Thesis Proposal:
Cardinality Constraints in Satisfiability Solving

Joseph E. Reeves
Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
Committee: Marijn J. H. Heule, Randal E. Bryant, and Ruben Martins
External Member: Armin Biere, University of Freiburg

Abstract

Automated reasoning (AR) is the practice of applying deductive reasoning systems at large scale to solve hard problems in mathematics and logic. While the development of modern AR tools has been largely driven by formal methods research focusing on hardware and software verification, the tools themselves have been applied to a broad spectrum of additional problems from cryptography, planning, computational geometry, cloud security, and neuro-symbolic machine learning. Much of the generalizability and success stems from efficient Boolean satisfiability (SAT) solvers and, specifically, their low-level input format: a formula in propositional logic. Forcing the transformation of an abstract problem description into propositional logic, a process known as *encoding*, allows a SAT solver to uniformly employ the highly optimized reasoning of conflict-driven clause learning (CDCL) on disparate problems ranging from quantum circuit synthesis to robot path planning.

AR tools deal with ostensibly intractable classes of problems, so the existence of any particular reduction to propositional logic far from guarantees a solver's success. Several choices concerning an encoding are made at the outset, and one misstep could turn an otherwise simple problem into something requiring years of computation. For common high-level constraints including cardinality, all-different, and XOR several types of encodings into propositional logic exist and choosing the best option often requires expert, problem-specific knowledge. This thesis will narrow the encoding problem to one important class of high-level constraints: cardinality constraints.

Cardinality constraints appear extensively in automated reasoning, e.g., "*at least* k packages must be delivered", or "a resource can have *at most one* status". We propose taking the task of encoding these constraints out of the users' hands, and instead extending the input format of SAT solvers to include cardinality constraints. There are several reasons to single out cardinality constraints. First, many researchers have studied cardinality constraints yielding collections of relevant and challenging benchmark families, various types of encodings, and special purpose solving techniques. Second, controlling the encoding of cardinality constraints on the solver side permits new methods for improving cardinality constraint encodings for sequential and parallel solving. And most importantly, cardinality constraints share a unique relation to CDCL and in a straight-forward way can be incorporated into the standard algorithm, in some cases foregoing the need to encode anything at all. This thesis seeks to provide several avenues for leveraging cardinality constraints to improve solver performance and ultimately support the claim that a cardinality-based input should be the standard for SAT.

Completed work. We implemented an extraction tool for encoded at-most-one (AMO) constraints, a special type of cardinality constraint, and modified the SAT solver CaDiCaL to handle a cardinality-based input. We showed that many existing benchmarks have sub-optimally encoded AMO constraints, and our proof-producing extraction and solving pipeline improved solving performance. In another work, we developed strategies to automatically improve the encodings of large cardinality constraints, solving significantly more problems than with the default encodings.

Proposed work. First, we will develop a general cardinality constraint extraction tool. General extraction is far more challenging than AMO extraction but will drastically improve our solving capabilities with existing benchmarks. Second, we will modify our solver to support encoding during solving (dynamic encoding), allowing us to incorporate our work on improved encodings as inprocessing. Third, we will use information from the cardinality-based input to devise new techniques for parallel SAT solving. Forth, we will instrument an end-to-end proof checking pipeline that will work for problems originating in a cardinality-based format. We present experimental data from prototypes for the proposed research topics, underlining their potential impact.

Contents

1 Introduction

Over the past few decades automated reasoning (AR) tools have been successfully applied to problems across various domains including hardware and software verification [8, 18], cryptography, planning [24], computational geometry, cloud security, neuro-symbolic machine-learning, and theorem proving [14]. Much of their success stems from the evolution of Boolean satisfiability (SAT) solvers. SAT solvers serve as the backbone for automated reasoning tools including bounded model checkers (BMC), satisfiability modulo theory (SMT) solvers, and maximum satisfiability (MaxSAT) solvers.

To use a SAT solver, a problem, first described as a series of high-level constraints, is encoded into a conjunction of clauses (CNF) where each clause is simply a disjunction of literals in propositional logic. The encoding process is crucial to the effectiveness of SAT solving. Modern SAT solvers employ the conflict-driven clause learxning (CDCL) algorithm [20] to either find a satisfying assignment or generate a proof certificate if the formula is unsatisfiable. CDCL solvers are highly optimized with fine-tuned heuristics, often outperforming other tools on complex problems while only reasoning at the clause level. A good encoding maintains some important structure from the original problem, allowing the solver to reason over useful abstractions in the form of propositional variables, whereas a bad encoding can take it away, crippling a solver's ability to learn important facts. Furthermore, an encoding might add several ultimately unimportant constraints that only serve to distract the solver and lead it away from a solution. We address these concerns for one specific type of high-level constraint: cardinality constraints.

A *cardinality constraint* asserts that the sum of a set of *literals* exceeds a given bound, e.g., $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$, where each literal ℓ_i is either a $0/1$ Boolean variable x_i or its complement $\overline{x}_i$. We propose a cardinality-based normal form (KNF) as input for SAT solvers that extends standard CNF with cardinality constraints [23]. We will explore the following topics for handling cardinality constraints: 1) Cardinality Extraction, 2) Dynamic Encoding, and 3) Parallel Solving with cardinality constraint partitioning. For each of these capabilities will provide 4) Trustworthy Solving within formally verified, proof-producing frameworks to establish trust and to support explainability.

Cardinality constraints appear frequently in problems solved via automated reasoning. In an evaluation of the approximately 5,000 SAT Competition Anniversary Track benchmarks from the past two decades of competitions, we found that more than half contained cardinality constraints. These benchmarks come from both industrial and combinatorial tracks, indicating that cardinality constraints appear in many common SAT applications. This is no surprise—counting occurs frequently in problem descriptions, e.g., "a node can have *at most one* color", "a truck can hold *at most* m *packages*", or "*at least half* of the agents should accept the updated identifier". A special case is optimization problems, where a single cardinality constraint sets the bound for some optimization parameter, e.g., "we want to deliver *at least* k *packages*." A common strategy for solving optimization problems is to create new formulas increasing the bound of the cardinality constraint until an unsatisfiable result is returned, and this implies the optimum was found in the prior step. In a related way, to solve optimization problems MaxSAT solvers incrementally add cardinality constraints to a formula [12].

Several groups have proposed cardinality-based normal forms similar to KNF [10, 19]. These groups sought to take advantage of cardinality-based propagation within the CDCL framework, but the results were not satisfactory. In related work, PB solvers have implemented cardinality-based reasoning using the cutting planes proof system [5, 11]. While cutting planes is stronger than the proof system underlying CDCL, these PB solvers have demonstrated success mostly on a narrower set of instances. CDCL still achieves better performance for many important applications from a diverse set of domains. As a result, most of the recent research on cardinality constraints has focused on improving clausal encodings to enhance solver performance. The PySAT [16] tool provides nine different options for encoding cardinality constraints as clauses, and there are several literature

reviews evaluating the performance of different encodings.

The first part of this proposal involves Research Problem 1: Cardinality Extraction. To use cardinality solving techniques on formulas that have already been encoded into CNF, including many available benchmark problems, we require a front-end tool that converts propositional formulas into the cardinality-based normal form KNF. With a powerful extraction tool, the decades of SAT competitions with diverse benchmark sets in CNF could serve as a testing ground for our KNF solving techniques. Benchmark sets in cardinality-based formats are typically specialized and meant to prove the strength of different systems like PB, whereas our goal is to strengthen CDCL solvers on a diverse set of problems. An extraction tool might also have benefits beyond simply converting a formula into KNF; for example, in observing the relationship between different variables to improve constraint encodings. Cardinality constraint detection has been studied for at-most-one (AMO) and at-most-two (AMT) constraints [6], but these implementations are only effective for simple encodings. We plan to explore new techniques for general cardinality extraction that build on our recently developed guess-and-verify framework [23]. This approach uses heuristic methods to identify potential constraints followed by function extraction to check the correctness and parameters of the constraint. The work was initially used for AMO constraints, but the guess-and-verify framework presents a starting point for extracting general cardinality constraints, a problem that has been largely unstudied and is much harder than AMO extraction.

The second part of this project will involve Research Problem 2: Dynamic Encoding. Many encodings have been proposed for cardinality constraints [2–4, 25]. Previous studies considered only how types of encodings compared with respect to size (number of clauses and new variables introduced), propagation power, and their impact on solver performance. In previous work we developed methods to automatically improve encodings for large cardinality constraints that neither affected the size or structure of the encoding. We did this by considering how variable ordering affects clausal encodings and generated good orderings using information extracted from the formula. This general method for improvement worked across the various types of encodings. Separately, we modified the SAT solver CaDiCaL to support cardinality-based propagation, finding that a hybrid approach switching between cardinality-based propagation and clausal encodings had potential for solving problems where either approach was weak. We plan to recast the notion of cardinality-based propagation with the addition of dynamic clausal encodings. We will examine the ways different clausal encodings can be partially encoded (i.e., not all of the encoding clauses and auxiliary variables are added to the formula) and evaluate several heuristics for configuring dynamic encodings. Within this framework we will incorporate our variable ordering work to improve encodings during inprocessing, after the solver has learned important information about the problem.

The third part of this project will involve Research Problem 3: Parallel Solving. We aim to leverage cardinality constraint encodings for problem partitioning. We will evaluate several heuristics for splitting on auxiliary variables to enable the cube-and-conquer parallel solving algorithm [15]. Automatically selecting splitting variables is notoriously difficult and an impediment to using cube-and-conquer. With a cardinality-based input, we have the option to encode cardinality constraints, and importantly, will know the semantics of auxiliary variables within the encoding. The auxiliary variables summarize information from the cardinality constraints, making them ideal candidates for splitting a formula into subproblems.

The fourth part of this project will involve Research Problem 4: Trustworthy Solving. Proof certificates have become a cornerstone of SAT solving technology. When a solver returns an UNSAT result, the solver also produces a clausal proof that can be checked by an independent, formally verified tool. This serves several purposes, increasing trust in the solver's results, foregoing the need to review solver's source-code, and improving debugging capabilities. We plan to use the standard DRAT proof format [28] in our proposed solving techniques. The zone of trust should also be extended

beyond solving. We would like the constraint extraction, as well as problems natively encoded in a cardinality-based normal form, to have formally verified proof checking support. To address these problems, we plan to add cardinality-based propagation to the DRAT checking algorithm.

Much of this work began with a project on exploiting cardinality constraints to improve SAT solver performance [23]. The initial work explored AMO constraint extraction and native cardinality constraint propagation. Since then, we have completed an additional project on automated encoding methods for large cardinality constraints. The results are presented in completed works sections and motivated the core ideas behind this proposal. There are many questions to explore within this proposal, but early indications suggest answers are both attainable and worthwhile.

2 Background

In this section we review the background information important for understanding the remainder of the thesis proposal. This includes topics on Boolean satisfiability, cardinality constraint clausal encodings, the cube-and-conquer parallel solving paradigm, and clausal proofs.

2.1 Boolean Satisfiability

The Boolean satisfiability problem (SAT) asks whether there exists a satisfying assignment to a formula in propositional logic. 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 ℓ is either a Boolean variable x (positive literal) or a negated variable $\overline{x}$ (negative literal).

An *assignment* α is a mapping from variables to truth values 1 (*true*) and 0 (*false*). Assignment α *satisfies* a positive (negative) literal ℓ if α maps $\mathsf{var}(\ell)$ to true (α maps $\mathsf{var}(\ell)$ to false, respectively), and *falsifies* it if α maps $\mathsf{var}(\ell)$ to false (α maps $\mathsf{var}(\ell)$ 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 formulas 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 formula F until a fixed point is reached: for all units α, remove clauses from F containing a literal in α and remove from the remaining clauses all literals negated in α. In cases where unit propagation yields the empty clause ($\bot$) we say it derived a *conflict*.

The most effective algorithm for evaluating the satisfiability of a formula is conflict-driven clause learning (CDCL) [20]. CDCL solvers iteratively expand a partial assignment by assigning values to variables and applying unit propagation. If a conflict is discovered, the solver learns new information, in the form of a clause, and backtracks. This procedure continues until the solver learns the empty clause, indicating the formula is unsatisfiable, or the solver assigns values to all variables without encountering a conflict, indicating the formula is satisfiable under this assignment. Modern solvers incorporate a number of pre- and in-processing techniques that transform the formula before and during solver operation. These typically reduce the number of variables and clauses or shrink the sizes of clauses.

2.2 Cardinality Constraints

A cardinality constraint on Boolean variables has the form $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$ and is satisfied by a partial assignment if the sum of the satisfied literals is at least k. The size of the cardinality constraint is the number of literals (s) it contains. 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 improve solver performance for some problems. It forms the basis of the *bounded-variable addition* formula transformation.

We refer to cardinality constraints as *klauses*, containing a bound k and the literals in the constraint. A standard clause is the special case of a klause with $k = 1$. A clausal encoding of a klause $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$ is *consistent* if assigning any $s - k + 1$ literals to false will always result in a conflict by unit propagation. It is *arc-consistent* [13] if it is consistent and unit propagation will assign all unassigned literals to true if exactly $s - k$ literals are assigned to false.

When $k = s - 1$, the klause can be viewed as an at-most-one (AMO) constraint by negating the literals, i.e., $\ell_1 + \ell_2 + \cdots + \ell_s \geq s - 1$ is equivalent to AMO($\overline{\ell}_1, \ldots, \overline{\ell}_s$). There are special encodings for transforming AMO constraints into clauses including pairwise, sequential counter [25], and commander [17].

For a general klause 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 klause are falsified, unit propagation should lead to a conflict. For these klauses there are many ways to generate a clausal encoding. The Python package PySAT [16] offers API support for generating cardinality constraints using the following encodings: totalizer [3], sorting network [4], cardinality network [2], and sequential counter [25].

▶ **Example 1.** Consider the following cardinality constraint:

$$(x_1 + x_2 + x_3 + \overline{x}_4 \leq 2)$$

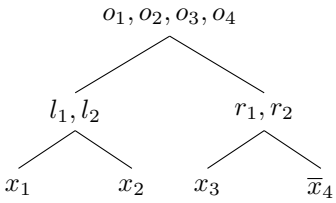
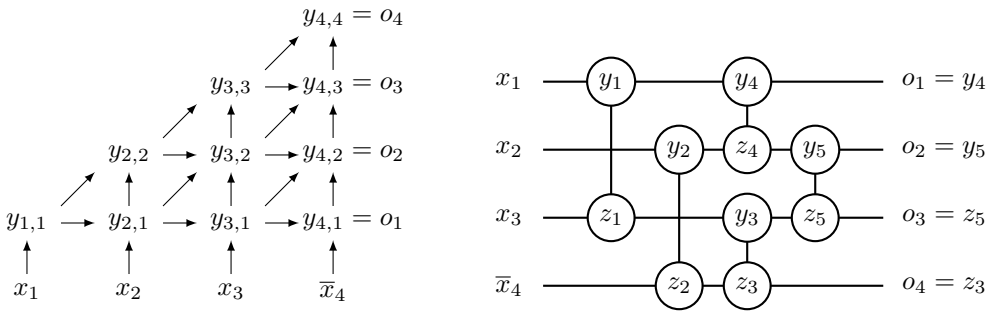


Figure 1 totalizer of Example 1

The totalizer, e.g., Figure 1, is one of the most used classes of encodings, with incremental variants appearing in MaxSAT solvers. We will spend additional time explaining the totalizer as it will appear in examples throughout the proposal. The totalizer uses a binary tree to incrementally count the number of true data literals at each level. Data literals form the leaves, and each node has auxiliary variables representing the unary count from the sum of its children counters. For example, in Figure 1, variable o_3 is true if either pairs l_1, r_2 or l_2, r_1 are true. The bound k is enforced by adding the unit $\overline{o}_{k+1}$. The modulus totalizer (mtotalizer) [22] uses a quotient and remainder at each node to reduce the number of auxiliary variables required to count the sum. The encoding can be simplified by only encoding the count to $k + 1$ at each node (kmtotalizer) [21]. The totalizer splits the sub problems symmetrically, so a solver can reason about x_3 and $\overline{x}_4$ via the auxiliary variables r_1 (at least one of the pair is true) and r_2 (both are true). The further apart literals are in the ordering, the more levels of abstraction are present in their shared counters. For example, x_1 and $\overline{x}_4$ do not share a counter until the root, two levels of abstraction away from the data literals. Furthermore, a node's counters can only be used to reason about all of the data literals below it together, e.g., o_1 means at least one of the four data literals is true.

The sequential counter, e.g., Figure 2a, uses auxiliary variables $y_{i,j} (1 \leq i \leq n, 1 \leq j \leq n)$ in a grid to count the true data literals, where $y_{i,j}$ is true if at least j of the first i data literals are true.



(a) sequential counter of Example 1

(b) sorting network of Example 1

Figure 2

The sequential counter is asymmetrical, with auxiliary variables on the left-hand side summarizing information from fewer data literals. The sorting network, e.g., Figure 2b, and cardinality network share a similar design, using networks that take the data literals as input and through a serious of swaps output the count in sorted order. For each swap, two auxiliary variables are introduced to represent the high (y_j) and low (z_j) outputs. The swaps proceed in layers until the output layer $o_i (1 \leq i \leq n)$, where o_i is true if at least i of the data literals are true. While the sorting network sorts all of the data literals, the cardinality network takes advantage of the bound of the cardinality constraint by implementing simplified merging networks that output at most approximately $k + 1$ bits. Both networks are implemented hierarchically by dividing the sorting into sub problems over subsets of inputs: sorting inputs in groups of two, merging groups then sorting groups of four, merging groups then sorting groups of eight, etc.

We refer to a general At-Least-K Conjunctive Normal Form (KNF) format for cardinality-based formulas throughout the remainder of this document. This format extends the standard file format for CNF to include declarations of cardinality constraints. The following illustrates ALK transformation of the cardinality constraint from Example 1 and its encoding in KNF:

$$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4 \geq 2 \qquad \texttt{k 2 -x1 -x2 -x3 x4 0}$$

As can be seen, klauses are written with a '`k`' followed by the bound and then the literals. A standard clause can be declared by simply listing its literals, or by declaring it as a klause with bound $k = 1$.

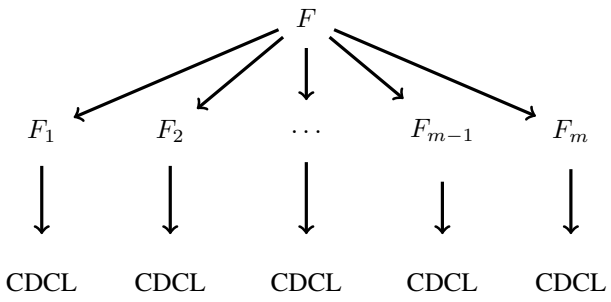


Figure 3 Cube-and-conquer heuristically splits a formula F into N subformulas F_1 to F_m and solves the subformulas using CDCL.

2.3 Cube-and-Conquer

The cube-and-conquer approach [15] aims to *split* a SAT instance F into multiple SAT instances $F_1, \ldots, F_m$ in such a way that F is satisfiable if, and only if, at least one of the instances F_i is satisfiable; thus allowing work on the different instances F_i to proceed in parallel. If $\psi = (c_1 \vee c_2 \vee \cdots \vee c_m)$ is a tautological formula in disjunctive normal form, then we have

$$\mathrm{SAT}(F) \iff \mathrm{SAT}(F \wedge \psi) \iff \mathrm{SAT}\left(\bigvee_{i=1}^{m}(F \wedge c_i)\right) \iff \mathrm{SAT}\left(\bigvee_{i=1}^{m} F_i\right),$$

where the different $F_i := (F \wedge c_i)$ are the instances resulting from the split.

Intuitively, each cube c_i represents a *case*, i.e., an assumption about a satisfying assignment to F, and soundness comes from ψ being a tautology, which means that the split into cases is exhaustive. If the split is well designed, then each F_i is a particular case that is substantially easier to solve than F, and thus solving them all in parallel can give significant speed-ups, especially considering the sequential nature of CDCL, at the core of most solvers.

Automated approaches to construct such ψ have been developed [14], but frequently one needs to manually construct ψ to obtain effective parallelism [26]. In preliminary work, discussed in Research Problem 3 (see Section 5.1), we observed that the auxiliary variables of encoding cardinality constraints can be useful splitting variables.

2.4 Clausal Proofs

CDCL solvers produce certificates for satisfiable formulas and refutation proofs for unsatisfiable formulas. Proofs are represented as a clause sequence, e.g., $F, C_1, C_2, \ldots, C_m$ is a clausal proof of C_m. Each subsequent clause addition step must meet the criteria of a chosen proof system. The case of $C_m = \bot$ serves as a refutation for F. CDCL learns clauses that are logically implied by the formula, and so these can be issued as proof clauses. Our solving techniques will produce proof steps using the same DRAT proof system as do other CDCL solvers [28]. This will allow us to use existing formally verified proof checkers, such as CAKE-LPR [27].

2.5 Magic Squares and Max Squares Benchmarks

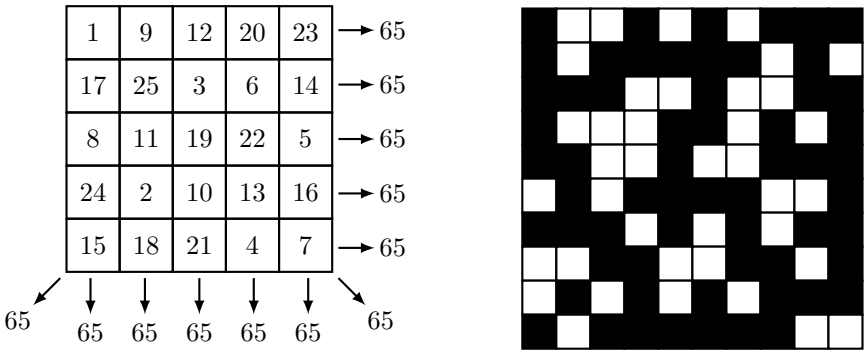


Figure 4 Left a magic square and right an optimal solution of a max square ($n = 10$, $m = 61$).

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 (the magic number M), shown on left in Figure 4. The **Max Squares** problem [29], shown in the right of Figure 4, 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 M_{opt} for each grid such that the Max Squares problem on M_{opt} is satisfiable and on $M_{\text{opt}+1}$ is unsatisfiable. We reference both problems in the research projects below.

3 Research Problem 1: Cardinality Extraction

A number of AR tools build upon existing SAT solvers. Their authors must encode any cardinality constraints as clauses; our experiments have found that these are often encoded inefficiently, in terms of both the number of clauses and the resulting SAT solver performance. While our proposed work will enable users to encode their cardinality constraints directly as klauses, we must support this transition via tools that can automatically detect and extract constraints from CNF formulas. An extraction engine can work as a front-end, transforming formulas from CNF to KNF.

In addition, the extraction engine can serve as a means for accessing the plethora of benchmarks accumulated over years of SAT competitions. These benchmarks come from many different domains and authors and are relevant to the community at large. The golden standard for SAT research is improving solver performance across these diverse benchmark sets, perpetuating the generalizability of SAT solving methods. The wide-spread adoption of KNF hinges on our ability to prove the efficacy of KNF solving techniques on benchmarks such as these.

While there exist other classes of solvers e.g., pseudo-Boolean, that develop some benchmarks in a cardinality-based format, these benchmarks are often only amenable to certain special purpose solving techniques. The KNF solving techniques presented in this proposal target a more general class of problems for which CDCL works well, making the SAT competition benchmarks more applicable.

An extraction engine can have benefits beyond simply converting a formula from CNF to KNF to then apply our proposed KNF solving methods. In related work we used the AMO extraction engine described in Section 3.1 to discover relationships between variables and improve cardinality constraint encodings. Further, several solvers employ special purpose solving techniques for cardinality constraints. The extractor can be used by those tools to improve their performance on certain problems appearing in CNF.

There have been some studies on extracting at-most-one (AMO) and at-most-two (AMT) constraints [6], but previous tools cannot find the full constraints for many of the encodings in current use. Extraction can be broken down into *syntactic* or *semantic* approaches. A syntactic approach looks for exact patterns at the clause level. This approach is used for finding pairwise encodings, which involves 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 [6]. Semantic approaches are more complex, requiring additional methods for reasoning about the Boolean function encoded by a set of clauses. One approach is to use the results of unit propagation to determine if some variable is in a cardinality constraint [6]. A major drawback of this approach is the enormous search space and difficulty determining which variables to propagate, leading to large runtimes and subpar results on common types of encodings. In the completed work we discuss a previous project that performed AMO extraction, outperforming existing tools. In the proposed work we detail an initial concept for a general cardinality constraint extractor along with the research problems it entails.

3.1 Completed Work

Published Works: *Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In Computer Aided Verification (CAV), 2024.*

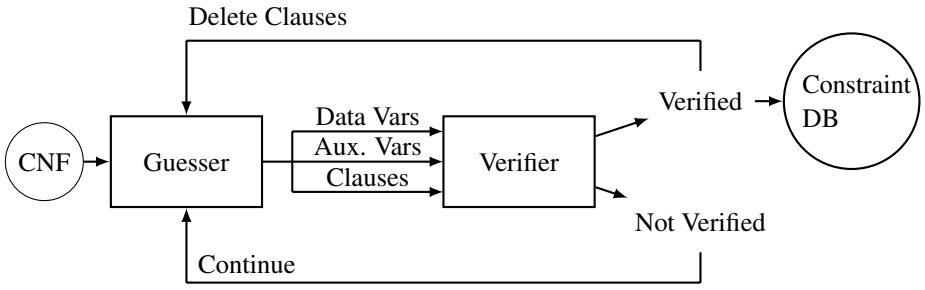


Figure 5 The guess-and-verify framework for extracting cardinality constraints from an input CNF formula. The guesser selects a set of data variables, auxiliary variables, and clauses representing a candidate cardinality constraint. If the verifier verifies the constraint, it is added to the constraint database and the constraint's clauses are removed from subsequent guesses.

In previous work [23], we developed a "guess-and-verify" approach (Figure 5) using ordered binary decision diagrams (BDDs) [7] to perform a semantic check on a guessed cardinality constraint. Specifically, 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. For the "guess" part, we presented methods for finding AMO constraints. The extractor 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 simplistic approach generally works for AMO constraints. The "verify" portion of the extractor uses BDD conjunction and quantification operations to extract the Boolean function represented by the identified clauses, as a function of the data variables. The verifier can traverse the BDD to determine whether it encodes a cardinality constraint, as well as the parameters of that constraint. The guess-and-verify approach balances an optimistic guesser with a conservative verifier to maximize the constraints extracted, while avoiding faulty results. The verifier is built to handle general cardinality constraints, but limited by the guesser, was only used for AMO constraints.

Table 1 Detecting size 10 AMO constraints on the 9 PySAT clausal encodings: pairwise, sequential counter, cardinality network, sorting network, totalizer, k-totalizer, mod k-totalizer, bitwise, and ladder. The table shows ✓when a constraint containing all original data variables was detected, and – otherwise. No approach detected the full constraint for the bitwise or ladder encodings.

Tool	Pair	SCnt	CNet	SNet	Tot	kTot	mkTot	Bit	Lad
Guess-and-Verify	✓	✓	✓	✓	✓	✓	✓	–	–
Lingeling (Syntactic)	✓	–	–	–	–	–	–	–	–
Riss (Semantic)	✓	–	–	–	–	–	–	–	–

PySAT [16] is a Python API for encoding cardinality constraints into clausal form. It supports 9 different encodings. Table 1 compares our guess-and-verify AMO extractor against published approaches in the tools Lingeling and Riss [6]. Our guess-and-verify tool found the original AMO constraint for the majority of commonly used encodings. The other tools find small nested AMO constraints of sizes 3-6, but they cannot find the entire AMO constraint for any encoding other than pairwise.

Table 2 Statistics running the cardinality constraint extractor with a $1{,}000$ second timeout for pairwise and then non-pairwise constraints on the $5{,}351$ SAT Competition Anniversary Track benchmarks. 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. ≥ 5 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 ≤ 15 seconds.

Found	Pairwise	NonPairwise	Both	≥ 5	$\geq 10 \times 10$	Average. (s)	≤ 15 s
3,414	3,089	55	270	64%	17%	69.0	78.0 %

We applied the guess-and-verify tool to the SAT Competition Anniversary Track benchmarks (a collection of formulas spanning a decade of SAT competitions) and summarize the results in Table 2 [23]. Over half of the formulas contain AMO cardinality constraints, and over 17% contain at least ten AMO constraints of size at least 10. This exhibits the prevalence of cardinality constraints within SAT formulas, thus motivating our goal of improving solver performance on these families of problems. These results capture only AMO and not general cardinality constraints. There are likely many more constraints that can be found with improved extraction algorithms.

3.2 Proposed Work

The next step is to move from AMO constraints to general cardinality constraints. It is significantly more difficult to extract general cardinality constraints than to extract AMO constraints. The encodings themselves are more diverse in their structure, often containing unit, binary, and ternary clauses. So, the heuristics we used for AMO extraction would not readily extend to the general case. Further, propagation within the encoded constraint is more complex since assigning a single data variable will not result in propagation of other data variables (assuming the bound is greater than 1). However, our guess-and-verify framework provides an entry point for solving this problem. If we can guess the data and auxiliary variables of a cardinality constraint, then we can use our BDD verifier to validate and characterize the constraint.

Below we present the ideas for a rudimentary extraction algorithm, along with a prototype and some initial experimental results. The guesser will make use of a new statistic called the blocking count. Without getting into details, the blocking count of a literal ℓ measures how many new clauses would be required to make $\mathsf{var}(\ell)$ functionally dependent on other variables. In general, auxiliary variables from encodings often depend on few variables, and therefore will have lower blocking counts than the data variables.

The first step in our guessing algorithm is to compute the blocking count for each literal. Using this information, we classify a variable as a data variable if it meets one of the following three criteria:

1. The blocking counts of both x and $\overline{x}$ are large.
2. Either x or $\overline{x}$ occur in a large clause.
3. Both x and $\overline{x}$ each occur in multiple clauses.

First, many auxiliary variables are defined by a small set of inputs; second, common encodings do not produce large clauses; and third, for some encodings auxiliary variables occur in few clauses. These criteria may misclassify auxiliary variables as data variables, but we can address this issue with additional calls to the verifier.

Given one of the classified auxiliary variables e, a cardinality constraint is guessed in the following way:

- Find the clauses, data variables, and auxiliary variables from the transitive closure over auxiliary variables starting with e.
- Run the BDD-based verification.
- If successful, add the extracted cardinality constraint to the database and remove the clauses.
- Else, convert some data variables to auxiliary variables, recompute the transitive closure, then repeat the process.

This process uses the same guess-and-verify framework from Figure 5 with additional guessing capabilities. The repetition is needed for encodings where it is hard to correctly classify auxiliary variables. For example, in the totalizer encodings the inner nodes introduce auxiliary variables that are both seen as outputs with respect to their children and inputs with respect to their parents, making them appear as data variables (based on the crude classification criteria). The iterative reclassification of some data variables can eventually lead to the detection of a complete totalizer encoding.

Table 3 Detecting Magic Square $N = 3$ cardinality constraints (9 AMO constraints, and 16 cardinality constraints with bound 12 of 24) on the 6 PySAT clausal encodings: sequential counter, cardinality network, sorting network, totalizer, k-totalizer, mod k-totalizer. For each encoding, all 25 cardinality constraints were successfully extracted. Additionally, the number of BDD verification failures and total runtime is listed.

	SCnt	CNet	SNet	Tot	kTot	mkTot
Total Runtime	1.4	8.55	7.0	28.1	14	26.6
Verification Succeses	25	25	25	25	25	25
Verification Failures	0	9	9	89	201	313

From the initial data in Table 3 it is clear that the prototype is not yet suitable for practical use, as the runtimes are too long for a problem with relatively small cardinality constraints. Further, on Magic Squares $N = 4$ the extractor times out at 1800 seconds for most the encodings. This is due in part to the many failed calls to the verifier, and in part to sub-optimal BDD variable orderings. However, the prototype does extract all of the cardinality constraints in Table 3 and is a good starting point for a practical tool given we can solve the following research problems.

When attempting to extract large cardinality constraints, the largest cost in the procedure comes from the verification calls. It is crucial to find a better way to automatically generate good BDD variable orderings in the verifier. For the AMO extraction, variable orderings were generated with a simple method that separated auxiliary variables by distance, but this technique will not work for general cardinality constraints that have a more complicated hierarchical structure. Initially, we plan to create hand-crafted orderings for each type of encoding and experimentally determine which work best. Next, we will attempt to generalize and automate the hand-crafted encodings. This would require recovering the hierarchical structure of the encodings, potentially through a modified breadth-first search over the encoded clauses.

Another problem involves the misclassification of auxiliary and data variables, which causes several additional verifier calls for the totalizer encodings. Each execution can be costly, especially when the BDD variable ordering is suboptimal. One approach to fixing this problem is by adding additional processing during the classification stage, possibly using machine-learning techniques to reclassify some data variables as auxiliary variables. Another approach is to place a filter in front of the verifier. A light-weight filter could reject obviously incomplete cardinality constraints, reducing the overhead of repeated calls to the verifier.

Finally, there are many heuristics involved in the extraction procedure, e.g., determining the cutoffs for criteria in classifying data variables, choosing which auxiliary variables to expand into guessed cardinality constraints, and deciding when to stop the inner verification loop. We plan

to optimize values for these heuristics through large scale experimental evaluations on the SAT Competition Anniversary Track benchmarks.

4 Research Problem 2: Dynamic Encoding

When working with problems that contain cardinality constraints, several layers of decisions must be made before solving ever begins. These decisions will affect the size of the formula, the way information is organized through auxiliary variables, and the type of reasoning a solver can perform. Each decision in the encoding process can have a significant impact on solver performance.

The least intensive approach is to use a format that includes cardinality constraints. For example, PB solvers will accept problems with cardinality constraints, and in some cases the cutting planes proof system is far more effective than resolution. In fact, some SAT solvers have incorporated these solving techniques as a form of preprocessing in an attempt to solve hard combinatorial problems without engaging the CDCL algorithm. Within this setting there exist verified PB proof-checkers that provide for trusted solving.

PB solvers are moderately orthogonal to CDCL and targeted at different classes of problems. If PB solvers are not up to the task, a small next step would be to use a SAT solver that handles cardinality constraints natively. These solvers do not use stronger reasoning like cutting planes and instead slightly modify the CDCL algorithm to support propagation on cardinality constraints. The benefits include smaller formulas, faster propagation, and less effort in the encoding stage; but the tradeoff can be immense. Tools with native propagation underperform, especially on unsatisfiable problems, and have not been widely accepted. A crucial setback to this format is the lack of auxiliary variables. A cardinality constraint encoding will introduce many auxiliary variables that can be used to reason over subproblems within the original cardinality constraint. For example, a solver could use auxiliary variables introduced in an inner node of a totalizer encoding to reason about the set of data literals forming the leaves under that node. This capability is extremely important for reasoning abstractly and finding short proofs.

This leads us to the final option: encoding the cardinality constraints into clauses. Several types of encodings exist (see Section 2.2), and they are compared by size (number of clauses and auxiliary variables), propagation power, and the impact on SAT solver performance. Tools like PySAT provide APIs with various types of encodings, making the process user friendly.

For many problems a clausal encoding of cardinality constraints will yield the best SAT solver performance. But selecting the best encoding is difficult and may require expert knowledge or information not known prior to solving. Furthermore, full encodings for large cardinality constraints can place a strain on the propagation procedure and possibly lead a solver into a fruitless search over auxiliary variables. These do not need to be all or nothing decisions. By combining native propagation and cardinality constraint encodings, we can push questions about handling encodings into the solver.

4.1 Completed Work

Published Works: *Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In Computer Aided Verification (CAV), 2024.*

Cardinality constraints can be propagated natively within a CDCL solver by modifying the propagation and analysis procedures to account for additional watch pointers. For problems with many large cardinality constraints, handling them natively will significantly reduce the size of the formula and increase the speed at which cardinality constraints propagate. We modified the SAT

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
NATIVE PROPAGATION	**0.18**	**1.42**	**6.56**	**12.01**	**46.37**	**460.82**	**164.61**
HYBRID	2.54	37.04	1070.97	887.71	–	–	–
CLAUSAL ENCODING	58.75	246.55	1099.65	4487.79	–	–	–

Max Squares							
	SAT (n,m)				UNSAT (n,m)		
	(7,32)	(8,41)	(9,51)	(10,61)	(7,33)	(8,42)	(9,52)
NATIVE PROPAGATION	0.12	15.01	539.88	660.25	217.62	–	–
HYBRID	0.02	0.92	**17.0**	101.42	1.07	1.27	58.53
CLAUSAL ENCODING	**0.01**	**0.62**	57.83	**24.33**	**0.18**	**0.72**	**22.82**

solver CADICAL to support native cardinality constraint propagation and stochastic local search for phase saving.

Early experiments suggested native propagation worked well on satisfiable problems and clausal encoding were important for unsatisfiable problems. We developed a HYBRID approach that takes both the clausal encoding and cardinality constraints as input, only using native propagation during SAT mode. 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. The solver always has access to the clausal encodings, but in SAT mode the native propagation will likely cause the solver to ignore auxiliary variables.

Table 4 shows results for the three approaches on the Magic and Max Squares problems. The Magic Squares has many cardinality constraints, and their sizes grow quickly, so at $n = 10$ the 30,000 cardinality constraints become over 22 million encoded clauses. The clausal encoding weighs down the solver and slows propagation, whereas the native propagation finds solutions for the larger instances. The Max Squares problem is much smaller, with only around 8,500 encoded clauses for $n = 9$. Auxiliary variables introduced in the clausal encoding are useful in both the satisfiable and unsatisfiable instances of the problem. HYBRID attempts to find a middle ground, performing somewhere between the other two approaches.

Under Submission: *Joseph E. Reeves, João Filipe Boucinha de Sá, Mindy Hsu, Marijn J. H. Heule, Ruben Martins. The Impact of Literal Sorting on Cardinality Constraint Encodings. In (AAAI), 2025.*

In a separate project we studied ways to automatically improve the encoding for cardinality

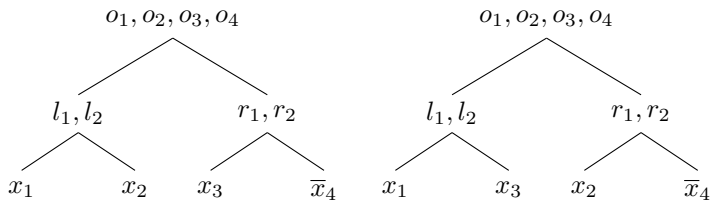


Figure 6 totalizer of Example 1 with two different data literal orderings.

constraints by sorting literals within the constraint. Figure 6 shows how sorting data literals can change the meaning of auxiliary variables without changing the size or structure of the encoding. This amounts to renaming some variables within the clausal encoding. The sorting can be important if certain variables within the cardinality constraint are related. For example, if at most one of x_1 and x_3 can be true, then $\bar{l}_2$ may be learned in the second encoding (right). No such reasoning can take place for the first encoding.

We developed several methods for automatically generating variable orderings. The most successful was based on the proximity of variables. Put simply, we grouped variables that occurred in short clauses together. This method was augmented by first extracting AMO constraints and placing a tighter grouping on variables occurring in the AMOs.

We ran an experimental evaluation on benchmarks from the MaxSAT competition. Each MaxSAT benchmark was converted to one satisfiable and unsatisfiable problem containing the hard clauses and a cardinality constraint over the soft clauses. We encoded the cardinality constraint using the mod-totalizer with three sorting options:

- Proximity: apply our proximity metric augmented by AMO extraction.
- Natural: use the original ordering provided in the problem.
- Random5: use 5 random orderings and present the best runtime.

The results are shown in Figure 7. The proximity ordering takes some time to compute, causing it to lag behind until around 400 seconds, but it solves the most instances by the timeout. The random orderings perform much worse than the others, highlighting the substantial cost of a bad ordering.

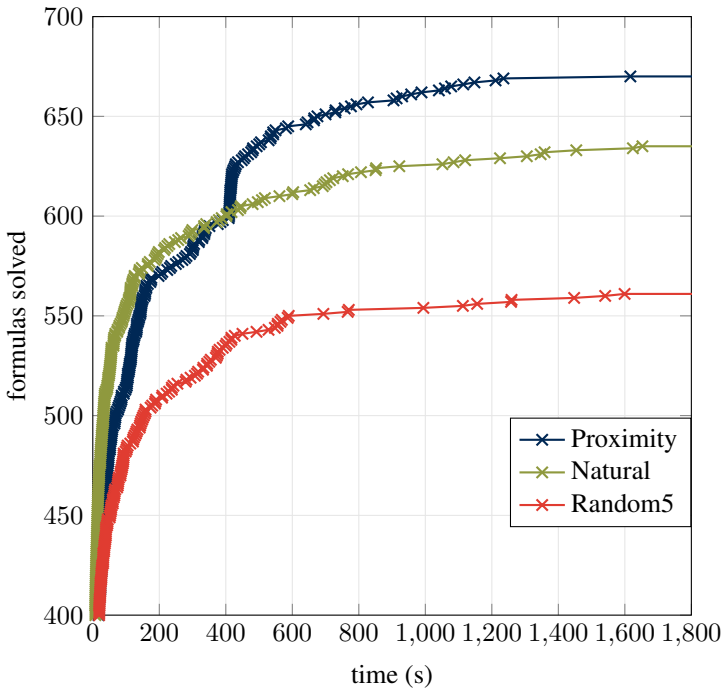


Figure 7 Instances solved on MaxSAT competition benchmarks using the three literal sorting methods.

4.2 Proposed Work

The idea behind a dynamic encoding is to incrementally add portions of an encoding to the solver over time. Dynamic encodings will address two separate questions: Can we decide during solving

whether to leverage native propagation or encode cardinality constraints? And can we create better encodings after running the solver and learning some information about the problem? Previously, these encoding decisions had to be made upfront, but with dynamic encodings they can be held off.

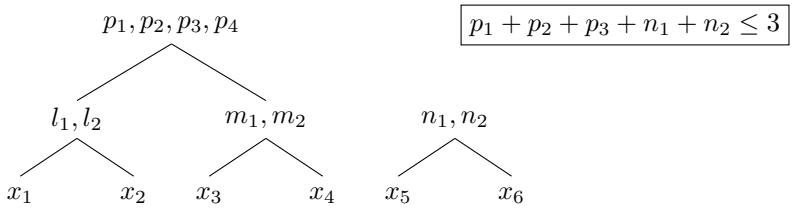


Figure 8 Partial totalizer encoding for the cardinality constraint $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 3$. Note, $\overline{p}_4$ is added to the clausal encoding to enforce the bound on that node.

A partial encoding of a totalizer is shown in Figure 8. Starting at the leaves, a partial encoding is formed by iteratively selecting pairs of nodes and merging them to create a new node. If this process ends before the full totalizer tree is constructed, the auxiliary variables from each of the unpaired nodes are added to a cardinality constraint to enforce the bound.

A partial encoding is a snapshot of a dynamic encoding. The solver could initially propagate on the constraint $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 3$ natively. After some time, it might decide to extend the encoding. It could add the encoding clauses for the partial totalizer in Figure 8 and replace $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 3$ with $p_1 + p_2 + p_3 + n_1 + n_2 \leq 3$. The solving would continue, then after more time it might decide to finish the encoding, removing the constraint $p_1 + p_2 + p_3 + n_1 + n_2 \leq 3$ and adding the remaining clauses.

We will explore multiple heuristics for determining when and how far to encode cardinality constraints. One option is to take a global approach, looking at the problem as a whole. If solving heuristics such as the average decision depth or glue value suggest the problem is hard, proving unsatisfiability may be intractable. It might make sense to avoid encoding the cardinality constraints, keeping the propagation fast and increasing the chances of getting lucky and finding a satisfying solution. On the other hand, if it seems the problem is likely unsatisfiable, encoding clauses could help the solver find a shorter proof. Additionally, we can consider the number of clauses in the formula. We might forego encoding if the number of clauses is very large. If the solver learns units after some time, the number of clauses in the formula would drop as well as the number of clauses created by an encoding, making it less expensive.

At a more local level, the activity of cardinality constraints can be tracked. Activity refers to the number of times a constraint appeared as a reason in conflict analysis. If the activity for a cardinality constraint is high, then the solver is propagating the constraint frequently and it may be beneficial to add a partial encoding of the constraint. This would provide the solver with auxiliary variables that enable better reasoning capabilities with respect to the cardinality constraint. Within a cardinality constraint each variable has its own activity. This value can guide which individual nodes in the totalizer are selected for further encoding.

Our previous work on literal sorting can also be incorporated into the dynamic encoding. In the example from Figure 8 assume only the first layer of the totalizer was encoded, i.e., the p_i variables were not introduced. After some time solving, learned clauses may suggest a strong relationship between the m_i and n_i variables. When the solver decides to extend the encoding, the two nodes (m_i and n_i) could be grouped under the p_i node, and this may be more effective than the l_i and m_i grouping. One way to test the relationship between variables is to use our proximity measure over learned clauses. Depending on when this procedure is enacted, it would be more efficient to only traverse a subset of the learned clauses, e.g., learned clauses with a small glue value.

This work is motivated by a project on dynamic encodings for sorting networks [1]. The approach in [1] uses separate propagation engines and variable activity to decide when to extend the encoding. We go beyond this by considering more heuristics, propagating natively within the same solver, and changing the encoding via literal sorting on the fly.

5 Research Problem 3: Parallel Solving

Given massive computing resources, cube-and-conquer has been the most effective parallel SAT solving approach for tackling hard combinatorial formulas. However, breakthrough results found via cube-and-conquer have typically required significant expertise in the problem domain and thorough experimental evaluations to find good splitting variables. Automated splitting techniques on general problems have been largely unsuccessful. The semantic meaning of variables is lost once the problem is encoded into clauses and attempting to recover insights about these variables is difficult.

When a formula is written in KNF the cardinality constraints are not yet encoded into clauses. We have the option to select an encoding for the cardinality constraints, and more importantly, we will know the semantics of the auxiliary variables generated by the encoding. This still does not provide much information about the data variables, but splitting on auxiliary variables may suffice on many problems. In prior work, cube-and-conquer was used to find the packing chromatic number of the infinite square grid [26]. Good splitting relied on auxiliary variables produced through SBVA. These variables summarized constraints on the square grid. In a similar way, auxiliary variables produced by a cardinality constraint summarize information over the problem variables. Thus, we will attempt to answer the following question: can we use auxiliary variables within clausal encodings for cardinality constraints to split and effectively parallelize solving for KNF formulas?

We will focus on a splitting approach targeted at optimization problems with one large cardinality constraint. Many of these optimization problems are difficult to scale with constraint programming techniques, and so any improvements could have a large impact on this domain.

5.1 Proposed Work

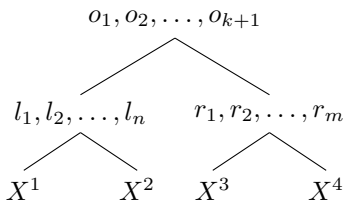


Figure 9 Visualization of the top portion of an at most k totalizer encoding where the X^i nodes represent a subtree over one fourth of the s data literals in the cardinality constraint. For example, X^1 is the subtree over the data literals $x_1, \ldots, x_{s/4}$.

We developed an initial approach for selecting splitting variables using the totalizer encoding. The totalizer uses unary encodings to count sums of data literals and then sums of the generated auxiliary variables. A unary encoding uses one new auxiliary variable per value, e.g., $o_1 = 1, o_2 = 2, ..., o_k = k$, and if a unary encoding variable is true then the sum of incoming variables is at least that value. Given the totalizer from Figure 9, selecting l_2 as a splitting variable will result in the following two cases:

- If l_2 is true, at least 2 of the data literals from the sets X^1 and X^2 are true. This also means that at most $k - 1$ data literals from the sets X^3 and X^4 are true.

- If l_2 is false, at most 1 of the data literals from the sets X^1 and X^2 is true. This provides no information about the data literals from the sets X^3 and X^4 are true.

Intuitively, splitting on an auxiliary variable at an internal node in the totalizer will designate how many true data literals are among its children. One simple approach for selecting a set of splitting variables is to process the internal nodes from left to right, top to bottom, collecting the auxiliary variables that split the node evenly. For example, for the first two nodes in Figure 9 the splitting variables $l_{n/2}$ and $r_{m/2}$ would be selected. The selection would continue until a desired count is reached, and the cubes could be generated by forming a balanced split on the selected variables.

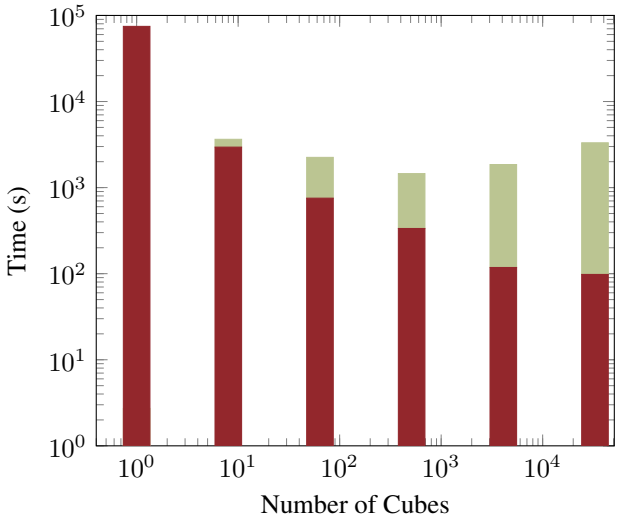


Figure 10 Solving time (CPU) for unsatisfiable MaxSquares $n = 10$ instance using cube-and-conquer with an increasing number of cubes for splitting. Green is total solve time and red is slowest subproblem.

Figure 10 shows the runtimes when splitting on auxiliary variables within the totalizer cardinality constraint encoding. A balanced split on j variables will result in 2^j cubes. The runtime drops dramatically to $3{,}500$ seconds when solving the eight subproblems from a three-variable split, and falls to $1{,}450$ with a-six variable split. The overhead from splitting on more variables eventually leads to an increase in the total runtime (green), but it consistently reduces the slowest subproblem (red). While these initial results are promising, there are still open questions about finding the best splitting variables, measuring how deep to perform the split, determining suitable types of encodings, and predicting if a problem is amenable to this sort of parallelization.

Regarding the selection of good splitting variables, the initial approach used a simple metric of selecting variables that halve the sum of the node. An alternative approach could select a sum that matches the original constraint's ratio of bound to size (k/s). Also, multiple variables can be selected from each node. Consider the case where l_2 is true. If l_3 was set false, then there must be exactly 2 true data literals from the sets X^1 and X^2. Using multiple splitting variables in this way could provide fine-grained control over the meaning of splits.

Once a criterion for variable selection is determined, we must decide how many variables to split on and whether the split should be balanced. In general, splitting is represented as a tree, with each node in the tree signifying a splitting variable with a true and false branch, and each path from the root to a leaf representing a cube. Some problems might require an unbalanced approach that splits deeper in some parts of the tree. In the example above, the case where l_2 is false may be easy, but setting l_2 to true may yield a harder subproblem that requires splitting further within the X^1 and X^2 subtrees.

Another consideration is the choice of encoding. Research Problem 2 discussed techniques to sort literals and therefore modify the meaning of auxiliary variables within the encoding. Reordering is useful for solving a problem sequentially, but another approach to sorting may be more effective given the goal of finding balanced splits. For example, randomizing the variable ordering might distribute information in a way that creates balanced splitting. Moreover, performing the splitting will leave parts of the original cardinality constraint in the formula, and applying the native propagation or dynamic encoding from Research Problem 2 may also improve performance.

An important part of this research will be finding problems for which parallelization is effective. Max Squares is an optimization problem with one large cardinality constraint over all of the variables in the formula. It is not yet known whether this approach requires similarly large cardinality constraints, or if the bound on the cardinality constraint must be close to the middle. An initial study on MaxSAT benchmarks might provide some insight.

6 Research Problem 4: Trustworthy Solving

Automated reasoning is commonly applied to settings that demand both efficient solving and a high confidence in the results. Automated reasoning tools are highly optimized and undergo constant revision to incorporate new techniques. Attempts at developing formally verified solvers have been unable to replicate the performance of existing tools. An alternative approach is to have the unverified tool produce a proof certificate of its result. This proof can be checked using a much simpler and easy to verify algorithm. That is, the solvers generate a log of their reasoning in the DRAT proof format which can then be checked by a formally verified DRAT proof checker. Proofs can be used for checking solver results, debugging solvers during development, and explaining solver reasoning via proof mining.

This proposal presents several methods for solving a formula written in CNF or in KNF. In the first case, the extractor is used to transform the formula into KNF (see Figure 11). In general, proof checking should take the entire pipeline into account and check a result against the original input format of the problem. We discuss ways this can be achieved using existing tools, as well as the possibility of developing a new proof checker with limited cardinality support.

6.1 Completed Work

Published Works: *Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In Computer Aided Verification (CAV), 2024.*

In previous work we narrowed our focus to AMO constraints and developed proof production for extraction and solving (either with native propagation, reencoding, or a hybrid combination). The derivation for reencoding involves adding clauses for the new encoding and potentially some additional clauses, then deleting clauses from the former encoding. The reencoded pairwise constraint can be added directly to the formula. Each binary clause is RUP since assigning two data literals to true would propagate a conflict in the original AMO constraint. It is less straight-forward for more complex AMO encoding. To use the linear encoding additional steps were added to the derivation to account for the backwards propagation of the auxiliary variables. The clauses do not appear in the encoding itself but must appear in the proof.

The native propagation produces RUP proof steps that can be checked against a formula with arc-consistent clausal encodings of the cardinality constraints. To see this, consider the condition when a cardinality constraint propagates: exactly $s - k$ literals are false. If $s - k$ data literals are false in an arc-consistent clausal encoding of the cardinality constraint, that encoding would also

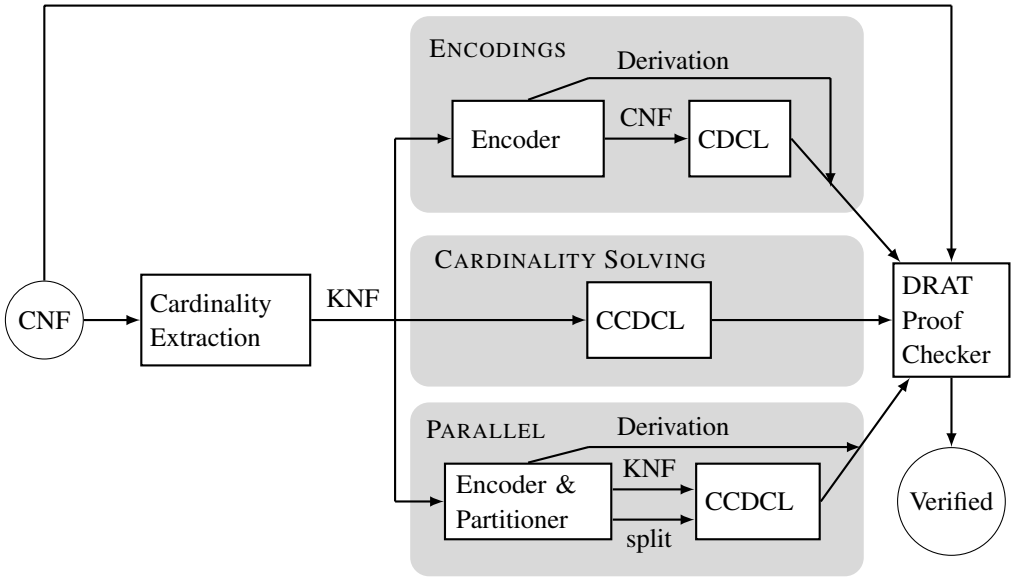


Figure 11 An input CNF formula is passed through the cardinality extraction tool and a KNF formula is generated. We present three separate configurations for solving the KNF formula, along with proof production and validation for each configuration. CCDCL stands for Cardinality-CDCL, and represents a solver equipped with dynamic encoding and cardinality propagation capabilities. In the PARALLEL configuration, the Encoder & Partitioner will split the KNF formula into many subproblems that may then be solved by independent solvers, or a single solver in sequence. The proofs produced by independent solvers are combined to create a proof for the initial problem.

propagate the remaining k data literals. Given a proof from solving with native propagation, a tool like DRAT-TRIM can find the propagation steps on the clausal encoding.

In this work, if a problem was given in KNF we either checked the solution against the KNF or encoded the KNF into CNF then checked the proof against the CNF formula. We made no effort to verify the encoding from KNF to CNF, leaving a gap in our trusted solving pipeline.

6.2 Proposed Work

The extraction of general cardinality constraints complicates the picture in Figure 11. If the extracted KNF is solved with native propagation, the solver will produce RUP steps that can be checked against an arc-consistent clausal encoding similar to the AMO case. However, if the cardinality constraints are encoded into clauses, generating a derivation for the reencoding is more difficult.

One way to solve this problem is by using a SAT solver to generate a DRAT proof showing the former clausal encoding implies the new clausal encoding. First, assume all of the clauses for a new encoding except the unit (o) enforcing the bound can be added as RAT steps. Then those clauses together with the former encoding can be solved under the assumption $\overline{o}$. The formula would be unsatisfiable and the proof would derive the bound o for the new encoding. This proof can be prepended to the DRAT proof produced during solving.

Handling a formula written in KNF is a different challenge. Our solving techniques either reencode the cardinality constraint, propagate on the constraint natively, or perform some combination therein. We would not allow the addition of cardinality constraints to the proof, only RAT clauses. As such, we do not require the stronger and more complicated cutting planes proof system. Instead, we can modify a DRAT proof checker to allow propagation on cardinality constraints. Reencoding would proceed in the way described above, with RAT additions for the new encoding clauses and a

derivation of the new encoding's output. For this situation, the derivation would be generated by a CDCL SAT solver with native propagation since the original constraint is in KNF. We will explore how the additional cardinality constraint propagation would complicate the lemmas underpinning a formally verified proof checker.

If we simply wanted a preprocessor that transformed a KNF into CNF, we could use verified encodings of cardinality constraints into CNF, such as those developed for the Lean4 proof assistant [9]. This would allow the remainder of the verification to be performed purely at the clausal level.

7 Timeline

Dates	Proposed Research Plan
November – February	(1) Finish developing the cardinality constraint extraction and dynamic encodings solver hack. Run experiments on the SAT Competition Anniversary Track benchmarks. (2) Find additional benchmarks for the automatic splitting techniques. (3) Develop a subset of cardinality constraint support for an up-to-date version of CaDiCaL. This includes KNF parsing, encoding with the three most performant encodings (sequential counter, totalizer, mod-totalizer), and native cardinality constraint propagation.
March – April	(1) Determine the best way to equip the tools with proof production and extend the DRAT-trim checker to supports cardinality constraint propagation. (2) Complete a large experimental evaluation of the different extraction and solving techniques. (3) Write and submit conference articles: one article on automatic splitting techniques with Zach Battleman, and one article on extraction and dynamic encodings with Marijn, Randy, and Ruben.
May – August	(1) Collaborate with Jermey Avigad's research lab to extend a verified DRAT checker with cardinality constraint support. (2) Work on new problems uncovered during the course of research. Dive deeper into cardinality constraint solving by understanding when and how native propagation is important along with experimentally determining the usefulness of auxiliary variables in proofs. (3) Discuss ideas with other research teams at various conferences and seminars to discover new use cases.
September – October	Write the thesis. Generate a list of project ideas for future work.
November	Defend the thesis. This could potentially be done in the Spring (April/May) depending on progress and alternative circumstances.

References

1 Ignasi Abío and Peter James Stuckey. Conflict directed lazy decomposition. In *Principles and Practice of Constraint Programming (CP)*, 2012.

2 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 167–180. Springer, 2009.

3 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, pages 108–122. Springer, 2003.

4 K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery (ACM).

5 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–6, 2010.

6 Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In *Theory and Applications of Satisfiability Testing* (*SAT*), volume 8561 of *LNCS*, pages 285–301. Springer, 2014.

7 Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

8 Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

9 Cayden Codel, Jeremy Avigad, and Marijn J. H. Heule. Verified encodings for SAT solvers. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2023.

10 Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation,*, 2(1-4):1–26, 2006.

11 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1291–1299, 2018.

12 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *LNCS*, 2006.

13 Ian P. Gent. Arc consistency in SAT. In *European Conference on Artificial Intelligence*, 2002.

14 Marijn J. H. Heule. Schur number five. In *AAAI Conference on Artificial Intelligence*, 2018.

15 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference (HVC)*, volume 7261 of *LNCS*, 2011.

16 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 428–437, 2018.

17 Will Klieber and Gihwon Kwon. Efficient CNF encoding for selecting 1 from n objects. In *Constraints in Formal Verification (CFV)*, page 39, 2007.

18 Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 389–391. Springer, 2014.

19 Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free (poster presentation). In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *LNCS*, pages 485–486. Springer, 2012.

20 João Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

21 Antonio Morgado, Alexey Ignatiev, and Joao Marques-Silva. Mscg: Robust core-guided MaxSAT solving: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:129–134, 12 2015.

22 Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Modulo based cnf encoding of cardinality constraints and its application to maxsat solvers. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 9–17, 2013.

23 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In *Computer Aided Verification (CAV)*, 2024.

24 Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12):1031–1080, 2006.

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

26 Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is 15. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 389–406. Springer, 2023.

27 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*, volume 12652 of *LNCS*, pages 223–241, 2021.

28 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 422–429, 2014.

29 Ed Wynn. A comparison of encodings for cardinality constraints in a SAT solver. *ArXiv*, abs/1810.12975, 2018.