We measure the performance of the solver in terms of the total number of clauses in the generated proofs of unsatisfiability. This metric tracks closely with the solver runtime and has the advantage that it is machine independent. We set an upper limit of 100 million clauses for the reported proof sizes.

## 4.1 Urquhart Parity Formulas

Urquhart [45] defined a family of formulas that require resolution proofs of exponential size. Over the years, two sets of SAT benchmarks have been labeled as "Urquhart Problems" [14, 35]. The formulas are defined over a class of undirected, bipartite graphs, having bounded degree five, and being parameterized by a size value $m$, such that the number of nodes in the graph is $2m^2$. To transform a graph into a formula, each edge $\{i, j\}$ in the set of edges $E$ has an associated variable $x_{\{i,j\}}$. (We use set notation to emphasize that the order of the indices does not matter). Each vertex is assigned a polarity $p_i \in \{0, 1\}$, such that the sum of the polarities is odd. The clauses then encode that the sum for all values of $i$ and $j$ of $x_{\{i,j\}} + p_i$ equals 0 modulo 2. This is false of course, since each edge is counted twice in the sum, and the sum of the polarities is odd.

The two families of benchmarks differ in how the graphs are constructed. Li's benchmarks are based on the explicit construction of *expander* graphs [25, 37], upon which Urquhart's lower bound proof is based. Simon's benchmarks are based on randomly generated graphs and thus depend on the random seed. We generated five different formulas for each value of $m$. It is unlikely that Simon's graphs satisfy the expander property, but they are still very challenging benchmarks for most SAT solvers.

[3] The solver, along with code for generating and testing a set of benchmarks, is available at https://github.com/rebryant/pgpbs-artifact.

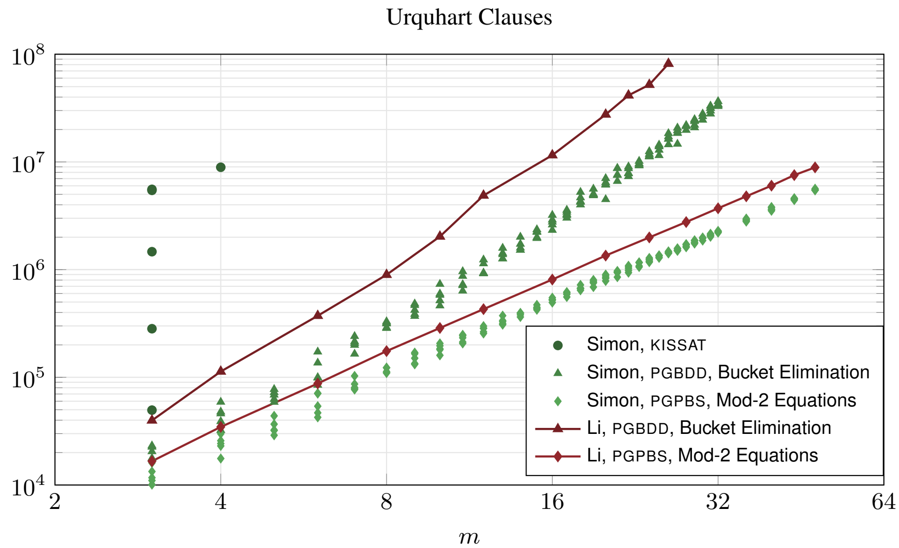


**Fig. 4.** Total number of clauses in proofs of two sets of Urquhart formulas.

Figure 4 shows the performance of the solvers, measured as the number of clauses as a function of $m$, for both Simon's and Li's benchmarks. The smallest instances of the benchmark have $m = 3$. As can be seen KISSAT is able to generate proofs for the Simon version for four cases with $m = 3$ and one with $m = 4$, but it is unable to handle any other cases, including not even the minimum instance for Li's benchmark. Measurements are shown for PGBDD running bucket elimination, a simple algorithm that processes clauses and intermediate terms with conjunction and quantification operations according to the levels of the topmost variables [21, 33]. It achieves polynomial scaling on both benchmarks, with only mild sensitivity to the random seeds. Running PGPBS with modulo-2 equation solving improves the performance even further, such that we were able to handle both families of benchmarks up to $m = 48$. Considering that the problem grows quadratically in $m$, this represents a major improvement over KISSAT.

### 4.2 Other Parity Constraint Benchmarks

Chew and Heule [16] introduced a benchmark based on Boolean expressions computing the parity of a set of Boolean values $x_1, \ldots, x_n$ using two different orderings of the inputs, with a randomly chosen variable negated in the second computation. The SAT problem is to find a satisfying assignment that makes the two expressions yield the same result—an impossibility due to the negated variable. With KISSAT, we found the results were very sensitive to the choice of random permutation, and so we ran the solver for five different random seeds for each value of $n$. We were able to generate proofs for instances with $n$ up to 47, but we also encountered cases where the proofs exceeded the limit of 100 million clauses starting with $n = 40$. The overall scaling is exponential.

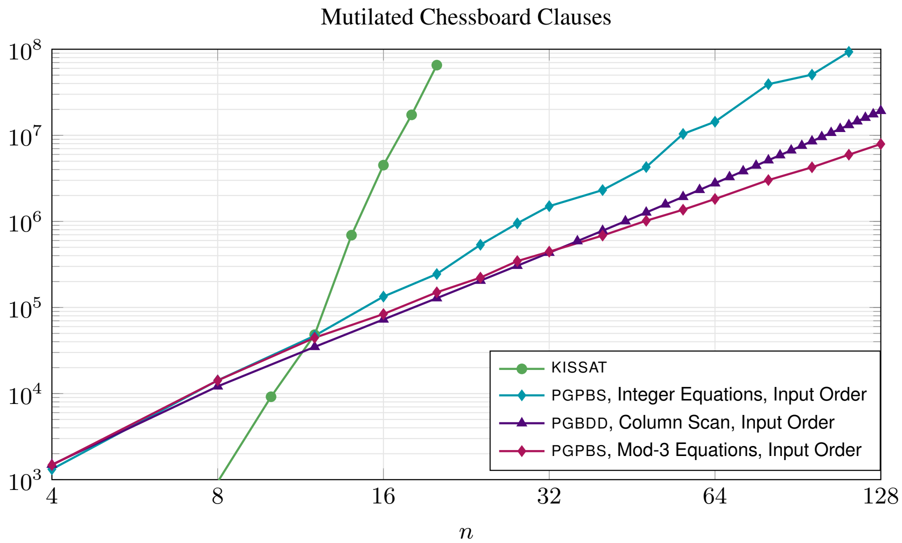


**Fig. 5.** Total number of clauses in proofs of $n \times n$ mutilated chess board problems.

Chew and Heule showed they could generate proofs for this problem that scale as $n \log n$. Using bucket elimination, PGBDD is able to obtain polynomial performance, handling up to $n = 3{,}000$ with a proof of 61 million clauses. PGPBS is able to apply Gaussian elimination with modulus $r = 2$, obtaining even better performance than did Chew and Heule. For $n = 10{,}000$, Chew and Heule's proof has 14 million clauses while the proof generated by PGPBS has less than 7 million.

For the 2016 SAT competition, Elffers and Nordström created the TSEITINGRID family of benchmarks based on grid graphs having fixed width but variable lengths [24]. These are designed to be challenging for SAT solvers while having polynomial scaling. The 2020 SAT competition included two instances of this benchmark, with $7 \times 165$ and $7 \times 185$ grids. None of the entrants could generate an unsatisfiability proof for either instance within the 5000 second time limit. On the other hand, PGPBS can readily handle both, generating proofs with less than 500,000 clauses and requiring at most 63 seconds. Indeed, PGPBS can solve the largest published instance, having a $7 \times 200$ grid, in 76 seconds. Clearly, parity constraint problems pose no major challenge for PGPBS.

### 4.3 Variants of the Mutilated Chessboard

The mutilated chessboard problem considers an $n \times n$ chessboard, with the corners on the upper left and the lower right removed. It attempts to tile the board with dominos, with each domino covering two squares. Since the two removed squares had the same color, and each domino covers one white and one black square, no tiling is possible. This problem has been well studied in the context of resolution proofs, for which it can be shown that any proof must be of exponential size [2].

The standard CNF encoding defines a Boolean variable for the boundary between each pair of adjacent squares, both horizontally and vertically. It encodes an exactly-one constraint for the variables forming the boundary of each square. Both the number of variables and the number of clauses scale as $\Theta(n^2)$. Figure 5 shows the performance of the different solvers as a function of $n$. KISSAT scales exponentially, hitting the clause limit with $n = 20$. The plot labeled "Column Scan" demonstrates that PGBDD performs very well on this problem when given a carefully crafted schedule and the proper variable ordering [11], requiring less than 20 million clauses for $n = 128$.

The plot labeled "Integer Equations, Input Ordering" shows that PGPBS can achieve polynomial scaling on this problem when performing Gaussian elimination on integer equations. It does not scale as well as column scanning, reaching $n = 96$ before hitting the clause limit. (The unevenness of the plot appears to be an artifact of the randomization used to break ties during pivot selection.)

Looking deeper, we can see that solver avoids the worst-case performance for Gaussian elimination on this problem. Let us assume that the omitted corners are both white, and so the board has $k$ black squares and $k - 2$ white squares, where $k = n^2/2$. Each variable occurs in one equation for a black square and in one for a white square. If we were to sum all of the equations for the black squares, we would get $\sum_{j=1,m} x_j = k$, where $m$ is the number of variables. Similarly, summing the equations for the white squares gives $\sum_{j=1,m} x_j = k - 2$. Subtracting the second equation for the first gives the unsolvable equation $0 = 2$. These sums and differences can be performed using pseudo-Boolean equations with coefficients $0$ and $+1$. Although Gaussian elimination combines equations in a different order, it maintains the property that the coefficients are limited to values $-1$, $0$, and $+1$.

The plot labeled "Mod-3 Equations, Input Ordering" demonstrates the benefit of modular arithmetic when solving systems of equations. The equation $0 = 2$, obtained by integer Gaussian elimination for this problem, has no solution for any odd modulus; modulus auto-detection chooses $r = 3$. This optimization achieves better scaling, due to the bounded width of the BDD representations. Indeed, it outperforms the best results obtained with PGBDD, generating a proof with less than 8 million clauses for $n = 128$. For the remaining measurements, we assume that modulus auto-detection is enabled.

The plots of Figure 6 illustrate how pseudo-Boolean reasoning makes PGPBS more robust than PGBDD. First, we consider the extension of the mutilated chessboard problem to a *torus*, with the sides of the board wrapping around both vertically and horizontally. As the plot labeled "Torus, PGBDD, Column Scan, Input Order" indicates, the performance of column scanning disintegrates for this seemingly minor change. The compact state encoding exploited by column scanning works only when there is a single frontier as the variables are processed from left to right. Second, the plot labeled "Board, PGBDD, Column Scan, Random Order" illustrates that column scanning is highly sensitive to the chosen BDD variable ordering. On the other hand, the four versions using auto-detected modular equations are only mildly sensitive to the topology (torus or board) or the variable ordering (input or random). For both topologies, the clause counts for the two different orderings (input and random) are so close to each other that they cannot be distinguished on the log-log scale. and so we show only the results for random orderings. These results show that pseudo-Boolean reasoning overcomes several major weaknesses of the pure Boolean methods of PGBDD. With its PB solver, PGPBS requires no guidance from the user regarding how to process the clauses,

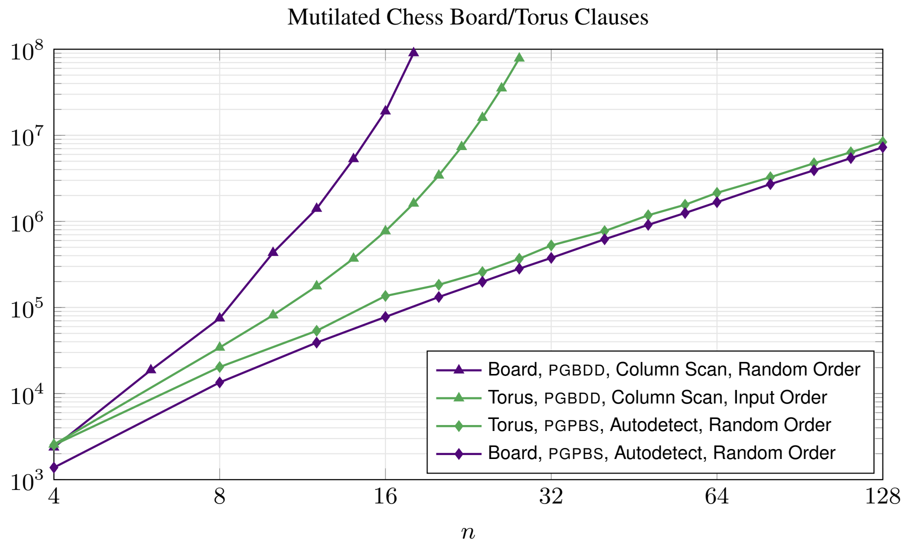


**Fig. 6.** Stress Testing: Changing the topology and variable ordering for mutilated chess. Autodetection enables the PB solver to use modulo-3 arithmetic.

nor does it require any guidance or heuristics to choose a good BDD variable ordering. Furthermore, it is less sensitive to the problem definition.

### 4.4 Pigeonhole Problem

The pigeonhole problem is one of the most studied problems in propositional reasoning. Given a set of $n$ holes and a set of $n+1$ pigeons, it asks whether there is an assignment of pigeons to holes such that (1) every pigeon is in some hole, and (2) every hole contains at most one pigeon. The answer is no, of course, but any resolution proof for this must be of exponential length [28].

The problem can be encoded into CNF with Boolean variables $p_{i,j}$, for $1 \leq i \leq n$ and $1 \leq j \leq n+1$, indicating that pigeon $j$ is placed in hole $i$. A set of $n$ AMO constraints indicates that each hole can contain at most one pigeon, and $n+1$ ALO constraints indicate that each pigeon must be placed in some hole. We experimented with two different encodings for the AMO constraints: the direct encoding requiring $n\,(n+1)/2$ clauses per hole, and the Sinz encoding [42], requiring $3n-1$ clauses.

Figure 7 shows the total number of clauses as functions of $n$ for this problem. KISSAT performs poorly, reaching the 100-million clause limit with $n = 14$ for the direct encoding and $n = 15$ for the Sinz encoding. Using PGBDD, we were unable to find any strategy that gets beyond $n = 16$ with a direct encoding. Our best results came from a “linear” strategy, simply forming the conjunction of the input clauses. For the Sinz encoding, on the other hand, we devised a column scanning technique similar to the method used to solve the mutilated chessboard problem. This approach scales very well, staying below 100 million clauses up to $n = 128$, although it can only reach $n = 17$ with a random variable ordering (plot not shown).

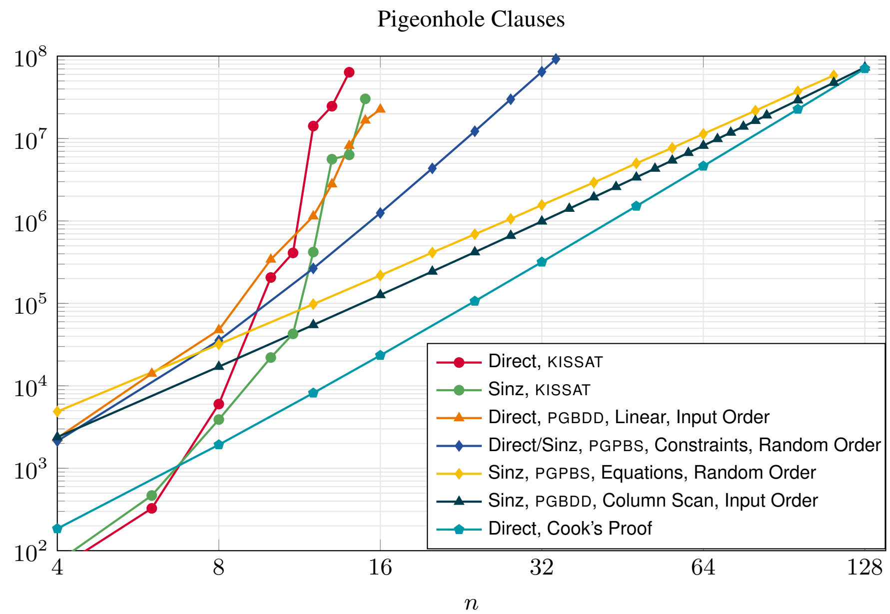


**Fig. 7.** Total number of clauses in proofs of pigeonhole problem for $n$ holes

Using pseudo-Boolean reasoning with Fourier-Motzkin elimination, we were able to achieve polynomial scaling, reaching $n = 34$ with both encodings and for both input and random ordering. The four results are so similar that they are indistinguishable on a log-log plot, and so we show the average for the two encodings with random orderings. Observe that each variable $p_{i,j}$ occurs with coefficient $-1$ in the AMO constraint for hole $i$ and with coefficient $+1$ in the ALO constraint for pigeon $j$. Thus, as described in Section 2.3, each step of Fourier-Motzkin elimination reduces the number of constraints by at least one, with the coefficients restricted to the values $-1$, $0$, and $+1$. Indeed, it can be seen that the solver, in effect, sums the $n$ AMO and $n+1$ ALO constraints to get the unsolvable constraint $0 \geq 1$.

The plot labeled "Sinz, PGPBS, Equations, Random Order" demonstrates the effect of adding constraints to enforce exactly-one constraints on both the pigeons and the holes. The solver applies modulus auto-detection to give a modulus of $r = 2$. Modulo-2 reasoning enables the solver to match the performance of column scanning, with the further advantages of being fully automated and being insensitive to the variable ordering. However, it requires additional constraints in the input file.

Finally, the plot labeled "Direct, Cook's Proof" shows the complexity of Cook's extended-resolution proof of the pigeonhole problem [18], encoded in DRAT format. Although it is very concise for small values of $n$, its scaling as $\Theta(n^4)$ lies between what we achieved by column scanning and equation solving, empirically measured as $\Theta(n^3)$, and what we achieved by constraint solving, measured as $\Theta(n^5)$. Of these, only Cook's proof and the solution by constraint solving are directly comparable, in that only these use a direct encoding and have only the minimum set of AMO and ALO constraints.

In summary, pseudo-Boolean reasoning makes this problem tractable with full automation, and it has minimal sensitivity to the variable ordering. Generating proofs by solving systems of ordering constraints is more challenging than by solving automatically detected modular equations, but both achieve polynomial scaling.

### 4.5 Other Cardinality Constraint Problems

Codel et al. [17] defined a general class of problems that includes the mutilated chessboard and the pigeonhole problems as special cases. Given a bipartite graph with vertices $L$ and $R$ such that $|L| < |R|$, the problem is to find a perfect matching, i.e., a subset of the edges such that each vertex has exactly one incident edge. For the mutilated chessboard, $L$ and $R$ correspond to the white and black squares, respectively, with edges based on chessboard adjacencies. For pigeonhole, $L$ corresponds to the holes and $R$ to the pigeons, and the graph is the complete bipartite graph $K_{n,n+1}$. No instance of this matching problem has a solution, since the sets of nodes are of unequal size.

Twelve instances of this problem were included in the 2021 SAT competition, based on randomly generated graphs with $n = |L|$ ranging from 15 to 20 and with $|R| = n+1$. Different methods were used to encode the AMO constraints, and some included clauses to convert both sets of constraints into exactly-one constraints. In the competition, all of the solvers could easily handle the benchmarks with $n = 15$, most could handle $n = 16$, with typical runtimes of around 1000 seconds, but none could solve any of the larger problems. PGPBS can easily handle all of the benchmarks, requiring at most 13 seconds and generating proofs with less than $500{,}000$ clauses.

## 5 Conclusions

Incorporating pseudo-Boolean reasoning into a SAT solver enables it to handle classes of problems encoded in CNF that are intractable for CDCL solvers. By having the PB solver generate BDD representations of its intermediate results, a BDD-based, proof-generating SAT solver can then generate clausal proofs of unsatisfiability on behalf of the PB solver in the standard, DRAT proof framework. Compared to the SAT solver operating on its own, including a PB solver enables greater automation with less sensitivity to problem definition, encoding method, and variable ordering.

We have shown that applying pseudo-Boolean reasoning to unsatisfiable instances of parity and cardinality constraint problems can yield proofs that scale polynomially. On the other hand, there are satisfiable variants of these problems, e.g., an ordinary chessboard or a pigeonhole problem with at least as many holes as pigeons. Gaussian elimination with modulus $r > 2$ and Fourier-Motzkin elimination are not guaranteed to find 0-1 valued solutions, and so our methods cannot be used to solve these instances. In our experience, however, these problems are easily solved by existing CDCL solvers, and so pseudo-Boolean reasoning is not required.

The method described here can be generalized to incorporate other reasoning methods into a proof-generating SAT solver. As long as intermediate results can be expressed as BDDs, a proof can be generated that the result of each step logically follows from the preceding steps. Thus, we could incorporate other pseudo-Boolean reasoning methods, such as cutting planes [27, 30], or we could add totally different reasoning methods.

## References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A new look at BDDs for pseudo-Boolean constraints. Journal of Artificial Intelligence Research **45**, 443–480 (2012)
2. Alekhnovich, M.: Mutilated chessboard problem is exponentially hard for resolution. Theoretical Computer Science **310**(1-3), 513–525 (Jan 2004)
3. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: AAAI Conference on Artificial Intelligence. pp. 15–20 (2010)
4. Bareiss, E.H.: Sylvester's identity and multistep integer-preserving Gaussian elimination. Mathematics of Computation **22**, 565–578 (1968)
5. Barnett, L.A., Biere, A.: Non-clausal redundancy properties. In: Conference on Automated Deduction (CADE). LNAI, vol. 12699, pp. 252–272 (2021)
6. Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In: Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions. Dep. of Computer Science Series of Publications B, vol. B-2016-1, pp. 44–45. University of Helsinki (2016)
7. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
8. Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing SAT solving. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 391–435. IOS Press, second edn. (2021)
9. Biere, A., Le Berre, D., Lonca, E., Manthey, N.: Detecting cardinality constraints in CNF. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 285–301 (2014)
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)
11. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I. LNCS, vol. 12651, pp. 76–93 (2021)
12. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. CoRR **abs/2105.00885** (2021)
13. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transition relations. In: VLSI91 (1991)
14. Chatalic, P., Simon, L.: ZRes: The old Davis-Putnam procedure meets ZBDD. In: Conference on Automated Deduction (CADE). LNCS, vol. 1831, pp. 449–454 (2000)
15. Chen, J.: A new SAT encoding of at-most-one constraint. In: Workshop on Constraint Modeling and Reformulation (2010)
16. Chew, L., Heule, M.J.H.: Sorting parity encodings by reusing variables. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 12178, pp. 1–10 (2020)
17. Codel, C., Reeves, J., Heule, M.J.H., Bryant, R.E.: Bipartite perfect matching benchmarks. In: Pragmatics of SAT (2021)
18. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News **8**(4), 28–32 (Oct 1976)
19. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: Conference on Automated Deduction (CADE). LNCS, vol. 10395, pp. 220–236 (2017)
20. Dantzig, G.B., Eaves, B.C.: Fourier-Motzkin elimination and its dual with application to integer programming. In: Combinatorial Programming: Methods and Applications. pp. 93–102. Springer (1974)

21. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1–2), 41–85 (1999)
22. Duff, I.S., Reid, J.K.: A comparison of sparsity orderings for obtaining a pivotal sequence in Gaussian elimination. IMA Journal of Applied Mathematics **14**(3), 281–291 (1974)
23. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal of Satisfiability, Boolean Modeling and Computation **2**, 1–26 (2006)
24. Ellfers, J., Nordström, J.: Documentation of some combinatorial benchmarks. In: Proceedings of the SAT Competition 2016 (2016)
25. Gabber, O., Galil, Z.: Explicit construction of linear-sized superconcentrators. Journal of Computer and System Sciences **22**, 407–420 (1981)
26. Gocht, S., Nordström, J.: Certifying parity reasoning efficiently using pseudo-Boolean proofs. In: AAAI Conference on Artificial Intelligence. pp. 3768–3777 (2021)
27. Gomory, R.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64**, 275–278 (1958)
28. Haken, A.: The intractability of resolution. Theoretical Computer Science **39**, 297–308 (1985)
29. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.D.: Verifying refutations with extended resolution. In: Conference on Automated Deduction (CADE). LNCS, vol. 7898, pp. 345–359 (2013)
30. Hooker, J.N.: Generalized resolution and cutting planes. Annals of Operations Research **12**, 217–238 (1988)
31. Hosaka, K., Takenaga, Y., Yajima, S.: Size of ordered binary decision diagrams representing threshold functions. Theoretical Computer Science **180**, 47–60 (1996)
32. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370 (2012)
33. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 54–60 (2006)
34. Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from N objects. In: Constraints in Formal Verification (CFV) (2007)
35. Li, C.M.: Equivalent literal propagation in the DLL procedure. Discrete Applied Mathematics **130**(2), 251–276 (2003)
36. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of Boolean formulas. In: Haifa Verification Conference. LNCS, vol. 7857 (2013)
37. Margulis, G.A.: Explicit construction of concentrators. Probl. Perdachi Info (Problems in Information Transmission) **9**(4), 71–80 (1973)
38. Markowitz, H.M.: The elimination form of the inverse and its application to linear programming. Management Science **3**(3), 213–284 (1957)
39. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
40. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J.ACM **12**(1), 23–41 (January 1965)
41. Rosser, J.B.: A method of computing exact inverses of matrices with integer coefficients. Journal of Research of the National Bureau of Standards **49**(5), 349–358 (1952)
42. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: Principles and Practice of Constraint Programming (CP). LNCS, vol. 3709, pp. 827–831 (2005)
43. Sinz, C., Biere, A.: Extended resolution proofs for conjoining BDDs. In: Computer Science Symposium in Russia (CSR). LNCS, vol. 3967, pp. 600–611 (2006)
44. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970. pp. 466–483. Springer (1983)

45. Urquhart, A.: The complexity of propositional proofs. The Bulletin of Symbolic Logic **1**(4), 425–467 (1995)
46. Wetzler, N.D., Heule, M.J.H., Hunt Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 422–429 (2014)
47. Williams, H.P.: Fourier-Motzkin elimination extension to integer programming problems. Journal of Combinatorial Theory (A) **21**, 118–123 (1976)
48. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: Design, Automation and Test in Europe (DATE). pp. 880–885 (2003)