




# From Clauses to Klauses\*

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

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

**Abstract.** Satisfiability (SAT) solvers have been using the same input format for decades: a formula in conjunctive normal form. Cardinality constraints appear frequently in problem descriptions: over 64% of the SAT Competition formulas contain at least one cardinality constraint, while over 17% contain many large cardinality constraints. Allowing general cardinality constraints as input would simplify encodings and enable the solver to handle constraints natively or to encode them using different (and possibly dynamically changing) clausal forms. We modify the modern SAT solver CADICAL to handle cardinality constraints natively. Unlike the stronger cardinality reasoning in pseudo-Boolean (PB) or other systems, our incremental approach with cardinality-based propagation requires only moderate changes to a SAT solver, preserves the ability to run important inprocessing techniques, and is easily combined with existing proof-producing and validation tools. Our experimental evaluation on SAT Competition formulas shows our solver configurations with cardinality support consistently outperform other SAT and PB solvers.

**Keywords:** Cardinality constraints, SAT solving, CNF Encoding

## 1 Introduction

Satisfiability (SAT) solvers have become remarkably effective automated reasoning engines in the last 25 years, with many applications in verification including bounded model checking [7] and automatic test generation [4]. Although many aspects of the solvers have changed, the top-tier solvers continue using conjunctive normal form (CNF) formulas as their input. There exist richer representations that allow for stronger reasoning techniques and make encoding problems much simpler. The most successful is Satisfiability Modulo Theories (SMT), which enables high-level reasoning (theory propagation). Higher-level reasoning is not always necessary, and for theories like strings [33] and bit vectors [39] a so-called eager SMT approach works well. This involves transforming the problem from SMT to SAT and using an off-the-shelf SAT solver.

Various groups have proposed a more modest deviation from CNF: a conjunction of cardinality constraints [20, 34, 43]. A cardinality constraint asserts that the sum of a

---

\*Supported by the U.S. National Science Foundation under grant CCF-2108521, and in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

set of literals exceeds a given bound, e.g.,  $l_1 + l_2 + \dots + l_s \geq k$ . Note that cardinality constraints generalize clauses because a clause  $l_1 \vee l_2 \vee \dots \vee l_s$  is equivalent to  $l_1 + l_2 + \dots + l_s \geq 1$ . Cardinality constraints appear frequently in problem descriptions, whether as at-most-one (AMO) constraints that may force some k-valued variable to be unique (e.g., the color of a vertex), or general at-least-k (ALK)/at-most-k (AMK) constraints that place a lower or upper bound on some resource, e.g., for optimization. In our evaluation of 5,354 SAT Competition formulas, we found that over 64% contained at least one cardinality constraint and over 17% contained at least 10 large cardinality constraints (see Section 7). There exist pseudo-Boolean (PB) solvers with stronger reasoning techniques than SAT solvers, but similar to strings and bit vectors, an eager approach transforming cardinality constraints into clauses is often desirable. SAT solvers work well across a wide range of problems and have a more developed verification toolchain. This work attempts to bridge the gap between cardinality constraints and clauses. We introduce an infrastructure for a cardinality-based input for SAT solving that makes encoding problems easier and significantly improves the performance on some problems with many cardinality constraints. These changes come without throwing away the well-developed verification toolchain and high-performance solving of modern SAT solvers.

Attempts to improve solver performance on problems with cardinality constraints have focused on either strengthening the underlying proof system, improving encodings, or natively propagating on constraints. Solvers can make use of stronger proof systems both on formulas with richer input structure and on formulas in CNF. The solver RoundingSAT [21] exploits the strength of the cutting planes proof system [16], allowing it to efficiently solve various problems that are hard for resolution. The solver SAT4J [5] implements cardinality extraction [10] and generalized resolution [26] in a preprocessing step to quickly solve formulas in CNF that contain cardinality constraints, with additional native handling for finding hamiltonian cycles [43]. Additionally, to make writing formulas simpler, solver engineers have provided support for cardinality-based representations. The solver MINISAT+ [20] supports cardinality-based input transforming the formula into clauses, and the solver package PYSAT [27] provides API calls for converting constraints into clauses. Lastly, the solver MiniCARD [34] substantially reduces the memory footprint and improves propagation by handling cardinality constraints natively. However, most of the recent work has focused on stronger proof systems or better encodings. This is partly because implementation details for cardinality-constraint propagation [34, 47] came before the development of modern preprocessing, did not account for proof generation, and were only evaluated on some crafted formulas. We revisit native cardinality constraint handling in the context of modern CDCL, showing that performance can be improved on some problems without compromising the verification toolchain (assuming no cutting planes are used).

First, consider propagation, the well-known bottleneck of SAT solvers. Improving propagation speed can boost solver performance on both satisfiable and unsatisfiable formulas. Cardinality constraint propagation can be supported with limited overhead by generalizing the watch-pointer data-structure [34]. Moreover, with modest-sized changes a cardinality-based representation can work with some other reasoning techniques in modern solvers, ranging from learned clause minimization to vivification.

These important techniques can be kept because cardinality-based propagation does not alter the relevant properties of the implication graph during conflict analysis. These changes do not require cutting planes or a new proof checker such as VERIPB [45], and are compatible with standard DRAT proofs. This is in contrast to other forms of native reasoning in CDCL solvers such as XOR parity reasoning. For example, to communicate propagated units and conflicts between XOR clauses and a CNF formula, the solver CRYPTOMINISAT makes use of BDD packages [13] to generate checkable proofs [44], adding overhead to the solving and producing large proofs.

Second, a cardinality-based representation allows for alternative ways to encode and reencode constraints. The choice of encoding can have a large impact on performance [38]. Alternatively, handling cardinality constraints separately, or dynamically encoding partial constraints [2], may also improve performance through faster propagation on frequently visited constraints and the lack of propagation on unimportant constraints. Modern CDCL solvers constantly switch between a SAT and UNSAT mode with differing heuristics [40]. We implement a hybrid solver that maintains reencoded clauses throughout solving but only propagates on cardinality constraints during SAT mode. The solver has access to auxiliary variables in the reencoded clauses along with the capability to propagate on cardinality constraints, improving performance on both satisfiable and unsatisfiable formulas.

**Contributions.** We incorporated cardinality-constraint handling into the modern CDCL SAT solver CADICAL, while still allowing several important preprocessing techniques including clause vivification and local search. We ensure clause learning from propagation on cardinality constraints produces valid proof steps.

We implemented a tool to extract cardinality constraints with a “guess and verify” approach that heuristically identifies possible encoded cardinality constraints in CNF formulas and then uses BDDs to validate and characterize them. We provide three solver configurations: one that uses cardinality-based CDCL reasoning, one that reencodes the cardinality constraints into CNF, and a hybrid approach that combines the two former configurations via. mode-switching. This hybrid configuration (HYBRID) represents a novel approach for incorporating both encoded clauses that are useful for unsatisfiable formulas and native cardinality constraints that are useful for satisfiable formulas. Proofs generated from the three solving configurations are checked with DRAT-TRIM.

We evaluated the cardinality extractor and solving configurations on the SAT Competition Anniversary Track formulas, finding cardinality constraints in over half the formulas, and many large cardinality constraints in over 17% of formulas. On these formulas, CDCL solvers outperformed PB solvers and the cardinality constraint handling further improved the CDCL solver performance. Additionally, solvers were evaluated on the Magic Squares and Max Squares problems, highlighting the importance of a good reencoding and the power of native cardinality constraint propagation.

## 2 Background

We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula  $F$  is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal  $\ell$  is

either a variable  $x$  (positive literal) or a negated variable  $\bar{x}$  (negative literal). The *phase* of a literal indicates whether it is positive or negative.

An *assignment*  $\alpha$  is a mapping from variables to truth values 1 (*true*) and 0 (*false*). Assignment  $\alpha$  *satisfies* a positive (negative) literal  $\ell$  if  $\alpha$  maps  $\text{var}(\ell)$  to true ( $\alpha$  maps  $\text{var}(\ell)$  to false, respectively), and *falsifies* it if  $\alpha$  maps  $\text{var}(\ell)$  to false ( $\alpha$  maps  $\text{var}(\ell)$  to true, respectively). An assignment satisfies a clause if the clause contains a literal satisfied by the assignment, and satisfies a formula if every clause in the formula is satisfied by the assignment. A formula is *satisfiable* if there exists a satisfying assignment, and *unsatisfiable* otherwise. Two formula are *logically equivalent* if they share the same set of satisfying assignments. Two formulas are *satisfiability equivalent* if they are either both satisfiable or both unsatisfiable.

A *unit* is a clause containing a single literal. *Unit propagation* applies the following operation to fix point: take all units  $\alpha$  in a formula  $F$  and removes from  $F$  clauses containing a literal in  $\alpha$  and removes from clauses all literals negated in  $\alpha$ . In cases where unit propagation yields the empty clause ( $\perp$ ) we say it derived a *conflict*.

## 2.1 Cardinality Constraints

A cardinality constraint on Boolean variables has the form  $\ell_1 + \ell_2 + \dots + \ell_s \geq k$  and is satisfied by a partial assignment if the sum of the assigned literals is at least  $k$ . The size of the cardinality constraint is the number of literals ( $s$ ) it contains. For this work, we do not permit duplicate literals in cardinality constraints as it would complicate the implementation of unit propagation seen in Section 5. Variables occurring in the cardinality constraint are *data variables*, and new variables added in a clausal encoding are *auxiliary variables*. The introduction of auxiliary variables is known to be beneficial to solvers for some problems, and has been studied in the context of the preprocessing technique bounded variable addition [35].

We refer to cardinality constraints as *klauses*, containing a bound  $k$  and the literals in the constraint. All clauses can be written as klauses with  $k = 1$ , corresponding to at-least-one (ALO) constraints. Throughout the rest of the paper, we refer to klauses with  $k = 1$  as clauses. When  $k = s - 1$ , the kause can be viewed as an at-most-one (AMO) constraint by negating the literals, i.e.,  $\ell_1 + \ell_2 + \dots + \ell_s \geq s - 1$  is equivalent to  $\text{AMO}(\bar{\ell}_1, \dots, \bar{\ell}_s)$ . There are multiple ways to encode an AMO constraint into CNF.

*Pairwise Encoding* for an AMO constraint is given by a set of binary clauses with negative literals and no auxiliary variables.  $\text{AMO}(\ell_1, \dots, \ell_s)$  is encoded as the conjunction of  $(\bar{\ell}_i \vee \bar{\ell}_j)$  with  $1 \leq i < j \leq s$ , resulting in  $s(s - 1)/2$  binary clauses.

*Linear Encoding* (an instance of the commander encoding [30]) for  $\text{AMO}(\ell_1, \dots, \ell_s)$  uses the pairwise encoding for  $s \leq 4$  and splits on  $s > 4$  using fresh auxiliary variables ( $y$ ) according to the following recursion:

$$\text{Linear}(\ell_1, \dots, \ell_s) : \text{Pairwise}(\ell_1, \ell_2, \ell_3, y) \wedge \text{Linear}(\bar{y}, \ell_4, \dots, \ell_s) \quad (1)$$

The encoding uses  $(s - 3)/2$  auxiliary variables and  $3s - 6$  clauses. The cutoff of  $s = 4$  (commonly used in practice) was selected as the “optimal” value to minimize the sum of the number of variables and the number of clauses.

For a general kause with  $1 < k < s - 1$ , an efficient encoding to CNF requires  $\Theta(s \cdot k)$  auxiliary variables to keep track of the count of data variables that have been

assigned. When  $s - k + 1$  literals in the clause are falsified, unit propagation should lead to a conflict. For these clauses we use the sequential counter (or *Sinz*) encoding [42].

A clausal encoding of a clause  $\ell_1 + \ell_2 + \dots + \ell_s \geq k$  is *consistent* if assigning any  $s - k + 1$  literals to false will always result in a conflict by unit propagation. It is *arc-consistent* [24] if it is consistent and unit propagation will assign all unassigned literals to true if exactly  $s - k$  literals are assigned to false. There are many encodings for cardinality constraints [3, 20, 28]. We use the Linear and Sinz encodings as a proof of concept, but these could easily be substituted with other arc-consistent encodings.

## 2.2 Conflict-Driven Clause Learning and Proofs of Unsatisfiability

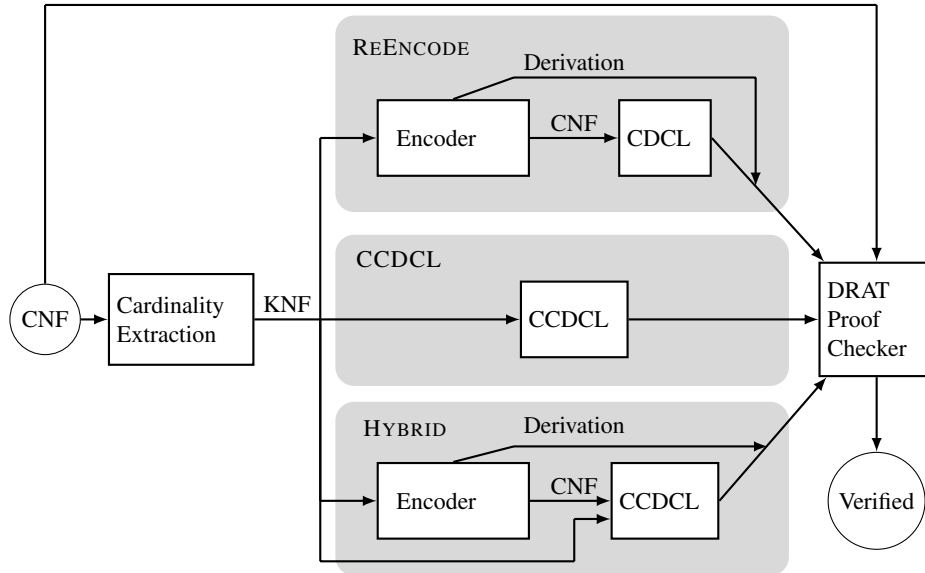
To evaluate the satisfiability of a formula, a CDCL solver [36] iteratively performs the following operations: First, the solver performs unit propagation and tests for a conflict. Two-literal watch pointers [37] enable efficient unit propagation. If there is no conflict and all variables are assigned, the formula is satisfiable. Otherwise, the solver chooses an unassigned variable through a variable decision heuristic [9, 32], assigns a truth value to it through a phase selection heuristic, and performs unit propagation. The selected variables are *decision variables*, and the assignment including decision variables and propagated variables is called the *trail*. If, however, there is a conflict, the solver performs conflict analysis potentially learning a short clause. In case this clause is the empty clause, the formula is unsatisfiable. In case it is not the empty clause, the solver revokes some of its variable assignments (“backjumping”) and then repeats the whole procedure. Additionally, modern solvers incorporate pre- and inprocessing techniques that change the formula in some way, usually reducing the number of variables and clauses or shrinking the sizes of clauses.

CDCL solvers produce satisfying assignments for satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. A clause  $C$  is *redundant* w.r.t. a formula  $F$  if  $F$  and  $F \cup \{C\}$  are *satisfiability equivalent*. The clause sequence  $F, C_1, C_2, \dots, C_m$  is a clausal proof of  $C_n$  if each clause  $C_i$  ( $1 \leq i \leq n$ ) is redundant w.r.t.  $F \cup \{C_1, C_2, \dots, C_{i-1}\}$ . The proof is a refutation of  $F$  if  $C_m$  is  $\perp$ . Clausal proof systems may also allow deletion.

The strength of a clausal proof systems is determined by the syntactic criterion it enforces when checking clause redundancy. The standard SAT solving paradigm CDCL learns clauses that are logically implied by the formula and fall under the *reverse unit propagation* (RUP) proof system. A clause is RUP if unit propagation on the falsified literals of the clause results in a conflict. The *Resolution Asymmetric Tautology* (RAT) proof system generalizes RUP. We make use of RAT proof steps in our derivations (see Section 6), but refer the reader to [29] for more details. Proofs are typically transformed to a format with hints, e.g. LRAT, before being passed to a formally-verified checker like CAKE-LPR [46].

## 3 At-Least-K Conjunctive Normal Form (KNF)

$$x_1 + x_2 + x_3 + \bar{x}_4 \geq 2 \quad \text{k } 2 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \mathbf{-x}_4 \quad 0 \quad (2)$$



**Fig. 1.** Three configurations for solving a KNF formula extracted from an input CNF formula.

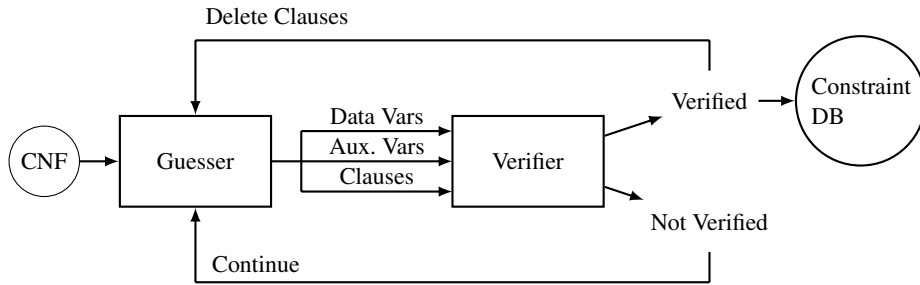
We propose enriching the input of SAT solvers and proof checkers to accept a conjunction of clauses (KNF). As an initial step, we provide backwards compatibility with CNF formulas, so KNF solvers can be used on existing benchmarks. Consistent AMO cardinality constraints can be extracted from the input CNF formula (see Section 4), then converted to clauses in the KNF format. In Equation (2) if the cardinality constraint on the left appears as clauses in a CNF formula, those clauses can be replaced by a single clause, shown on the right, in the corresponding KNF formula. Clauses with  $k > 1$  are written with a ‘ $k$ ’ followed by the bound and then the literals. All other clauses in the CNF formula can be placed directly in the KNF formula.

In Figure 1 we present three independent configurations for solving a KNF extracted from a CNF formula: REENCODE, CCDCL, and HYBRID. Each configuration produces a DRAT proof (or satisfying assignment) for the input CNF formula.

REENCODE, the encoder reencodes the clauses into clauses, and the resulting CNF formula is solved by a CDCL solver. Since the CDCL solver is using reencoded clauses that do not appear in the original CNF formula, a DRAT derivation for the clausal reencoding must be prepended to the solver’s DRAT proof. This derivation explains how the reencoded clauses can be added to the original CNF formula.

CCDCL, cardinality-CDCL (CCDCL) seen in Section 5 is used to solve the formula in KNF directly by natively propagating on clauses. The DRAT proof generated by the CCDCL solver can be verified against the input CNF formula if the extracted constraints were arc-consistent (otherwise a derivation is added).

HYBRID, a CCDCL solver takes in both the formula in KNF along with a clausal reencoding of clauses as input. The reencoded clauses are kept throughout solving as irrelevant formula clauses (never deleted by the solver). The clauses are only watched



**Fig. 2.** Guess and verify framework for extracting cardinality constraints from an input CNF formula. The guesser selects a set of data variables, auxiliary variables, and clauses representing a candidate cardinality constraint. If the verifier verifies the constraint, it is added to the constraint database and the constraint’s clauses are removed from subsequent guesses.

and propagated on during SAT mode. So, while the solver is in UNSAT mode when propagating on cardinality constraints it exclusively makes use of the reencoded clauses. This gives the solver access to auxiliary variables within the encoding and these variables can be extremely important for finding short proofs of unsatisfiable formulas. While the solver is in SAT mode it can propagate natively on clauses, allowing faster propagation that bypasses the auxiliary variables. This can be important for quickly solving satisfiable formulas. In general the solver moves back and forth between SAT and UNSAT modes with increasing limits, and will roughly spend half of its time in either mode. Clauses learned in either mode can be kept during the mode switch, but certain heuristics are modified for each mode. The proof requires a derivation as in REENCODE. Further details on verification are found in Section 6.

The three different solving configurations highlight the flexibility provided by the KNF format. In some cases, a smaller representation and fast propagation on clauses is beneficial. In other cases, reencoding clauses introducing auxiliary variables can lead to a much shorter proof. And a combination of the two approaches may work best for unknown problems. These configurations can be implemented with straightforward changes to a CCDCL solver, and the proof checker DRAT-TRIM is used as is.

## 4 Cardinality Constraint Extraction and Analysis

### 4.1 Extraction

Several researchers have devised techniques for automatically extracting cardinality constraints from CNF representations either as part of a preprocessing step [10] or dynamically within a pseudo-Boolean constraint solver [22]. We implemented our own preprocessor that detects cardinality constraints within a CNF file, converts these into clauses, and emits both these and the remaining clauses as a KNF file.

AMO constraints with pairwise constraints can be detected by finding cliques in the graph having a node for each literal and an edge between two literals if they occur in the

same binary clause [10]. Although finding maximal cliques is NP-hard, simple greedy approaches work well for this task.

We use a “guess-and-verify” approach for detecting non-pairwise constraints, shown in Figure 2. Our method of guessing looks for patterns of clauses in the CNF representation that could be cardinality constraints, including classifying the variables in these clauses as either data or auxiliary variables. To do this it examines the binary clauses in the formula and classifies each variable as being either *unate*—always having the same phase, or *binate*—occurring with both phases. Data variables are assumed to be unate, while auxiliary variables must be binate. Starting with a binate variable, the extractor forms the transitive closure of all binate variables that occur in clauses with other variables in the set. It then selects as data variables all unate variables that occur in these clauses.

We have found this approach to guessing effective at detecting standard encodings of AMO constraints, including all of those handled by previous extractors [10]. It can fail when a data variable is used with one phase for some constraint and with some other phase for another. It will also find patterns that meet the phase requirements but do not encode cardinality constraints. Fortunately, these will be rejected in the “verify” stage. Although our verifier could determine whether a set of clauses encodes a non-AMO cardinality constraint, we have been unable to devise a reliable strategy for distinguishing these clauses from the other clauses in a file. We plan to extend the extraction to general cardinality constraints in the future.

## 4.2 Analysis with BDDs

We use a BDD-based analysis to verify that our guessed cardinality constraints are in fact cardinality constraints. Given a set of clauses and a classification of the variables into a set of data variables  $X$  and a set of auxiliary variables  $Y$ , we construct the representation of the associated Boolean function  $f(X)$  as an Ordered Binary Decision Diagram (BDD) [12]. We generate the BDD for  $f(X)$  using *bucket elimination*, a systematic way to perform conjunctions and quantifications [17, 41]. That is, we create a total ordering of the data and auxiliary variables, described below, and use this ordering for the BDDs and as the *bucket ordering*. For each  $y \in Y$ , we associate a set  $B_y$ , which we refer to as the “bucket” for variable  $y$ . We also have a set  $B_d$ , which we refer to as the “data bucket”. At each point in the processing, we maintain a set of *terms*, where each term  $T$  is a BDD depending on a set of variables  $D(T) \subseteq X \cup Y$ . Term  $T$  is placed in bucket  $B_y$  when  $y = \min(D(T) \cap Y)$  and in the data bucket when  $D(T) \cap Y = \emptyset$ . The initial set of terms consists of the BDD representations of the clauses.

Bucket elimination processes the terms via conjunction and quantification operations until the only nonempty bucket is  $B_d$ . That is, let  $y$  be the maximum variable for which  $B_y$  is nonempty. While this bucket contains more than one element, we remove two, compute their conjunction, and place the result in the proper bucket. This must be in some bucket  $B_{y'}$  such that  $y' \leq y$  or in the data bucket  $B_d$ . When bucket  $B_y$  contains a single term, we form its existential quantification with respect to  $y$  and place the result in the proper bucket. This will either be in some bucket  $B_{y'}$  for which  $y' < y$  or in the data bucket. Eventually, the only terms will be in the data bucket. We form their conjunction to get the BDD representation of  $f(X)$ .



Building BDDs and performing bucket elimination requires defining a total ordering of all of the variables in  $X \cup Y$ . Our approach targets the layered structure that arises in many encodings of cardinality constraints. We start with the auxiliary variables in  $Y$  by building an undirected graph with a node for each variable  $y$  and an edge  $(y, y')$  of length 1.0 when some clause contains a literal of  $y$  and a literal of  $y'$ . In addition, we add an edge  $(y, y')$  with length 0.75 when there is some data variable  $x$  such that there is some clause containing literals of  $x$  and  $y$  and another clause containing literals of  $x$  and  $y'$ .

We identify a “source” node  $s$  and a “sink” node  $t$  and (conceptually) view the edges as elastic, enabling us to stretch the graph between these two endpoints into a single line and order the nodes according to where they lie on this line. The two edge types will tend to group the auxiliary variables first by their occurrence in clauses with matching data variables and second by their occurrence with each other. Our implementation of this idea starts by looking for endpoints  $s$  and  $t$  for which the shortest path between the two nodes is maximal. Starting with some random node, we jump to the most distant node (in terms of shortest path), and from there to the most distant node, iterating as long as the distance increases. We perform these iterations from multiple starting points and take the most distant pair as the graph endpoints. Then, we order the variables first in terms of their proximity to  $s$  and secondarily in terms of their distance from  $t$ . Finally, each data variable  $x \in X$  is inserted into the ordering to be near the first variable  $y$  for which some clause contains a literal of  $x$  and a literal of  $y$ . For a layered graph, this approach will tend to find opposite corners as endpoints  $s$  and  $t$  and generate a layered ordering of the variables. For a graph having a tree structure, it will produce an ordering that approximates what would be obtained via an inorder traversal of the tree. Both of these make good BDD orderings.

Once the BDD for the function  $f(X)$  has been constructed, detecting whether it encodes a cardinality constraint and the parameters of that constraint can readily be inferred from the structure of the BDD. Let us number the data variables as  $x_1, x_2, \dots, x_N$ . For a set of literals  $\{\ell_1, \ell_2, \dots, \ell_N\}$ , where each  $\ell_i \in \{x_i, \bar{x}_i\}$ , function  $f$  can encode both a lower bound  $L$  and an upper bound  $H$ , giving a *two-sided* constraint:

$$L \leq \ell_1 + \ell_2 + \dots + \ell_N \leq H \quad (3)$$

Lower bound  $L$  can degenerate to  $L = 0$ , while upper bound  $H$  can degenerate to  $H = N$ . As examples, an at-most-one constraint has  $L = 0$  and  $H = 1$ , while a clause has  $L = 1$  and  $H = N$ .

Our task is to determine the two bounds and the phase of each literal, or to reject  $f$  as not encoding a cardinality constraint. The BDD encoding of the constraint (3) has a simple, layered structure [1, 13]. In detail, let us say that the pair of integers  $(i, j)$  is *feasible* if there is some satisfying assignment for the constraint where the first  $i - 1$  variables have  $j$  literals assigned to true. More precisely, the following conditions must be satisfied for  $(i, j)$  to be feasible:

- $i$  satisfies  $1 \leq i \leq N + 1$
- $j$  satisfies  $0 \leq j \leq i - 1$
- There must be some value  $k$  such that  $0 \leq k \leq N - i + 1$  and  $L \leq j + k \leq H$ .

**Table 1.** Detecting size 10 AMO constraints on the 9 PySAT exactly-one clausal encodings: pairwise, sequential counter, cardinality network, sorting network, totalizer, k-totalizer, mod k-totalizer, bitwise, and ladder. The table shows the number of data variables/auxiliary variables in the largest AMO constraint detected by the extraction tool on a given encoding. A 10/0 represents the full constraint on all of the data variables. No approach detected an AMO constraint in the bitwise encoding.

Tool	Pair	SCnt	CNet	SNet	Tot	kTot	mkTot	Bit	Lad
Guess-and-Verify	10/0	10/0	10/0	10/0	10/0	10/0	10/0	–	9/0
LINGELING (Syntactic)	10/0	1/2	2/1	2/1	2/1	2/1	2/1	–	1/2
RISS (Semantic)	10/0	3/2	4/2	4/1	3/2	3/2	3/2	–	3/2

The BDD will have a node  $u_{i,j}$  for each feasible pair  $(i, j)$ . This node can either be  $L_1$ , the leaf node representing Boolean constant 1, or it can be a nonterminal node labeled by variable  $x_i$ . When  $u_{i,j}$  is nonterminal and  $\ell_i = x_i$ , then its positive (respectively, negative) child will be node  $u_{i+1,j+1}$  (resp.,  $u_{i+1,j}$ ) if pair  $(i+1, j+1)$  (resp.,  $(i+1, j)$ ) is feasible and leaf node  $L_0$  (representing Boolean constant 0) otherwise. If  $\ell_i = \bar{x}_i$ , then the two children will be reversed. Starting with the root node, the literal assignments and values of  $L$  and  $H$  can be determined by examining the BDD level-by-level, while also determining whether or not the structure matches that of an cardinality BDD.

### 4.3 PySAT Encodings Experimental Evaluation

In this section we compare our Guess-and-Verify G&V tool against the two extraction techniques presented in [10]. LINGELING implements the static detection of pairwise constraints and RISS implements the static pairwise and two product encoding detection along with the merging operation and semantic detection. The merging operation involves taking two AMO constraints of the form  $-x + \ell_1 + \ell_2 + \dots + \ell_s \geq s$  and  $x + j_1 + j_2 + \dots + j_n \geq n$  and resolving on the opposing literal  $x$  to produce  $\ell_1 + \ell_2 + \dots + \ell_s + j_1 + j_2 + \dots + j_n \geq n + s - 1$  (where duplicate literals are removed and the bound is updated appropriately). The semantic detection involves using unit propagation to detect AMK cardinality constraints with an arc-consistent encoding. In short, starting from a clause, unit propagation is used to determine if literals can extend the candidate cardinality constraint. This approach may be disrupted by auxiliary variables within the encoding such that the unit propagation produces only truncated versions of the cardinality constraints.

For this evaluation, we modified LINGELING and RISS to run cardinality detection, print the detected constraints, then exit. Neither solver provides command line options for this operation, or the ability to produce a formula in any format similar to KNF with clauses and auxiliary variables from the extracted cardinality constraints removed. PySAT [27] is a Python API for encoding cardinality constraints into clausal form. It supports 9 different encodings, and these contain the most common AMO encodings. We performed unit propagation and pure literal elimination on the generated PySAT encodings, and added an ALO constraint on the data variables, making the constraint exactly-one. We add this clause because data variables will appear in both polarities

in a typical formula (otherwise they would be propagated by pure literal elimination and removed from the AMO constraint). Table 1 shows a comparison between G&V, LINGELING, and RISS [10] on the PySAT encodings for AMO constraints of size 10. Our G&V tool found the original AMO constraints for 7 of the PySAT encodings, and found the core of the original AMO constraint for the Ladder encoding (missing a single data variable). The other tools found small nested AMO constraints of sizes 3-6, but they could not find the core of the AMO constraint for any encoding other than pairwise. Finding many small AMO constraints is less useful, since the propagation power is weaker and a reencoding would only consider the sub constraints individually; whereas finding a larger AMO constraint that combines several sub constraints is far more effective for applying native cardinality constraint handling or reencoding.

The semantic detection from RISS detects some AMO2 constraints among the encodings, but they do cover a majority of the original problem variables. The merge operation from RISS can generate constraints of size 4-6, but this operation is less helpful in our setting since it does not allow the deletion of the smaller constraints used in the merge. We plan to explore additional heuristics to encapsulate all commonly used encodings in subsequent iterations of the extraction tool.

In future work we plan to extend the G&V framework to general cardinality constraint extraction. The general case is much more difficult than the AMO case, and is relatively unexplored in the literature. Our tool would require more sophisticated heuristics for guessing since general cardinality constraint encodings may contain auxiliary variables in varying polarities as well as varying clause structures. The verifier would also require modifications. It is well-known that BDDs have size limitations, and this could be a factor for large general cardinality constraints. In addition, it would be important to have a verifier that could dynamically adapt the constraint it is characterizing so that the guessing algorithm could provide an under approximation of a given constraint (e.g., providing a set of clauses to the verifier that contains all of the clauses used in a cardinality constraint as well as other clauses not used in the cardinality constraint).

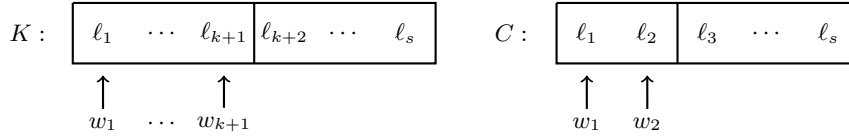
## 5 Cardinality Conflict-Driven Clause Learning

In this section we describe cardinality-CDCL (CCDCL), an extension of CDCL with propagation on clauses. For problems with many large clauses, handling them natively will significantly reduce the size of the formula and increase the speed at which cardinality constraints propagate.

$$K_1 : x_1 + x_2 + x_3 + \bar{x}_4 + x_5 \geq 3 \qquad C_1 : x_1 + x_2 + x_3 + \bar{x}_4 + x_5 \geq 1$$

*Example 1.* The partial assignment  $\bar{x}_1 \bar{x}_2$  forces the extension  $x_3 \bar{x}_4 x_5$  for  $K_1$  to be satisfied. The partial assignment  $\bar{x}_1 \bar{x}_2 \bar{x}_3 x_4$  forces the extension  $x_5$  for  $C_1$  to be satisfied.  $K_1$  can propagate at most 3 literals, whereas  $C_1$  can propagate at most 1 literal.

Example 1 shows the added propagation power of clauses over clauses, not to mention the many auxiliary variables that must be propagated in a clausal encoding. Clauses



**Fig. 3.** Klausel (left) of the form  $\ell_1 + \dots + \ell_s \geq k$  and Klausel (right) of the form  $\ell_1 + \dots + \ell_s \geq 1$ , with watch pointers for the first  $k + 1$  literals in the Klausel.

can be handled natively with minimal changes to a CDCL solver, and no changes to the proof logging.

CCDCL incurs a few tradeoffs. Some preprocessing techniques need to be restricted or disabled, and the propagation/analysis algorithms become more complicated. More importantly, the auxiliary variables in clausal encodings may be important for learning useful clauses. These limitations are further discussed in the experimental evaluation.

## 5.1 Implementation Details

A Klausel requires more watch pointers ( $k + 1$ ) than a clause (Figure 3), since  $s - k$  literals must be falsified in order to propagate the Klausel [34]. The invariant on a non-conflicting Klausel is that at least  $k$  watched literals are either unassigned or satisfied. If this is not the case, then at least  $s - (k + 1)$  literals are falsified and therefore the Klausel is falsified.

Propagation on clauses is unchanged. For a Klausel, assuming the watch pointer in question is  $w_i$  for assigned literal  $\ell_i$ , the first unassigned or satisfied literal starting from  $\ell_{k+2}$  is swapped with  $\ell_i$ , then  $w_i$  is released and a new watch is created for the swapped literal. If no such literal exists, then the watched literals  $\ell_1, \dots, \ell_{k+1}$  (not including  $\ell_i$ ) are assigned to true, and their *reason* is all of the literals  $\ell_i, \ell_{k+2}, \dots, \ell_s$  that are falsified. If any of the would-be propagated literals is already falsified, then there is a conflict and the propagation algorithm breaks. The conflict clause contains the reason literals,  $\ell_i$ , and first falsified watched literal other than  $\ell_i$ .

Conflict analysis works the same as in CDCL, where the implication graph is traversed backwards from the conflict clause to the first unique implication point in order to produce a learned clause. An *implication graph* is a data structure capturing the ordering and dependencies of decided or propagated literals, where each node is a literal assigned to true and incoming edges to a node are the reason literals for why the node was propagated. Intuitively, the clauses learned by CDCL are RUP because they represent a cut in the implication graph, from which unit propagation will derive a conflict. It is similar for CCDCL. Consider a literal propagated by a Klausel. In the implication graph, the reason for the literal is exactly the literals that propagated the Klausel. Since important properties of the implication graph are unchanged, clause minimization can be applied to learned clauses. We have not considered the affect of clauses on chronological backtracking, and therefore only allow backjumping.

## 5.2 Inprocessing Techniques

In order to support a selection of the most important inprocessing techniques, we split the clause database into clauses ( $k = 1$ ) and clauses ( $k > 1$ ). When one clause is a subset of another clause it can subsume (or replace) the other clause. This operation can be performed on all clauses, without considering clauses. We allow bounded variable elimination (BVE) [18] on all variables not occurring in clauses. Variables in clauses are frozen [19] so they are not selected as candidates for BVE. Variable elimination relies on resolving the clauses containing a variable with themselves. This would not work with clauses since we provide no corresponding inference rule for clause resolution. We allow vivification [31] on all clauses. During the vivification procedure, the literals in a clause are falsified and propagated. If a conflict is derived, conflict analysis is used to strengthen the clause. We enable propagation on clauses during vivification.

We support Stochastic Local Search (SLS) for phase saving [8]. The SLS algorithm within CADICAL is simple and only relies on break values, i.e., the number of clauses falsified after flipping a literal. In our implementation, a falsified cardinality constraint adds additional weight to the break value of a literal depending on how many falsified literals are contained within the cardinality constraint.

There are additional inprocessing techniques that we plan to include in future work, but are less important than the implemented techniques. These include failed literal probing [23] and Equivalent Literal Substitution (ELS). While we do not allow duplicate literals in a clause, ELS could be performed by adding clauses for literal equivalence, then substituting literals in all clauses but not in clauses.

## 6 Proof Checking

A formula is transformed from CNF to KNF by iteratively detecting cardinality constraints and replacing the clauses encoding the constraint with a corresponding clause, leaving the remaining clauses unchanged. We only detect consistent clausal encodings to ensure correct proof generation. We did not encounter any non-consistent AMO constraints during extraction; however, if this were the case or if it occurred for general cardinality constraints, we could use a BDD to generate a derivation of a consistent constraint from the extracted constraint [14]. There are two possible results produced by the solver: a satisfying assignment or a clausal proof of unsatisfiability. And for each there are three cases to consider, propagating natively on the KNF, reencoding the clauses into clauses, or a hybrid approach.

### 6.1 Satisfying Assignments

It is possible that some variables from the original formula are removed when generating the KNF formula since certain extracted constraints use auxiliary variables that will not appear in the corresponding clause. This will not affect proof generation for unsatisfiable problems but will affect satisfying assignments. If a solver produces a satisfying assignment for the KNF formula, the auxiliary variables from the original CNF formula will be unassigned. Every partial assignment that satisfies a cardinality

constraint must be extendable to an assignment that satisfies the clausal encoding of the constraint. So, calling a secondary SAT solver on the original CNF formula under the partial assignment given by the solver will produce a satisfying assignment that includes the auxiliary variables. For configurations where clauses are reencoded, the new clauses may contain auxiliary variables not appearing in the original CNF formula. These can simply be removed from the satisfying assignment produced by the solver, then the same procedure for calling a secondary SAT solver is followed.

## 6.2 Clausal Proofs

For configurations that make use of reencoded constraints, we must generate a derivation of these constraints proving they are redundant and can be added to the original CNF formula. To derive the pairwise encoding, we add the clauses from the encoding to the formula. Each binary clause is RUP since assigning two literals in the constraint to true must propagate a conflict. The derivation of the linear encoding is similar to its clausal encoding, with an additional clause for each auxiliary variable:

$$\text{Deriv}(\ell_1, \dots, \ell_s) = \text{Pairwise}(\ell_1, \ell_2, \ell_3, y), (\ell_1 \vee \ell_2 \vee \ell_3 \vee y), \text{Deriv}(\bar{y}, \ell_4, \dots, \ell_s) \quad (4)$$

The Linear derivation makes use of so-called RAT proof steps since new auxiliary variables are being added to the formula. Then, the proof produced by the solver with reencoded clauses is appended to the derivation, and this serves as a complete proof for the original CNF formula.

For configurations that propagate natively on cardinality constraints, the clauses learned by the solvers are RUP with respect to an arc-consistent clausal encoding of the cardinality constraints. To see this, consider when a propagation on an AMO cardinality constraint occurs – exactly when one literal is set to true – and this will propagate the remaining literals to false for both the native propagation and an arc-consistent clausal encoding. So, for formulas with arc-consistent encodings (the vast majority), the proof produced by our natively propagating configurations can be checked against the original CNF formula as is. In the special case where the clausal encoding is consistent but not arc-consistent, a derivation is prepended to the proof.

## 6.3 Starting with KNF Input

Finally, we consider if the original formula is in KNF. A satisfying assignment can be verified by checking if each clause in the KNF is satisfied. If the formula is unsatisfiable, we can use any of the three solver configurations above to produce a DRAT proof. We then transform the KNF formula to a CNF formula using an arc-consistent encoding. The proof can be checked against this CNF formula. To increase trust, one can use a formally verified KNF to CNF encoder [15]. Alternatively, one could verify the transformation from KNF to CNF with a PB checker [25]. As a long-term solution, existing DRAT proof checkers can be modified to accept KNF formulas as input, with moderate changes to parsing and unit propagation. This approach would avoid the overhead required to check a KNF to CNF translation since propagation on clauses would be handled natively by the checker.

## 7 Experimental Evaluation

All experiments were performed in the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [11]. We ran 64 experiments in parallel per node with 5,000 second timeouts. Therefore, each process held approximately 4GB of memory. This was not a limiting factor except for the only Java based solver SAT4J which failed from memory outs more than timeouts. We report the PAR-2 score for each solver. PAR-2 is the sum of completed runtimes added to the number of timeouts and memory outs multiplied by two times the timeout (10,000), averaged over the number of formulas solved by some configuration.

We implemented the CCDCL algorithm on top of the award winning CDCL solver CADICAL [6]. The base CADICAL is run with all default inprocessing enabled, in contrast with the CCDCL-based approaches that disable some reasoning techniques. Converting the KNF formulas to the pseudo-Boolean input format OPB format is purely syntactical. As such, we are able to run both ROUNDINGSAT [21] and SAT4J [5] (with cutting planes enabled) after extracting cardinality constraints. These two solvers provide a baseline for comparing stronger reasoning techniques against our resolution-based CDCL solvers. We use the following configurations, given an input CNF formula, an extracted KNF formula, and an OPB formula from the KNF formula:

- CADICAL : run CADICAL on the input CNF formula.
- REENCODE : run CADICAL on reencoded formula (Linear encoding for AMOs).
- CCDCL : run CCDCL on the extracted KNF formula.
- HYBRID : run CCDCL on the extracted KNF formula plus linearly encoded AMO constraints. Klausen are present during SAT mode.
- ROUNDINGSAT : run ROUNDINGSAT on the extracted KNF formula (converted to OPB).
- SAT4J : run SAT4J with combined cutting planes and resolution on the extracted KNF formula (converted to OPB).

The runtimes presented in the experimental evaluation include only each solver’s runtime when given the proper input formula. We do not include the extraction time or translation time because we intend to compare solvers as if a user had generated the input formula in different formats. From our experience, when a user has a CNF formula generator, it is simple to modify the generator to output both KNF or OPB formulas. The repository containing our solver, experiment configurations, and experiment data can be found at [temp\\_url](#).

### 7.1 SAT Competition Benchmarks

We evaluated solvers and the cardinality extraction tool on the SAT Competition Anniversary Track formulas. First we performed unit propagation, removing one from the set that was solved immediately, leaving 5,354 formulas. We applied cardinality extraction on each formula with a timeout of 1,000 seconds per AMO constraint type (pairwise or non-pairwise), producing a corresponding KNF formula. For the PB solvers, we translated the KNF formula into an OPB formula (a line-by-line syntactic translation).

**Table 2.** Statistics running the cardinality extractor with a 1,000 second timeout for pairwise and then non-pairwise constraints on the 5,354 competition formulas. Found is the number of formulas containing extracted cardinality constraints, Pairwise is the count with exclusively pairwise encodings, Non-Pairwise is the count with exclusively non-pairwise encodings, and Both is the count with a mixture of encodings. *geq5* is the percent of formulas with at least one constraint of at least size 5, and  $\geq 10 \times 10$  is at least 10 constraints of at least size 10. We show the average runtime and the percentage of these formulas that took  $\leq 15$  seconds.

Found	Pairwise	Non-Pairwise	Both	$\geq 5$	$\geq 10 \times 10$	Average. (s)	$\leq 15$ s
3,415	3,090	55	270	36%	17%	69.0	78.0 %

Table 2 shows that of the 3,415 formulas with cardinality constraints the vast majority contained pairwise encoded constraints and we extracted non-pairwise encoded constraints from only a few hundred formulas. While an expert may know that for many problems the pairwise encoding can be improved for unsatisfiable problems with more compact encodings, these results show that many formulas still implement the pairwise encoding. Furthermore, we log the sizes of constraints extracted, and found that 1,946 formulas contained a cardinality constraint of at least size 5, and 933 formulas contained at least 10 cardinality constraints of size 10 or more. A large fraction of formulas appearing in the SAT Competitions contained many large cardinality constraints, indicating our approach could impact many potential users. By breaking down the formula set in this way, we are able to gauge the performance of the various solving configurations on general formulas versus formulas with many large cardinality constraints. Cardinality constraint extraction was fast (less than 15 seconds) for the majority of formulas, but again, we expect it would be easy for benchmark authors to rewrite the problems in KNF.

Table 3 shows the performance of solvers on two increasingly more restrictive formula sets. It is expected that a reencoding approach or native cardinality propagation would work better on problems with many large cardinality constraints, where the difference between encodings or propagation is more pronounced. This motivated the curation of the formula sets.

At a high-level, the table shows the separation of our three cardinality-based configurations from the default CDCL and PB solvers. For each of the PAR-2 scores, a cardinality-based configuration has the best result. While there are some crafted instances in the formula set that SAT4J and ROUNDINGSAT can solve instantly, neither solver performs well over all formulas. As expected, for a general set of formulas with only AMO constraints extracted the CDCL solvers perform better. The PB solvers are more suited for special cases where formulas contain many general cardinality constraints.

REENCODE performs best for unsatisfiable formulas but does not perform as well as CADICAL on satisfiable formulas. Since many of the formulas originally use the pairwise encoding, this result suggests that for some satisfiable formulas the pairwise encoding outperforms the more compact Linear encoding. CCDCL and HYBRID solve the most satisfiable formulas. The cardinality-based propagation is more effective when the formula contains larger cardinality constraints, seen in the larger difference in PAR-



**Table 3.** From top to bottom, the first set of results are for the 1,946 (847 SAT, 716 UNSAT) formulas with at least one cardinality constraint of at least size 5. The second set of results are for the 933 (405 SAT, 345 UNSAT) formulas with at least 10 cardinality constraints of at least size 10. PAR-2 score is the sum of all completed solving times plus twice the timeout for each unsolved benchmark that was solved by another configuration, averaged over the number of solved formulas. Combined is both SAT and UNSAT formulas. Solving times do not include constraint extraction.

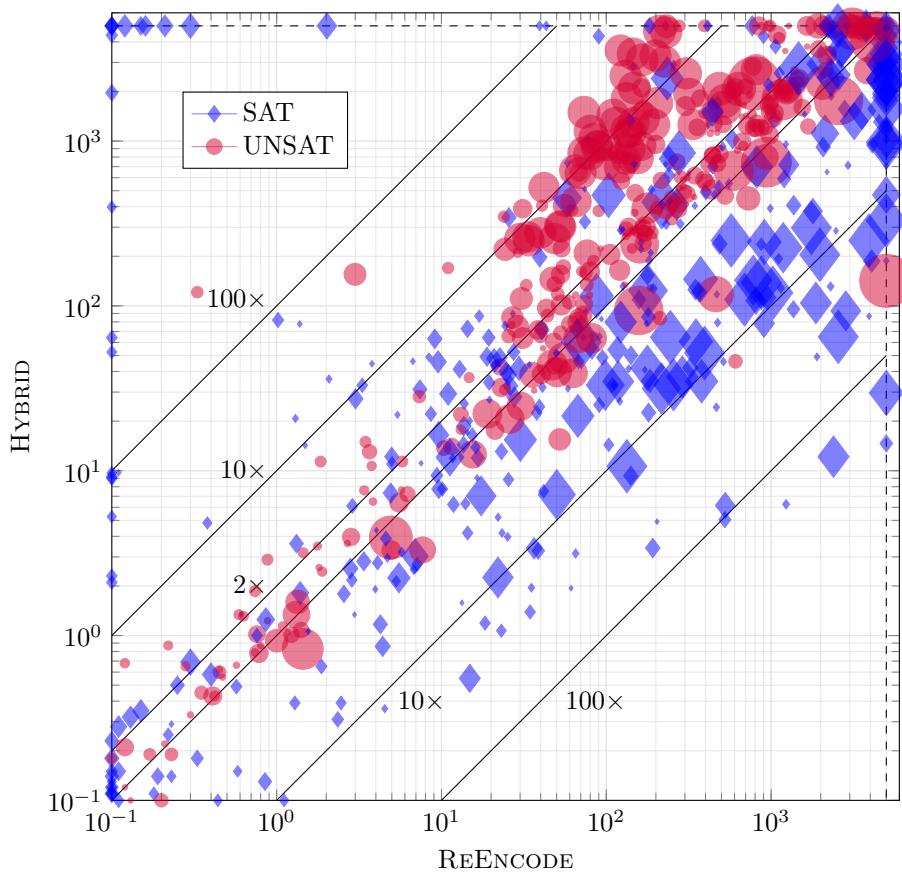
Configuration	SAT / UNSAT	SAT PAR-2	UNSAT PAR-2	Comb. PAR-2
At least one size 5				
CADICAL	790 / 619	931.68	1850.31	1352.5
CCDCL	789 / 593	953.19	2292.22	1566.59
HYBRID	<b>795</b> / 585	<b>897.54</b>	2427.31	1598.31
REENCODE	787 / <b>636</b>	965.55	<b>1560.70</b>	<b>1238.18</b>
ROUNDINGSAT	647 / 475	2592.14	3823.39	3156.17
SAT4J	373 / 240	5676.72	6720.69	6154.95
At least 10 size 10				
CADICAL	373 / 269	1062.52	2824.08	1872.84
CCDCL	<b>380</b> / 254	<b>928.64</b>	3315.55	2026.62
HYBRID	377 / 262	1017.63	3104.27	1977.49
REENCODE	372 / <b>282</b>	1108.04	<b>2297.03</b>	<b>1654.98</b>
ROUNDINGSAT	294 / 185	2975.97	4924.31	3872.21
SAT4J	166 / 103	5953.62	7065.07	6464.89

2 score between CADICAL and CCDCL on the bottom set of the table. HYBRID solves many more unsatisfiable formulas than CCDCL on the second formula set, showing the possible benefit of a configuration that targets both satisfiable and unsatisfiable formulas with many cardinality constraints.

Figure 4 presents the tradeoff between cardinality-based propagation and encodings. HYBRID implements mode-switching that trades approximately half the time between the cardinality-based propagation, leading to a slow down on average for solving unsatisfiable formulas. This is made clear by the  $2\times$ 's line in the scatter plot containing many of the unsatisfiable formulas. On the other hand, HYBRID is able to solve the satisfiable instances with many cardinality constraints much faster due to the native cardinality propagation. Our heuristic-based extraction only works for AMO constraints, so many general cardinality constraints may have been missed. If the problems were first encoded in KNF containing general cardinality constraints, we expect the results to improve significantly. We explore this possibility in the following section with two problems encoded directly in KNF.

## 7.2 Magic Squares and Max Squares

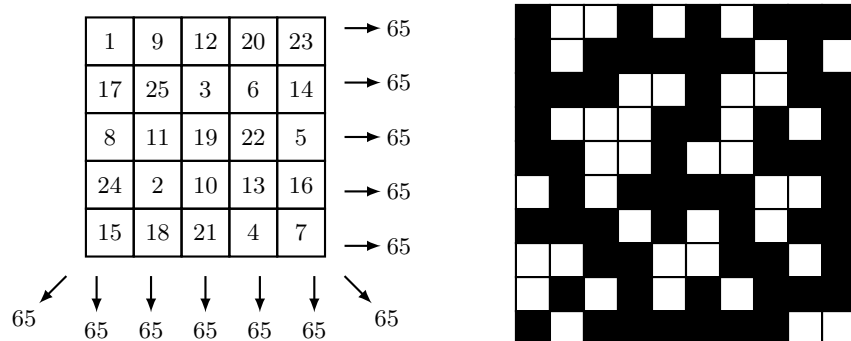
In this section we explore the Magic Squares and Max Squares problems. These problems demonstrate the effectiveness of cardinality-based propagation on satisfiable formulas with general cardinality constraints, as well as the importance of good encodings



**Fig. 4.** Comparison between solver configurations on the 933 formulas with at least 10 extracted constraints of size 10 or more. The size of a mark is proportional to the number of extracted constraints size 10 or more, i.e., formulas with many large AMO constraints have large marks.

for unsatisfiable formulas. Both problems were generated in KNF, so no cardinality extraction is necessary.

The **Magic Squares** problem asks whether the integers from 1 to  $n^2$  can be placed on an  $n \times n$  grid such that the sum of integers in each row, column, and diagonal all have the same value (a.k.a. the magic number  $M$ ) see Figure 5. Problem variables denote the integer value of a cell. We add a unary encoding of values such that the pop count of these encoded values in a row, column, or diagonal is the corresponding sum. We use the following constraints: (a) ALO constraints stating each cell is assigned to a value, (b) AMO constraints stating no two nodes can have the same value, (c) klausal constraints stating the sum of each row, column, and diagonal is at least  $M$ , (d) klausal constraints stating the difference between the total  $n \times n$  and the sum of each row, column, and diagonal is at least the total  $n \times n - M$ .



**Fig. 5.** Left a magic square ( $n = 5$ ) and right an optimal solution of a max square ( $n = 10$ ,  $m = 61$ ).

**Table 4.** Solving times for Magic Squares (top) and Max Squares (bottom), timeout of 5,000 s.

Magic Squares								
Configuration	$n$							
	5	6	7	8	9	10	11	12
CCDCL	<b>0.18</b>	<b>1.42</b>	<b>6.56</b>	<b>12.01</b>	46.37	<b>460.82</b>	<b>164.61</b>	<b>766.07</b>
HYBRID	2.54	37.04	1070.97	887.71	–	–	–	–
REENCODE	58.75	246.55	1099.65	4487.79	–	–	–	–
ROUNDINGSAT	3.88	8.24	4.41	264.83	631.46	4212.7	1150.16	–
SAT4J	0.78	8.3	5.31	23.45	<b>17.99</b>	958.56	247.94	3177.64

Max Squares								
Configuration	SAT ( $n,m$ )				UNSAT ( $n,m$ )			
	(7,32)	(8,41)	(9,51)	(10,61)	(7,33)	(8,42)	(9,52)	(10,62)
CCDCL	0.12	15.01	539.88	660.25	217.62	–	–	–
HYBRID	0.02	0.92	<b>17.0</b>	101.42	1.07	1.27	58.53	–
REENCODE	<b>0.01</b>	<b>0.62</b>	57.83	<b>24.33</b>	<b>0.18</b>	<b>0.72</b>	<b>22.82</b>	–
ROUNDINGSAT	0.06	1582.62	–	–	2.31	1046.83	–	–
SAT4J	5.75	–	–	–	26.24	–	–	–

When encoding the problem with the correct magic number, it is satisfiable for any  $n \times n$  grid. Table 4 shows the solving times on the Magic Squares formulas of increasing size. CCDCL configuration significantly outperforms the solvers, finding satisfying assignments for large values of  $n$ . Only the PB solvers SAT4J and ROUNDINGSAT get close to the performance of CCDCL. This shows that for some crafted instances with many cardinality constraints, improved propagation alone can perform better than a stronger reasoning system like cutting planes. Still, the addition of encoded constraints in REENCODE and HYBRID can significantly worsen the performance. The mode-switching of HYBRID gives it a slight edge over REENCODE.

The **Max Squares** problem [48] asks whether you can set  $m$  cells to true in an  $n \times n$  grid such that no set of four true cells form the corners of a square. There exists an

optimal value  $opt$  for each grid such that the Max Squares problem on  $opt$  is satisfiable and on  $opt+1$  is unsatisfiable. Problem variables denote whether a cell is in the solution. We use the following constraints: (a) clauses with 4 literals blocking the 4 corners of each possible square in the grid, (b) a klausal constraint stating at least  $m$  cells are set to true.

The results in Figure 4 show the solving times on several configurations with satisfiable formulas ( $m = opt$ ) and unsatisfiable formulas ( $m = opt + 1$ ). For these formulas, both cardinality-based propagation and PB reasoning are ineffective. The two configurations with encoded constraints, REENCODE and HYBRID are able to solve much larger unsatisfiable formulas. This problem is unique in that it contains one large cardinality constraint unlike Magic Squares with many cardinality constraints. This may explain the worse performance of CCDCL on even the satisfiable formulas.

The problems above highlight the main difficulty with handling clauses and reencodings: sometimes encoded constraints make the problem much easier, yet sometimes keeping the cardinality constraints abstract make the problems easier. We attempt to address this dilemma with the combined configuration HYBRID that has access to auxiliary variables throughout solving, and clauses during SAT modes. For Magic Squares, HYBRID outperforms REENCODE, and for Max Squares HYBRID outperforms CCDCL. For future work, we plan to improve the combined approach HYBRID by modifying solver heuristics. For example, variable scores can be modified to prefer deciding on auxiliary variables at different stages of the search. With these and other changes, we believe HYBRID can get closer to the performance of a virtual portfolio of REENCODE and CCDCL.

## 8 Conclusion and Future Work

We argue the input format for SAT solvers and proof checkers should be enriched with clauses. In this work, we present several solver configurations that take as input KNF formulas extracted from CNF formulas. In an experimental evaluation we show that with modifications to the state-of-the-art solver CADICAL, our three cardinality-based configurations outperform default CDCL and PB solvers on SAT Competition and Magic/Max Squares formulas. The CCDCL configuration performs well on satisfiable formulas, the REENCODE configuration on unsatisfiable formulas, and HYBRID on a mixture of both. We plan to extend this further by incorporating partial encodings dynamically during runtime. By partially encoding the cardinality constraints as the solver runs, we can guide the solver to focus on cardinality constraints that appear more important, and provide auxiliary variables for those cardinality constraints in case the problem appears to be unsatisfiable.

This initial step opens many avenues for future work. We plan to incorporate more complex propagation-based cardinality constraint detection in the extractor in order to go beyond AMO constraints. We plan to modify a DRAT proof-checker to take KNF formulas as input and propagate on clauses, comparing the verification tool chain against corresponding pseudo-Boolean toolchains. And finally, we plan to explore the possibility of using KNF to enhance other paradigms including local search and parallel solving.

## 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. Abío, I., Stuckey, P.J.: Conflict directed lazy decomposition. In: *International Conference on Principles and Practice of Constraint Programming* (2012)
3. Bailleux, O., Boufkhad, Y.: Efficient cnf encoding of boolean cardinality constraints. In: *Principles and Practice of Constraint Programming (CP)*. pp. 108–122. Springer (2003)
4. Becker, B., Drechsler, R., Eggersglüß, S., Sauer, M.: Recent advances in sat-based atpg: Non-standard fault models, multi constraints and optimization. In: *Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. pp. 1–10 (2014)
5. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* **7**, 59–6 (2010)
6. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, and YalSAT entering the SAT competition 2017 (2017), unpublished
7. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)* (1999)
8. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020 (2020), unpublished
9. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: *Theory and Applications of Satisfiability Testing (SAT)*. LNCS, vol. 9340, pp. 405–422 (2015)
10. 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. Springer (2014)
11. Brown, S.T., Buitrago, P., Hanna, E., Sanielevici, S., Scibek, R., Nystrom, N.A.: Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research, pp. 1–4. Association for Computing Machinery, New York, NY, USA (2021)
12. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers* **35**(8), 677–691 (1986)
13. Bryant, R.E., Biere, A., Heule, M.J.H.: Clausal proofs for pseudo-Boolean reasoning. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, vol. 12651, pp. 76–93 (2022)
14. Bryant, R.E., Biere, A., Heule, M.J.H.: Clausal proofs for pseudo-boolean reasoning. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. p. 443–461. Springer (2022)
15. Codel, C.: Verifying SAT Encodings in Lean. Master’s thesis, Carnegie Mellon University Pittsburgh, PA (2022)
16. Cook, W., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. *Discrete Applied Mathematics* **18**(1), 25–38 (1987)
17. Dechter, R.: Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* **113**(1–2), 41–85 (1999)
18. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: *Theory and Applications of Satisfiability Testing (SAT)*. LNCS, vol. 3569, pp. 61–75. Springer (2005)
19. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. *Electron. Notes Theor. Comput. Sci.* **89**(4), 543–560 (2003)
20. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**(1-4), 1–26 (2006)
21. Elffers, J., Nordström, J.: Divide and conquer: Towards faster pseudo-Boolean solving. In: Lang, J. (ed.) *International Joint Conference on Artificial Intelligence (IJCAI)*. pp. 1291–1299. ijcai.org (2018)

22. Elffers, J., Nordström, J.: A cardinal improvement to pseudo-Boolean solving. In: Conference on Artificial Intelligence (AAAI). pp. 1495–1503. AAAI Press (2020)
23. Freeman, J.W.: Improvements to Propositional Satisfiability Search Algorithms. Ph.D. thesis, University of Pennsylvania, USA (1995)
24. Gent, I.P.: Arc consistency in sat. In: European Conference on Artificial Intelligence (2002)
25. Gocht, S., Martins, R., Nordström, J., Oertel, A.: Certified CNF translations for pseudo-Boolean solving. In: Meel, K.S., Strichman, O. (eds.) Theory and Applications of Satisfiability Testing (SAT). LIPIcs, vol. 236, pp. 16:1–16:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
26. Hooker, J.: Generalized resolution and cutting planes. *Annals of Operations Research* **12**, 217–239 (1988)
27. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A Python toolkit for prototyping with SAT oracles. In: SAT. pp. 428–437 (2018)
28. Jabbour, S., Sais, L., Salhi, Y.: A pigeon-hole based encoding of cardinality constraints. *Theory and Practice of Logic Programming* **13** (2013)
29. Jarvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370. Springer (2012)
30. Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from  $n$  objects. In: Constraints in Formal Verification (CFV). p. 39 (2007)
31. Li, C.M., Xiao, F., Luo, M., Manyà, F., Lü, Z., Li, Y.: Clause vivification by unit propagation in CDCL SAT solvers. *Artificial Intelligence* **279**(C) (feb 2020)
32. Liang, J., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9710, pp. 123–140 (2016)
33. Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Majumdar, R., Nowotka, D.: Solving string constraints using sat. In: Enea, C., Lal, A. (eds.) Computer Aided Verification (CAV). pp. 187–208. Springer, Cham (2023)
34. Maglalang, J.C.: Native cardinality constraints: More expressive, more efficient constraints (2019)
35. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of Boolean formulas. In: Haifa Verification Conference (HVC). LNCS, vol. 7857, pp. 102–117 (2013)
36. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
37. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Annual Design Automation Conference. p. 530–535. ACM (2001)
38. Nguyen, V.H., Nguyen, V.Q., Kim, K., Barahona, P.: Empirical study on sat-encodings of the at-most-one constraint. In: The 9th International Conference on Smart Media and Applications. p. 470–475. Smart Media and Applications (SMA), ACM, New York, NY, USA (2021)
39. Niemetz, A., Preiner, M.: Bitwuzla. In: Computer Aided Verification (CAV). p. 3–17. Springer (2023)
40. Oh, C.: Between sat and unsat: The fundamental difference in cdcl sat. In: Theory and Applications of Satisfiability Testing (SAT). pp. 307–323. Springer International Publishing (2015)
41. Pan, G., Vardi, M.Y.: Search vs. symbolic techniques in satisfiability solving. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3542, pp. 235–250 (2005)
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. Soh, T., Le Berre, D., Roussel, S., Banbara, M., Tamura, N.: Incremental sat-based method with native boolean cardinality handling for the hamiltonian cycle problem. In: European Conference on Logics in Artificial Intelligence. vol. 8761, p. 684–693. Springer (2014). [https://doi.org/10.1007/978-3-319-11558-0\\_52](https://doi.org/10.1007/978-3-319-11558-0_52)
44. Soos, M., Bryant, R.E.: Combining CDCL, Gauss-Jordan elimination, and proof generation. In: Pragmatics of SAT (2022)
45. Stephan Gocht, Ciaran McCreesh, J.N.: Veripb: The easy way to make your combinatorial search algorithm trustworthy. In: From Constraint Programming to Trustworthy AI (2020)
46. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake\_lpr: Verified propagation redundancy checking in CakeML. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II. LNCS, vol. 12652, pp. 223–241 (2021)
47. Whittemore, J., Kim, J., Sakallah, K.: Satire: A new incremental satisfiability engine. In: Design Automation Conference (DAC). p. 542–545. DAC '01, ACM, New York, NY, USA (2001)
48. Wynn, E.: A comparison of encodings for cardinality constraints in a SAT solver. ArXiv [abs/1810.12975](https://arxiv.org/abs/1810.12975) (2018)