Cardinality Constraints in Boolean Satisfiability Solving

Joseph E. Reeves

CMU-CS-25-147

November 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work is based on material that was supported by the National Science Foundation under grant CCF-2108521, CF-2229099, CCF-2415773, and was supported 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)

Keywords: Automated Reasoning, Cardinality Constraints, Boolean Satisfiability Solving

For my parents, for always believing in me.

Abstract

Automated reasoning tools are used in a wide range of applications from hardware and software verification to mathematical discovery. The workhorses underpinning many of these tools are Boolean satisfiability (SAT) solvers. To use a SAT solver, a problem must first be *encoded* in propositional logic as a conjunctive normal form (CNF) formula. It may seem strange that a problem as complex as quantum circuit synthesis can be represented in CNF as a series of disjunctive constraints on variables that can either be `true` or `false`, and furthermore, that such a representation is useful. But, thanks to several decades of algorithmic advancements, SAT solvers implementing the conflict-driven clause learning (CDCL) algorithm can efficiently solve a plethora of hard problems that no other tool can handle.

The main drawback to CNF is that it requires the encoding of *all* high-level constraints, and this can be both difficult for a user to find an optimal encoding and the encodings themselves can prevent a solver from leveraging structural information from the constraints during solving. One of the most commonly occurring high-level constraints in SAT problems are cardinality constraints. A cardinality constraint on Boolean variables counts the number of `true` variables and compares the sum against a bound. Cardinality constraints arise any time a problem requires counting. For example, "synthesize a quantum circuit with *at most k* swap gates", or "each value from 1 to 9 may appear *at most once* in every row of a Sudoku puzzle".

This thesis proposes a new standard input for SAT solvers: AtLeastK Conjunctive Normal Form (KNF), with native support for cardinality constraints, which is both easier to use and will help improve solver performance. We develop several methods that take advantage of cardinality information in the KNF format to improve solving. First, we explore how the meaning of new variables introduced in cardinality constraint encodings can be improved by exploiting the relationships between variables in the formula. Second, we implement native cardinality constraint propagation as an extension of a modern CDCL solver and show that the faster propagation is especially helpful on satisfiable problems. Third, we develop a method that uses information from cardinality constraint encodings to split a formula into thousands of independent components that can then be solved in parallel. To evaluate our KNF solving techniques, we developed cardinality constraint extractors that detect AtMostOne constraints and implicit ExactlyOne constraints transforming a formula in CNF to KNF. We used our extractor to convert thousands of benchmarks from the past two decades of SAT competitions into KNF and found that our KNF solving techniques improved solver performance on many of these problems. This work has culminated in an open sourced solver that has been used by several collaborators in mathematical discovery, artificial intelligence explainability, and hardware synthesis research.

Acknowledgments

First, I would like to thank my advisors Marijn and Randy. I am so grateful to have worked with such an amazing pair of mentors. Marijn's enthusiasm and treasure trove of ideas kept the research interesting, and Randy's experience and sage advice helped mold me into a better researcher and educator. I would like to thank Ruben and Armin for their many insights that underpinned some of the work in this thesis, and all of the undergraduate and graduate students at CMU that I had the privilege of working with. I would like to thank Todd Wilson for introducing me to logic and theorem proving.

Next, I would like to thank the friends I made over the course of my studies; the board games, pickleball, yoga, eclipse gazing, and book club helped make Pittsburgh feel like home. I would like to thank my friends still in California; our time spent together online made me feel like I was never far from home. I would like to thank my family – my parents Dion and Tammy, my brother Tim, and my sisters Katelyn and Juliana, for their continued support and multiple visits over the six years.

Finally, I would like to thank my partner Hillary. I am excited to step into the next stage of my academic journey with you by my side. And I would like to thank our dogs Griffon and Brillo for being wily but good companions, sitting with me as I wrote the majority of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Automated reasoning (AR) is a branch of artificial intelligence that applies various forms of reasoning to automatically solve problems in mathematics and logic. AR tools solve problems through the use of deductive inference techniques and exhaustive search, producing provably correct results along with certificates that log their reasoning steps. Note that this is in contrast to the black box prediction engines in machine learning that are built from large swaths of data and computationally intensive training.

In a world driven by technology, where hardware and software touch every part of our lives, trust is paramount.

> "Can I trust that new applications built mostly from code generated by large language models will not introduce security and privacy threats to my devices?"
>
> "Can I trust that the highly optimized processors running on my computer will not fail because of a division-by-zero error?"
>
> "Can I trust that the cryptocurrency I invested in will not lose millions of dollars to hackers due to a logical defect in its blockchain protocol designs?"

AR tools can help bridge this gap in trust by providing "will not" answers to important problems. But, before a problem can be solved by an AR tool (often referred to as a *solver*), the problem must first be *encoded* as a set of constraints in some standardized input format. The decisions made during the encoding process have a cascading effect on a solver's ability to find a solution in a reasonable amount of time. There are many different input formats corresponding to different systems of logic, each with its own set of solvers and types of reasoning. The formats vary in expressivity, meaning that not every problem can be expressed naturally in each format. In fact, several layers of encoding might be needed to express a problem in the simplest format, *Boolean logic*. While encoding into simpler formats can unlock faster and more efficient solver reasoning, it often obfuscates structural information from the problem, and this may hinder solving. To make these ideas more concrete, we consider the puzzle Sudoku.

Sudoku is constructed from a 9×9 grid with 81 cells in total. To solve the puzzle, each cell must be assigned a value and in conjunction the assigned values must all adhere to the following constraints:

1. Each cell can take a value from 1 to 9.
2. Some cell's values are already given (clues).

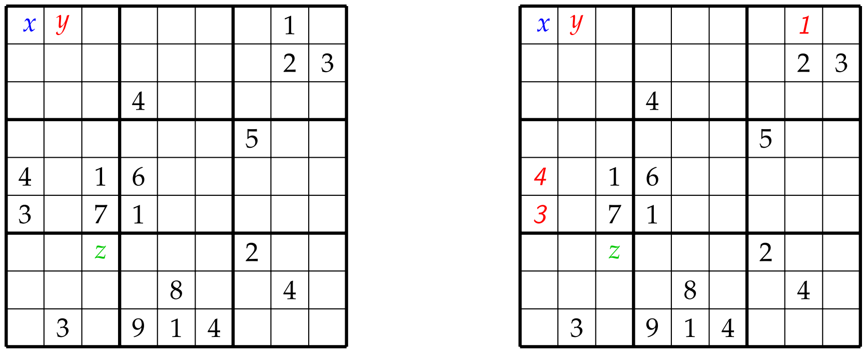


Figure 1.1: Example Sudoku puzzle with 18 clues and three variables x, y, z (left), and colored clues (red) that constrain the variable x (right).

3. Cells in the same row, column, or marked inner 3×3 square must all have different values.

Now, consider the constraints on variable x from Figure 1.1, "x can take a value from 1 to 9, and the clues given state x cannot take values 1, 3, or 4". We show how this constraint can be represented in different formats, with each subsequent format providing less expressivity and therefore requiring additional encoding.

`constraint programming:`	$x \in \{1, 2, \ldots, 9\} \setminus \{1, 3, 4\}$
`integer arithmetic:`	$(1 \leq x \leq 9) \wedge (x \neq 1) \wedge (x \neq 3) \wedge (x \neq 4)$
`pseudo-Boolean logic:`	$(x^1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 \geq 1) \wedge$ $(x^1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 \leq 1) \wedge$ $(\overline{x}^1 \geq 1) \wedge (\overline{x}^3 \geq 1) \wedge (\overline{x}^4 \geq 1)$
`Boolean logic:`	$(x^1 \vee x^2 \vee x^3 \vee x^4 \vee x^5 \vee x^6 \vee x^7 \vee x^8 \vee x^9) \wedge$ $(\overline{x}^1 \vee \overline{x}^2) \wedge (\overline{x}^1 \vee \overline{x}^3) \wedge \cdots \wedge (\overline{x}^8 \vee \overline{x}^9) \wedge$ $(\overline{x}^1) \wedge (\overline{x}^3) \wedge (\overline{x}^4)$

Constraint programming considers variables that can be viewed as objects from a set. We can represent the possible values of x as the set $\{1, 2, \ldots, 9\}$, and this matches how humans might think about Sudoku. Note, for constraint programming the labels could be interchanged, e.g., we could imagine the values in Sudoku as 9 different colors or letters instead of numbers. On the other hand, in integer arithmetic we deal explicitly with integer valued variables, so we must constrain x as being at least 1 and at most 9. Moving into the less expressive Boolean formats, variables can only take the values `true` or `false`. To represent 1 to 9, we must introduce 9 variables $x^1 \ldots x^9$, where if x^i is `true` this means x has value i. In pseudo-Boolean we have constraints on the sum of variables and will say at least one of $x^1 \ldots x^9$ is `true` (x has some value between 1 and 9), and at most one of $x^1 \ldots x^9$ is `true` together meaning x takes exactly one value between 1 and 9. Finally, in Boolean logic, the least expressive format, we must convert the AtMostOne constraint into clauses. This process will be described later in this section. For now, it suffices to view the four different encodings and consider how the less expressive formats look less and less like the original problem. There exist different solvers that can each take as input one of the presented formats, so one might ask why would we ever go through the process of encoding all the way to Boolean logic? The answer is that solvers for Boolean logic use the conflict-driven clause learning (CDCL) algorithm and in combination with highly optimized software and heuristics they can solve many problems that the other solvers (with their corresponding input formats) cannot. But it is not as simple as always encoding a problem and its high-level constraints all the way down to Boolean logic, because information that could be crucial for solving certain problems might be lost.

In this thesis, we develop solving techniques that leverage the structural information of one specific type of high-level constraint: *cardinality constraints*. First, a *literal* is a Boolean variable x, or a negated Boolean variable $\overline{x}$. A cardinality constraint counts a set of Boolean literals and checks that the sum is greater than or less than some bound. Cardinality constraints arise any time a problem requires counting. For example, "Can I deliver a set of m packages with *at most* k delivery trucks?", "Can I synthesize a functionally equivalent circuit using *at most* k transistors?", or "Do all variables in the first row of the Sudoku have *at most one* of each value from 1 to 9?". Standard practice when using a CDCL solver is to encode the cardinality constraint into Boolean logic, hiding the cardinality constraint's original structure from the solver. We propose an alternative approach that brings cardinality constraints directly into the CDCL solver giving access to the following three types of reasoning:

Group Reasoning adds meaning to encodings to allow the solver to reason over groups of variables at the same time. In Sudoku this would be akin to reasoning about whether $x \leq 4$ or $x > 5$. If we grouped y as well, we could make inferences of the sort: if $y \leq 4$ and $x \leq 4$, then z has value 4. In general, a CDCL solver can only reason about these types of groups if they are predetermined in the input formula.

Native Reasoning allows the solver to reason over a single high-level constraint in one step. In Sudoku this would be like reasoning about the constraint "x has at most one value from 1 to 9" natively, so if x has value 5, we know right away that x cannot have any of the values $1, 2, 3, 4, 6, 7, 8, 9$. Such reasoning must be broken up into several steps in a clausal encoding.

Parallel Reasoning allows multiple solvers to reason about different scenarios based on groups. In Sudoku we could have one solver try and solve the puzzle if $x \leq 4$ and at the same time another solver try to solve the puzzle if $x > 5$. We can find these sorts of groups to split on by accessing information about cardinality constraints.

Next, we provide background on Boolean logic and cardinality constraints to concretize these three reasoning techniques.

1.1 Boolean Satisfiability (SAT)

The Boolean satisfiability problem (SAT) takes as input a propositional formula in Boolean logic and asks whether there exists a truth assignment under which the formula evaluates to `true`. In SAT, we often consider propositional formulas in conjunctive normal form (CNF). Formulas contain a set of Boolean variables (x, y, z etc.) that can take the value `true` (1) or `false` (0). A *literal* (ℓ_1, ℓ_2, etc.) is either a variable x (positive) or the negation $\overline{x}$ of a variable x

(negative). The building blocks of CNF are clauses. A *clause* is a disjunction of literals, e.g., $(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_r)$. A CNF formula is a conjunction of clauses, e.g., $C_1 \wedge C_2 \cdots \wedge C_m$. A truth *assignment* ($\alpha$) is a function mapping variables to truth values. A positive literal x is satisfied by an assignment α if $\alpha(x) = 1$, otherwise it is falsified by α, and a negative literal $\overline{x}$ is satisfied by α if $\alpha(x) = 0$ otherwise it is falsified by α. A clause is satisfied by an assignment α if at least one literal is the clause is satisfied, and a CNF formula is satisfied by α if all clauses are satisfied by α. A CNF formula is *satisfiable* if there exists a satisfying assignment, otherwise, the formula is *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 ($\perp$) we say it derived a *conflict*.

Example 1.1

Consider the two formulas in CNF:

$$\begin{array}{ll} F_1 & (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3) \\ F_2 & (x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_2) \wedge (\overline{x}_2) \end{array}$$

The first formula F_1 is satisfiable. One of many satisfying assignments is $x_1 = x_3 = 1, x_2 = 0$. The second formula F_2 is unsatisfiable, with no satisfying assignments.

We will now reexamine Sudoku in the context of CNF. As previously stated, Boolean variables can only take truth values so for each cell x we need to introduce 9 new variables $x^1, x^2, \ldots, x^9$. As an example, if the assignment sets x^3 to `true` then x has value 3, and if it sets $\overline{x}^4$ to `true` then x does not have value 4.

A possible assignment in Sudoku therefore must set one variable for each cell to `true` and the remaining variables to `false`. The Sudoku puzzle is satisfiable if there exists such an assignment that satisfies all of the puzzle's constraints. We often refer to a satisfying assignment as a *solution* to the problem. The Sudoku puzzle is unsatisfiable if there is no possible solution. All Sudoku puzzles played by humans should be satisfiable – it would be terrible to give someone a puzzle with no solutions.

If a formula is unsatisfiable, a solver need not test out every possible solution and show they all fail. Indeed, this strategy would be infeasible. In a Sudoku puzzle with 18 clues there would be 9^{63} ways to fill in the remaining cells, and it would be impossible to enumerate all of these possible assignments and check if any are valid solutions. Instead, the solver will reason through inferences (e.g., learning facts like the one shown as an example for group reasoning above) and construct an argument with many (sometimes millions of) steps concluding no solutions exist.

Now that we have a set of variables, we must enforce their meaning with a new set of constraints. We need one constraint that shows a cell has some value, and this can simply be represented as a clause $(x^1 \vee x^2 \vee x^3 \vee x^4 \vee x^5 \vee x^6 \vee x^7 \vee x^8 \vee x^9)$. Then, we need to enforce the constraint that each cell has at most one value, but this is a cardinality constraint $(x^1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 \leq 1)$. We cannot translate this directly into CNF – no single clause can represent the cardinality constraint – so instead we must encode the constraint as a set of clauses.

1.2 Cardinality Constraints

An AtLeastK cardinality constraint on Boolean literals asserts that the sum of a set of literals exceeds a given bound, e.g., $\ell_1 + \ell_2 + \cdots + \ell_r \geq k$.

Example 1.2

Consider the two formulas with clauses and cardinality constraints:

$$\begin{array}{ll} F_1 & (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3) \wedge (x_1 + \overline{x}_2 + \overline{x}_3 \geq 2) \\ F_2 & (\overline{x}_2) \wedge (\overline{x}_1 \vee x_3) \wedge (x_1 + x_2 + \overline{x}_3 \geq 2) \end{array}$$

The first formula F_1 is satisfiable. It has exactly one satisfying assignment: $x_1 = x_3 = 1, x_2 = 0$. The second formula F_2 is unsatisfiable, with no satisfying assignments.

Cardinality constraints generalize clauses. A clause $(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_r)$ could be viewed as an AtLeastOne cardinality constraint $\ell_1 + \ell_2 + \cdots + \ell_r \geq 1$ which exactly matches the semantics of a clause stating that at least one of the literals is `true`.

An AtMostOne cardinality constraint on literals $\ell_1, \ell_2, \ldots, \ell_r$ (written as AMO $(\ell_1, \ell_2, \ldots, \ell_r)$) can be viewed as either $\ell_1 + \ell_2 + \cdots + \ell_r \leq 1$ or alternatively as $\overline{\ell}_1 + \overline{\ell}_2 + \cdots + \overline{\ell}_r \geq r - 1$. Any AtMostK can be transformed into an AtLeastK by negating the literals and updating the bound as the number of literals minus the previous bound. AtMostOne constraints appear frequently in SAT problems and are often used for selecting unique values. For example, in Sudoku the AMO $(x^1, x^2, \ldots, x^9)$ states that the cell x will select at most one value between 1 and 9.

The simplest encoding of an AtMostOne into clauses is the *direct* (a.k.a. Pairwise) encoding, given by the following set of binary clauses:

$$\mathsf{AMO}(\ell_1, \ldots, \ell_r) = \bigwedge_{i=1}^{r} \bigwedge_{j=i+1}^{r} (\overline{\ell}_i \vee \overline{\ell}_j)$$

Intuitively, this encoding blocks every pairwise combination of two variables from being `true` at the same time.

In Sudoku, the Pairwise encoding for AMO$(x^1, x^2, \ldots, x^9)$ would include the clauses $(\overline{x}^1 \vee \overline{x}^2)$, etc. To satisfy the first clause in the encoding, $(\overline{x}^1 \vee \overline{x}^2)$, either x^1 or x^2 must be `false`. Put another way, if both x^1 and x^2 were `true`, the clause could not be satisfied, so there are no satisfying assignments where both x^1 and x^2 are `true`. Creating these clauses for every pair of possible values of x will give the desired effect that x cannot take more than one value simultaneously.

One drawback of the Direct encoding is its size, introducing a quadratic number of clauses. But more importantly, the Direct encoding does not allow the solver to reason about groups. We can account for both problems with the introduction of *auxiliary variables*. These are fresh variables not appearing anywhere else in the formula. They can be used to both make an encoding smaller and improve a solver's reasoning capabilities.

The Split encoding for AtMostOne constraints introduces new variables that split the original constraint into sub-constraints, then applies the Pairwise encoding to the sub-constraints. One instance of the Split encoding would be to split the AtMostOne on x into two smaller constraints:

$$\mathsf{Pairwise}(x^1, x^2, x^3, x^4, \overline{x}_{\leq 4}) \wedge \mathsf{Pairwise}(x^5, x^6, x^7, x^8, x^9, \overline{x}_{\geq 5}) \wedge (\overline{x}_{\leq 4} \vee \overline{x}_{\geq 5})$$

In this encoding, the auxiliary variable $x_{\leq 4}$ is `true` if x has a value between 1 and 4, and the auxiliary variable $x_{\geq 5}$ is `true` if x has a value between 5 and 9. The binary clause ensures that both auxiliary variables cannot be `true` at the same time, x must either be less than 5 or greater than 4. These auxiliary variables will allow a solver to reason about the groups of variables they represent.

Example 1.3

Consider the puzzle in Figure 1.1, and the fact learned in the **group reasoning** example:

If x and y are both less than 5, then z must be 4.

Using a Pairwise AtMostOne encoding for each of the cells, the fact can be represented as the clause

$$(x^5 \vee x^6 \vee x^7 \vee x^8 \vee x^9 \vee y^5 \vee y^6 \vee y^7 \vee y^8 \vee y^9 \vee z^4)$$

This clause can only be falsified if x is less than 5, y is less than 5, and z is not 4, violating the fact above.

Now, assume we use the Split encoding with variables $x_{\leq 4}$, $x_{\geq 5}$ and $y_{\leq 4}$, $y_{\geq 5}$. We can represent the same fact with the much shorter clause

$$(\overline{x}_{\leq 4} \vee \overline{y}_{\leq 4} \vee z^4)$$

Example 1.3 shows one instance where auxiliary variables in cardinality constraint encodings can be used to reason about groups of values. In general, auxiliary variables are very powerful tools in SAT solver reasoning and can turn an otherwise hard problem into an exponentially easier one. However, choices made during encoding can change the meaning of auxiliary variables, making them less useful to reason with. If we had performed a split on $x \leq 5$ and $x \geq 6$ for x, and $y \leq 3$ and $y \geq 4$ for y, we would not have been able to infer the short fact from Example 1.3 because the auxiliary variables would be representing different groups of values. Therefore, it is important to create auxiliary variables with useful *meanings* when encoding cardinality constraints.

Example 1.4

Consider the puzzle in Figure 1.1, and the propagation of clues constraining x described in the **native reasoning** example. With the Pairwise encoding on $(x^1 + x^2 + \cdots + x^9 \leq 1)$, propagating values will take several steps. If x has value 5, setting x^5 to `true`, the solver will need to lookup up each relevant binary clause, for example, starting with $(\overline{x}_1 \vee \overline{x}_5)$ to propagate x^1 to `false`.

The Split encoding is no better in this regard and will incur more propagations because it needs to assign values to the auxiliary variables. If x^5 is `true`, then $x \geq 5$ is `true` so $x \leq 4$ must be propagated to `false` before the remaining values of x can be propagated to `false`.

If the cardinality constraint is propagated natively, the solver would look up the single constraint $(x^1 + x^2 + \cdots + x^9 \leq 1)$, then propagate every variable except x^5 to `false` in one step.

Example 1.4 describes a case where native reasoning can propagate several values at once, whereas unit propagation on a clause can only propagate a single value at a time. Storing the cardinality constraints natively in the solver improves propagation speed and reduces the size of the formula since you no longer need to add the often large clausal encodings. Unfortunately, enabling native reasoning on cardinality constraints does not always lead to faster runtimes. There exist solvers that already implement native reasoning, and they often cannot compete with solves that take as input a clausal encoding. For some problems group reasoning is crucial for fast solving times, but group reasoning is only made possible via auxiliary variables in clausal encodings. The important question is *when* to perform native reasoning and *how* to combine native reasoning and group reasoning.

Example 1.5

Consider the puzzle in Figure 1.1, and the example for **parallel reasoning**. Assume all of the constraints for the Sudoku puzzle are represented by formula F. We can generate four new formulas:

$$\begin{aligned}
F_1 &= F \wedge x_{\leq 4} \wedge y_{\leq 4} \\
F_2 &= F \wedge x_{\leq 4} \wedge y_{\geq 5} \\
F_3 &= F \wedge x_{\geq 5} \wedge y_{\leq 4} \\
F_4 &= F \wedge x_{\geq 5} \wedge y_{\geq 5}
\end{aligned}$$

Each formula represents a subproblem created by splitting on groups of values for x and y, and together the subproblems cover all possible values. Each formula can be solved by an independent solver. If one is satisfiable, then the original formula F has a solution, but if they are all unsatisfiable, then F has no solutions.

Example 1.5 shows how auxiliary variables can be used to split a formula into several subformulas which can then be solved in parallel. Similar to group reasoning, parallel reasoning depends heavily on the meaning of auxiliary variables in the encoding.

Now, imagine you are trying to solve Sudoku with an off-the-shelf SAT solver. For group reasoning and parallel reasoning, you would need to find an optimal encoding that made the most out of auxiliary variable meanings, and for native reasoning you would need to decide which cardinality constraints to use natively. Trying to make these decisions as a user is difficult, and even experts may go through a series of trial and error. Towards this end, we introduce a new input format for SAT solvers: **AtLeastK conjunctive normal form (KNF)**. KNF is a small extension to CNF that adds AtLeastK cardinality constraints to the format. With KNF, questions like "should I propagate natively?", "what encoding should I choose?", "how can I parallelize solving?" will be moved from the user into the solver. This is not simply a matter of making solvers easier to use. Some of these questions, like how and when to natively reason about cardinality constraints, cannot be answered prior to solving and should be handled dynamically during solver execution. Furthermore, a solver can leverage information from the cardinality constraints to discover optimal encodings that would be nonobvious to SAT experts.

It should be said that there are many tools that already take cardinality constraints as input. The closest to a SAT solver is a pseudo-Boolean (PB) solver, which can use stronger cardinality reasoning to solve certain problems exponentially faster than a SAT solver. We purposefully extend CNF to KNF, focusing solely on CDCL SAT solvers, for several reasons. First and foremost, CDCL can solve a wide range of problems that other solving paradigms (including PB solvers) cannot. We want to preserve the general effectiveness of CDCL and at the same time improve performance on problems containing cardinality constraints. Second, there are many existing problems that contain cardinality constraints for which CDCL is the best choice, and these problems are ripe for improvement. Third, we only extend CNF with cardinality constraints and not general pseudo-Boolean constraint because the CDCL algorithm can be modified to support cardinality with only slight changes, but support for PB would require more significant engineering. We will provide more context for these decisions in the following chapter.

1.3 Thesis Outline

The main goal of this thesis is to provide the groundwork for making KNF the new SAT standard, elevating cardinality constraints to first class citizens of the SAT world.

Thesis Statement: Extending the input format of SAT solvers from CNF to KNF will make tools easier to use and allow us to develop automated encoding and solving techniques that improve solver performance. **Ease-of-use** refers to the ease in which an automated reasoning tool can be picked up by a non-expert and applied to whatever novel problem they wish to solve. One of the largest barriers of entry to SAT solving is the encoding process, with most of the prominent results derived by SAT solvers coming at the hands of experts in the field. The KNF format will make the encoding process easier by taking one choice away from users: how to encode and solve problems with cardinality constraints. Furthermore, KNF will allow us to **improve performance**. By using the KNF format we will have access to cardinality constraints in context of the entire formula, enabling us to capture relationships amongst literals that appear hidden in the formula, and we can use this information to automatically improve clausal encodings. In addition, the KNF input will allow us to choose, within the solver, whether to encode a cardinality constraint or propagate natively, increasing the capabilities of the solver without requiring any additional interaction from the user.

First, in Chapter 2 we will motivate the thesis through a discussion of state-of-the-art research on cardinality constraints in SAT. Then, we will explore the three main research topics: extraction, encodings, and solving.

In Chapter 3 Extraction, we will present a framework for detecting encoded AtMostOne (Section 3.2) and ExactlyOne (Section 3.3) cardinality constraints in CNF formulas. This will provide a means for transforming a formula from CNF into KNF, allowing us to apply our new encoding and solving techniques to existing problems.

In Chapter 4 Encoding, we will discuss three techniques for improving cardinality constraint encodings. Each of these leverage the concept of *group reasoning*. First, in Section 4.1 we will show how a combination of extraction and encoding can turn Pairwise AtMostOne constraints into linear AtMostOne constraints, and this will lead to consistently better solver performance. Second, in Section 4.2 we will describe several techniques for sorting literals within cardinality constraints. Our proximity-based sorting algorithm will create more useful groups of variables to reason over, and we will show that these new encodings will outperform existing encodings. Third, in Section 4.3 we will describe a method that reencodes implicit and explicit ExactlyOne constraints. A key part of this reencoding procedure is to *align* groups of variables across cardinality constraints to produce better groupings.

In Chapter 5 Solving, we will describe our implementation of native reasoning in the state-of-the-art CDCL SAT solver CaDiCaL. In Section 5.1 we will explore the tradeoffs inherent in equipping the solver with native propagation. This involves both accounting for modern reasoning techniques and synchronizing with group reasoning from encodings so that native propagation can be seamlessly incorporated into the solver without inhibiting performance. Then, in Section 5.2 we will propose a small extension to KNF with conditional cardinality constraints and show how conditional cardinality propagation can be used to quickly solve certain discrete geometry problems.

In Chapter 6 Parallel Solving, we will present an approach to parallelize solving using cardinality constraints. This approach leverages the KNF format, allowing us to generate our own clausal encodings of cardinality constraints. Form these encodings we can select splitting variables that best divide the search space.

Finally, in Chapter 7 Conclusion, we will conclude with a discussion on ongoing research and a path towards a KNF standard.

Most of the sections within the thesis are derived from published research. Below are the four conference papers, along with the chapters and sections they appear in.

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. *From Clauses to Klauses*. In *Computer Aided Verification (CAV)*. (2024). [91] (Sections 3.2,4.1,5.1)

Joseph E. Reeves, João Filipe, Min-Chien Hsu, Ruben Martins, and Marijn J. H. Heule. *The Impact of Literal Sorting on Cardinality Constraint Encodings*. In *Conference on Artificial Intelligence (AAAI)*. (2025). [92] (Section 4.2)

Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule. *Reencoding Unique Literal Clauses*. In *Theory and Applications of Satisfiability Testing (SAT)*. (2025). [96] (Sections 3.3, 4.3)

Zachary Battleman, Joseph E. Reeves, and Marijn J. H. Heule. *Problem Partitioning via Proof Prefixes*. In *Theory and Applications of Satisfiability Testing (SAT)*. (2025). [18] (Chapter 6)

Experimental Setup

We will present results from several experimental studies in the different chapters. Most of the experiments were performed on the Pittsburgh Supercomputing Center (PSC) [28]. Each node has 128 cores and 256 GB RAM. In the parallel experiments (Chapter 6) we run 32 solver instances per node so each solver has approximately 8 GB of RAM. In the remaining experiments on the PSC we run 64 solver instances per node so each solver has approximately 4 GB of RAM. We did not find this memory allocation limiting except in the case of the Java-based solver SAT4J.

Only the experiments in Section 4.2 were run on a separate cluster. They were ran on StarExec [101] with specs found on their website [99]. Each solver instance had 32 GB of RAM.

When comparing solver runtimes, we use real time and often compare the average PAR2 scores. PAR stands for penalized average runtime. PAR2 is computed by summing runtimes for all benchmarks on which a solver found a solution, then adding two times the timeout for all benchmarks on which the solver did not find a solution (either due to a timeout or a memory out). The average PAR2 divides the PAR2 score for a solver by the number of benchmarks.

1.4 Additional Published Research

Over the course of my PhD studies I have worked on several research projects related to SAT but outside of this thesis's focus on cardinality constraints. My published projects are listed below, along with short descriptions for each.

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. *Moving Definition Variables in Quantified Boolean Formulas*. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. (2022). [88]

In this work, we developed a technique for automatically reordering variables in the prefix of a quantified Boolean formula (QBF). QBF extends SAT with the addition of universal and existential quantifiers. We found that existentially quantified *definition* variables are often placed in the innermost quantifier but can be moved to the quantifier of their defining variables without changing the satisfiability of the formula. We implemented a tool that detected definition variables using SAT preprocessing tools, moved these variables to different quantifier levels in the QBF prefix, and output a proof that each movement was sound. We showed that variable movement improved performance across the board for a representative set of modern QBF solvers.

This work ties into a general interest in preprocessing and formula transformations that improve performance. Furthermore, the focus on definition variables and their ordering is similar to the sorting of literals within cardinality constraints in that it allows the solver to reason more compactly over certain auxiliary variables. Lastly, this work required sophisticated proof logging to prove the correctness of each instance of variable movement. Similar styles of proof logging appear throughout the reencoding work in this thesis.

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. *Preprocessing of Propagation Redundant Clauses*. In *Journal of Automated Reasoning (JAR)*. (2023). [89]

In this work, we developed a preprocessing technique (PRELEARN) that learns binary and unit propagation redundant (PR) clauses. Adding these clauses to a CNF formula will often improve solving time, especially on hard combinatorial problems. PR clauses are derived from a stronger proof system than typical SAT reasoning but determining whether or not a clause is PR is NP-complete. Therefore, we implemented a tool that uses a SAT solver to test whether heuristically selected binary and unit clauses were in fact PR. These techniques were adapted to some SAT solvers appearing in the annual SAT competition in 2023 including our own submission PRELEARN with KISSAT earning a bronze medal in the UNSAT track and KISSAT-MAB-PROP which reimplemented our tool among others changes and earned a silver medal in the overall track [15].

A key part of this work was developing heuristics that measure locality of variables within a formula when constructing candidate binary and unit PR clauses. This motivated a similar approach in the cardinality constraint literal sorting work, where heuristics are used to construct groups of variables based on their proximity within a formula.

Amar Shah, Twain Byrnes, Joseph Reeves and Marijn J. H. Heule. *Learning Short Clauses via Conditional Autarkies*. In *Formal Methods in Computer-Aided Design (FMCAD)*. (2025). [95]

In this work, we developed a preprocessing technique that learns PR clauses from conditional autarkies and show that in some cases these PR clauses can help a solver find shorter proofs. The approach depends heavily on heuristics for strengthening PR clauses and filtering away less-useful PR clauses, ensuring the solver only accesses PR clauses that will positively impact its reasoning.

This work required tight integration of the conditional autarky algorithm within the search procedure of a SAT solver. This presented a similar set of challenges to those from the tight integration of native cardinality propagation in search.

Joseph E. Reeves, Benjamin Kiesl-Reiter, and Marijn J. H. Heule. *Propositional Proof Skeletons*. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. (2023). [90]

In this work, we developed several techniques for summarizing a SAT solver's reasoning via proof skeletons. Proof skeletons are a subset (often less than one percent) of the reasoning steps used by a SAT solver to prove that a problem is unsatisfiable. We compared the strength of different proof skeletons by measuring how costly it was to expand the skeleton into a complete proof, finding that activity-based metrics produced proof skeletons that captured the most important solver reasoning.

We explored ways to parallelize solving by using a proof skeleton. Similar parallelization approaches based on important variables were used to parallelize cardinality constraint solving. Several of the metrics we considered for the proof skeleton work, including activity, are generally useful and make appearances in various sections of this thesis.

1.5 Repositories

We strive to make all benchmarks, solvers, and evaluation data described in this thesis open source.

The modified version of CaDiCaL with native propagation and hybrid mode-switching, the tool for parallel solving using splitting variables from the Totalizer encoding, and the AtMostOne extractor can be found at Cardinality-CDCL. At the time of writing, this repository is under active development and is soon to include new KNF solving capabilities and extraction techniques.

The extraction and reencoding tool for ULCs and XLCs implemented as a preprocessing routine inside of CaDiCaL can be found at ulc-cadical.

The literal sorting techniques and scripts for converting MaxSAT problems into KNF can be found at LiteralSorting.

Chapter 2

State-of-the-Art: Cardinality in SAT

SAT gained relevance in the 1970's as the core problem in the study of NP [34]. It was shown that many important problems could be reduced to SAT, yet the task of solving SAT was provably hard. Even still, over the last 25 years automated reasoning has hit the mainstream. Modern SAT solvers, backed by several key algorithmic advancements, can turn many theoretically hard problems into practical solutions.

Some recent automated reasoning applications include:

- Solving open problems in mathematics, including the Pythagorean Triples Problem [54], Schur Number Five [51], and the Happy Ending Problem [52].
- Verifying hardware [32], with new advancements in machine learning and neural network verification [60, 115].
- Verifying software [64], including ongoing development for modern programming languages like Rust [65].
- Verifying blockchain protocols [107], for example smart contracts in solidity [4].
- Synthesizing hardware [27], also following the latest trends into quantum circuit synthesis [114].
- Verifying access policies with over a billion solver queries a day at Amazon [93].

There is a vast collection of automated reasoning tools to solve problems across many domains with incongruous representations. For most of these, including bounded model checkers [32], hardware synthesis tools [27], theorem provers, SMT solvers [83, 116], and maximum satisfiability (MaxSAT) solvers [8, 77], SAT solvers serve as the backbone. Therefore, any improvements to SAT solving technology will have a far reach, touching many automated reasoning applications.

On the one hand, the fast pace of open-sourced solver development in the last two decades has spurred the ongoing adoption of automated reasoning and formal methods in industries that care about correctness, with projects often led by research scientists with high levels of expertise. On the other hand, it can be hard for non-experts to take advantage of off-the-shelf automated reasoning tools. A crucial step when using any automated reasoning tool is the choice of encoding. Selecting the optimal set of constraints to express a problem can be the difference between a solver finding a solution in one second and one year. Furthermore, theoretical results underpin the use of different tools and knowing when to opt for different types of solver reasoning (like SMT over SAT), requires years of experience.

In this thesis, we will narrow our focus and explore questions of encoding and solving for SAT problems containing cardinality constraints. First, we will motivate the focus on cardinality constraints by cataloging some relevant problems containing cardinality constraints (Section 2.1). Then, we will discuss the state-of-the-art in cardinality constraint encoding (Section 2.2) and solving (Section 2.3). Finally, we will describe how this thesis addresses existing challenges and paves a new path for handling cardinality constraints in SAT (Section 2.4).

2.1 Cardinality Constraints in the Wild

Cardinality constraints arise in any problem that requires counting. Counting comes in many different flavors. For example, when selecting an object from a set we can use an AtMostOne constraint: "the cell x has *at most one* value from 1 to 9" and "a package out for delivery can be assigned to *at most one* truck". Also, we can use AtLeastK constraints to mark minimum progress: "in the next second *at least half* of the robots should move closer to their final destination" and "in the next turn the player must move *at least two* of his game pieces", or we can use AtMostK constraints to enforce an optimization bound: "plan routes for delivering all packages using *at most k* truck drivers" and "synthesize a quantum circuit with *at most k* swap gates". In this section, we will give more detail on the different types of cardinality constraints and discuss several SAT benchmarks containing cardinality constraints.

2.1.1 Solver Competitions

It is common practice in the AR community to hold annual competitions between solvers. In a typical year, each area (SAT, MaxSAT, SMT, first order logic) receives dozens of solver submissions and hosts a competition to determine which solver achieves the best runtime performance. Crucially, the competition organizers curate a set of new benchmarks that consist of difficult industrial and crafted problems. These benchmarks span a wide set of application domains, testing how robust solvers are as general purpose tools. The competitions are open-sourced, allowing researchers to access previous years' benchmarks and solver source code. This has led to the rapid development and improvement of critical reasoning techniques, yielding solver extensions that dramatically outperform their predecessors year after year.

We will consider benchmarks from two competitions. The first is the SAT competition which uses formulas in CNF. The SAT Anniversary Track was a compendium of the past two decades of competitions, totaling approximately $5{,}000$ benchmarks. In the following section we will discuss how often cardinality constraints occurred in these problems. The second is the MaxSAT competition which uses formulas in an extended version of CNF that represent an optimization variant of SAT. Later, we will show how these formulas can be converted into KNF.

2.1.2 AtMostOne Constraints

AtMostOne constraints are a special class of cardinality constraints that are ubiquitous in SAT problems. In an evaluation of the approximately $5{,}000$ SAT Competition Anniversary benchmarks, we found that more than half contained AtMostOne constraints [91]. In around 750 problems, over half of the clauses in the problem come from AtMostOne encodings. These problems come from various application domains including planning, petri net concurrency, edge-matching, hidoku, and scheduling.

We also found that approximately one fifth of the benchmarks contained implicit ExactlyOne constraints [96]. These constraints often occur in problems that require the selection of a value. For example, in graph coloring problems each node in the graph should select exactly one color. These implicit exact-ones can also be found in FPGA-routing, pigeonhole, and petri net concurrency problems.

2.1.3 Maximum Satisfiability and AtLeastK Constraints

MaxSAT is the optimization variant of SAT where the goal is to find a solution that satisfies as many clauses as possible. There are several flavors of the MaxSAT problem, but for simplicity we consider unweighted partial MaxSAT. Each problem consists of a set of hard clauses and soft clauses, and soft clauses can be seen as units through a standard preprocessing procedure that adds new variables. A solution must satisfy all hard clauses, and an optimal solution will satisfy as many soft clauses as possible.

A MaxSAT problem can be encoded into a satisfiable and unsatisfiable formula by viewing the soft units as a cardinality constraint $s_1 + s_2 + \cdots + s_n \geq \mathit{opt}$, and either setting the bound to opt (satisfiable) or $\mathit{opt} + 1$ (unsatisfiable). If the optimal value is not known, the solver can be executed on successive formulas from $\mathit{opt} = 0$ then incrementing the bound by one until the formula becomes unsatisfiable, or from $\mathit{opt} = n$ then decrementing the bound by one until the formula becomes satisfiable.

Therefore, any MaxSAT problem can also be encoded as a SAT problem (or series of SAT problems) by using cardinality constraints. Optimization problems in SAT come from a wide variety of domains, for example, in hardware synthesis optimizing the number of transistors and quantum circuit synthesis optimizing the number of swap gates, in scheduling with the nurse rostering problem optimally assigning nurses to shifts, and in planning with robot warehouse path planning optimizing the number of time steps to move objects without collisions. There are many more examples of scheduling and planning problems coming from MaxSAT studies and the general field of operations research that can be solved with SAT [31, 66, 67].

2.1.4 Conditional AtLeastK Constraints

Conditional cardinality constraints present a small extension to KNF and can enable more natural encodings for certain problems. A conditional cardinality constraint has the form $c \implies \ell_1 + \ell_2 + \cdots + \ell_r \geq k$. If the conditional literal c is `true`, the cardinality constraint is evaluated as normal, and if c is `false`, the constraint is automatically evaluated to `true`.

Example 2.1

Consider neural network verification of a binarized neural network (BNN). A node may be activated (with activation symbol c) when at least k of its inputs $\ell_1, \ell_2, \ldots, \ell_r$ are true. This can be written as the conditional cardinality constraints $\overline{c} \implies \ell_1 + \ell_2 + \cdots + \ell_r < k$ and $c \implies \ell_1 + \ell_2 + \cdots + \ell_r \geq k$. The variable c is true iff the node is activated.

Conditional cardinality can help encode mathematical discovery problems for example finding candidate solutions to the various point discrepancy problems [103], machine learning explainability discovering abductive and counterfactual explanations for k-nearest neighbors [16], neural network verification [60, 115], and SAT-based association rule mining [25].

2.2 Encodings

The most widely used approach for handling cardinality constraints in SAT is to encode the constraints into clauses then solve the CNF formula with a SAT solver. Researchers have developed several different types of encodings, often focusing on optimizing the size of the encoding (number of clauses and auxiliary variables) and the propagation power of the encoding.

Given a cardinality constraint $\ell_1 + \ell_2 + \cdots + \ell_r \geq k$, a clausal encoding is *consistent* if falsifying $r - k + 1$ literals always yields a conflict, and *arc-consistent* [47] if unit propagation also sets the remaining literals to `true` once $r - k$ are falsified. Most clausal encodings for cardinality constraints are arc-consistent, so other structural details and experimental evaluations are used to differentiate them. We denote the size of the cardinality constraint as its number of literals r. Some encodings are tuned for small cardinality constraints, while others are used mainly for large cardinality constraints.

2.2.1 Auxiliary Variables

Cardinality constraint encodings often introduce auxiliary variables that make the encodings smaller. Auxiliary variables also serve another purpose, allowing a solver to reason about groups of literals compactly. The theoretical importance of auxiliary variables is well-known. For instance, the pigeonhole and mutilated chessboard problems have exponentially sized resolution proofs [3, 50] but polynomially sized extended resolution (ER) proofs [35]. ER is a proof system that allows the introduction of new variables via definitions [110].

ER is difficult to use in practice, since at any given point there are infinitely many ways to introduce new variables. The most effective approach has been to find specific patterns of clauses and use them as candidates for bounded variable addition (BVA) [72].

Example 2.2

Given the formula:

$$(a \vee x_1 \vee x_2) \wedge (a \vee x_2 \vee x_3) \wedge (a \vee x_4) \wedge (b \vee x_1 \vee x_2) \wedge (b \vee x_2 \vee x_3) \wedge (b \vee x_4)$$

BVA detects the repeated pattern on a and b then introduces the auxiliary variable y to create the new, smaller formula:

$$(y \vee x_1 \vee x_2) \wedge (y \vee x_2 \vee x_3) \wedge (y \vee x_4) \wedge (\overline{y} \vee a) \wedge (\overline{y} \vee b)$$

In 2023 the standard BVA preprocessing technique was improved with heuristics for ordering patterns, leading to the structured BVA (SBVA) preprocessing technique [49] which was used by the winning solver in the 2023 SAT competition.

In the case of cardinality constraints, there is only one chance to introduce meaningful auxiliary variables, during the encoding process. BVA can only detect simple patterns and would not be able to reencode an entire AtLeastK cardinality constraint encoding. Furthermore, the auxiliary variables introduced in all but the Pairwise clausal encodings would prevent BVA from finding large patterns. Therefore, the task of creating useful auxiliary variables falls to the encoder.

2.2.2 AtMostOne Encodings

There are several types of encodings designed specifically for AtMostOne constraints. Unlike for general cardinality constraints, the Direct (Pairwise) encoding on AtMostOnes provides good propagation properties since it is constructed from binary clauses. Therefore, many of the Non-Pairwise AtMostOne encodings strike a balance between introducing new variables to divide up the constraint and applying the Pairwise encoding to the sub constraints.

The Pairwise encoding, described in the introduction, is commonly used for AtMostOne constraints, especially for smaller AtMostOnes. The Pairwise encoding is often the first choice for users less familiar with the more complex encodings and appears far more frequently than Non-Pairwise encodings in SAT competition benchmarks from the last two decades. While the Direct encoding is almost never used for general cardinality constraints, the AtMostOne variant's simplicity, nice propagation properties, and lack of auxiliary variables can make it a suitable choice especially for satisfiable problems.

The Linear encoding (an instance of the Commander encoding [62]) for AMO $(\ell_1, \ldots, \ell_r)$ uses the Pairwise encoding for $r \leq 4$ and splits on $r > 4$ using fresh auxiliary variables (y) according to the following recursion:

$$\mathsf{Linear}(\ell_1, \ldots, \ell_r) : \mathsf{Pairwise}(\ell_1, \ell_2, \ell_3, y) \wedge \mathsf{Linear}(\overline{y}, \ell_4, \ldots, \ell_r) \tag{2.1}$$

The encoding uses $(r-3)/2$ auxiliary variables and $3r-6$ clauses. The cutoff of 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. The encoding can be modified by adding the auxiliary variable to the end of the recursive call or by changing the size of the splits.

There are several more types of AtMostOne encodings including the Sequential Counter encoding [97], Ladder encoding [7] and the Two-Product encoding [81], as well as variants of the Commander encoding [62]. In practice, when dealing with unsatisfiable problems that contain large AtMostOnes it is best to use one of the Non-Pairwise encodings; and for satisfiable problems or problems containing smaller AtMostOnes the Pairwise encoding will often suffice. While the introduction of auxiliary variables is known to be beneficial to solvers for some problems, auxiliary variables may be detrimental in the satisfiable case, possibly confusing some solver inprocessing techniques like stochastic local search for phase-saving.

Group reasoning is affected by the choice of encoding and the order of literals within the encoding. For example, if we shuffle literals then apply the Linear encoding, we could get two separate groups, and these groups will change how the solver can reason.

Example 2.3

$$\mathsf{Linear}(\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6) : \mathsf{Pairwise}(\ell_1, \ell_2, \ell_3, y) \wedge \mathsf{Pairwise}(\ell_4, \ell_5, \ell_6, y)$$

$$\mathsf{Linear}(\ell_1, \ell_3, \ell_5, \ell_2, \ell_4, \ell_6) : \mathsf{Pairwise}(\ell_1, \ell_3, \ell_5, y) \wedge \mathsf{Pairwise}(\ell_2, \ell_4, \ell_6, y)$$

The change in literal ordering modifies the meaning of auxiliary variable y from representing the groups $\{1, 2, 3\}, \{4, 5, 6\}$ to representing the groups $\{1, 3, 5\}, \{2, 4, 6\}$.

2.2.3 AtLeastK Encodings

General cardinality constraints, with the exception of AtMostOne and AtMostTwo, are almost never encoded directly because the direct encodings uses too large a number of clauses. Most encodings support both an AtLeastK and AtMostK variant with slight modifications, but it is always possible to transform an AtLeastK constraint into an AtMostK constraint and vice versa. Simply negate the literals in the constraint, then compute the new bound as the size of the constraint minus the old bound. It is often helpful to consider the version of a constraint (AtLeastK or AtMostK) that creates the smallest bound since the bound impacts the size of the encoding.

Several widely used encodings are provided in the Python module PySAT [58], including the Totalizer [12], Mod-Totalizer [84], KMod-Totalizer [78], Sorting Network [17], Cardinality Network [10], and Sequential Counter [97]. Each of these encodings are arc-consistent. They differ in how they recursively divide the constraint, affecting the number of clauses and auxiliary variables in the encoding, as well as the meaning of the auxiliary variables. There are several literature reviews evaluating the performance of different encodings [24, 73, 82], showing that different encodings perform best on different types of problems.

In general, the recursive structure of the encoding can be viewed as a set of intermediary levels, with auxiliary variables at each level that summarize information from the prior levels. As a representative example, we will consider the Totalizer. The Totalizer is a highly effective encoding and is used by most state-of-the-art MaxSAT solvers.

The Totalizer forms a tree with data literals at the leaves. Data literals are the literals from the original cardinality constraint. Each node can be thought of as a merge unit, outputting the

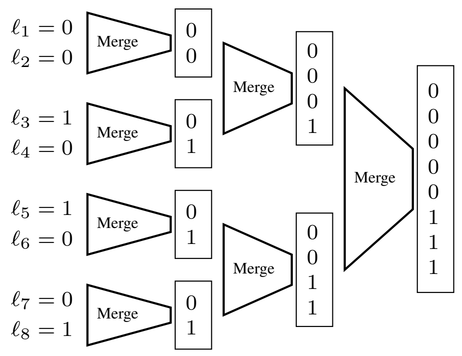


Figure 2.1: Visual representation of a Totalizer on 8 inputs.

sorted number of true input literals, as shown in Figure 2.1. The bound on the input cardinality constraint is enforced by constraining the auxiliary variables at the root. The auxiliary variables o_i refer to the output of the final merge unit at the root of the tree. An output auxiliary variable o_i can be set to `true` to ensure there are at least i `true` data literals or set to `false` ensuring there are at most $i - 1$ `true` data literals. Notably, unlike the sorting or cardinality networks, the merge nodes in the Totalizer do not introduce intermediary auxiliary variables and instead use clauses to tie the input literals to the output literals. This structural property is important because it makes the meaning of the auxiliary variables more clear, and this can be leveraged by our improved encodings and parallel cardinality solving techniques.

AtLeastK encodings can be modified for conditional cardinality constraints by equipping each clause in the encoding with the condition literal. In most cases this will impinge the arc-consistency of the encoding, but through small modifications the Pigeon-Hole encoding, Sorting Network, and Sequential Counter can regain arc-consistency [25].

As is the case for AtMostOnes, the meaning of auxiliary variables in AtLeastKs can be modified by changing the ordering of data literals in the cardinality constraint. For AtLeastKs this can have a larger impact due to the hierarchical structure of the encoding, where auxiliary variables represent successively larger groups of the input literals (seen by the larger merge units in the Totalizer).

2.2.4 Maximum Satisfiability Algorithms

MaxSAT has been a testing bed for cardinality constraint research, with MaxSAT algorithms adapting cardinality constraints in many different ways. In the *linear* MaxSAT algorithm, hard clauses are input into an incremental SAT solver along with soft units encoded as one large cardinality constraints: $s_1 + s_2 + \cdots + s_n \geq b$. Assumptions are used to set the bound of the cardinality constraint, either starting from 0 and incrementing the bound between each execution until the problem becomes unsatisfiable, or starting from n and decrementing until the problem becomes satisfiable [46]. The Mod-Totalizer was developed in this context to handle MaxSAT problems with a large number of soft clauses [84], and its effectiveness translates to SAT problems with large cardinality constraints.

In the class of *core-based* MaxSAT algorithms, an incremental SAT solver is used to find successive unsatisfiable cores, each time updating the upper bound on the number of satisfied soft units [6]. An unsatisfiable core, or core, is an unsatisfiable subset of constraints from the formula. When a core is found the solver must add a new constraint stating at most one clause from the core may be falsified. For example, given a core with clauses $C_1, \ldots, C_d$, a solver can add selectors $(C_i \vee y_i)$ and a clausal encoding for $y_1 + \cdots + y_d \leq 1$ to allow successive executions to falsify at most one clause from the core. Incremental cardinality constraints were developed to handle the case when variables from previous cores appeared in a new core, requiring a modification of the underlying cardinality constraints [76]. This work led to a stronger understanding of encodings, finding that the way in which Totalizers were merged would impact runtime, and this was an impetus for our study of variable groupings in Totalizer encodings.

2.2.5 Native Encodings – KNF

There are several formats that extend CNF to include cardinality constraints [41, 66, 70, 98].

Our format, KNF, was built to coincide with native propagation support and therefore allows AtLeastK, and conditional AtLeastK cardinality constraints.

$$x_1 + x_2 + x_3 + \overline{x}_4 \geq 2 \qquad \texttt{k 2 x}_1\texttt{ x}_2\texttt{ x}_3\texttt{ -x}_4\texttt{ 0} \tag{2.2}$$

$$x_5 \implies x_1 + x_2 + x_3 + \overline{x}_4 \geq 2 \qquad \texttt{g 2 }\overline{\texttt{x}}_5\texttt{ x}_1\texttt{ x}_2\texttt{ x}_3\texttt{ -x}_4\texttt{ 0} \tag{2.3}$$

Note, the conditional literal is negated in the KNF format. This choice corresponds with the propagation algorithm: the conditional literal can only be propagated to `false`, satisfying the conditional cardinality constraint in the case that the cardinality constraint is falsified.

Since we do not support propagation on ExactlyK cardinality constraints, we force users to break the constraint into two AtLeastK constraints. Furthermore, we do not support pseudo-Boolean propagation and therefore do not allow duplicate literals. Such constraints can be handled in a processing stage.

2.3 Solving

There are two main ways to handle cardinality constraints during solving, either by encoding them into clauses and passing the CNF as input to a SAT solver or by passing the cardinality constraints directly as input to a solver with native cardinality support. The predominant SAT solving technique is conflict-driven clause learning (CDCL). Native cardinality support comes in a few forms as native propagation within the CDCL framework, stronger cardinality reasoning in preprocessing, or stronger cardinality reasoning during search. Each method has its own strengths, and when trying to solve a new problem containing cardinality constraints it is not always clear beforehand which approach will work best.

2.3.1 Conflict Driven Clause Learning

To evaluate the satisfiability of a formula, a CDCL solver [74] iteratively performs the following operations: First, the solver performs unit propagation and tests for a conflict. Two-literal watch pointers [79] 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 [21, 69], 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.

Modern CDCL solvers constantly switch between a SAT and UNSAT mode with differing heuristics [85]. Mainly these heuristics change how often the solver restarts and how variable decisions are made, and they are generally tuned for satisfiable or unsatisfiable instances. The solver spends approximately the same time in each mode, with the amount of time before a mode switch increasing as solving progresses.

Clause learning is based on the resolution rule. Given two clauses C_1 and C_2 with $x \in C_1$ and $\overline{x} \in C_2$, *resolution* on the pivot x will produce the resolvent clause $(C_1 \setminus \{x\} \vee C_2 \setminus \{\overline{x}\})$.

Additionally, modern solvers incorporate pre- and inprocessing techniques that change the formula in some way, usually reducing the number of variables and clauses or removing literals from clauses. Some of the most useful inprocessing techniques are bounded variable elimination (BVE) [39], clause vivification [68], failed literal probing [45], equivalent literal substitution (ELS), and stochastic local search (SLS) for phase-saving [23]. Until the last few years, most competitive SAT solvers used resolution-based reasoning techniques for adding new clauses to the formula. This changed recently with the success of stronger proof systems like propagation redundancy (PR) and extended resolution (ER) most notably used in SBVA [49].

2.3.2 Cardinality Reasoning

Cardinality Propagation

Native Cardinality Propagation enables direct inference on a cardinality constraint. Given $\ell_1 + \ell_2 + \ell_3 + \ell_4 \geq 2$ and $\ell_2 = \ell_3 = 0$, the solver can immediately propagate $\ell_1 = \ell_4 = 1$. Unit propagation in CNF proceeds more slowly, propagating at most one literal from a clause at a time. Native propagation is implemented efficiently using additional watch pointers. The solver MINICARD implements native propagation but does not incorporate important inprocessing techniques or modern CDCL heuristics. Results from experimental evaluations showed that the native propagation in MINICARD is only effective on a limited set of satisfiable formulas [70].

Dynamic encoding is a framework for encoding only part of a cardinality constraint and propagating natively on the unencoded portion. During solving, different metrics like constraint activity can be used to guide which constraints should be encoded further [1]. This study helped highlight the potential gains from encoding only some cardinality constraints some of the time, but the technique was never adapted to modern tools. It combined separate propagation engines, slowing the solver and hampering the benefits of native propagation.

Native propagation on conditional cardinalities and variants have been explored in works specifically targeted at neural network verification, either as an extension of MiniSAT [60] or an extension of cryptoMiniSAT [115]. These implementations did not fully incorporate the conditional cardinality constraints into the modern SAT solver ecosystem; for example, not supporting a local search algorithm with cardinality constraints that could be used for phase-saving.

Cardinality Resolution

Generalized Resolution extends clausal resolution to PB (cardinality constraints with coefficients) [57] and is an instance of the *Cutting Planes* proof system [36]. We can consider a restricted case of generalized resolution on two cardinality constraints $\ell_1 + \ell_2 + \cdots + \ell_r + x \geq k$, and $j_1 + j_2 + \cdots + j_p + \overline{x} \geq b$ that do not share any literals in common. Generalized resolution on x would produce $\ell_1 + \ell_2 + \cdots + \ell_r + j_1 + j_2 + \cdots + j_p \geq k + b$, and this constraint could be further simplified by removing sets of opposing literals and decrementing the bound by one for each such set. The restriction on shared literals ensures that the resulting constraint is in fact a cardinality constraint; otherwise, the output could be a PB constraint (with coefficients greater than one).

This restricted form of cutting planes can be implemented in a preprocessing procedure that uses an algorithm like Fourier-Motzkin Elimination (FME) with the requirement that resolved cardinality constraints do not share literals. However, most tools that implement cardinality reasoning support operations on general PB constraints.

Cutting Planes

In addition to generalized resolution, cutting planes includes the *division rule*. Given the PB constraint with coefficients a_i, $\sum a_i \ell_i \geq k$, the division rule allows the inference of $\sum \frac{a_i}{c} \ell_i \geq \lceil \frac{k}{c} \rceil$ for a positive integer c that divides all of the coefficients a_i.

These inference rules are provably stronger than resolution-based rules, meaning that there exist some problems that require exponentially sized resolution proofs but only polynomially sized cutting planes proofs. Several PB solvers implement cutting planes reasoning and these include RoundingSAT [42] and the PB-variant of Sat4j [19]. Even though cutting planes is stronger than the proof system underlying CDCL, these solvers have demonstrated success mostly on crafted instances. CDCL still achieves better performance for many important applications and on diverse sets of benchmarks. However, PB reasoning can also be useful when targeted at specific tasks. In recent SAT competitions some solvers used a symmetry-breaking preprocessor [5], and this tool justifies its symmetry breaking constraints using the cutting planes proof system.

In the KNF format we do not include PB constraints, so our solver only propagates natively on cardinality constraints. Cardinality constraint propagation can be implemented as a natural extension to the CDCL watched-literal and conflict analysis schemes with only minor changes to the underlining clause data structure. By making only small changes we can preserve the superior performance of CDCL SAT solvers on a wide range of problems while also improving performance on some problems containing cardinality constraints. On the other hand, general PB propagation requires more complex data structures and algorithms, and this could impinge the propagation speed and heuristics of a solver, potentially degrading performance most of the time only to improve a small class of problems. For a similar reason we do not support cutting planes reasoning. Even though the proof system is stronger, cutting planes is generally not helpful on the majority of problems a SAT solver will encounter. In other words, cardinality constraints strike the right balance between the amount of effort required to modify the solver, the positive impact on problems containing cardinality constraints, and the overhead incurred by the solver modifications. We do not however eliminate the possibility that future versions of our solver will support cardinality resolution. It is possible to support resolution on cardinality constraints without opening P(B)andora's box as long as we enforce the restriction that all resolvents are also cardinality constraints.

2.3.3 Extraction

Some problems (e.g. pigeon-hole) that are exponentially harder for resolution than for cutting planes [50] may appear either directly or as a subproblems in SAT benchmarks. The worst-case behavior of resolution can be avoided by first extracting cardinality constraints then applying a light-weight cardinality reasoning algorithm like FME. Even if extraction and FME could not solve the problem, the procedure might output useful binary and unit clauses to be included during the CDCL search.

Such an approach was explored by Biere et al. [22]. They developed a syntactic extraction approach for the Pairwise AtMostOne, the Two-Product encoding, and Direct AtMostTwo encoding, implemented in the solver LINGELING. They also developed a semantic approach for extracting AtLeastK constraints based on unit propagation, implemented in the solver RISS.

The extraction was only effective on a small set of encodings and often failed to detect AtMostOne encoding that used auxiliary variables. The purpose of the technique was to solve hard-for-resolution problems quickly, without ever engaging the CDCL search engine.

An alternative approach to extraction makes use of runtime information within a PB solver, where the solver would store and track building blocks of cardinality constraints during solving [43]. For example, the inference $(\ell_1 \land \ell_2 \land \ell_3) \implies \overline{\ell}_4$ could be seen as a building block for an at-most-3 constraint containing those literals. This technique was limited by the size of inferences it could process, with larger sizes requiring more building blocks to form a complete constraint. Furthermore, it was implemented in the context of a PB solving framework and therefore cannot be used as a preprocessing extraction engine. However, it provides and interesting path towards cardinality constraint extraction for constraints revealed during solving.

In some sense, BVA performs extraction by finding patterns of clauses and reencoding them with new variables. BVA can reencode the Pairwise encoding of AtMostOne constraints into an AtMostOne encoding with auxiliary variables. This is not a complete form of extraction because it detects only portions of the original cardinality constraint at a time and does not perform well on encodings with auxiliary variables. When reencoding AtMostOnes, we want to extract the entire cardinality constraint so that we can improve the encoding all at once.

Ideally, we would like an extractor to transform a problem from CNF to KNF. This would allow us to use any of our KNF reencoding or solving techniques. To do so, we would need an extractor that works as a preprocessor and detects complete cardinality constraints, so that the detected constraint clauses can be removed and replaced with the native cardinality constraint. None of the tools described above provide a robust mechanism for this sort of KNF transformation.

2.3.4 Proofs

SAT solving is often applied in settings that demand both fast solving and a high degree of trust in the results. Modern SAT solvers are highly optimized and thanks to the annual competitions are constantly evolving. This makes formal verification of the entire solver impractical and a potential roadblock to innovation. Instead, the SAT solving community has moved the locust of trust from the solver itself to the solver's output. To this end, SAT solvers generate *proof certificates* for unsatisfiable problems, and these proof certificates can be checked by external tools. Proofs serve multiple purposes: certifying the correctness of solver answers, supporting debugging during solver development, and providing insight into solver reasoning through proof mining.

CDCL solvers produce *clausal proofs* represented as a clause sequence, for example the sequence $F, C_1, C_2, \ldots, C_m$ 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.

Most inprocessing and clause learning techniques can be expressed in the Reverse Unit Propagation (RUP) proof system.

A clause C is RUP in a formula F if unit propagation on $F \wedge \neg C$ derives a conflict. The negation of a clause $C = \{\ell_1, \ldots, \ell_r\}$, written as $\neg C$, is the formula $(\overline{\ell}_1 \wedge \cdots \wedge \overline{\ell}_r)$.

Intuitively, if a formula F and the negation of a clause C are unsatisfiable, then C is logically implied by F and therefore can be added to F. The restriction that the unsatisfiability is determined through unit propagation ensures that the RUP check is polynomial.

RUP checks logical equivalence, meaning that adding or deleting a RUP clause from a formula will not change the satisfying assignments of the formula. Some techniques go beyond resolution and therefore need a stronger proof system. The Resolution Asymmetric Tautology (RAT) proof system captures extended resolution and can be used to express most common inprocessing techniques [59].

A RAT clause may not preserve logical equivalence, but it does preserve satisfiability equivalence, meaning if a problem is satisfiable then it will remain satisfiable after the addition or deletion of a RAT clause. A RAT clause C is redundant in formula F under some *repair* such that if a satisfying assignment is found for $F \wedge C$, it can be repaired (by flipping the value of at most one literal) to form a satisfying assignment of F.

RAT with deletions (DRAT) proof logging is the most widely supported format. Typically, a solver emits a DRAT proof, and this proof is *trimmed* by a tool like DRAT-TRIM [111]. Trimming removes irrelevant information and equips the proof with hints (LRAT) so that it can be checked in linear time by a formally verified DRAT checker such as CAKE-LPR [106].

Our solving techniques will produce proof steps using the DRAT proof system to match the standard logging of modern CDCL solvers [111]. This will allow us to use existing formally-verified proof checkers, such as CAKE-LPR [106].

PB solvers using the cutting planes proof systems use the checking tool VERIPB [100]. VERIPB was notoriously slow, with much larger proof checking overheads than its DRAT counterparts, but advancements in constraint propagation have begun to change the story [63].

Native cardinality propagation with CDCL-like clause learning does not require the power of the cutting planes proof system but cannot be represented directly in DRAT. It is possible to avoid the slowdown of a cutting planes checker by starting with a DRAT checker and only adding a restricted set of capabilities for handling cardinality constraints, allowing the checker to optimize for cardinality propagation [115].

2.4 A New Frontier

The current state of the research on cardinality constraints presents several challenges and opportunities among our three main topics: extraction, encoding, and solving.

For extraction, syntactic techniques cannot keep up with the changing landscape of non-pairwise AtMostOne encodings, and previous semantic techniques were not mature enough to work across a general set of benchmarks. This sets the stage for our new extraction framework that combines syntactic heuristic-driven detection and semantic verification to tackle all of the most common AtMostOne encodings. Furthermore, we develop a new syntactic detection procedure for discovering implicit and explicit ExactlyOne constraints. Extraction opens the door for using our KNF-based solving techniques on existing problems. We will show how the extraction and solving pipeline can be equipped with proof-logging to ensure a solver's end-result correctly corresponds to the original problem.

For encoding, much of the research has focused on the general structure of encodings (number of new variables and number of clauses), and how these encodings compare across different sets of benchmarks. We will turn the focus towards using auxiliary variables to perform group reasoning on data literals, showing that features from a given problem can inform how to organize data literals within an encoding. Furthermore, previous encoding techniques were agnostic to whether a problem contained multiple cardinality constraints. We will show that the encodings of multiple cardinality constraints can be aligned, allowing a solver to more effectively reason over multiple encodings at once.

For solving, native propagation has been relatively understudied due to its lack of success. We will show that a crucial component to the success of native propagation is the integration with modern inprocessing techniques and search heuristics. Furthermore, if KNF solving is to be incorporated into mainstream SAT solvers, it also needs proof-logging capabilities that balance ease-of-use and checking time.

The KNF format will also unlock the potential for parallel solving with cardinality constraints. We will show that by controlling the encoding, thereby specifying the meaning of auxiliary variables, we can select variables that create good splits on a formula.

These challenges are worth addressing and have already led to several collaborations. Our cardinality solver has been used experimentally in hardware synthesis improving the tool ABC on some benchmarks [27], mathematical discovery finding candidate solutions to the various point discrepancy problems [103], machine learning explainability discovering abductive and counterfactual explanations for k-nearest neighbors [16], and finding mathematical objects in Ramsey theory. In addition, there are several new problems arising in neural network verification, discrete geometry, and planning. The KNF format, along with a refined set of improved encoding and solving techniques, stands to make SAT more accessible, broadening its impact to new domains.

Chapter 3

Extraction

Extraction is the task of detecting clausal encodings of cardinality constraints in a CNF formula. For our purposes, extraction is a means for transforming a formula in CNF to a formula in KNF by replacing all extracted cardinality constraint encodings with their explicit KNF representation.

While our thesis explicitly describes ease-of-use and performance gains for problems already encoded into the KNF format, we developed an extraction tool as a way to access the thousands of benchmarks already encoded into CNF. A robust extraction tool will allow us to transform thousands of existing SAT competition benchmarks encoded in CNF into KNF. These formulas are important to the SAT community and represent problems from a wide range of domains that are challenging for modern solvers. If we could show performance gains on these existing benchmarks using our KNF solving techniques, this would promote the adoption of KNF and help lead us to a future where new users encode a problem first as KNF and extraction is no longer necessary.

Previous studies on extraction focused mainly on AtMostOne constraints. AtMostOnes are a good target for extraction because they 1) appear frequently in SAT problems, 2) have simpler encodings than general cardinality constraints, and 3) can be used in conjunction with basic cardinality reasoning procedures to solve problems that are otherwise hard for resolution. We discuss our approach to AtMostOne extraction in Section 3.2.

We also look at extraction for implicit and explicit ExactlyOne constraints. ExactlyOne constraints are often used when selecting a value for an object and can appear in a formula as unique literal clauses. A unique literal clause is a clause containing literals that appear only once in the formula. We can detect these clauses with a simple syntactic check. We will discuss ExactlyOne extraction in Section 3.3.

3.1 AtMostOne, ExactlyOne, and AtLeastK Encodings

In the second chapter we defined the Pairwise and Linear encodings for AtMostOne constraints, and the Totalizer for AtLeastK constraints. In this section, we will define the Sequential Counter encoding for AtLeastK constraints and ExactlyOne constraints. Next, we will define the Order encoding for ExactlyOne constraints. Our extraction algorithm will work on the majority of the PySAT encodings, which can also be used as AtLeastK encodings. We will provide definitions for each of these as well. We will use k to represent the bound of a cardinality constraint and r to represent the size (number of literals in the cardinality constraint).

Sequential Counter Encoding

$$\begin{tikzcd}
 & & & s_{4,4} = o_4 \\
 & & s_{3,3} \arrow[r] \arrow[ur] & s_{4,3} = o_3 \arrow[u] \\
 & s_{2,2} \arrow[r] \arrow[ur] & s_{3,2} \arrow[r] \arrow[u] \arrow[ur] & s_{4,2} = o_2 \arrow[u] \\
s_{1,1} \arrow[r] \arrow[ur] & s_{2,1} \arrow[r] \arrow[u] \arrow[ur] & s_{3,1} \arrow[r] \arrow[u] \arrow[ur] & s_{4,1} = o_1 \arrow[u] \\
\ell_1 \arrow[u] & \ell_2 \arrow[u] & \ell_3 \arrow[u] & \ell_4 \arrow[u]
\end{tikzcd}$$

Figure 3.1: Sequential Counter for four literals ℓ_1, ℓ_2, ℓ_3, ℓ_4.

The Sequential Counter, e.g., Figure 3.1, uses auxiliary variables to count the number of `true` data literals. Specifically, the auxiliary variables $s_{i,j}$ with $1 < i < r$ and with $1 < i < r$ form a grid where $s_{i,j}$ is `true` if at least j of the first i data literals are `true`. Literals in the last column, $s_{n,j}$, correspond to output literals o_j stating that j data literals are `true`. The bound k is enforced with the unit $\overline{o}_{k+1}$. The encoding is simplified by only using the first $k+1$ rows, in Figure 3.1 removing $s_{4,4}$, and by removing auxiliary variables $s_{i,j}$ if $i < k$ and $j > i$ (e.g., $s_{1,2}$). Auxiliary variables interact locally in the grid, e.g., $s_{i,j} \rightarrow s_{i+1,j}$ and $(s_{i-1,j} \wedge \ell_i) \rightarrow s_{i,j+1}$. The sequential counter is asymmetrical, with auxiliary variables on the left-hand side summarizing information from fewer data literals. For example, in Figure 3.1, a solver can reason about the pair ℓ_1, ℓ_2 via the auxiliary variables $s_{2,1}$ (at least one of the pair is `true`) and $s_{2,2}$ (both are `true`), but no two auxiliary variables allow similar reasoning about the pair ℓ_3, ℓ_4.

The Sequential Counter encoding can be defined for AtMostOne constraints using a single layer of auxiliary variables, allowing us to remove the row index and use s_i. To enforce that at most one of the literals is `true`, we add the single output literal. This will result in an encoding with only binary clauses. We can modify Sequential Counter to become an ExactlyOne constraint by making the definitions of the auxiliary variables bidirectional. So, for the first auxiliary variable we have that $s_1 \leftrightarrow \ell_1$, and for the remaining auxiliary variables we have that $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$. We keep the clauses $s_i \rightarrow s_{i+1}$ and add the clause $\ell_{r-1} \rightarrow s_r$.

Example 3.1

Consider the constraintExactlyOne (ℓ_1, ℓ_2, ℓ_3). The sequential counter encoding uses the following clauses:

$$\underbrace{(s_1 \vee \overline{\ell}_1) \wedge (\overline{s}_1 \vee \ell_1)}_{s_1 \leftrightarrow \ell_1} \wedge \underbrace{(\overline{s}_2 \vee s_1 \vee \ell_2) \wedge (s_2 \vee \overline{s}_1) \wedge (s_2 \vee \overline{\ell}_2)}_{s_i \leftrightarrow (s_{i-1} \vee \ell_i)} \wedge \underbrace{(\overline{s}_1 \vee \overline{\ell}_2) \wedge (\overline{s}_2 \vee \overline{\ell}_3)}_{\overline{s}_{i-1} \vee \overline{\ell}_i} \wedge \underbrace{(s_2 \vee \ell_3)}_{\overline{s}_{r-1} \rightarrow \ell_r}$$

Order Encoding

The Order encoding [104] uses order variables $o_{\leq i}$ with $1 < i < b$ to express if one of $\ell_1, \dots, \ell_b$ is `true`. Similar to the Sequential Counter, two order variables must be used to state that a data literal is `true`: ℓ_i is `true` when $o_{\leq i}$ is `true` and $o_{\leq i-1}$ is `false`. This is slightly different for ℓ_1, which is `true` iff $o_{\leq 1}$ is `true`. For an ExactlyOne encoding, ℓ_r is `true` iff $o_{\leq k+1}$ is `false`. The order encoding uses the following clauses $\overline{o}_{\leq i} \vee o_{\leq i+1}$ stating that if one of the first i literals is `true`, then one of the first $i+1$ is `true`.

Given an ExactlyOne constraint on the literals $\ell_1, \ell_2, \dots, \ell_r$, the Order encoding eliminates the var(ℓ_i) variables from the formula by replacing all their occurrences with $o_{\leq i}$ variables. Each literal is replaced in the following way:

- all literals ℓ_1, $\overline{\ell}_1$, ℓ_r, and $\overline{\ell}_r$ are replaced by $o_{\leq 1}$, $\overline{o}_{\leq 1}$, $\overline{o}_{\leq r-1}$, and $o_{\leq r-1}$, respectively.
- all literals $\overline{\ell}_i$ with $1 < i < k$ are replaced by $o_{\leq i-1} \vee \overline{o}_{\leq i}$.
- all clauses with literals ℓ_i with $1 < i < k$ are duplicated with one copy using $\overline{o}_{\leq i-1}$ and the other copy using $o_{\leq i}$ instead of ℓ_i.

We do not provide a mechanism for extracting the Order encoding but will revisit it when considering ways to reencode detected ExactlyOne constraints in Section 4.3.

Totalizer Encoding

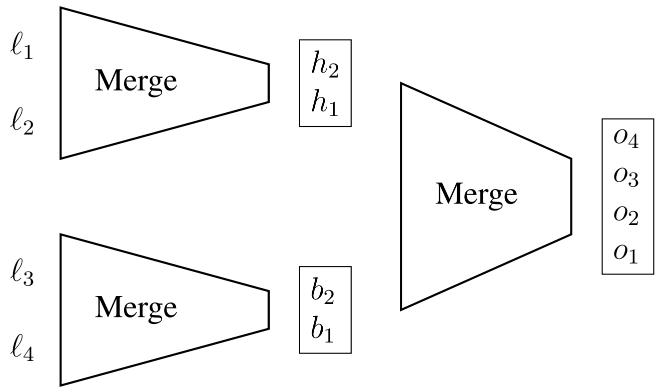


Figure 3.2: Totalizer for four literals ℓ_1, ℓ_2, ℓ_3, ℓ_4.

The Totalizer, e.g., Figure 3.2, uses a binary tree to incrementally count the number of true data literals at each level. In this visualization of the Totalizer the merge units output the explicit auxiliary variables. Data literals form the leaves, and each merge unit has auxiliary variables representing the unary count from the sum of its children counters. For example, in Figure 3.2, variable o_3 is `true` if either pairs h_1, b_2 or h_2, b_1 are `true`. The bound k is enforced by adding the unit $\overline{o}_{k+1}$. The modulo Totalizer (Mod-Totalizer) [84] uses a quotient and remainder at each merge unit to reduce the number of auxiliary variables required to count the sum. The encoding can be simplified by only encoding the count up to $k+1$ at each merge unit (KMod-Totalizer) [78]. Unlike the Sequential Counter, the Totalizer splits the subproblems symmetrically, so a solver can reason about ℓ_3 and ℓ_4 via the auxiliary variables b_1 (at least one of the pair is `true`) and b_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, ℓ_1 and ℓ_4 do not share a counter until the root, two levels of abstraction away from the data literals. Furthermore, a merge unit's counters always capture all of its descendant data literals, e.g., the output literal o_1 means at least one of all of the data literals is `true`.

Sorting and Cardinality Network Encodings

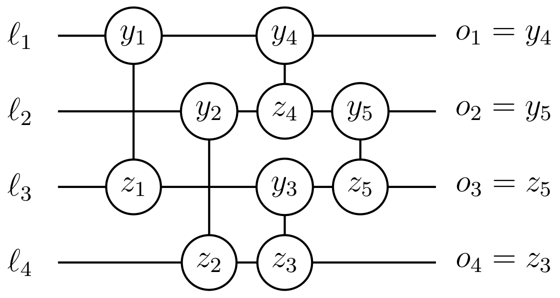


Figure 3.3: Sorting Network for four literals ℓ_1, ℓ_2, ℓ_3, ℓ_4.

The Sorting Network, e.g., Figure 3.3, and Cardinality Network share a similar design, using networks that take the data literals as input and, through a series of swaps, output the count in sorted order. This is in contrast to the Totalizer which does not use swaps or intermediary auxiliary variables within each merge unit. 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`. The bound is enforced by making $\overline{o}_{k+1}$ unit. 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 $k+1$ bits. Both networks are implemented hierarchically by dividing the sorting into subproblems 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. These networks are symmetrical, with the grouping of literals in the subnetworks determined by their order.

3.2 AtMostOne Extraction

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. *From clauses to klauses*. In *Computer Aided Verification (CAV)*. (2024). [91]

The most prominent work on extracting AtMostOne encodings comes from Biere et al. [22]. They devised a syntactic approach for detecting the Pairwise, Nested, and Two-Product AtMostOne encodings as well as the direct AtMostTwo encoding. Separately, they developed a semantic approach based on unit propagation for detecting AtLeastK constraints. The semantic detection uses unit propagation to iteratively extend arc-consistent AtMostK constraints. This

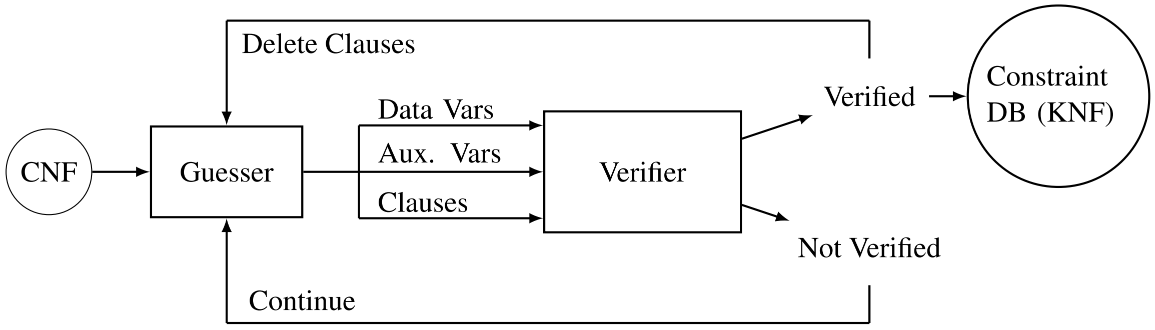


Figure 3.4: Guess-and-verify framework for transforming a formula in CNF into KNF.

procedure depends on the sequence of variables picked for extending the current constraint and can be misled by auxiliary variables to produce truncated constraints. Both syntactic and semantic configurations support a merging operation for finding nested AtMostOne constraints. The merging operation resolves two AtMostOnes to remove auxiliary variables. For example, resolving $-y + \ell_1 + \ell_2 + \cdots + \ell_r \geq k$ and $y + j_1 + j_2 + \cdots + j_p \geq b$ on the opposing literal y will produce the AtMostOne $\ell_1 + \ell_2 + \cdots + \ell_r + j_1 + j_2 + \cdots + j_p \geq k + b - 1$. Notably, the merged constraint is logically implied but does not justify the deletion of y from the formula. Additional bookkeeping would be required to enable a full CNF to KNF transformation that involved deleting and replacing clausal encodings.

They implemented the extraction procedures inside the SAT solvers LINGELING and RISS in combination with the Fourier-Motzkin elimination (FME) algorithm to either find a quick refutation or export any learned binary or unit clauses to the CDCL engine. These extraction techniques were also used in SAT4J, and extracted constraints were used by the PB variant of the solver which could apply either native propagation or generalized resolution. The syntactic extraction works well for simple AtMostOne encodings but fails to extract the majority of PySAT encodings, and the semantic extraction also underperforms on these encodings (see Table 3.1).

Another approach is to detect constraints during solving. Over time a solver may learn clauses that were previously missing from a possible encoding, or it may delete variables that were obfuscating encodings. Furthermore, during conflict analysis a solver can log relationships between variables, for example that $(\ell_1 \land \ell_2 \land \ell_3) \implies \overline{\ell}_4$ which could be a building block for an AtMostThree constraint containing those literals. Such an approach was studied in the context of PB solving [43], but the success of extraction relied heavily on the decision heuristics of the solver and in finding the full set of building blocks for an encoding.

With the advent of SAT encoding tools such as PySAT, users can, with little effort, move away from simple and often suboptimal AtMostOne encodings towards more complex encodings without the fear of introducing defects. As such, more complex AtMostOne encodings have been appearing in the wild, and many rest beyond the capabilities of the aforementioned extraction tools, opening the door for our new extraction framework.

We developed a novel extraction tool using a guess-and-verify (G&V) framework, shown in Figure 3.4. The guesser detects a candidate cardinality constraint consisting of a set of clauses, data variables, and auxiliary variables. Then, the verifier uses some semantic check to determine if the candidate represents an valid cardinality constraint. We describe how this framework was successively applied to Non-Pairwise AtMostOne encodings in Section 3.2.2, but first we look at techniques for extracting Pairwise AtMostOnes .

3.2.1 Syntactic Pairwise AtMostOne Extraction

Pairwise AtMostOnes can be extracted syntactically without using the G&V framework. Consider the variable incidence graph on binary clauses (VIG$_{\text{bin}}$). Each node in the VIG$_{\text{bin}}$ represents a literal in the formula, and undirected edges are added between two nodes if their corresponding literals occur together in a binary clause. Detecting a Pairwise AtMostOne encoding is as simple as finding a clique in the VIG$_{\text{bin}}$. Indeed, for any clique Q all clauses of the form $(\ell_i \vee \ell_j)$ for $edge(\ell_i, \ell_j) \in Q$ must be in the formula by the definition of the VIG$_{\text{bin}}$, and this is exactly the Pairwise AtMostOne encoding on the negated literals in the clique.

Finding the max clique of a graph is NP-complete. One way to get around this problem's complexity is by using a greedy algorithm. A greedy approach to clique finding will start with a single literal in the clique. Then, in each iteration it will then try and find a literal that occurs in binary clauses with all other literals in the clique, and if so add this new literal to the clique. The clique expansion ends when no new literals are found that can be added to the clique. When a clique is found the binary clauses forming the clique are removed and clique expansion starts again with another literal. This was the approach we used in [91], but it risks finding suboptimal cliques (see Example 3.2).

Example 3.2

Consider the formula generated from the Pairwise encoding of the following constraints:

$$\mathsf{AMO}(\ell_1, \ell_2, \ell_3, \ell_4, \ell_5) \wedge \mathsf{AMO}(\ell_1, \ell_6) \wedge \mathsf{AMO}(\ell_2, \ell_3, \ell_6)$$

A greedy clique finding algorithm starting from ℓ_1 may first expand the clique with ℓ_6. It can then expand the clique with ℓ_2 using the binary clause $(\overline{\ell}_1 \vee \overline{\ell}_2)$ from the first AtMostOne and the binary clause $(\overline{\ell}_2 \vee \overline{\ell}_6)$ from the third AtMostOne. It can expand the clique again with ℓ_3. Removing the clauses for the detected constraint $\mathsf{AMO}(\ell_1, \ell_2, \ell_3, \ell_6)$ will prevent the first AtMostOne from being fully detected. Either $\mathsf{AMO}(\ell_1, \ell_4, \ell_5)$ or $\mathsf{AMO}(\ell_2, \ell_3, \ell_4, \ell_5)$ may be extracted, and if the greedy algorithm picks suboptimal expansions again it may choose the smaller of the two AtMostOnes .

We found that for many benchmarks the greedy approach worked well. We added an additional heuristic that always picked variables for an expansion in the natural order. The natural order is defined by the variables' IDs. In Example 3.2, this would involve expanding the clique with ℓ_2 first since it is ordered before ℓ_6. This heuristic is not robust against variable shuffling and provides no theoretical guarantees. Alternatively, one could use an off-the-shelf max clique solver. In each iteration the detected max clique is stored as an AtMostOne then removed from the graph. This continues until no cliques above a desired size threshold remain. While this approach is more rigorous and in practice max clique solvers are quite fast, we did not find the additional computation necessary for finding large AtMostOnes on competition benchmarks.

3.2.2 Semantic Non-pairwise AtMostOne Extraction

Non-Pairwise AtMostOnes are extracted using our G&V framework. The guesser first classifies variables into two sets. If variables occur only positively or only negatively in binary clauses they are marked as data variables, and if they occur both positively and negatively in binary clauses they are marked as auxiliary variables. Then, the guesser tries to build up AtMostOne encodings starting from the auxiliary variables.

The process starts by picking a single auxiliary variable to form the initial set. The guesser expands the set by finding the transitive closure on auxiliary variables in clauses that contain other auxiliary variables from the set. Once this expansion is finished, the candidate encoding is defined as all clauses containing auxiliary variables from the set, and the data variables are all non-auxiliary variables occurring in these clauses. Example 3.3 shows how the guesser would process a Sequential Counter encoding.

Example 3.3

Consider the Sequential Counter encoding on the AtMostOne $\ell_1 + \ell_2 + \ell_3 + \ell_4 + \ell_5 \leq 1$:

$$(\bar{\ell}_1 \vee s_1) \wedge (\bar{\ell}_2 \vee s_2) \wedge (\bar{\ell}_3 \vee s_3) \wedge (\bar{\ell}_4 \vee s_4) \wedge (\overline{s}_1 \vee s_2) \wedge (\overline{s}_2 \vee s_3) \wedge (\overline{s}_3 \vee s_4) \wedge$$
$$(\bar{\ell}_2 \vee \overline{s}_1) \wedge (\bar{\ell}_3 \vee \overline{s}_2) \wedge (\bar{\ell}_4 \vee \overline{s}_3) \wedge (\bar{\ell}_5 \vee \overline{s}_4)$$

The data variables (ℓ_i's) appear only negatively in binary clauses while the auxiliary variables (s_i's) appear both positively and negatively. Assume the starting auxiliary variable is s_1. The closure procedure first adds s_2 from the clause $(\overline{s}_1 \vee s_2)$, then adds s_3 from the clause $(\overline{s}_2 \vee s_3)$, and finally s_4 from the clause $(\overline{s}_3 \vee s_4)$. The candidate encoding is all of the clauses touched by the auxiliary variables s_1, s_2, s_3, or s_4, and the data variables are all non-auxiliary variables occurring in these clauses.

There are a few potential drawbacks to the heuristics used by the guesser. It can fail to detect AtMostOne constraints when a data variable appears in both polarities, but by limiting the classification to binary clauses we mitigate this possibility. Furthermore, if an encoding is not simplified, meaning it contains unit or pure auxiliary literals, these will be misclassified as data literals. This can be avoided by running unit propagation and pure literal elimination on the formula. It can also detect spurious AtMostOne candidates from sets of binary clauses that do not represent AtMostOne encodings. While misclassification can increase the runtime of the extraction tool, it is not a serious threat to soundness because all candidate AtMostOne constraints will be passed through the verifier.

3.2.3 BDD Verification

The verifier takes as input the candidate constraint – a set of clauses with labelled auxiliary variables (Y) and data variables (X) – and returns a `verified` or `not verified` result as to whether the candidate is in fact a cardinality constraint. In the case of a `verified` result, the verifier will also produce the full cardinality constraint giving the polarities of data literals and the upper and lower bounds on the constraint.

Internally, the verifier takes the input clause set and constructs an Ordered Binary Decision Diagram (BDD) [29] through conjunction operations on the clauses and quantifications to eliminate the auxiliary variables. The resulting BDD, which we refer to as the function $f(X)$, can be characterized as a cardinality constraint if it contains the correct layered structure [2, 30].

The procedure we use to construct the BDD is *bucket elimination* [38, 86], a systematic way to perform conjunctions and quantifications. Bucket elimination is guided by a total ordering of the data and auxiliary variables, which defines both the ordering for the BDDs and 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 $f(X)$.

The variable ordering is crucial when dealing with BDD. The ordering directly impacts the memory footprint and runtime of the bucket elimination procedure. Most AtMostOne encodings form a linear (instead of hierarchical) structure, motivating the following approach for automatically generating a variable ordering:

1. Build an undirected graph over auxiliary variables with edges of weight 1.0 between two nodes if the two auxiliary variables share a clause and edges of weight 0.75 if two auxiliary variables share a clause with the same data variable.

2. Find two auxiliary variables forming the endpoints (start and end) of the graph. 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.

3. Order auxiliary variables first by their distance from the start then by their distance from the end. Data variables are inserted into the ordering adjacent to the first auxiliary variable they co-occur in a clause with.

Conceptually, this ordering *stretches* the AtMostOne encoding from the two end points, linearizing the internal auxiliary variables. For a layered graph, this approach will tend to find opposite corners as endpoints and generate a layered ordering of 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. Specifically, we can obtain both the polarities of data variables as they occur in the cardinality constraint (which we denote as the literals $\{\ell_1, \ell_2, \ldots, \ell_r\}$) and the lower ($L$) and upper ($R$) bounds on the cardinality constraint:

$$L \leq \ell_1 + \ell_2 + \cdots + \ell_r \leq H$$

For a BDD representing a one-sided cardinality constraint either $L = 0$ or $R = r$. For an AtMostOne constraint $L = 0$ and $H = 1$, and for a clause $L = 1$ and $H = N$. If the cardinality constraint is one-sided and $L = 0$ we can extract the AtLeastK constraints $\overline{\ell}_1 + \overline{\ell}_2 + \cdots + \overline{\ell}_r \geq r - H$. If the cardinality constraint is two-sided, we can extract the two AtLeastK constraints $\ell_1 + \ell_2 + \cdots + \ell_r \geq L$ and $\overline{\ell}_1 + \overline{\ell}_2 + \cdots + \overline{\ell}_r \geq r - H$.

We use the layered structure of cardinality constraints represented in BDDs [2, 30] to determine the two bounds and the phase of each variable. 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 $1 \leq i \leq r + 1$
- j satisfies $0 \leq j \leq i - 1$
- There must be some value k such that $0 \leq k \leq r - i + 1$ and $L \leq j + k \leq H$.

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 $\ell_i = \overline{x}_i$, 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. If at any point the structure does not match that of a cardinality BDD the verifier exits returning the result `not verified`.

3.2.4 AtMostOne Extraction on PySAT Encodings

The PySAT encoding library serves a good baseline for comparing different extraction tools since it implements many of the most-used AtMostOne encodings. Our G&V tool was compared against the syntactic detection in LINGELING and semantic detection in RISS [22].

Table 3.1: Detecting size 10 PySAT AtMostOne encodings: Pairwise, Sequential Counter, Cardinality Network, Sorting Network, Totalizer, Mod-Totalizer, KMod-Totalizer, Bitwise, and Ladder. Shown are the number of data variables/auxiliary variables in the largest AtMostOne detected. $10/0$ represents the full constraint on all of the data variables.

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

For the purpose of this evaluation we modified LINGELING and RISS to run cardinality detection, print the detected constraints, then exit. Neither solver provides command line options for extraction or the functionality to output a new formula with the extracted constraints added and clausal encodings removed. Each formula contained the PySAT encoding of an AtMostOne constraint on ten literals as well as an AtLeastOne constraint on those literals. Without the AtLeastOne constraint, the data variables would appear only negatively in the encodings. This would likely not happen in practice since those literals could be removed with pure literal elimination. We also performed unit propagation and pure literal elimination on the encodings. Units and pure literals could be avoided by refining the encoding program, but PySAT does not make these optimizations.

The results in Table 3.1 show a stark contrast between G&V and the other extraction tools. Our G&V tool found the the original AtMostOne constraints for seven of the nine PySAT encodings and found the core of the original AtMostOne constraint for the Ladder encoding (missing a single data variable). The other tools found small nested AtMostOne constraints of sizes 3-6, but they could not find the core of the AtMostOne constraint for any encoding other than Pairwise. The semantic detection in RISS also detects some AtMostTwo constraints for each of the encodings, but the AtMostTwos do not use a majority of the original data variables. The merge operation generates some additional constraints of size 4-6, but as mentioned previously this is less useful in our setting as it does not permit the deletion of the smaller constraints used in the merge.

PySAT does not contain the Linear encoding or the Two-Product encoding, both of which LINGELING and RISS can extract. Indeed, most of the PySAT encodings are more widely used for AtLeastK instead of AtMostOne constraints. However, the sequential counter is a common AtMostOne and AtLeastK encoding and only the G&V tool can extract it. Furthermore, the robust performance of G&V across differing encodings increases confidence in the tool's ability to extract unknown encodings from new problems.

3.3 ExactlyOne Extraction with Unique Literal Clauses and Exclusive Literal Clauses

Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule. *Reencoding Unique Literal Clauses*. In *Theory and Applications of Satisfiability Testing (SAT)*. (2025). [96]

A constraint that occurs commonly in SAT problems is the selection of an object from a set. For example, in a graph coloring problem each node will select one color from a set of k colors. A selection constraint can be thought of as an ExactlyOne constraint, stating that exactly one of the objects from the set will be chosen in a solution.

An ExactlyOne constraint can be encoded explicitly as the combination of an AtMostOne and AtLeastOne constraint (a clause). To extract this type of encoding, we could use the extractor from the previous section to find an AtMostOne constraint, then check if the clause corresponding to the literals in the AtMostOne constraint appears in the formula.

In certain cases, an ExactlyOne constraint can also be encoded implicitly by foregoing the

AtMostOne encoding. This can happen when the literals in the ExactlyOne constraint do not appear in the rest of the formula, therefore appearing only in the single clause defined by the ExactlyOne constraint. We call these clauses *unique literal clauses* (ULCs).

Example 3.4

Consider a graph-coloring problem with k colors and n nodes. The problem contains two sets of constraints:

1. Each node is assigned a color
2. Adjacent nodes cannot have the same color

For each node x, the constraint stating that x has a color is encoded directly as $(x_1 \vee \ldots \vee x_k)$, where x_i is `true` if x has color i. We can add the additional constraint AMO $(x_1, \ldots, x_k)$ making it an explicit exactly one constraint, but this is redundant. To see this, consider the formula without the AtMostOne constraint. If there exists a solution to the formula where a node is assigned more than one color, then each of these colors must also satisfy the constraints that adjacent nodes cannot have the same color. Therefore, we can simply select one of the assigned colors for the node and it will preserve the graph coloring.

In this section, we discuss an extraction technique guided by ULCs. Once we detect the ULCs, we can add the corresponding AtMostOne to the formula either natively in KNF or encoded in CNF. However, we have found that it is beneficial to perform additional processing on the ULCs before converting them to KNF. We may detect multiple sets of ULCs that share variables, which we will call *clashing* ULCs. We can resolve clashing ULCs on shared variables to form larger ULCs. This is desirable because larger cardinality constraints benefit more from the various KNF solving techniques.

The closest related work on extracting ExactlyOnes is by Manthey and Steinke [71]. They detected ExactlyOnes that used the Pairwise encoding for the AtMostOne constraint, then reencoded them with the Sequential Counter. Their detection only worked for explicit ExactlyOnes but missed implicit ExactlyOnes represented by ULCs.

3.3.1 Unique Literal Clauses (ULCs)

A clause C is a *unique literal clause (ULC)* in a formula F if none of the literals in C occur in $F \setminus \{C\}$. We can compute the set of ULCs in a formula F in linear time in the size of F by first computing which literals occur only once in F and then determining which clauses only contain such literals.

In general, a clause over a set of objects says at least one of those objects is `true`. However, since the literals in a ULC are unique, it is satisfiability equivalent to say exactly one of those objects is `true`.

To see this, consider an assignment τ that satisfies some formula F. Assume a clause $C \in F$ is a ULC. The assignment τ must satisfy at least one literal in C since it satisfies F. If τ satisfies more than one literal in C, we can flip all but one of the literals to `false`. Since they are unique, flipping them will not falsify any clause in F. Therefore, any satisfying assignment can be modified to satisfy exactly one literal in the ULC. Note, this is not solution preserving because it forbids assignments that satisfy multiple objects from the ULC at once.

3.3.2 Clashing ULCs

Ideally, we want to detect as many large ULCs as possible, as this will present more options for the KNF solving techniques. This goal can be hampered if variables are shared among ULCs. If two ULCs share a variable, we can only add the AtMostOne constraint for one of the ULCs since adding the AtMostOne would make the shared variable no longer unique. Furthermore, shared variables may be auxiliary variables used to split selection constraints in a similar way as the Linear AtMostOne encoding. We would like to remove these variables and recover the original, larger selection constraint to maximize the effect of extraction. We can address these problems by handling ULC clashes with variable elimination.

Two ULCs C and D *clash* on a literal ℓ if there exist $\ell \in C$ and $\overline{\ell} \in D$. We also say that they are a clashing pair on ℓ. If none of the ULCs in a formula clash, we say the formula is *clash-free*.

The clash-free property is important because it will allow each ULC to be transformed into an ExactlyOne constraint. To see this, consider an assignment τ that satisfies some formula F. For each ULC in F we flip all but one satisfied literal to `false`. Since there are no clashes, flipping these literals will not falsify any other ULCs. And since the literals are unique, flipping them will not falsify any other clauses in F.

Resolving two clashing ULCs will either results in a tautology (if there are multiple clashing literals) or a new ULC. So, if a formula has a pair of clashing ULCs, we can either remove the pair (if the resolvent is a tautology) or replace the pair with a new ULC by eliminating the clashing variable.

Any formula can be made clash-free by eliminating all pairs of clashing ULCs via variable elimination, and this procedure is confluent so the order of the eliminations is not important. To see this, consider a formula F that contains clashing ULCs. First, we form partitions of the clashing ULCs in F such that i) for each partition each ULC does not clash with any ULC in another partition and ii) each ULC in a partition clashes with at least one other ULC in the partition. Then, we perform elimination on the clashing pairs of each partition. If a partition with n ULCs contains more than $n-1$ clashing pairs, variable elimination will remove all of the ULCs in the partition. If a partition with n ULCs contains exactly $n-1$ clashing pairs, variable elimination will result in a single new ULC, and this ULC will not share variables with any of the other partitions and therefore cannot introduce a new clashing pair.

3.3.3 Exclusive Literal Clauses (XLCs)

We discussed earlier that explicit ExactlyOne constraints are encoded using an AtMostOne constraint and a clause. We can combine the concept of explicit ExactlyOne constraints with unique literals to generalize ULCs with exclusive literal clauses (XLCs).

An clause C is an XLC if the literals in C can be partitioned into two sets U and X such that every literal in U is unique and for every pair of literals $\ell, \ell' \in X$ the binary clause $(\overline{\ell} \vee \overline{\ell}')$ appears in F. We could generalize this definition further by stating that the literals in X appear as an AtMostOne constraint in F, but this would complicate the extraction.

So, a ULC is an XLC in which all literals are unique. In our experimental evaluation it is convenient to consider XLCs that are not ULCs separately, and we call them *proper* XLCs.

When an XLC is extracted and we add the AtMostOne constraint to the formula, all all binary clauses of the form $(\overline{\ell} \vee \overline{\ell}')$ on literals in the XLC become redundant and can be deleted.

Similar to a ULC, an XLC can be considered an implicit EXACTLYONE constraint as all missing binary clauses can be added via blocked clause addition. The notion of a pair of clashing XLCs is the same as for ULCs: the pair contains complementary literals. In contrast to ULCs, it is not the case that resolving two XLCs always results in a new XLC.

Example 3.5

Consider the formula $\Gamma = (x \vee z) \wedge (x \vee u) \wedge (y \vee \overline{u}) \wedge (y \vee z) \wedge (\overline{z})$. The clauses $(x \vee u)$ and $(y \vee \overline{u})$ are clashing XLCs with u and $\overline{u}$ being unique literals. Resolving these clauses results in $(x \vee y)$, which is not an XLC. Moreover, it cannot be turned into XLC, because adding the missing binary clause $(\overline{x} \vee \overline{y})$ does not preserve satisfiability.

The above issue arises when resolving on unique literals. When resolution on a pair of clashing XLCs is restricted to non-unique literals, then the resulting clause will be an XLC. However, performing variable elimination on a non-unique literal can cause a blowup in the size of the formula if the variable occurs often in different polarities. Therefore, we do not resolve XLCs that clash on non-unique literals, and instead only add an AtMostOne constraint for one of the clashing XLCs.

3.4 Extracting ULCs and XLCs in CADICAL

We implement the detection of ULCs and XLCs as a preprocessing step in the solver CADICAL. To classify clauses as ULCs and XLCs we first compute occurrence counts for each literal in the formula F and a lookup table storing all binary clauses. Next, we classify each clause in order. We determine if a clause is a ULC by checking if each literal in the clause is unique. If the clause is not a ULC, we check if it is an XLC by searching the lookup table for all binary clauses forming an AtMostOne constraint on all non-unique literals in the clause.

Once all clauses are classified, we proceed to handling clashes on ULCs. For each literal ℓ in a ULC with $\mathsf{occ}(\ell, \Gamma) = 1$ and $\mathsf{occ}(\overline{\ell}, \Gamma) = 1$ we mark the literal as clashing only if $\overline{\ell}$ occurs in a ULC, and store a clause lookup table $L(\ell)$ and $L(\overline{\ell})$ pointing to the ULCs ℓ and $\overline{\ell}$ occur in. We then process each ULC with a clashing literal ℓ, resolving both $L(\ell)$ and $L(\overline{\ell})$. We delete $L(\ell)$ and $L(\overline{\ell})$, eliminating the variable $\mathsf{var}(\ell)$ from the formula. The resolvent may take one of two forms: i) the resolvent is a tautology in which case we delete it; ii) the resolvent is a clause with at least two literals in which case we add it to the formula and mark it as a ULC. The resolvent cannot be empty because we propagate unit clauses in a separate procedure, and the resolvent cannot be unit because we only resolve two ULCs. We must also update the lookup table for any clashing literals contained in the resolvent because it may also be clashing with an existing ULC. After all clashes have been processed, we extract all XLCs and ULCs that have at least 5 literals. We use this size cutoff to avoid small constraints that are typically ineffectual in KNF solving.

The use of variable elimination on ULCs to remove clashing pairs makes use of RAT deletion

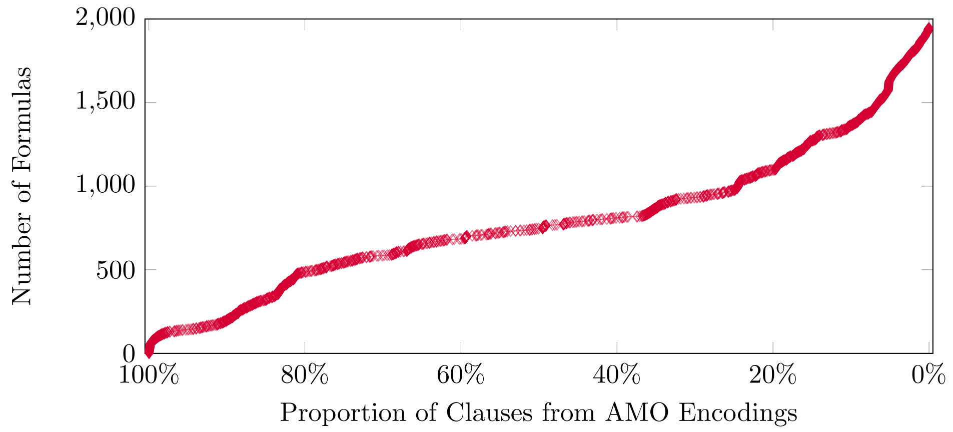


Figure 3.5: Number of formulas that are composed of some proportion of encoding clauses from AtMostOnes.

proof steps. In the case that the formula is satisfiable, we will need to use a reconstruction procedure to assign a value to the deleted variables. Reconstruction can be handled internally by CaDiCaL if we wish to use the solver for both extraction and solving.

3.5 Results

We applied our extraction tools to the $5{,}355$ SAT Competition Anniversary Track benchmarks. We discuss first the results of our AtMostOne detection, then our ULC and XLC detection.

3.5.1 AtMostOne Extraction

We ran our AtMostOne extraction tool with $1{,}000$ second timeouts for each AtMostOne extraction type (Pairwise and Non-Pairwise). The Pairwise extraction was run first but did not remove clauses for AtMostOnes with less than four literals. This is because many Non-Pairwise AtMostOne encodings use Pairwise AtMostOnes on subsets of the encodings, often over four or fewer literals. After the Non-Pairwise extraction, we check the smaller AtMostOnes and if they still exist in the formula we extract them.

The results in Table 3.2 show that AtMostOnes occur in the majority of studied SAT problems (64%); and furthermore, most of the extracted AtMostOnes use the Pairwise encoding. While the

Table 3.2: The table shows number of formulas with Pairwise, Non-Pairwise, or both Pairwise and Non-Pairwise AtMostOnes extracted. ≥ 5 is the percent of formulas with at least one extracted constraint of at least size 5, and $\geq 10 \times 10$ is at least 10 extracted constraints of at least size 10. We show the average runtime and the percentage of formulas that took ≤ 15 seconds.

Pairwise	Non-Pairwise	Both	≥ 5	$\geq 10 \times 10$	Average s	≤ 15 s
3,090	55	270	36%	17%	69.0	78.0 %

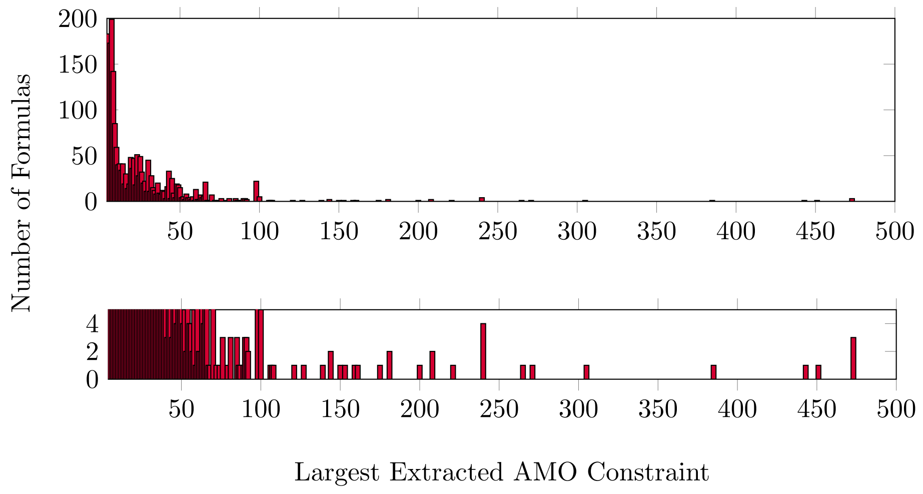


Figure 3.6: Histogram grouping formulas by the largest AtMostOne constraint extracted. Size is defined by the number of data variables. The bottom plot is a zoomed-in version of the full plot.

extractor only found a few hundred formulas with Non-Pairwise constraints, it could be the case that more existed. Since we do not have the ground truth, it is impossible to gauge how well Non-Pairwise extraction performed. The average runtime is fast on average, but can be improved on large problems by implementing additional filtering heuristics that prevent the guesser from trying to build a constraint from each classified auxiliary variable. Over 17% of the formulas contained several large cardinality constraints. This matches the results in Figure 3.5 showing that around 750 formulas are mostly composed of cardinality constraints, with over half of the clauses in the formula coming from AtMostOne encodings. These formulas are prime candidates for applying different cardinality solving methods, whereas formulas with very few small AtMostOnes would likely not benefit from cardinality solving.

The histograms in Figure 3.6 show the size of the largest AtMostOne extracted for each formula. Most problems contained smaller AtMostOne constraints, size 25 or below, but there were several problems containing much larger AtMostOnes. The largest Pairwise AtMostOnes were found in planning (up to size 473), petrinet (451), and edge-matching (140) problems, and the largest Non-Pairwise AtMostOnes were in planning (240), hidoku (80), and scheduling (43) problems.

3.5.2 ExactlyOne Extraction

Our ULC and XLC extraction configurations are as follows:

- ULC : extract ULCs with at least 5 literals. Including variable elimination for clashing ULCs.
- ULC-clash : ULC extraction without variable elimination.
- XLC-proper : XLC extraction on formulas that contain at least one proper XLC.
- XLC-full : XLC extraction (this includes both ULCs and proper XLCs).

Table 3.3 shows a comparison of the various extraction configurations. The combination of ULC and proper XLC (XLC-full) extraction found constraints in $1{,}739$ ($\sim 32\%$) formulas, showing just how often object selection appears often in SAT problems. The one hundred formula difference between ULC-clash and ULC is explained by the many formulas containing only small ULCs (under our cutoff of size 5) that would typically be filtered out by our extraction heuristics. However, applying variable elimination on clashing ULCs generates results in larger ULCs that then meet our size criteria. The average runtimes for the ULC and XLC reencoding configurations are often quite fast because the patterns can be detected in linear time, so formulas without any detectable constraints can be dispatched quickly.

3.6 Remarks on Extraction

In this chapter, we discussed our novel guess-and-verify framework for AtMostOne cardinality constraint extraction. The G&V tool, guided by its classification heuristics for auxiliary and data variables, works on the majority of PySAT encodings, outperforming other extraction tools by a large margin. Furthermore, the BDD-based verification provides a guarantee that the clauses do in fact represent a cardinality constraint. We performed an exploratory experiment on the SAT Competition Anniversary Track benchmarks and found that the majority contained AtMostOnes and these mostly appeared as the Pairwise encoding.

In addition, we described a technique for extracting implicit and explicit ExactlyOne constraints based on ULCs and XLCs. These constraints appear in about one third of the SAT Competition Anniversary Track benchmarks, and most of these formulas contained ULCs. ULCs would not otherwise be detected by an AtMostOne extraction tool, exhibiting the harmony between our two extractors.

Our extraction tools have generated a large set of KNF formulas for testing any new KNF solving techniques. This can help us make the case for KNF, showing that KNF solving can improve solver performance on many existing benchmarks. Extraction can also provide a means for backwards compatibility with CNF producing tools, allowing them to harness the power of KNF solvers without having to rewrite parts of their codebase.

The obvious next step is the extraction of AtLeastK constraints. General cardinality constraints have more complicated, often hierarchical encoding structures with binary and ternary clauses. Our G&V framework is a good starting point, but the guesser would need more sophis-

Table 3.3: Number of formulas with extracted constraints, average extraction time, number of formulas taking longer than 20 seconds, the medium number of encoded XLCs, and the medium of the maximum sizes of encoded XLCs.

	Found	Avg. (s)	$\geq$ 20s	Med. #	Med. Max
ULC-clash	891	4	29	760	45
ULC	1014	4	43	760	40
XLC-proper	825	1	9	1001	70
XLC-full	1739	3	49	867	52

ticated classification heuristics for distinguishing auxiliary variables from data variables. Furthermore, the BDD-based verification works well on AtMostOnes but may have trouble scaling with more complex encoding structures. To address this challenge, we will need new strategies for generating variable orderings, as well as the addition of light-weight verification techniques using either a SAT solver or random testing to serve as alternatives for much larger cardinality constraints.

Chapter 4

Encoding

Encoding high-level constraints into CNF is a crucial step when solving any problem in SAT. There is a plethora of encodings to choose from when encoding AtLeastK constraints and even more for the specific cases of AtMostOne and ExactlyOne constraints. The choice of encoding will affect the size and propagation properties of the formula, and the auxiliary variables introduced by an encoding will determine how a solver might reason abstractly over groups of literals within the constraint. Some inprocessing techniques such as SBVA can add auxiliary variables during solving, but these only work on simple patterns and cannot reconfigure an entire cardinality constraint encoding. In other words, we cannot count on separate inprocessing techniques or solver heuristics to make up for a poor choice of encoding.

The KNF format offers a way around this choice: users can first generate a problem in a cardinality normal form and then use our tools that handle encodings automatically. While we strive to make SAT solving more user friendly, performance is another core component of our thesis. We will show how by leveraging cardinality information from the KNF input we can improve encodings for many different types of cardinality constraints.

In this chapter, we will describe three ways for improving the encodings of cardinality. First, in Section 4.1, we will start with a simple example, showing that our AtMostOne extraction and reencoding framework can transform the Pairwise encoding into the Linear encoding. By doing so, we introduce auxiliary variables that can improve solver performance on unsatisfiable formulas. This framework is proof producing, meaning the reencoding steps produce valid DRAT derivations.

Second, in Section 4.2, we take a step beyond the existing encoding framework that asks *which* types of encodings are best and instead ask *how* can any type of encoding be improved. We developed several methods for sorting data literals prior to encoding in an effort to enhance a solver's ability to perform group reasoning. Literal sorting changes the grouping of literals within the encoded constraint, and we show that better groupings can lead to much faster solving times on problems containing one large AtLeastK constraint.

Third, in Section 4.3, we tie the ideas of extraction and reencoding together with literal sorting, developing a reencoding framework for ExactlyOne constraints that build on our ULC and XLC extractor. Often in practice cardinality constraints are viewed as independent objects with therefore independent encodings. We challenge this notion, finding that the relationships between cardinality constraints can be complex and this should be reflected in their encodings. We present a technique that we call alignment which sorts literals amongst all cardinality constraints at once. This allows a solver to simultaneously reason over groups of literals across cardinality constraints. We show that for many problems reencoding alone is not enough to improve solver performance, but with alignment we can modify the meaning of auxiliary in such a way that dramatically benefits the solver.

4.1 Reencoding AtMostOne Constraints

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. *From Clauses to Klauses*. In *Computer Aided Verification (CAV)*. (2024). [91]

It is well-known that the Pairwise AtMostOne encoding performs poorly on unsatisfiable problems. The auxiliary variables introduced in Non-Pairwise encodings can allow the solver to reason about subcomponents of the AtMostOne, effectively removing large portions of the search space with few decisions. This can yield shorter, stronger learned clauses leading to more compact reasoning and faster solving times. Example 4.1 shows how auxiliary variables can summarize groups of literals in a learned clause. Importantly, for a solver to learn the clauses in Example 4.1 without auxiliary variables, it would need to make a chain of several subsequent decisions on the relevant set of data variables, but only a single decision when using auxiliary variables. The impact of this type or reasoning compounds when auxiliary variables from different AtMostOnes are used together in clause learning.

Example 4.1

Consider the AtMostOne constraint on data variables $x_1, x_2, \ldots, x_6$. By introducing two auxiliary variables y_1, y_2, the AtMostOne can be decomposed as:

$$\mathsf{AMO}(x_1, \ldots, x_6) \iff \big((x_1 \lor x_2 \lor x_3) \Rightarrow y_1\big) \land \big((x_4 \lor x_5 \lor x_6) \Rightarrow y_2\big) \land \\ \mathsf{AMO}(x_1, x_2, x_3) \land \mathsf{AMO}(x_4, x_5, x_6) \land (\overline{y}_1 \lor \overline{y}_2).$$

Here, y_1 summarizes $\{x_1, x_2, x_3\}$ and y_2 summarizes $\{x_4, x_5, x_6\}$. A solver can branch on y_1 and set it to false, immediately pruning x_1, x_2, x_3.

Fewer Clauses. If the formula contained the clause $(x_1 \lor x_2 \lor x_3 \lor x_7)$, the solver could learn the binary clause $(\overline{y}_1 \lor x_7)$. With the Pairwise encoding, the solver would instead need to learn the following three binary clauses to capture the same implication: $(\overline{x}_1 \lor x_7)$, $(\overline{x}_2 \lor x_7)$, and $(\overline{x}_3 \lor x_7)$.

Shorter Clauses. Assume the following implication is contained implicitly in the formula, $\overline{x}_1 \land \overline{x}_2 \land \overline{x}_3 \implies (x_7 \lor x_8)$. Without auxiliary variables the solver could learn $(x_1 \lor x_2 \lor x_3 \lor x_7 \lor x_8)$, and with auxiliary variables the solver could learn $(y_1 \lor x_7 \lor x_8)$.

In this section, we present a reencoding framework that takes the extracted AtMostOne constraints using our G&V extractor and reencodes them with the Linear encoding. We show how this process can be equipped with DRAT proof production so that results produced from the reencoded formulas can be verified against the original formula.

4.1.1 Proof Production

In the context of our G&V AtMostOne extractor, we do not generate proofs when transforming the formula into KNF. Instead, we assume the original CNF used arc-consistent encodings for the extracted constraints and generate a derivation when reencoding the AtMostOnes into clauses. This derivation can be prepended to the DRAT proof generated for the reencoded formula and will serve as a proof for the original formula.

For the Pairwise encoding, the derivation is simply the set of clauses from the encoding added to the formula. Each binary clause is RUP since assigning two literals in the constraint to `true` must propagate a conflict in the original formula. The derivation of the Linear encoding is similar to its clausal encoding, with an additional clause for each auxiliary variable:

$$\mathsf{Deriv}(\ell_1, \ldots, \ell_s) = \mathsf{Pairwise}(\ell_1, \ell_2, \ell_3, y), (\ell_1 \vee \ell_2 \vee \ell_3 \vee y), \mathsf{Deriv}(\overline{y}, \ell_4, \ldots, \ell_s)$$

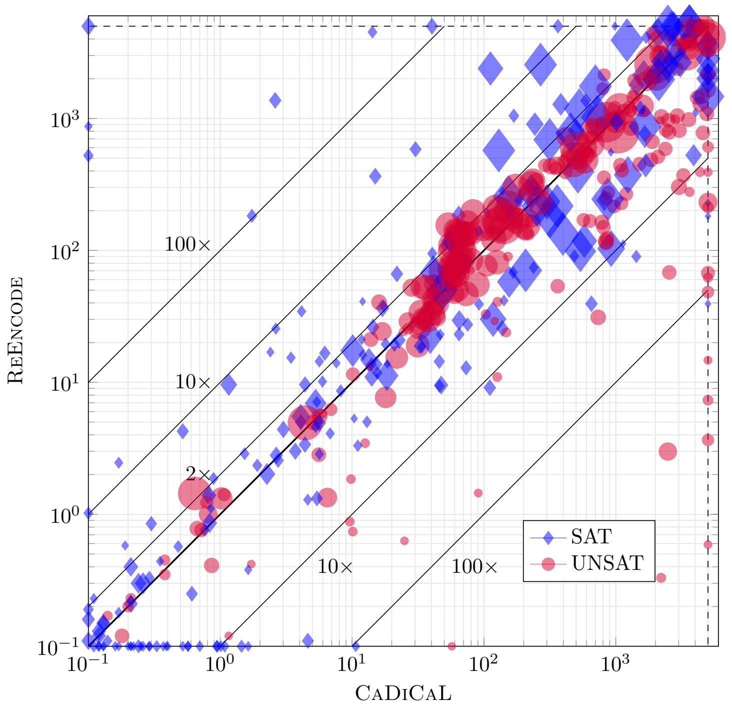
The Linear derivation makes use of RAT proof steps since new auxiliary variables are being added to the formula.

In the case the reencoded formula is satisfiable the solution may not directly translate to the original formula. Extraction may have removed some auxiliary variables from the original formula, and reencoding may add new auxiliary variables to the reencoded formula. Given a solution on the reencoded formula, first we discard variables not occurring in the original formula, then we add the partial assignment as units to the original formula and run a SAT solver to find an assignment on any auxiliary variables only appearing in the original formula. Note, it is easy to assign values to auxiliary variables if the values for data variables are given.

4.1.2 Results

For this evaluation we used a $5{,}000$ second timeout, and do not include the runtime of our extractor. We consider the subset of 933 formulas from the SAT Competition Anniversary Track in which we extracted at least 10 extracted constraints of size 10 or more.

Figure 4.1 reinforces the conception that Non-Pairwise AtMostOnes are stronger on unsatisfiable formulas, with several formulas in the top plot only solved within the $5{,}000$ second timeout after reencoding. Note, a small portion of the extracted AtMostOnes were Non-Pairwise (see Table 3.2) and these were reencoded as well, but with little effect. The bottom plot shows that formulas with no Pairwise AtMostOnes did not benefit from reencoding. The effect of reencoding is mixed on satisfiable formulas. Local search is crucial for solving some satisfiable problems, and auxiliary variables from Non-Pairwise encodings can inhibit local search. Notably, this plot shows that many of the SAT competition benchmarks use sub-optimal AtMostOne encodings, and an extraction plus reencoding pipeline could greatly benefit non-experts.



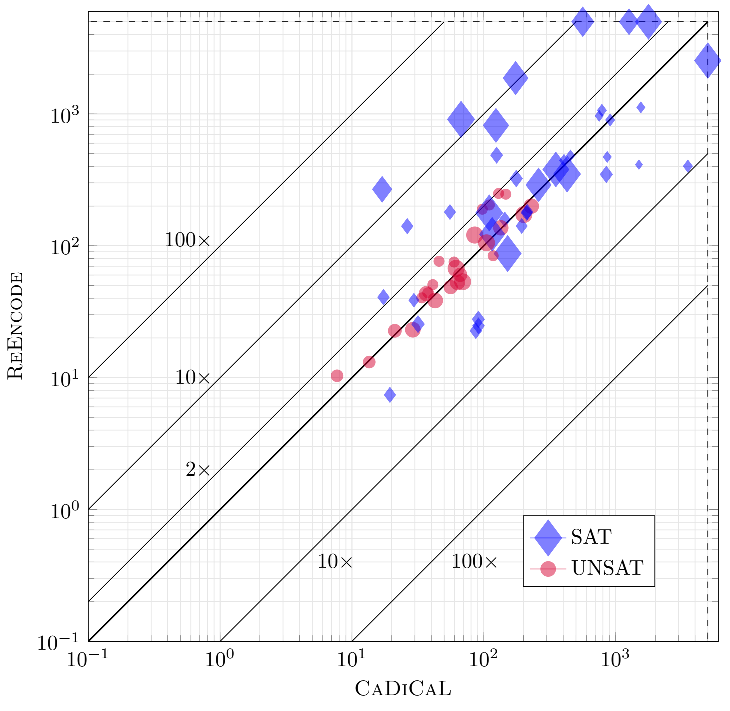


Figure 4.1: Reencoding on formulas with Pairwise encoded AtMostOnes (836) (top) and those without (97) (bottom). The size of a mark is proportional to the number of extracted constraints of size 10 or more, i.e., formulas with many large AtMostOne constraints have large marks.

4.2 Literal Sorting for AtLeastK Encodings

Joseph E. Reeves, João Filipe, Min-Chien Hsu, Ruben Martins, and Marijn J. H. Heule. *The Impact of Literal Sorting on Cardinality Constraint Encodings*. In *Conference on Artificial Intelligence (AAAI)*. (2025). [92]

Selecting the best encoding for a set of cardinality constraints is not only about *which* type of encoding is best but also *how* the encoding is used. One way to alter any standard encoding is to change the ordering of data literals prior to encoding.

Example 4.2

x Consider the AtMostOne constraint on $x_1, x_2, \ldots, x_6$ from Example 4.1. The meaning of the auxiliary variables y_1 and y_2 depends on the literal order. In the original encoding, y_1 summarizes $\{x_1, x_2, x_3\}$ and y_2 summarizes $\{x_4, x_5, x_6\}$.

If we change the order of the data literals to $\mathsf{AMO}(x_1, x_3, x_5, x_2, x_4, x_6)$, the new encoding will be:

$$\begin{aligned}\mathsf{AMO}(x_1, x_3, x_5, x_2, x_4, x_6) \iff{} & \big((x_1 \vee x_3 \vee x_5) \Rightarrow y_1\big) \wedge \big((x_2 \vee x_4 \vee x_6) \Rightarrow y_2\big) \wedge \\ & \mathsf{AMO}(x_1, x_3, x_5) \wedge \mathsf{AMO}(x_2, x_4, x_6) \wedge (\overline{y}_1 \vee \overline{y}_2).\end{aligned}$$

Now, y_1 summarizes $\{x_1, x_3, x_5\}$ and y_2 summarizes $\{x_2, x_4, x_6\}$. Assume the formula contained the clause $(x_2 \vee x_4 \vee x_6)$. With the new ordering, the solver can learn the unit y_2 via probing, then use unit propagation to set $y_1 = x_1 = x_3 = x_5 = 0$. Similar reasoning could not be performed for the original ordering because the auxiliary variables summarize different sets of data literals.

The intuitions behind literal ordering for AtMostOne encodings shown in Example 4.2 can be extended to more complicated Totalizer encodings shown in Figure 4.2. A solver can reason about a certain group of literals via auxiliary variables from the first node at which all literals are merged. If this merge unit is larger (contains several unrelated literals) then the solver will need to reason over many literals at once; however, if the merge unit is smaller, then the solver can reason more succinctly over the related literals. If we know the optimal grouping of literals, we can sort the literals such that groups occur in sequence and this will restructure the Totalizer to permit more compact solver reasoning.

Coming up with the right literal ordering is not always a straight-forward task. The problem in Example 4.3 shows an instance where a more natural ordering $1, 2, 3, \ldots, 100$ is not as useful as an odd then even ordering $1, 3, \ldots, 99, 2, 4, \ldots, 100$. In the following sections we will discuss several ways that a good ordering might be found automatically based on features from a given problem.

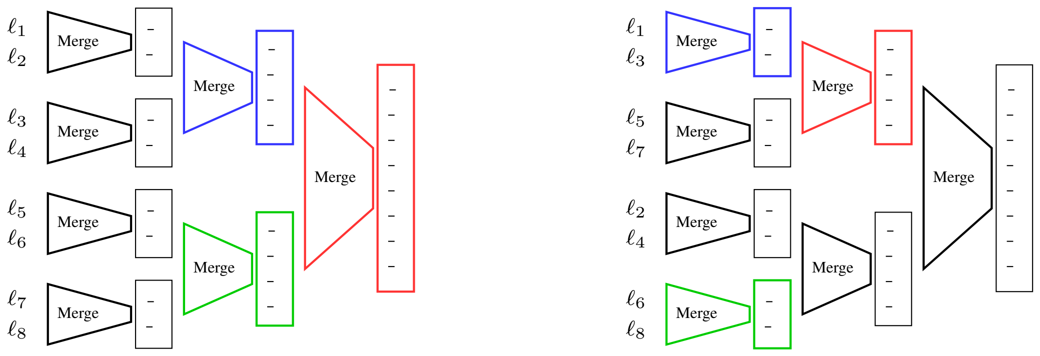


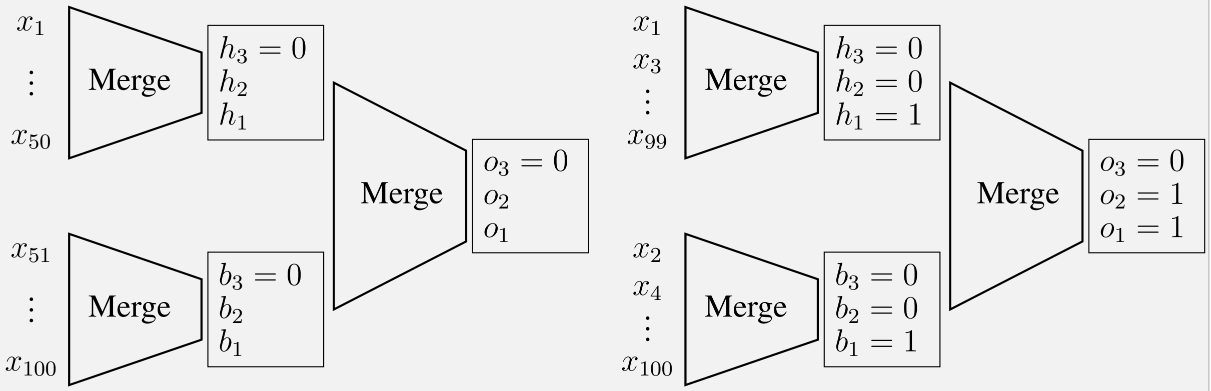
Figure 4.2: Totalizer with two input orderings. Groupings of related literals are ℓ_1, ℓ_3 (blue), ℓ_1, ℓ_5, ℓ_7 (red), and ℓ_6, ℓ_8 (green). A solver can perform more compact reasoning over smaller highlighted merge units (Totalizer on the right).

Example 4.3

Consider the following formula with one cardinality constraint:

$$(x_1 + x_2 + \cdots + x_{100} \leq 2) \wedge$$
$$(x_1 \vee x_3 \vee \cdots \vee x_{99}) \wedge (x_2 \vee x_4 \vee \cdots \vee x_{100}) \wedge F$$

The cardinality constraint may be encoded as a Totalizer. Two options for the encoding based on different literal orderings are shown below. The auxiliary variables h_i, b_i, o_i are counters for `true` input data literals, so h_3, b_3, o_3 are set to `false` (0), to enforce the bound of at most two. Additional values, `true` denoted by (1), can be derived in (b) via probing.



(a) **(b)**

In (b), a solver can learn the units h_1 and b_1 through probing: Assigning h_1 to `false` would propagate all the input literals for the left subtree to `false`, causing a conflict with the odd-literal clause. Similarly, b_1 can be learned. This propagates o_1 and o_2 to `true` and then h_2 and b_2 to `false`, since the sum of the children cannot exceed 2. In short, the auxiliary variables h_1 and b_1 allow the solver to reason about the entire set of data literals in either subtree and cheaply derive units. In (a), h_1 and b_1 summarize different sets of data literals preventing reasoning about the clauses. Leaving the solver to reason over the entire Totalizer encoding with no units learned.

4.2.1 Literal Sorting Methods

When we discuss hierarchical encodings like the Totalizer it is natural to think about groups of literals, but most encoding APIs like PySAT take as input a list of literals. Therefore, instead of grouping literals we will describe methods for sorting literals based on some order. This will allow us to use PySAT in the evaluation, and will still lead to better encodings, even if some information about the relationships between literals is lost due to the linearization of groups.

Example 4.4

$$\texttt{CardEnc.atleast}(\texttt{lits} = [1, 2, 3, 4, 5, 6], \texttt{bound} = 3, \texttt{encoding} = \texttt{EncType.totalizer})$$
$$\texttt{CardEnc.atleast}(\texttt{lits} = [1, 3, 5, 2, 4, 6], \texttt{bound} = 3, \texttt{encoding} = \texttt{EncType.totalizer})$$

Two API calls in Pysat for the encoding of $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 3$ and the same constraint with data literals reordered as $x_1 + x_3 + x_5 + x_2 + x_4 + x_6 \geq 3$.

We consider five orderings, the first three of which we discuss briefly, then the proximity ordering and community ordering we describe in more detail below.

- **Natural**: sort by variable IDs. Often when a benchmark is created there is some structure given to the variable names, for example, a row-order enumeration of cells in a matrix, and the natural ordering captures this structure.
- **Occurrence**: sort by the occurrence count (number of times a variable occurred positively or negatively in the formula). This will place more prominent variables close together, in an attempt to localize important parts of the encoding.
- **Random**: shuffle the literals. This will obfuscate any relationship between literals.
- **Proximity**: approximate the spatial distance between literals by looking at co-occurrence in clauses. The closer literals are in the formula, the closer they will appear in the ordering.
- **Community**: generate the variable-incidence graph (VIG) by taking variables as nodes and clauses as weighted edges connecting variables. The community detection algorithm [109] groups the variables, and those groups are linearized by smallest variable ID.

We will use the following formula as an example for the different sorting methods:

$$(x_1 + x_2 + x_3 + \overline{x}_4 \leq 2) \wedge$$
$$(x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_2) \wedge (\overline{x}_2 \vee x_3 \vee x_4) \wedge (\overline{x}_4 \vee x_5)$$

The Natural ordering is the simplest method, given by increasing variable IDs: $x_1 + x_2 + x_3 + \overline{x}_4 \leq 2$. While this method typically reflects the underlying problem's structure, it may not capture relationships between literals that are important to a specific cardinality constraint in the problem.

Another simple ordering is Random, acquired by randomly permuting the list of variables. This method disassociates the cardinality constraint from any of the problem's underlying structure. It is sometimes useful to randomly shuffle several copies of a formula to solve a satisfiable instance in parallel, since a solution might be found quickly for one copy if the solver gets lucky.

Next, we consider orderings that are derived from the clausal structure of the input problem. The occurrence method orders all variables in decreasing order based on the number of clauses a variable occurs in (taking the sum of the occurrences of both positive and negative polarities): $x_2 + x_1 + \overline{x}_4 + x_3 \leq 2$. Counting the number of occurrences is inexpensive and can be done with a single pass over the formula. The resulting ordering will be unbalanced with respect to the formula as groups of the most occurring variables will summarize large parts of the formula, and groups of sparsely occurring variables will summarize small parts of the formula. While most types of encoding are symmetric, the Sequential Counter provides more reasoning capabilities for the data literals earlier in the order, motivating the choice to count occurrence in decreasing order, e.g., most important to least important.

4.2.2 Proximity Ordering

The proximity ordering approximates the spatial distance between literals by looking at co-occurrence in clauses. The method can be enhanced by first extracting large AtMostOne constraints then using them to help guide the proximity metric.

The ordering procedure first initializes the score of all variables to 0, then proceeds as follows:

1. Select the unprocessed variable v (i.e., v is not yet a part of the ordering) with the highest score. If the highest score is 0, select the most occurring unprocessed variable. If there is a tie, select the variable that was seen first. A variable is seen the first time its score is incremented.
2. Append v to the ordering.
3. If AtMostOne detection is enabled, for each AtMostOne K that contains v, increment the scores of unprocessed variables occurring in K by $len(K)^2$, where the length of an AtMostOne constraint is the number of variables it contains.
4. For each clause C that contains v, increment the scores of unprocessed variables occurring in C by $\frac{1}{len(C)}$ if $len(C) \geq 3$ or $len(C)^2 = 4$ for a binary clause.
5. If all variables in cardinality constraints are processed, return the ordering; otherwise, return to step 1.

Example 4.5

We show an application of Proximity to the running example. Applying the Proximity algorithm without AtMostOne detection will first select the most occurring variable x_2. The clauses x_2 occurs in are then processed. For $(x_1 \vee x_2)$, x_1's score is incremented by 4. For $(\overline{x}_1 \vee x_2)$, x_1's score is again incremented by 4. For $(\overline{x}_2 \vee x_3 \vee x_4)$, both x_3 and x_4 are incremented by $\frac{1}{3}$. The second variable selected is x_1 with a score of 8. x_1 only occurs in clauses with the already processed variable x_2. The third variable selected is x_3 (seen before x_4) with a score of $\frac{1}{3}$, and finally x_4, which yields the ordering: $\{x_2, x_1, x_3, x_4\}$.

4.2.3 Community Ordering

Finally, we consider orderings extracted from graphs constructed from the literals and clauses of the problem. The VIG is an undirected, unweighted graph where the set of nodes represent each variable. An edge connects two nodes if the corresponding variables, regardless of their polarity, share a clause.

We use the Louvain Community Detection algorithm [109]. Each node is placed in its own set, and then nodes are moved to other sets if the move increases the modularity. Next, sets are lifted to nodes, and the algorithm is repeated until some threshold is met. The order nodes are processed affects the resulting communities, so they are shuffled using a random seed at the start of each execution. To identify the most promising community structures we use up to 50 executions, with a 300 second timeout enforced after the first run.

From these multiple runs, we select the executions that contain the highest number of communities. A higher number of communities typically indicates a more fine-grained partitioning of the graph, which might help capture intricate relationships between literals or clauses. Furthermore, since the variables within each community are ordered by their IDs, a higher number of communities will also lessen the impact of the natural ordering. We then choose the execution with the set of communities that has the smallest deviation from the average community size, aiming for a more balanced community structure. Finally, we determine the variable order by concatenating the variables from all the communities, processing each community sequentially by smallest variable ID.

Example 4.6

Suppose the Louvain Community Detection algorithm is executed three times, yielding the following sets of communities:

$$S_1 = \{C_{1,1} = \{x_1, x_3, x_5\}, C_{1,2} = \{x_2, x_4\}\}$$
$$S_2 = \{C_{2,1} = \{x_1, x_3, x_5\}, C_{2,2} = \{x_2\}, C_{2,3} = \{x_4\}\}$$
$$S_3 = \{C_{3,1} = \{x_1, x_3\}, C_{3,2} = \{x_2, x_5\}, C_{3,3} = \{x_4\}\}$$

We would first select the second and third executions, as they contain the highest number of communities. Next, we would choose the third execution, as the community sizes show a smaller deviation from the average size. Lastly, we concatenate the variables from all communities, yielding: $x_1 + x_3 + x_2 + \overline{x}_4 \leq 2$.

4.2.4 Results

For this evaluation we used a $1{,}800$ second timeout. This timeout is specific to the configuration of StarExec. We evaluated literal sorting on the 398 2023 MaxSAT competition unweighted track benchmarks with known optimums. For each problem we generated a satisfiable instance using the cardinality constraint $s_1 + s_2 + \cdots + s_n \geq opt$ and an unsatisfiable problem $s_1 + s_2 + \cdots + s_n \geq opt + 1$. We then used one of our various literal ordering techniques before calling PySAT to encode the cardinality constraint into CNF. The resulting formula was solved with CaDiCaL.

For the proximity algorithm we considered both with and without AtMostOne extraction. When using extraction, our AtMostOne extractor was run with a 50 second timeout. In either case, proximity sorting was run to completion, i.e., every literal in the cardinality constraint was added to the ordering.

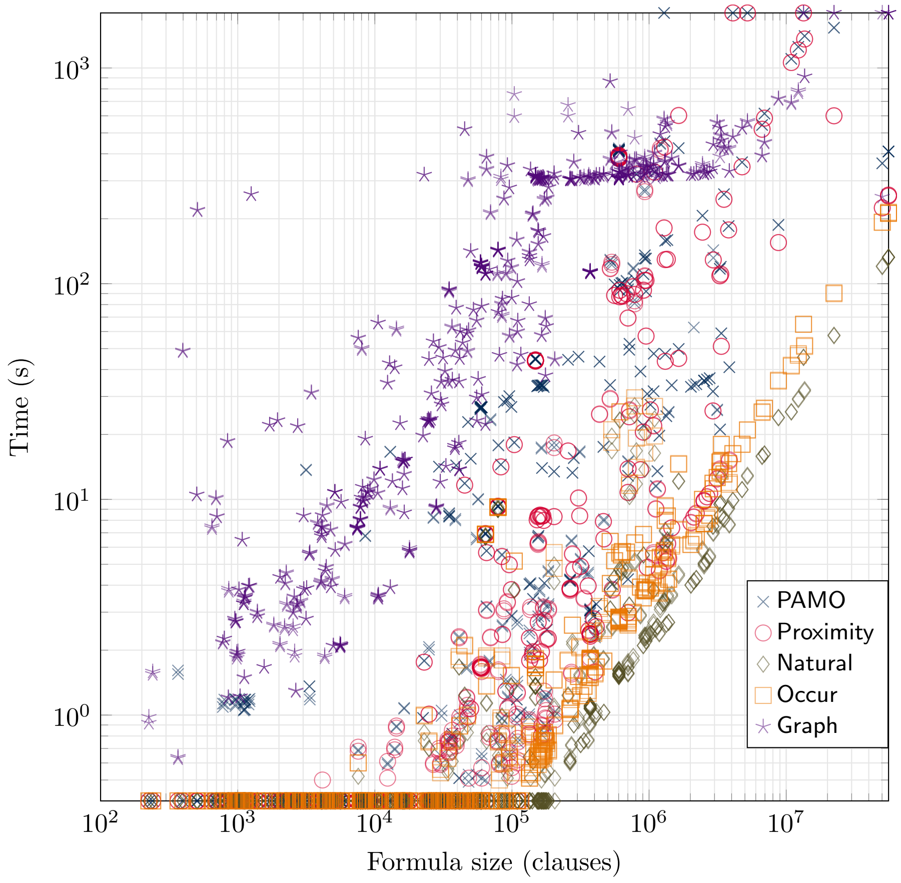


Figure 4.3: Preprocessing time for each formula and literal sorting method. Formulas are ordered by number of clauses.

We used the Louvain community detection algorithm [108] for the community detection ordering. Groupings are dependent on the order nodes are processed, so we ran 50 randomized executions (or until a 300 second timeout) then selected the execution yielding the largest number of communities with the smallest deviation from the average community size. We refer to the community detection approach as Graph.

Preprocessing Time for Sorting Methods

The preprocessing time for each method is shown in Figure 4.3. The preprocessing time includes both the time to sort literals and the time to encode the cardinality constraint as a Totalizer using PySAT. The runtimes of Natural and Occur (occurrence counts can be computed during parsing) track the cost of the PySAT encoder since no or little computation for the literal sorting takes place. The proximity methods scale with the formula size as well since they perform a type of breadth-first-search over the clauses. The difference between PAMO and Proximity is more pronounced on smaller formulas when AtMostOne detection with a 50 second timeout outweighs the cost of Proximity. Graph has the longest runtimes, partly due to the algorithms 50 executions per formula. In some cases, the 50 executions are not completed, seen by the formulas clustering

Encoding	Natural	PAMO+Occur	Proximity	Occur	BestRandom	WorstRandom
KMod-Totalizer	635	670	655	588	561	528
Mod-Totalizer	623	653	643	572	547	514
Cardinality Network	608	639	629	565	544	503
Sorting Network	602	645	631	561	532	506
Sequential Counter	597	617	611	563	539	507

Table 4.1: Number solved on 398 satisfiable and 398 unsatisfiable formulas using different types of PySAT encoding and different literal sorting techniques.

around 300 seconds.

While some amount of cost in preprocessing is worth the benefit to solving, timeouts during preprocessing are not acceptable. To mitigate this, we considered two additional configurations:

- **Natural+PAMO**: run Natural and the SAT solver for 100 seconds and if the formula is not solved then restart and run PAMO.
- **PAMO+Occur**: run PAMO for formulas under 1,000,000 clauses; otherwise, run the much cheaper Occur method.

It is common practice in SAT solving to run low effort solving techniques like lucky search before throwing the full set of inprocessing tools and CDCL heuristics at a problem. The hope is that certain easy-to-solve problems are dispatched before more complex but ultimately unnecessary solving techniques are employed. This motivated Natural+PAMO. For some large formulas not solved in 100 seconds the Proximity computation after the restart may timeout, but at least an initial attempt was made to solve these problems. In addition, some inprocessing techniques are simply too costly to be applied to large formulas and are disabled or postponed. This strategy motivated PAMO+Occur.

Solving Time for Sorting Methods

Solving times are the sum of literal sorting, encoding, and solving. Table 4.1 shows the effect of literal sorting on each of the five encoding types. Comparing the Natural ordering across the types of encodings gives a sense for what off-the-shelf encoding is most performant on these sets of problems. The KMod-Totalizer solves several more formulas than any other encoding, marking it as the best default choice on these optimization problems with one large cardinality constraint. However, after applying the PAMO+Occur ordering the Mod-Totalizer, Cardinality Network, and Sorting Network solve more instances than the KMod-Totalizer with the Natural ordering. This means that picking the best literal ordering is more important than picking the best type of encoding. When both the optimal encoding and literal ordering are combined, the PAMO+Occur KMod-Totalizer solves 15 more formulas than any other configuration.

On the other hand, a bad choice of literal ordering is far worse than a bad choice of encoding. The worst ordering of five shuffled attempts generally solves around one hundred fewer formulas than the Natural ordering. The best random ordering does not do much better than the worst random ordering, mostly helping for satisfiable formulas. Furthermore, a simple heuristic like

Ordering	Solved		PAR2		Best Runtime	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
VBS	363	332	358	653	–	–
PAMO+Occur	**353**	**317**	**492**	**799**	–	–
Natural+PAMO	351	315	499	806	–	–
PAMO	347	312	560	857	40	35
Proximity	343	312	591	856	49	46
Graph	332	312	762	949	6	14
Natural	334	301	635	916	148	157
Occur	317	271	792	1189	22	21
BestRandom	313	248	818	1388	98	59
WorstRandom	284	244	1106	1434	0	0

Table 4.2: Number solved and average PAR2 on 398 satisfiable and 398 unsatisfiable formulas using the PySAT KMod-Totalizer and different literal sorting techniques broken down by SAT and UNSAT. VBS picks the best literal sorting method for each formula. Best runtime shows number of formulas a given configuration solved the fastest.

Occur cannot match the performance of Natural. This reinforces the belief that users add a lot of structure and meaning to their encodings when they decide how to express constraints and generate variable IDs.

In comparing the different types of encodings, it seems that hierarchical encodings (everything but the Sequential Counter) benefit the most from literal sorting. This may be explained by the structure of the Sequential Counter, where auxiliary variables do not necessarily summarize groups of data literals but instead summarize a prefix or suffix of the sequential counter. Data literals appearing at the ends of the ordering may be concisely summarized by auxiliary variables, but those appearing in the middle of the ordering cannot be accessed. Somewhat surprisingly, the Sorting Network overtakes the Cardinality Network with PAMO+Occur. This is likely the cause of noise as both encodings have varying results for the different literal sorting techniques.

Table 4.2 shows a closer look at the various literal sorting techniques applied to the KMod-Totalizer encoding. It is expected that a good literal ordering would mostly benefit unsatisfiable formulas since the new auxiliary variables would enable compact reasoning towards a short proof. However, the PAR2 scores for PAMO+Occur show that a good ordering can improve performance for both the satisfiable and unsatisfiable formulas. This is likely the case because the cardinality constraint enforces the optimal bound and therefore must be used in the solver's reasoning. That said, a solver can get lucky and quickly find a satisfying assignment, and this is seen in the difference between the random orderings. The improvement on unsatisfiable formulas is much smaller between the best and worst random ordering.

The PAR2 score includes the preprocessing time, so the apparent loss in Figure 4.3 for Proximity is made up for with improved solving. Whereas the Graph sorting solves more formulas than Natural but with a worse PAR2 score, meaning the literal orderings were effective but the preprocessing time to apply the community detection algorithm was often not worth the cost. The

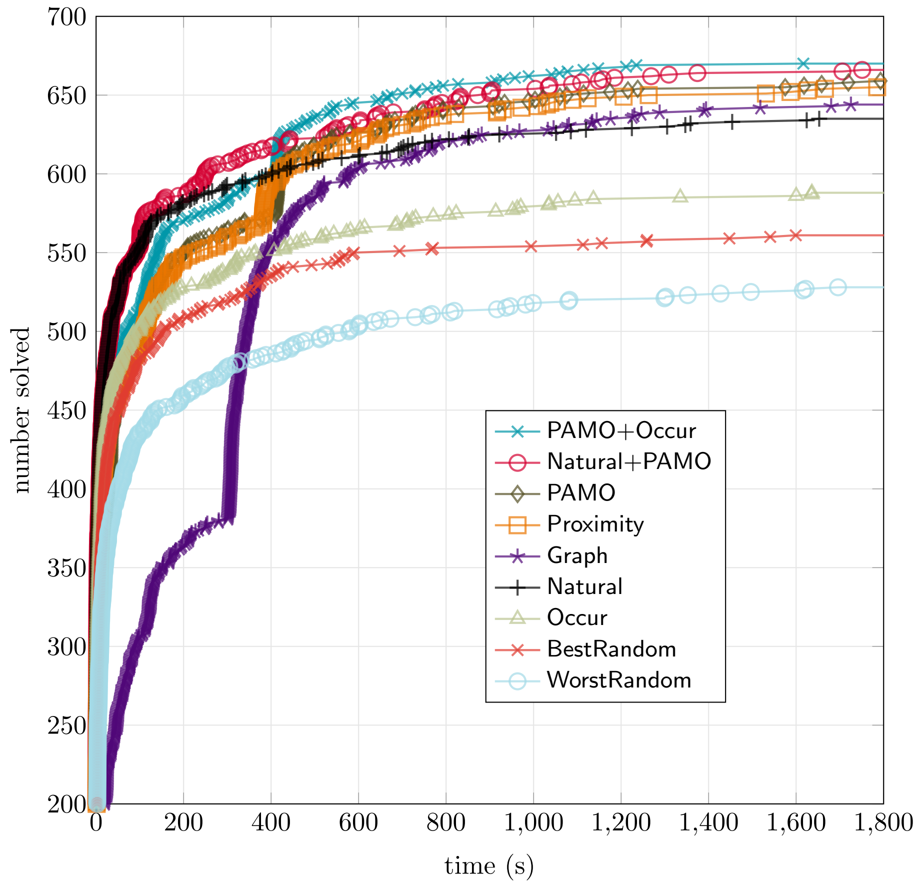


Figure 4.4: Number of solved formulas with the KMod-Totalizer configurations from Table 4.2, combining the satisfiable and unsatisfiable formulas.

preprocessing time also explains the reason behind Natural accumulating far more best solving times than the other approaches. With no overhead, Natural will start solving almost immediately and be the first to solve problems where the Natural order is as good (if not better) than the other more costly approaches. Both PAMO+Occur and Natural+PAMO have no best solving times because they would be binned in PAMO or Occur and Natural respectively. Even with the additional time to extract AtMostOne constraints, PAMO solves many problems faster than any other approaches. The Graph ordering can find good orderings but does not have a large share of best runtimes, so its orderings are often just as good as the Proximity approaches.

Figure 4.4 highlights the initial cost of preprocessing, with more complicated orderings like PAMO and Graph lagging behind Natural in the first couple hundred seconds. Then, for harder problems where the ordering becomes more important, the better techniques surpass Natural around the 400 second mark. This is slightly more delayed for the Graph technique that spends more time in preprocessing.

4.3 Reencoding ExactlyOne Constraints

Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule. *Reencoding Unique Literal Clauses*. In *Theory and Applications of Satisfiability Testing (SAT)*. (2025). [96]

In Section 4.1 we discussed the extraction and reencoding of AtMostOne cardinality constraints, then in Section 4.2 we presented ways to improve general cardinality constraint encodings with literal sorting. In this section, we will combine and extend both concepts, using the extractor from Section 3.3 to detect ExactlyOne constraints, then aligning these constraints via literal sorting prior to reencoding.

As mentioned before, the Direct encoding for ExactlyOne constraints does not use auxiliary variables and therefore does not allow reasoning over groups of variables. Other ExactlyOne encodings, such as the Sequential Counter encoding and the Order encoding include such auxiliary variables. The Order encoding has been shown to improve solver performance on various problems [37, 44, 87, 105], and the CSP solver Sugar employs the Order encoding [104]. The wide-ranging success of these encodings makes them good targets for reencoding.

For our reencoding technique, we will first detect and XLCs and perform variable elimination to remove clashing pairs of ULCs. Then, we will replace an XLC clause $(\ell_1 \lor \cdots \lor \ell_r)$ with the ExactlyOne variant of the Sequential Counter encoding. The new encoding replaces one clause by roughly $3k$ clauses, but any clauses of the form $(\overline{\ell}_i \lor \overline{\ell}_j)$ become redundant and may be deleted if present.

We can take the reencoding one step further and transform the Sequential Counter encoding into the Order encoding by eliminating the data variables from the formula using variable elimination. This process implicitly maps the s_i variables of the Sequential Counter encoding to the $o_{\leq i}$ variables of the Order encoding.

4.3.1 Aligning ULCs and XLCs

It is clear from the results in Section 4.2 that sorting literals in a cardinality constraint will impact solving. Common ExactlyOne encodings are non-hierarchical and literal sorting for a single ExactlyOne may not have much impact. That said, literal sorting applied to multiple ExactlyOne constraints simultaneously in a way that preserves relationships between literals across constraints can have a large impact. We will call the literal sorting of multiple constraints simultaneously *alignment*. ULCs and proper XLCs are aligned using the same procedure, so in the remainder of this section we will refer only to XLCs.

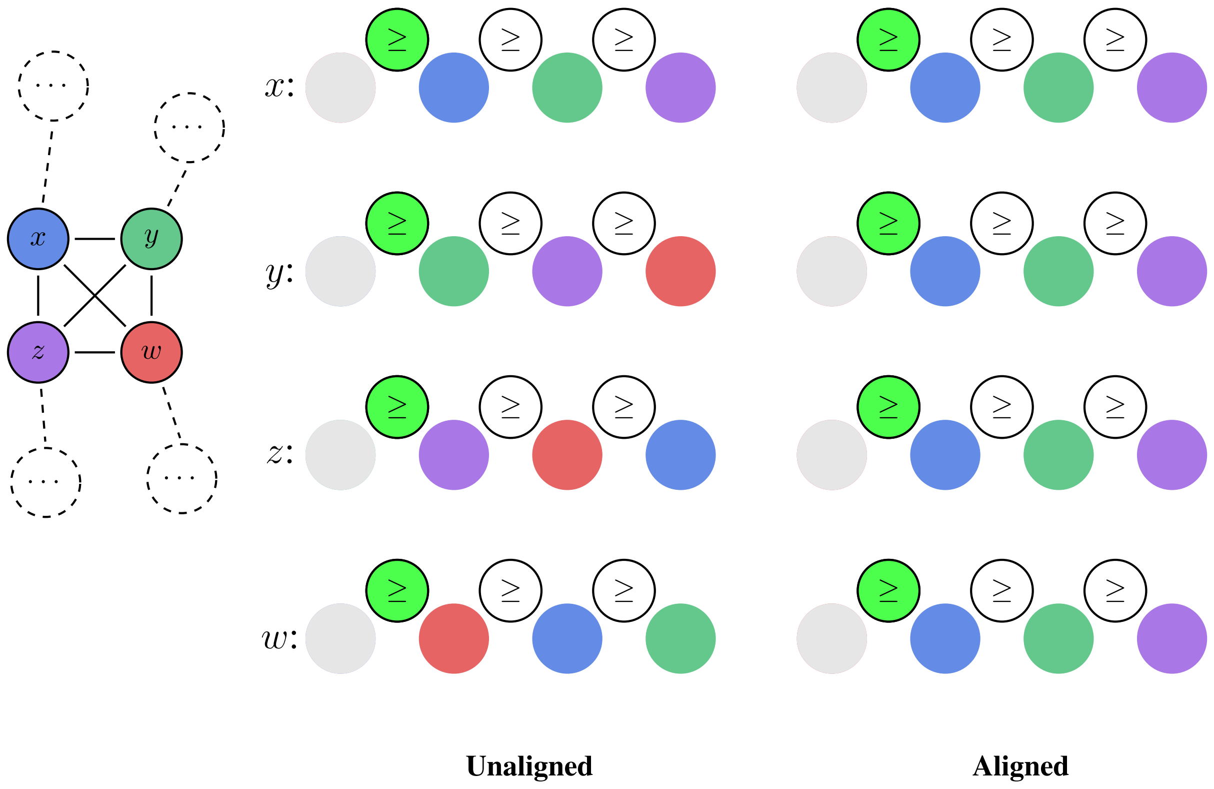


Figure 4.5: Unaligned and aligned ULCs for graph coloring a four clique x, y, z, w with four colors. The $\geq$ circles represent auxiliary variables from the sequential counter encodings for each node. The bright green $\geq$ circles are set `true`, meaning a given node must select a color from the right of the $\geq$ circle. In the unaligned set of ULCs, there is a possible coloring from the last three colors of each ULC, shown by the coloring on the graph to the left. In the aligned ULCs, there is no possible coloring when the auxiliary variables are set.

Example 4.7

Consider the graph-coloring problem in Figure 4.5 with four colors (r, b, g, p). The graph contains a 4-clique among the vertices x, y, z, w, and additional nodes and edges that are not shown. Assume the variables n^c are `true` if node n has color c. The direct encoding contains the ULCs $(x^r \vee \ldots \vee x^p)$, $(y^r \vee \ldots \vee y^p)$, $(z^r \vee \ldots \vee z^p)$, $(w^r \vee \ldots \vee w^p)$. The variables x^i denote that vertex x has color i.

The sequential counter auxiliary variables are denoted by the $\geq$ nodes, and the auxiliary variable is `true` for a node x when the color of x is not selected from the colors to the left of the auxiliary variable. If the first auxiliary variable for each of x, y, z, and w are set to `true`, that would mean each of these nodes have three remaining colors to select from. If the XLCs are aligned (shown on the right of Figure 4.5), then indeed the four-clique cannot be colored with only blue, green, and purple leading to a contradiction. This would allow the solver to learn a clause stating that at least one of the nodes selects the first color (red).

The XLCs may be unaligned, meaning the colors of variables between XLCs do not line up by column. So, even if each node has three colors to choose from, since the colors are out of order it is possible to find a coloring on the four clique, shown in the graph of Figure 4.5. Therefore, you cannot learn the short clause stating at least one of the nodes selects their first color.

Next, we provide a set of definitions that make the concept of alignment more concrete. Let F be a formula whose XLCs are $\{C_1, \ldots, C_m\}$. We say that F is aligned if for every pair of binary clauses $(\overline{a} \vee \overline{b}), (\overline{c} \vee \overline{d}) \in F$ with $a, c \in C_i$ and $b, d \in C_j$, it holds that literal a occurs before c in C_i if and only if b occurs before d in C_j.

A formula is *independent* if it is trivially aligned, that is, it does not contain any binary clauses sharing variables from different XLCs. If a formula is not independent and there exists a permutation of the literals in its XLCs such that it becomes aligned the formula is *alignable*, and otherwise the formula is *unalignable*.

These definitions are somewhat arbitrary; we could instead consider all clauses that share literals from different XLCs and not only binary clauses. Also, we could have gradations of alignability instead of an all-or-nothing classification. However, the experimental results suggest that alignment on binary clauses, as well as our crude classification scheme, do well to separate problems for which alignment and reencoding can improve performance.

Alignment Algorithm

To tackle the alignment problem, we must sort literals in all XLCs simultaneously. This allows us to create encodings that introduce auxiliary variables with meanings that are aligned amongst the various constraints so that a solver can more effectively reason abstractly over multiple encodings at once.

The alignment procedure makes use of a graph VIG^{-}_{bin}, which is the standard variable incidence graph containing binary clauses except that a binary clause $(\overline{\ell}' \vee \overline{\ell}) \in \Gamma$ is only included in VIG^{-}_{bin} if ℓ and ℓ' occur in distinct XLCs. Next, The literals within XLCs are sorted by variable ID (the natural ordering) to avoid noise from clause shuffling or preprocessing. XLCs are processed in order based on size, from largest to smallest. For each $C \in XLCs$ and for each $\ell \in C$:

- If ℓ is in an alignment set, continue.
- Otherwise, ℓ is placed in a new alignment set. This set is labelled by a counter that tracks the total number of sets.
- All literals in the connected component of VIG^{-}_{bin} containing ℓ are also placed in the same set as ℓ. If any of these literals also occurs in C, the problem is marked as unalignable.

Once all XLCs are processed, each literal will be in an alignment set with labelled by the set's counter and the clauses are sorted based on these label values. No two literals from the same XLC will be in the same alignment set unless the problem is unalignable. If the problem, is unalignable, we sort XLCs lexicographically by alignment set then variable ID. If there are no edges in VIG^{-}_{bin}, the entire procedure can be bypassed since it will result in the natural ordering by variable IDs.

Example 4.8

Consider the following formula:

$$C_1 : (x_1 \vee x_2 \vee x_3) \wedge C_2 : (x_4 \vee x_5 \vee x_6) \wedge (\overline{x}_1 \vee \overline{x}_6) \wedge (\overline{x}_2 \vee \overline{x}_4) \wedge \Gamma$$

Assume C_1 and C_2 are the only XLCs in the formula and that Γ does not contain any binary clauses having negated literals from C_1 and C_2. The first step is placing literals in alignment sets. We consider the literals from C_1 in natural order. First x_1 is placed in alignment set 1, then x_6 is placed in the same set because of the binary clause $(\overline{x}_1 \vee \overline{x}_6)$. Next, x_2 is placed in a new alignment set 2, along with x_4. Then, x_3 is placed in a new alignment set 3. All literals in C_1 have been placed in alignment sets and so C_1 can be sorted as $(x_1 \vee x_2 \vee x_3)$. When C_2 is processed, both x_4 and x_6 are skipped because they already appear in alignment sets. x_5 is placed in a new alignment set 5, and the clause is sorted as $(x_6 \vee x_4 \vee x_5)$. If the binary clause $(\overline{x}_2 \vee \overline{x}_5)$ was added to Γ, the formula becomes unalignable because x_4 and x_5 would both be placed in the alignment set 2.

The graph VIG^{-}_{bin} is never constructed explicitly in the implementation; instead, it suffices to process binary clauses lazily using an occurrence list. Each binary clause is still processed at most once: after a binary clause is processed both literals will be placed in an alignment set so if one of the literals appears in a XLC that has yet to be aligned it will be skipped during its turn in the algorithm. There are multiple possible alignments for a set of alignable XLCs. For instance, in Example 4.8, ℓ_5 can be placed at the beginning of C_2 and the XLCs would still be aligned. In general, the alignment sets can be assigned counters in any way, then the sorting will be performed based on the new ordering of alignment sets.

The alignment procedure will always produce an aligned formula if one exists (i.e., the problem is alignable). In short, the procedure places all literals within the same connected component into the same alignment set, and this preserves the definition of alignment.

4.3.2 Proof Production

In this section, we describe DRAT proof logging for our XLC reencoding framework. Note, while deletions need not be checked in a refutation, they may be important if the formula is satisfiable. In several places we apply DRAT deletion steps (e.g., when eliminating variables in clashing ULCs), and those steps must be logged in order to reconstruct solutions for satisfiable formulas.

The first step of our method, described in Section 3.3.2, makes the formula clash-free. In the case that resolution on a pair of clashing ULCs results in a new ULC, the new ULC is added as a RUP step in the proof; otherwise, no new clauses are added. In either case, the clashing ULCs are blocked on the clashing literal and are deleted from the formula. Since clauses are sets of literals, one can shuffle the literals around without any proof logging, so aligning is not explicit in the proof.

The reencoding consists of four stages for the Sequential Counter encoding and an additional fifth stage for the Order encoding. We will explain the proof logging procedure for reencoding ULCs on a clash-free formula. The procedure for XLCs is similar but somewhat more complicated.

1. Make the ExactlyOne constraint explicit. Before reencoding a ULC $(\ell_1 \vee \cdots \vee \ell_r)$, we need all the binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ with $1 \leq i < j \leq r$ to be present in the formula. Each of the missing ones can be added using blocked clause addition. For ULCs (and not proper XLCs), we can use the Linear encoding derivation instead of the Pairwise encoding for a shorter proof.
2. Add the definitions. Next, we add the definitions of the Sequential Counter encoding: $s_1 \leftrightarrow \ell_1$ and $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$ for $1 < i < r$. The clauses corresponding to these definitions can be added using blocked clause addition by adding them in increasing order of i.
3. Express AtMostOne and AtLeastOne. Now, we need to express that exactly one of the $\ell_1, \ldots, \ell_r$ literals can be `true` using the new definitions. For the at-most-one part, the clauses $(\overline{s}_{i-1} \vee \overline{\ell}_i)$ are included for $1 < i < r$, while for the at-least-one part we add the clause $(s_{r-1} \vee \ell_r)$. All these clauses are RUP w.r.t. the formula and can be added in arbitrary order.
4. Remove the original clauses. At this point, we no longer need the original clauses, including the added binary clauses, and we remove them. After the prior step, these clauses become RUP, so deleting them will not introduce additional satisfying assignments.
5. Eliminate the original variables. In case of a transformation to the Order encoding, we need to apply variable elimination on all the original variables in the direct encoding. When reencoding ULCs, this step will decrease the number of clauses. However, when reencoding XLCs, this step could substantially increase the number of clauses.

The main difference between reencoding ULCs and proper XLCs is the need for careful accounting in the latter case. It may not be allowed to remove some of the original binary clauses.

4.3.3 ULC and XLC reencoding in CaDiCaL

We implement XLC reencoding inside the state-of-the-art solver CaDiCaL [23]. Reencoding is performed as preprocessing immediately before entering the CDCL loop. By default, no preprocessing is performed in CaDiCaL; however, the order of literals within clauses may be changed by unit propagation during parsing and by propagation during lucky search [94]. We perform reencoding after lucky search so that trivial problems are still solved quickly.

1. Classify ULCs and XLCs
2. Perform variable elimination on variables in clashing ULCs
3. Optionally, align all ULCs and XLCs
4. Add the Sequential Counter encoding on ULCs and non-clashing XLCs
5. Remove tautologies and clauses made redundant by the Sequential Counter
6. Optionally, perform variable elimination on original variables to create the order encoding

The first two steps were described in the section on XLC extraction (see Section 3.3).

In the third step, we optionally align XLCs. Alternatively, the user may specify the natural or a shuffled ordering for literals within XLCs. The ordering of literals within each XLC will determine which clauses are generated in the Sequential Counter encoding. Even for independent formulas we apply the natural ordering because propagations during the parsing and lucky search can change the ordering of literals within a clause since newly watched literals are swapped to the front.

Before the fourth step of reencoding we perform several checks. We encode all XLCs of size 5 and up, preventing the reencoding of small XLCs that will have little impact on the formula. If the clause is an XLC and not a ULC, we must check if the XLC is clashing on a unique literal with another XLC. If so, we can only encode one of the XLCs, and choose to encode the first one that is seen. Finally, before adding the Sequential Counter encoding, we may first need to add clauses to the proof to ensure that the sequential encounter encoding can be derived. This is only necessary if the binary clauses described in Step 1 of Section 4.3.2 do not exist in the formula, in which case we can add them directly or can add a more compact set of clauses.

In the fourth step, we reencode each XLC by adding the Sequential Counter encoding to the formula. We generate new variables for the encoding with IDs starting from one plus the maximum variable ID in the formula. This strategy of adding new variables is not viable in the incremental setting when new variables may be added through the API between solver calls, but this limitation can be resolved and is not the focus of this work.

In the fifth step, we remove tautologies and redundant binary clauses. We use a time stamp to make this procedure efficient by marking the negation of literals occurring together in Sequential Counter encodings. For example, binary clauses formed of two negated literals from the same XLC can be removed after encoding the Sequential Counter, and this can be done by checking if the time stamp of both literals in the binary clause is the same. However, if a literal appears in multiple XLCs, it will have an updated time stamp. In this case, we must delete the binary clauses using a lookup table. Further, any binary clause added to the proof before adding the Sequential Counter encoding is deleted. Finally, we delete all reencoded XLCs from the formula.

In the sixth step, we optionally perform variable elimination on the literals occurring in reencoded ULCs. This will transform the Sequential Counter encoding into the Order encoding. Variable elimination for all ULCs is done in a single pass over the clauses in the formula by using a variable substitution mapping. For each clause containing a negated literal from a ULC, literals in that clause are mapped to their replacements in a new clause, and the old clause is deleted. Note that some negated literals from ULCs will be replaced by two literals from the Sequential Counter encoding, and this could turn many binary clauses into clauses of length three or four, potentially slowing down solver propagations. Then, clauses from the Sequential Counter encodings that contain ULC literals are deleted. While we perform variable elimination by hand, it is possible that the solver would have eliminated some or all of these literals in the bounded variable elimination (BVE) inprocessing steps. On the other hand, the solver may eliminate auxiliary variables from the Sequential Counter encoding.

All of our clause additions and deletions use the DRAT proof API inside CaDiCaL. This uses the extension stack for RAT deletions to reconstruct solutions for the original set of variables given a solution for the encoded formula. CaDiCaL supports proof logging in LRAT, and in future work, we plan to support LRAT proof production.

The reencoding preprocessor is called once before solving and checks every clause in the formulas for the XLC criteria. In addition, this could be performed as inprocessing [59]. The solver cannot learn a ULC directly through conflict analysis, because prior to clause learning, the literals in the learned clause must have been pure and therefore fixed. However, techniques like subsumption, probing, or variable elimination that remove literals from clauses and delete clauses may produce ULCs. To detect ULCs in inprocessing, it would suffice to check the occurrence counts for irredundant clauses. If an irredundant clause meets the ULC criteria, we delete all redundant clauses containing the literals from the ULC. Then, this clause could be reencoded as a ULC. Also, the solver could learn binary clauses that allow some other clause to meet the XLC criteria. We do not implement these inprocessing procedures because the preprocessing approach already performs well on a wide range of problems containing ULCs and does not improve performance much on problems containing XLCs.

Table 4.3: Number of formulas reencoded, broken down by alignment type, average preprocessing time, number of formulas taking longer than 20 seconds, the medium number of encoded XLCs, and the medium of the maximum sizes of encoded XLCs. SBVA is only run on the 1739 formulas for which an XLC was detected.

	Reencoded	Alignable	Ind.	Unalign.	Avg. (s)	$\geq$ 20s	Med. #	Med. Max
ULC-clash	891	415	145	331	4	29	760	45
ULC	1014	459	216	339	4	43	760	40
XLC-proper	825	305	187	333	1	9	1001	70
XLC-full	1739	791	294	654	3	49	867	52
SBVA	1616	679	314	623	30	356	–	–

Table 4.4: Performance of ULC reencoding across different configurations, with a virtual best solver (VBS) picking the best configuration for each formula. *SBVA uses original problem for the 123 formulas it does not find a pattern to reencode.

	Alignable (459)				Independent (216)				Unalignable (339)			
	# Solved		PAR2		# Solved		PAR2		# Solved		PAR2	
	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS
Original	118	123	1652	3081	**62**	90	**593**	823	**122**	103	1534	1534
SEQSHUFF	124	122	1237	3066	58	90	1173	899	121	102	1811	1596
SEQNAT	125	137	1109	2088	56	**91**	1364	825	**122**	102	**1526**	1560
SEQALI	**127**	152	**951**	1159	56	**91**	1364	831	**122**	103	1544	1470
ORDALI	125	**155**	1149	**993**	58	89	1225	1011	115	90	2019	2595
SBVA*	122	137	1395	2068	61	**91**	616	**713**	120	**107**	1687	**1108**
VBS	137	163	256	513	63	93	301	471	137	112	293	672

4.3.4 Results

In this evaluation we used the 5,355 formulas from the SAT Competition Anniversary Track. For solving we used a 5,000 second timeout. SBVA was allowed to run for 200 seconds (if it does not terminate by then) before solving the reencoded formula with CADICAL as in the original paper [49]. All runtimes include reencoding time.

Extraction with Alignment Classifications

In Table 4.3 we present the extraction results for ULCs and XLCs alongside the formulas' alignment classifications. Of the 1,739 formulas, 1,014 ($\sim$ 19%) contained ULCs and 459 ($\sim$ 9%) of those are alignable. These will be the problems for which our reencoding technique yields the biggest improvements.

We also show the number of formulas for which SBVA finds a pattern to reencode. It is no surprise that SBVA finds patterns in many of the same formulas that contains XLCs, since XLCs are often accompanied by sets of binary clauses that constrain the objects between them. We will see later that while SBVA can extract and reencode some patterns in these formulas, it cannot perform as well as our XLC extraction and reencoding framework.

ULC Reencoding

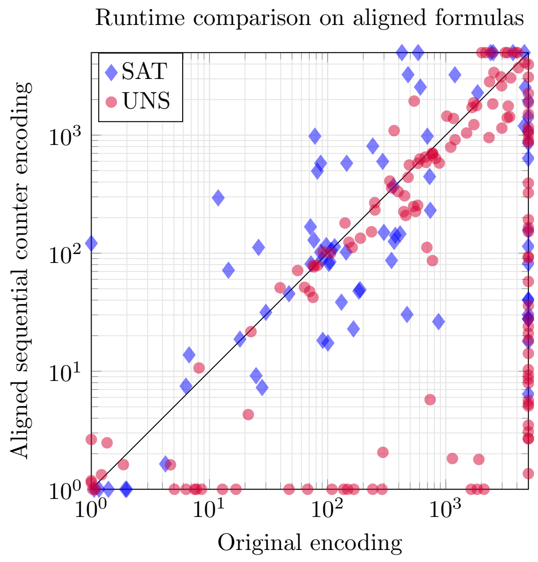
The different options for reencoding, each of which performing variable elimination on clashing ULCs prior to reencoding, are as follows:

- SEQSHUFF: Literals in each ULC are shuffled then reencoded with the Sequential Counter encoding.
- SEQNAT: Literals in each ULC are sorted by the natural ordering then reencoded with the Sequential Counter encoding.

- SeqAli: Literals in each ULC are aligned then reencoded with the Sequential Counter encoding.
- OrdAli: Literals in each ULC are aligned then reencoded with the Order encoding.

Table 4.4 presents the solving performance for the various reencoding options on the 1,014 formulas containing at least one ULC of size 5 or larger. Results are broken down by alignment type and by the satisfiability of the formula. Reencoding is most effective on both satisfiable and unsatisfiable formulas that are alignable. On the independent and unalignable formulas the original encoding performs almost as well if not better than the reencoding configurations. This suggests that the Sequential Counter encoding is not a one-size-fits-all solution but instead should be applied to certain problems where relationships between constraints can be exploited. Furthermore, the encoding alone is not enough, it requires alignment to achieve superior performance. SeqShuff solves far fewer unsatisfiable formulas than SeqAli on the alignable formulas. In the previous section we showed that shuffling on AtLeastK constraints could significantly degrade solver performance, but for this application suboptimal reencoding with shuffling still performs on par with the original encoding and is often not harmful.

The impact of alignment versus shuffling on individual alignable instances can be seen in the two plots of Figure 4.6. Alignment leads to many more solved unsatisfiable formulas (points on the y-axis), and in some cases solving times that are thousands of times faster than the original encoding (points on the x-axis). Many unsatisfiable formulas are solved slightly faster when comparing the shuffled reencoding to the original encoding, again suggesting that some encoding is better than no encoding.



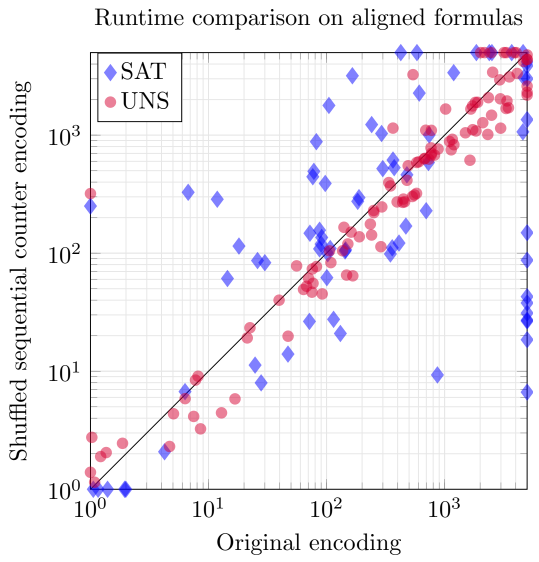


Figure 4.6: Sequential Counter encoding on alignable instances: Aligned vs. original (left) and shuffled vs. original (right).

Shuffled Benchmarks

Some benchmarks submitted to the SAT competition are shuffled to encourage robust solver development. Shuffling may involve reordering clauses, renaming variables, flipping literals, or reordering literals within a clause. We classified a subset of competition benchmarks as shuffled if the keywords "Shuffled" or "Random" appeared in their name. Table 4.5 shows solver performance on this subset of benchmarks. Alignment has an outsized impact on this set of benchmarks, solving an additional 18 unsatisfiable formulas. However, benchmarks do not need to be artificially shuffled for alignment to have an impact. On the full set of alignable benchmarks SeqAli solves over 30 more instances than the original encoding.

Independent and Unalignable Formulas

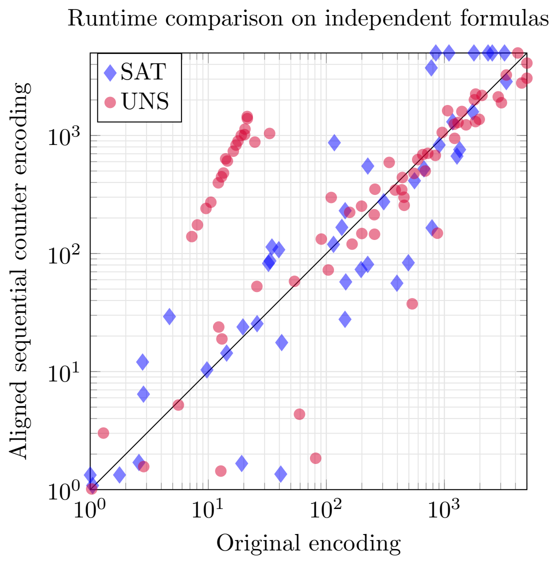
Figure 4.7 shows how reencoding on independent or unalignable formulas often does not improve the performance of the solver on unsatisfiable instances, and in some cases, can make performance worse. Satisfiable instances are generally more fragile, a small perturbation to the formula could interact with solver heuristics in such a way that it gets lucky and finds a solution quickly. Note, the alignment procedure is performed before reencoding, so in practice the our tool can check the alignment type of the formula and forego any reencoding on unalignable and independent formulas if we believe reencoding may worsen performance.

Order Encoding

We can take the reencoding one step further by replacing the Sequential Counter encoding with the Order encoding. This process not only adds clauses but replaces all instances of original variables from the ULCs with the new encoding variables through a sequence of variable eliminations. Even if we only add the Sequential Counter encoding, some of these variable eliminations may occur during BVE in the solver's inprocessing stages. However, the solver may choose to eliminate some of the auxiliary variables from the Sequential Counter instead, making it impossible to delete the remaining original variables.

Table 4.5: Number solved on alignable formulas with "Shuff" or "Rand" in benchmark name (satisfiable: 63, unsatisfiable: 76, unknown: 74)

	# Solved		Avg. PAR2	
	SAT	UNSAT	SAT	UNSAT
Original	62	55	159	3341
SeqShuff	**63**	57	**21**	3016
SeqNat	62	58	158	2891
SeqAli	**63**	72	22	1018
OrdAli	**63**	**73**	30	**906**
SBVA	62	60	158	2458
VBS	63	76	21	566



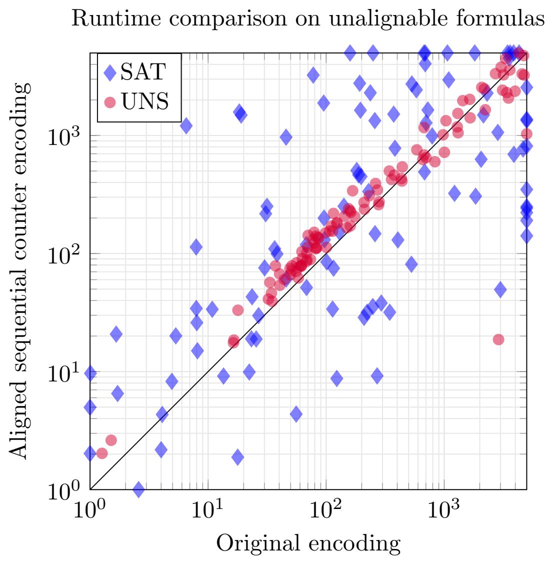
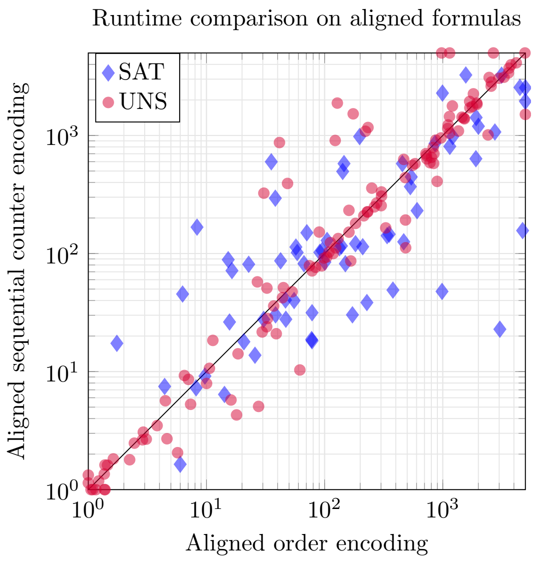


Figure 4.7: Runtime comparison between the aligned Sequential Counter encoding and the original encoding on independent formulas (left) and unalignable formulas (right). The cluster of UNSAT formulas in the top left of the independent plot are the shift1add formulas mentioned in Section 4.3.3.

Figure 4.8 shows the comparison between the Sequential Counter encoding and the Order encoding on aligned formulas. The runtimes for both approaches appear similar with a few deviations, but the number conflicts from each approach are almost identical. The difference in runtimes also suggest that when the Sequential Counter encoding is applied, the solver does not always eliminate the original variables in ULCs and sometimes it is better to keep them. Furthermore, in Figure 4.9, the runtimes become consistently worse for the Order encoding but the number of conflicts are mostly identical. So, the Order encoding can produce a similar number of reasoning steps to solve a problem, but it takes longer. One cause of this might be that the Order encoding replaces many binary clauses containing original variables with ternary clauses containing order variables, and the ternary clauses take longer to propagate.

Comparison with Structured Bounded Variable Addition

Figure 4.10 shows a comparison between our best approach and SBVA. The Sequential Counter encoding with alignment can solve dozens of alignable formulas that SBVA cannot solve, suggesting that they reencode the formulas differently. The Sequential Counter reencoding also improves performance on many satisfiable problems, showing that alignment can improve performance across the board. On unalignable and independent formulas, the results are mixed, where several formulas are solved faster with SBVA. Note, SBVA may find patterns that share no correspondence to ULCs, and for these cases it is possible to run ULC reencoding then SBVA on the reencoded formula.



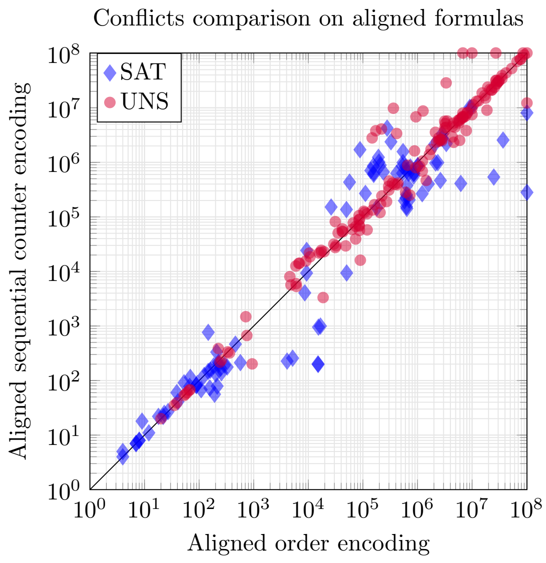
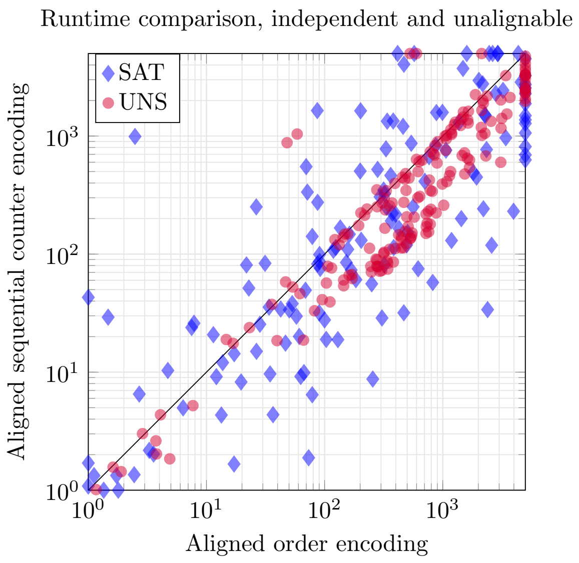


Figure 4.8: A comparison in runtime (left) and the number of conflicts (right) between the Sequential Counter encoding and the order encoding on alignable formulas with ULCs.



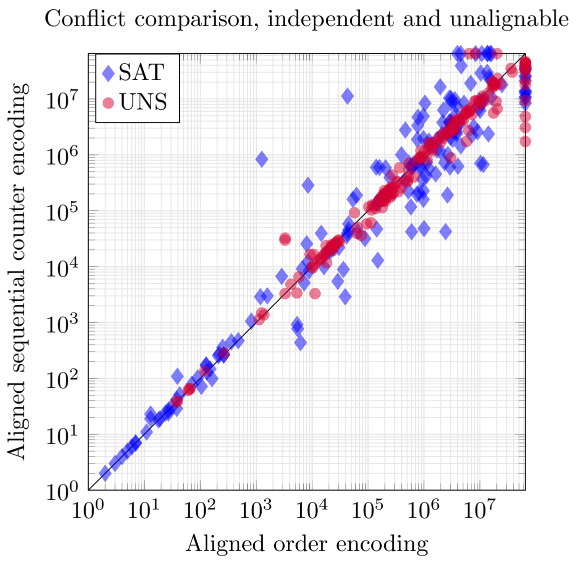


Figure 4.9: A comparison in runtime (left) and the number of conflicts (right) between the Sequential Counter encoding and the order encoding on unalignable and independent formulas with ULCs.

XLC Reencoding

Table 4.6 and Figure 4.11 show the performance of Sequential Counter encoding on formulas containing proper XLCs. For this set of formulas, reencoding improves unsatisfiable instances but harms satisfiable instances. This is consistent with the results for AtMostOne reencodings. Proper XLCs contain the full Pairwise encoding and reencoding them using auxiliary variables

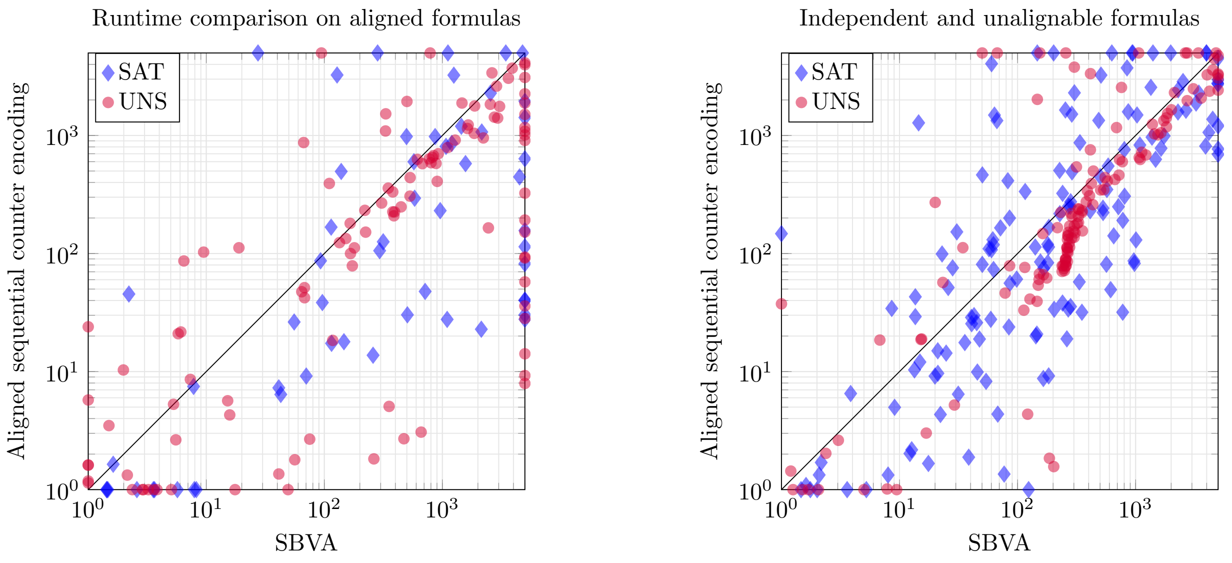


Figure 4.10: A runtime comparison between aligned Sequential Counter encoding and SBVA on alignable formulas (left) and unalignable and independent formulas (right). The runtime includes the reencoding time. Formulas for which SBVA does not perform reencoding are filtered out.

can be detrimental in the satisfiable case. The impact is also less pronounced on this set of alignable formulas than it was for alignable formulas containing only ULCs. For these formulas, it appears that having any explicit ExactlyOne encoding whether it be Direct or the Sequential Counter is enough to improve performance. For the independent and unalignable formulas there is an improvement on unsatisfiable formulas when using reencoding. Again, this translates to the AtMostOne reencoding work where a replacement of the Pairwise encoding with an encoding containing auxiliary variables can be beneficial for unsatisfiable formulas.

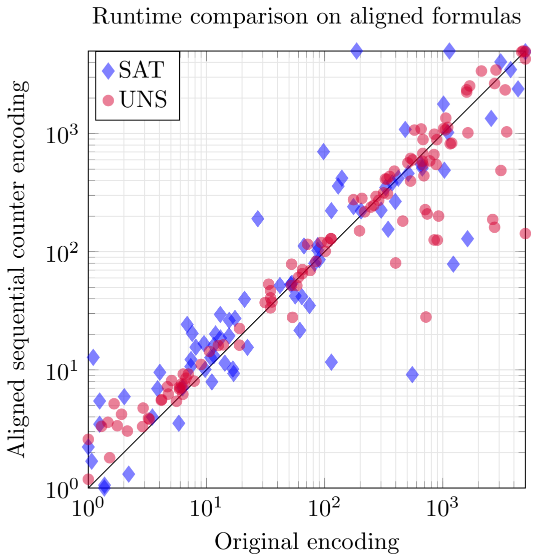
Benefited Benchmark Families

In this section we take a closer look at alignable benchmark families where reencoding provided significant benefits.

- **Pigeonhole** Pigeonhole formulas try to put n pigeons into m holes and consist of many ULCs (every pigeon must be in one hole). This problem is provably hard for resolution and

Table 4.6: Performance of XLC reencoding

	Alignable (305)				Independent (187)				Unalignable (333)			
	# Solved		PAR2		# Solved		PAR2		# Solved		PAR2	
	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS
Original	**119**	132	**318**	723	**102**	47	**644**	1461	**144**	135	**263**	471
XLC	118	**134**	397	**525**	100	**48**	762	**936**	143	**138**	343	**229**
VBS	120	135	207	433	105	49	270	783	145	138	155	206



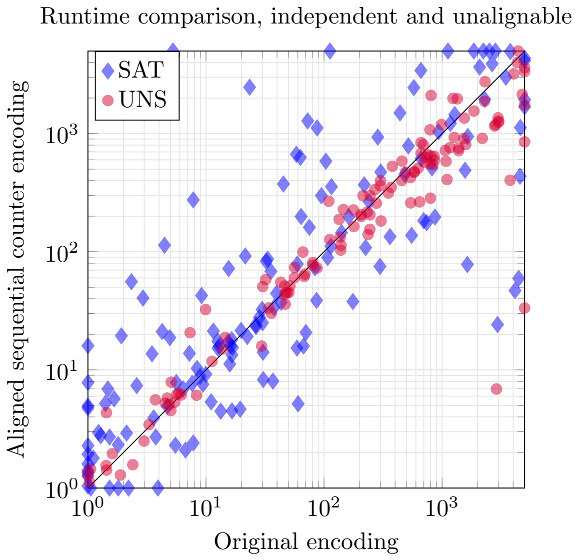


Figure 4.11: Performance comparison between the Sequential Counter and the original encoding on formulas with at least one proper XLC. Left, the alignable formulas and right, the unalignable and independent formulas.

will always have exponential growth for resolution-based learning, but the better encodings for can significantly improve the runtime on smaller instances. For example, consider the formula `php-018-014.shuffled`, a shuffled formula that tries to put 18 pigeons in 14 holes. SBVA times out whereas alignment and reencoding leads to a 192 second runtime. Similar speed ups were found in related families, such as relativized pigeonhole formulas.

- **Graph Coloring** Another family is graph coloring, also having many ULCs (every vertex has one color). Our method works remarkably well on this class of problems. For example, our method solved `queen14_14.col.14`[55] in 9 seconds and `le450_15b.col.15` in 28 seconds, while SBVA and original encoding timed out on both.
- **FPGA-routing** These formulas are Field-Programmable Gate Arrays routing constraints encoded as large SAT problems, in which satisfying assignments correspond to feasible routing solutions. The ULCs in these formulas encode connectivity constraints to ensure the existence of a conductive path for each two-pin connection [80]. Reencoding ULCs shows significant performance improvement on almost all such formulas. For example, `homer19.shuffled` timed out with original encoding and took SBVA 75 seconds, but with our method it was solved in 2 seconds.
- **Petri Net Concurrency** Petri nets are a model for concurrent computation that uses places and transitions [26]. Each formula represents whether there exists a valid partitioning of the Petri net's places into distinct subsets (units) respecting a given concurrency relation. The ULCs in these formulas encode that every place belongs to a unit. Our method dramatically improved solver performance on these formulas. For example, `vlsat2_30744_3925645.dimacs`[14], a formula that timed out with both the original encoding and SBVA, was solved by our method in 177 seconds.

We also noticed that the majority of unalignable formulas belong to the Graph/Subgraph Isomorphism family. These formulas consider two random graphs and the question of whether one is a subgraph of the other. This is encoded by searching for a permutation of one of the graphs such that they overlap [9]. The formula contains many unalignable ULCs encoding the permutations.

4.4 Remarks on Encodings

In this chapter, we presented three ways of (re)encoding cardinality constraints. First, we showed that simply reencoding Pairwise AtMostOnes into Linear AtMostOnes could improve performance on many unsatisfiable problems while potentially lowering performance on satisfiable problems. Second, we presented several techniques for sorting literals within AtLeastK constraints, finding that a good literal ordering is more important than the choice of AtLeastK encoding for improving performance. Third, we devised a technique for reencoding ExactlyOne constraints, and we showed that the alignment of sets of constraints prior to encoding was crucial for enhancing solver performance. Each of these studies looked at the importance of auxiliary variables in encodings from different perspectives, first as a simple reencoding in AtMostOnes, then as sorting within a single AtLeastK, and finally as the relationship between sets of ExactlyOnes. In each case, encodings could be improved with our automated techniques.

It would be difficult for users to come up with these encodings by hand. Indeed, in some cases the best encodings are hidden behind complex relationships between variables and constraints. Using KNF allows us to move these encoding questions out of the user's hands and into the solver. We can let our automated tools find the best encodings, instead of hoping we can extract and reencode a user's suboptimal encoding.

The next steps for improving cardinality constraint encodings are to combine the ideas of the three projects into a unified framework. We know that literal sorting within a single AtLeastK is important, and that alignment between multiple ExactlyOnes is crucial. Next, we can explore how to align multiple AtLeastK constraints within a formula. This would likely require a new type of encoding that could represent hierarchical relationships between groups of variables across AtLeastKs.

Chapter 5

Solving

There are several ways to solve a problem presented in KNF or some similar cardinality-based format. One of the most effective approaches, described in the previous chapter, is to use clausal encodings then a SAT solver. Clausal encodings are not one-size-fits-all and some problems require a stronger proof system for efficient solving; but, a typical tradeoff with cutting planes reasoning is that solver performance degrades on many problems that CDCL is generally good at. A minimally obstructive extension to standard CDCL is the addition of native cardinality constraint propagation. Several solvers have implemented native cardinality constraint propagation [70, 112], and these tools have excelled on satisfiable formulas. Unfortunately, these implementations came before the development of modern inprocessing techniques, did not account for proof generation, and were targeted at a small set of crafted instances.

We revisit native cardinality constraint propagation in the context of the modern CDCL ecosystem. Native propagation can be seamlessly incorporated into a CDCL solver with small changes to the clausal propagation algorithm, and the most used inprocessing techniques can be supported with minor modifications. We will describe our implementation of native propagation within the state-of-the-art solver CaDiCaL, discussing the main changes to the watched-literal data structures and analysis algorithm, as well as updates to the local search algorithm to provide cardinality constraint support. In addition, we will describe how native propagation can produce proofs in the DRAT proof format.

Our work on encodings showed that group reasoning made available through encodings is crucial for solving unsatisfiable problems. One benefit of KNF is that we can decide whether to encode constraints or propagate natively. We show that a hybrid configuration switching between native propagation and clausal encodings can help alleviate the performance gap between satisfiable and unsatisfiable instances.

The combination of native propagation and encoding highlights the versatility of the KNF input format. It will allow a user to solve very large problem with cardinality constraints that could not feasibly be encoded lest the formula become too large for a modern solver. However, our thesis also stresses usability, and providing native propagation as another option during solving would add yet another choice that a user must make. Thus, we developed our hybrid configuration to relieve this burden and make decisions about propagation and encoding within the solver.

5.1 Native Cardinality Constraint Propagation

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. *From Clauses to Klauses*. In *Computer Aided Verification (CAV)*. (2024). [91]

In this section, we describe cardinality-CDCL (CCDCL), an extension of CDCL with propagation on cardinality constraints. 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.

Example 5.1

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

The partial assignment $\overline{x}_1 = 1\, x_2 = 0$ forces the extension $x_3 = x_5 = 1\, x_4 = 0$ for K_1 to be satisfied. The partial assignment $x_4 = 1\, x_1 = x_2 = x_3 = 0$ forces the extension $x_5 = 1$ for C_1 to be satisfied. K_1 can propagate at most 3 literals, whereas C_1 can propagate at most 1 literal.

Example 5.1 shows the added propagation power of cardinality constraints over clauses, not to mention the many auxiliary variables that must be propagated in a clausal encoding. Cardinality constraints can be handled natively with minimal changes to a CDCL solver, and no changes to the proof system.

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

5.1.1 Native Propagation in CaDiCaL

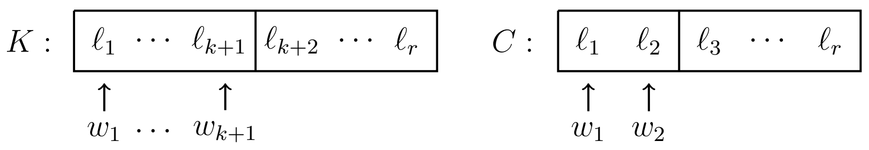

K :
ℓ_1 $\cdots$ ℓ_{k+1}
ℓ_{k+2} $\cdots$ ℓ_r
w_1 $\cdots$ w_{k+1}
C :
ℓ_1 ℓ_2
ℓ_3 $\cdots$ ℓ_r
w_1 w_2

Figure 5.1: cardinality constraint (left) of the form $\ell_1 + \cdots + \ell_r \geq k$ and clause (right) of the form $\ell_1 + \cdots + \ell_r \geq 1$, with watch pointers for the first $k+1$ literals in the cardinality constraint.

A cardinality constraint requires more watch pointers ($k+1$) than a clause (Figure 5.1), since $r-k$ literals must be falsified in order to propagate the cardinality constraint [70]. The invariant on a non-conflicting cardinality constraint is that at least k watched literals are either unassigned or satisfied. If this is not the case, then at least $r-(k+1)$ literals are falsified and therefore the cardinality constraint is falsified.

Propagation on clauses is unchanged. For a cardinality constraint, assuming the watch pointer in question is w_i for assigned literal $\overline{\ell}_i$, we have the three following cases:

1. If there exists an unassigned or satisfied literal starting from ℓ_{k+2}, it is swapped with ℓ_i, then w_i is released and a new watch is created for the swapped literal.
2. If no such literal exists, the watched literals $\ell_1, \ldots, \ell_{k+1}$ (not including ℓ_i) are assigned to `true`, and their *reason* is all of the literals $\ell_i, \ell_{k+2}, \ldots, \ell_r$ that are falsified.
3. If any of the would-be propagated literals are already falsified, then there is a conflict and the propagation algorithm exits and conflict analysis begins. The conflict clause contains the reason literals, ℓ_i, and the first falsified watched literal other than ℓ_i.

In the case of propagation with no conflict, it could be that some or all of the watched literals are already satisfied. There is no guarantee that k unassigned literals will be propagated, and often in practice only some of the watched literals are unassigned at the time of propagation. In the case of a conflict, if there are multiple falsified watched literals other than ℓ_i we only need to pick one to add to the conflict clause. Instead of selecting the first falsified watched literal, one could evaluate the effect of each literal on the size of the learned clause or compare the depth each of the falsified watched literals were assigned at.

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 falsified reason literals from the constraint that caused the propagation. 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 cardinality constraint. In the implication graph, the reason for the literal is exactly the literals that propagated the cardinality constraint. Since important properties of the implication graph are unchanged, clause minimization can be applied to learned clauses. We have not considered the effect of cardinality constraints on chronological backtracking, and therefore only allow backjumping.

5.1.2 Local Search with Cardinality Constraints

CaDiCaL uses Stochastic Local Search (SLS) for phase saving [23], based on the SLS solver ProbSAT [13]. The algorithm starts by generating a random assignment. It then randomly selects a falsified constraint and uses a heuristic to select a literal from that constraint to flip, thereby satisfying the constraint.

To adapt the ProbSAT algorithm to cardinality constraints we need to modify the heuristic for selecting a literal within a falsified constraint. The original heuristic is based on the break value of literals. The *break value* of a literal is the number of constraints that were satisfied with the current assignment and will be falsified if the literal is flipped. This heuristic is problematic for cardinality constraints since flipping a single literal may not satisfy or falsify a cardinality constraint but will still affect how close it is to being satisfied or falsified. Given a cardinality constraint with r literals, s satisfied literals and bound k, we consider three ways to compute the break value for cardinality constraints.

Linear : $k - s$

Multiplicative : $(k - s) \times r$

Quadratic : $(k - s)^2$

Thei break count is only computed for cardinality constraints that are falsified after flipping a literal within the constraint from `true` to `false`. A more thorough local search algorithm could account for satisfied cardinality constraints or cardinality constraints that become less falsified. However, since the SLS algorithm in CaDiCaL is used for phase-saving it is only run for a short period of time and is not meant to be a robust standalone SLS solver.

Example 5.2

Consider the following KNF formula:

$$(\overline{x}_1 + \overline{x}_2 + x_3 + x_4 \geq 2) \wedge (\overline{x}_1 + x_2 + x_3 + \overline{x}_4 + x_5 \geq 3) \wedge (x_2 \vee x_3)$$

Given the assignment $x_1 = x_3 = 1$, $x_2 = x_4 = x_5 = 0$, The break value for literal x_3 on the first cardinality constraint is 1 (linear), 4 (multiplicative), or 1 (quadratic), for the second cardinality constraint is 2 (linear), 10 (multiplicative), or 4 (quadratic), and for the clause is 1.

Example 5.2 shows that the multiplicative and quadratic break values deter a flip that over falsifies a cardinality constraint. This is especially pronounced for AtMostOne constraints, since the bound is $r - 1$.

In addition to break value, we may want to update the heuristics for picking a falsified constraint. We support three options:

1. Pick randomly from the set of falsified clauses, and if all clauses are satisfied pick a falsified cardinality constraint.
2. Pick randomly from the set of falsified cardinality constraints, and if all cardinality constraints are satisfied pick a falsified clause.
3. Pick a falsified clause or cardinality constraint at random.

These options will determine if the algorithm focuses on satisfying clauses or cardinality constraints.

In our implementation we support each option described above, and the default configuration uses linear weighted break values and random selection from the set of both clauses and cardinality constraints. The other configurations may be more valuable for problems containing many large AtLeastK constraints.

5.1.3 In-processing Techniques and Native Propagation

To support a selection of the most important inprocessing techniques, we split the constraint database into clauses ($k = 1$) and cardinality constraints ($k > 1$). 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 cardinality constraints. We allow bounded variable elimination (BVE) [39] on all variables not occurring in cardinality constraints. Variables in cardinality constraints are frozen [40] 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 cardinality constraints since we provide no corresponding inference rule for cardinality resolution. We allow vivification [68] 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 cardinality constraints during vivification.

There are additional inprocessing techniques that we plan to include in the future but are less important than the implemented techniques. One of the candidate techniques is Equivalent Literal Substitution (ELS). While we do not allow duplicate literals in a cardinality constraint, ELS could be performed by adding clauses for literal equivalence, then substituting literals in all clauses but not in cardinality constraints.

5.1.4 KNF Solving Configurations

CCDCL is our solving configuration that takes as input a formula in KNF and solves the formula with native propagation on cardinality constraints. The solver enables the inprocessing techniques described above, using the linear break values for SLS.

REENCODE is our baseline comparison for clausal encodings. It first encodes the cardinality constraints into clauses before applying standard CDCL. This uses the Linear encoding for AtMostOnes and Sequential Counter encodings for AtLeastKs.

HYBRID a CCDCL solver that takes as input a formula in KNF as well as the clausal encodings of all cardinality constraints from the KNF. The clausal encodings are kept throughout solving as if they were irredundant clauses from the original formula. The solver uses the internal mode switching heuristics to determine when native propagation is enabled. In SAT mode, cardinality constraints are watched and propagated, and in UNSAT mode, cardinality constraints are ignored. Since the clausal encodings are included, it is safe to forego native propagation on cardinality constraints. When cardinality constraints are watched they can potentially speed up the propagation algorithm and ignore auxiliary variables, helping the solver find solutions quickly for satisfiable instances. When cardinality constraints are unwatched, the solver has direct access to auxiliary variables from the encodings, benefitting unsatisfiable instances. In general, the solver moves back and forth between SAT and UNSAT modes with increasing limits and will spend roughly half of its time in either mode.

The three different solving configurations highlight the flexibility provided by the KNF format. In some cases, a smaller representation and fast propagation on cardinality constraints is beneficial. In other cases, encoding cardinality constraints introducing auxiliary variables can lead to a much shorter proof. And a combination of the two approaches may work best for unknown problems.

5.1.5 Proof Production

The three solving configurations and along with their proof-producing capabilities are shown in Figure 5.2. Currently, proofs generated by the KNF solver are checked against an arc-consistent encoded CNF using a standard DRAT checker. To increase trust, one can use a formally verified

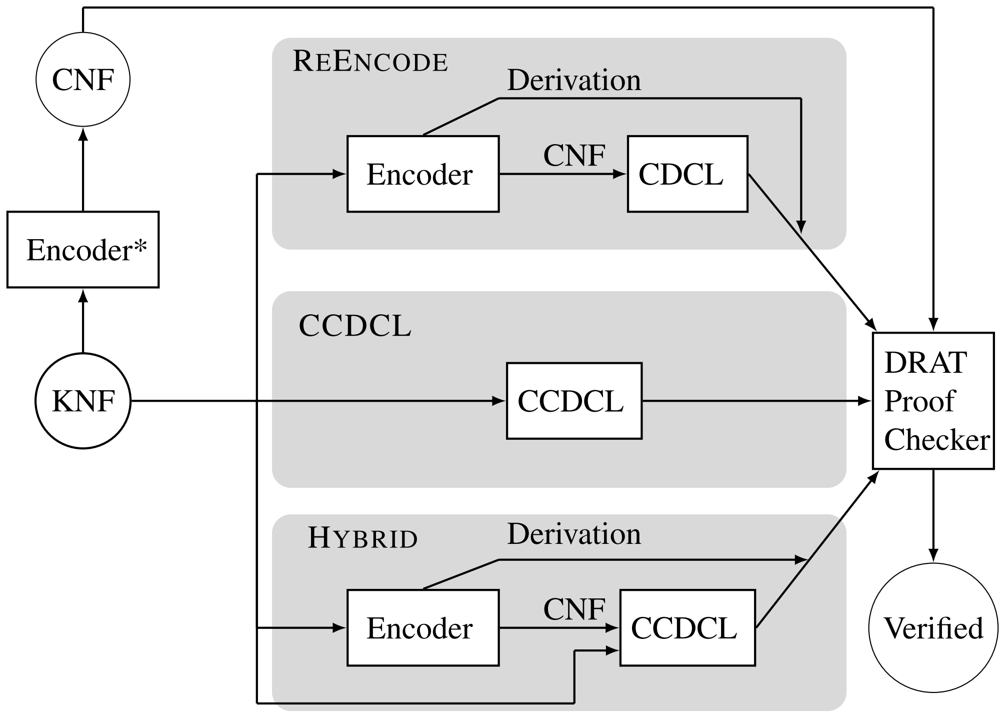


Figure 5.2: Three configurations for solving a KNF formula with DRAT proof production. Encoder* uses fresh variables and not those introduced by the solvers' encoders.

KNF to CNF encoder [33]. Alternatively, one could verify the transformation from KNF to CNF with a PB checker [48]. It would also be possible to export PB proofs from the solvers and user a PB checker like VeriPB, or to modify a DRAT checker to accept and propagate natively on cardinality constraints.

Satisfying assignments are easy to verify by checking that each constraint in the KNF is satisfied. For configurations where cardinality constraints are reencoded, the new clauses may contain auxiliary variables not appearing in the original KNF formula. These variables can simply be removed from the satisfying assignment produced by the solver.

Native Propagation

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 a cardinality constraint with r literals and bound k occurs – exactly when $r - k$ literals are assigned to `true`– and this will propagate the remaining literals to `false` for both the native propagation and an arc-consistent clausal encoding. This means that the speedups in propagation within the solver will be unwound in the proof checker, since the RUP check will need to trace the cardinality constraint's propagation over its encoded clauses. This is often not a problem since native propagation is most useful on satisfiable formulas which do not require proof checking.

Encoding

For configurations that make use of an encoder, we must generate a derivation of the encoded constraints proving they are redundant and can be added to the original KNF 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:

$$\mathsf{Deriv}(\ell_1, \ldots, \ell_s) = \mathsf{Pairwise}(\ell_1, \ell_2, \ell_3, y), (\ell_1 \vee \ell_2 \vee \ell_3 \vee y), \mathsf{Deriv}(\overline{y}, \ell_4, \ldots, \ell_s) \tag{5.1}$$

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

Extraction

The proof checking procedure looks very similar when starting from a CNF formula and using AtMostOne extraction to produce a KNF formula. The original CNF is passed to the proof checker instead of an encoded version of the KNF. As long as the extracted cardinality constraints were originally encoded with an arc-consistent encoding, the clauses produced by native propagation will be RUP for the reasons stated above.

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 cardinality constraint. 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 solver on the original CNF formula under the partial assignment given for the extracted KNF will produce a satisfying assignment that includes the auxiliary variables.

5.1.6 Results

We implemented the CCDCL algorithm on top of the award winning CDCL solver CaDiCaL [20]. 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 [42] and Sat4j [19] (with cutting planes enabled). These two solvers provide a baseline for comparing stronger reasoning techniques against our resolution-based CDCL solvers. For this evaluation we used a $5{,}000$ second timeout. We run the following KNF solving configurations:

- ReEncode : run CaDiCaL on reencoded CNF formula.
- CCDCL : run CCDCL on the KNF formula.
- Hybrid : run CCDCL on the KNF formula plus the clausal encodings of cardinality constraints.
- RoundingSAT : run RoundingSAT on the KNF formula converted to OPB.

Table 5.1: Top are the 1,946 (847 SAT, 716 UNSAT) formulas with at least one AtMostOne cardinality constraint of at least size 5, bottom are the 933 (405 SAT, 345 UNSAT) formulas with at least 10 AtMostOne cardinality constraints of at least size 10. Average PAR 2 is shown for SAT, UNSAT, and the combination thereof.

Configuration	# Solved		PAR2		
	SAT	UNSAT	SAT	UNSAT	Combined
At least one constraint of size 5					
CADICAL	790	619	932	1850	1353
CCDCL	789	593	953	2292	1567
HYBRID	**795**	585	**898**	2427	1598
REENCODE	787	**636**	966	**1561**	**1238**
ROUNDINGSAT	647	475	2592	3823	3156
SAT4J	373	240	5677	6721	6155
At least 10 constraints of size 10					
CADICAL	373	269	1063	2824	1873
CCDCL	**380**	254	**929**	3316	2027
HYBRID	377	262	1018	3104	1977
REENCODE	372	**282**	1108	**2297**	**1655**
ROUNDINGSAT	294	185	2976	4924	3872
SAT4J	166	103	5954	7065	6465

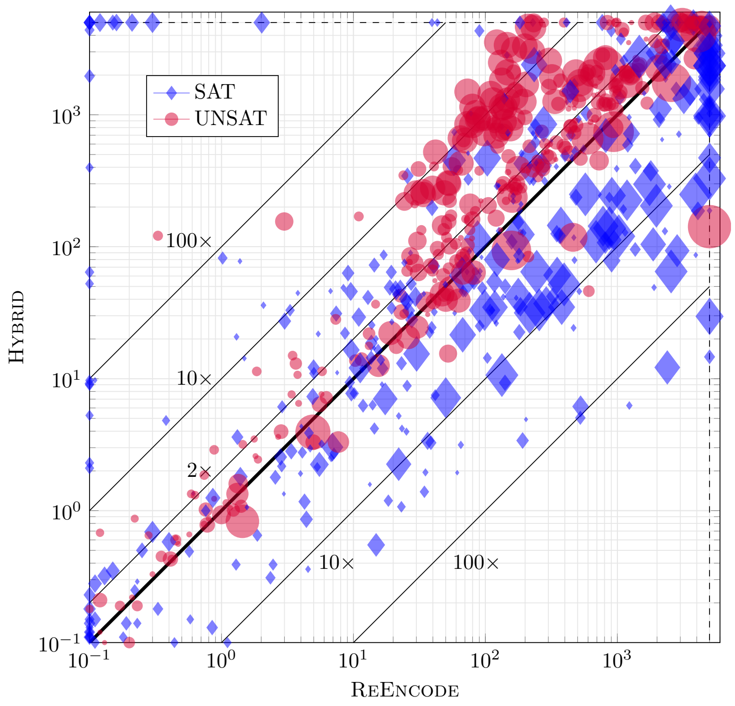
- SAT4J : run SAT4J with combined cutting planes and resolution on the KNF formula converted to OPB.

Sat Competition Extracted KNFs

In this section, we present a comparison of our KNF solving techniques on the formulas from the SAT Competition Anniversary Track after applying our AtMostOne extraction tool. Only solver runtimes are reported. 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 KNF. 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. In addition to the KNF solving techniques, we include the baseline solver CADICAL run on the original CNF formula.

We present two sets of formulas in Table 5.1. The first set contains all formulas with at least one cardinality of size 5, but the impact of KNF solving on small cardinality constraints is vanishing. So, we also show results for the restricted set with at least 10 cardinality constraints of at least size 10. 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.

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 PAR2 scores, a cardinality-based config-



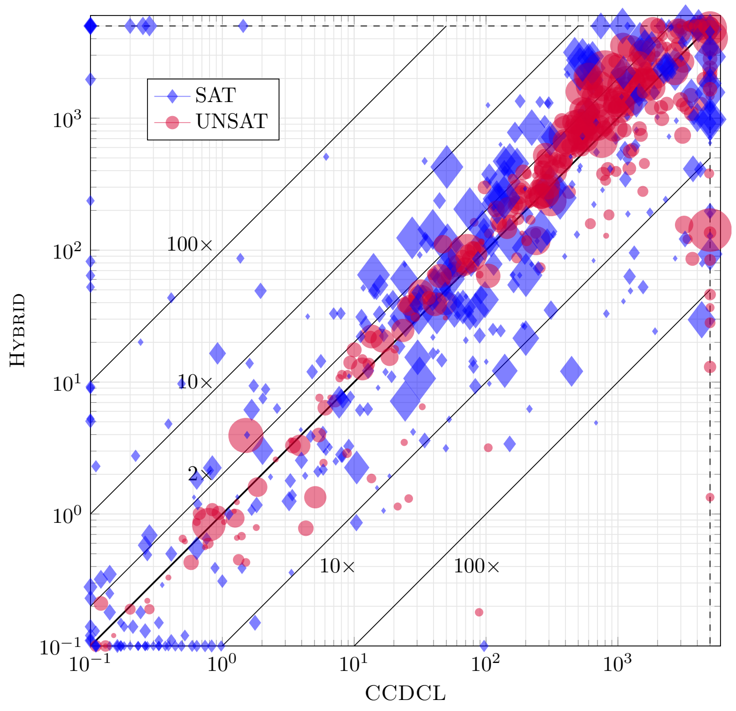
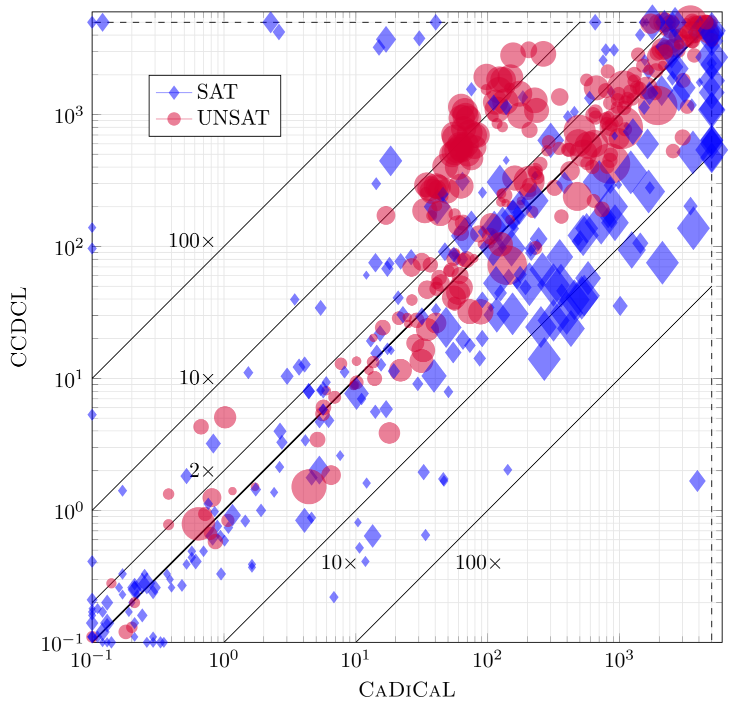


Figure 5.3: Comparison between HYBRID and other KNF solver configurations on the 933 formulas with at least 10 extracted constraints of size 10 or more. The size of a mark is proportional to the number of extracted constraints of size 10 or more, i.e., formulas with many large AtMostOne constraints have large marks.



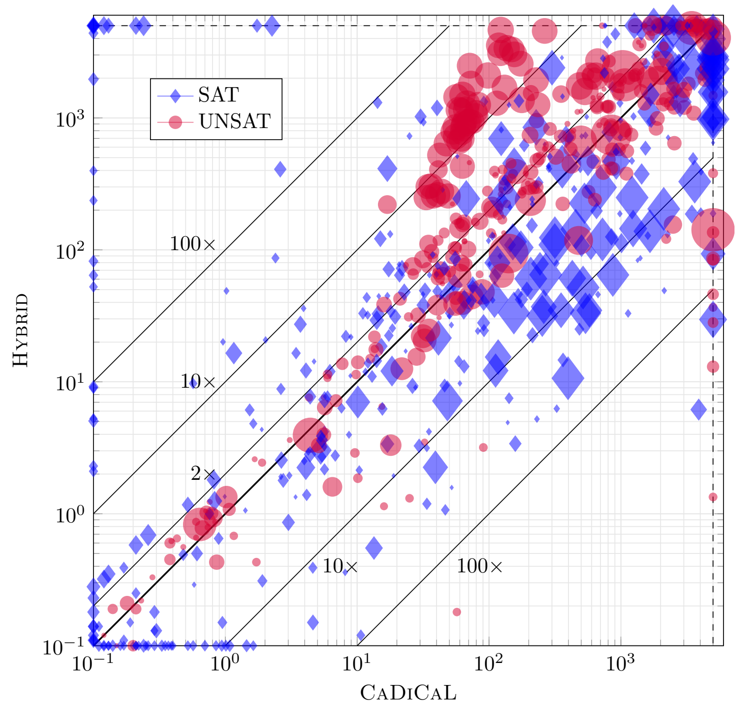


Figure 5.4: Comparison between KNF solver configurations and baseline CaDiCaL on the 933 formulas with at least 10 extracted constraints of size 10 or more. The size of a mark is proportional to the number of extracted constraints of size 10 or more, i.e., formulas with many large AtMostOne constraints have large marks.

uration 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. The PB solvers are more suited for special cases where formulas contain many AtLeastK cardinality constraints.

REENCODE performs best for unsatisfiable formulas but does not perform as well as CADICAL on satisfiable formulas. These results were described in Section 4.1. CCDCL and HYBRID solve the most satisfiable formulas. The native propagation is more effective when the formula contains larger cardinality constraints, seen in the larger difference in PAR2 score between CADICAL and CCDCL in the bottom set of the table. Of course, for a much larger AtMostOne constraint native propagation can potentially propagate all but one of the literals in the constraint in a single step, whereas propagation on a clausal encoding would need to work through several layers of auxiliary variables. 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.

The top plot in Figure 5.3 shows the tradeoff between native propagation and encodings. HYBRID implements mode-switching that trades approximately half the time between the native propagation and clausal encodings, leading to a slow down on average for solving unsatisfiable formulas. This is made clear by the many unsatisfiable instances falling around the $2\times$'s line in the scatter plot. On the other hand, HYBRID is able to solve the satisfiable instances with many cardinality constraints much faster due to the native propagation. The bottom plot in Figure 5.3 shows that HYBRID can make drastic improvements for a small set of unsatisfiable instances compared with CCDCL but incurs the a times cost on most of the formulas. This is especially pronounced for hard formulas taking over 500 seconds to solve.

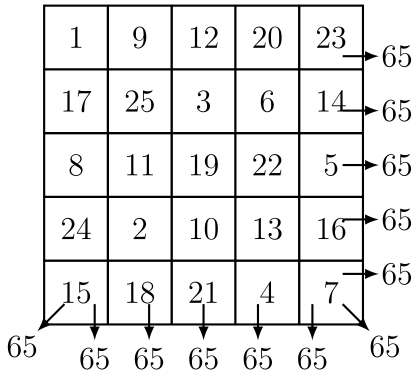
Figure 5.4 shows the KNF solving techniques using native propagation compared against the baseline CADICAL. In general, native propagation will help solve satisfiable instances faster, and this is true for both CCDCL and HYBRID. The encodings in HYBRID allow it to solve several unsatisfiable instances that CADICAL cannot, and this almost never occurs for CCDCL.

Our heuristic-based extraction only works for AtMostOne constraints, so any AtLeastK cardinality constraints were missed. If the problems were first encoded in KNF and contained AtLeastK cardinality constraints, we expect the results to improve significantly. We explore this possibility in the following section with two problems encoded directly in KNF.

Magic Squares and Max Squares

In this section, we consider both the Magic Squares and Max Squares problems. These problems demonstrate the effectiveness of native propagation on satisfiable formulas with general cardinality constraints, as well as the importance of good encodings for unsatisfiable formulas. Both problems were generated in KNF.

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 Table 5.5. 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



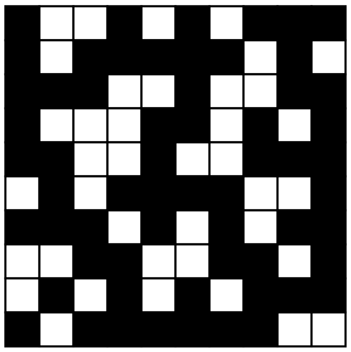


Figure 5.5: Left a magic square ($n = 5$) and right an optimal solution of a max square ($n = 10$, $m = 61$).

Table 5.2: 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	**0.18**	**1.42**	**6.56**	**12.01**	46.37	**460.82**	**164.61**	**766.07**
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	**17.99**	958.56	247.94	3177.64

Max Squares								
	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	**17.0**	101.42	1.07	1.27	58.53	–
REENCODE	**0.01**	**0.62**	57.83	**24.33**	**0.18**	**0.72**	**22.82**	–
ROUNDINGSAT	0.06	1582.62	–	–	2.31	1046.83	–	–
SAT4J	5.75	–	–	–	26.24	–	–	–

can have the same value, (c) cardinality constraints stating the sum of each row, column, and diagonal is at least M, (d) cardinality constraints stating the difference between the total $n \times n$ and the sum of each row, column, and diagonal is at least the total $n \times n - M$.

When encoding the problem with the correct magic number, it is satisfiable for any $n \times n$ grid. Table 5.2 shows the solving times on the Magic Squares formulas of increasing size. CCDCL significantly outperforms the other 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 [113] 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 cardinality constraint stating at least m cells are set to true.

The results in Figure 5.2 show the solving times on several configurations with satisfiable formulas ($m = opt$) and unsatisfiable formulas ($m = opt + 1$). For these formulas, both native 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 poor performance of CCDCL on even the satisfiable formulas.

The problems above highlight the main difficulty with handling cardinality constraints and reencodings: sometimes encoded constraints make the problem much easier, yet sometimes keeping the cardinality constraints abstract makes the problems easier. We attempt to address this dilemma with the combined configuration Hybrid, which has access to auxiliary variables throughout solving and native propagation during SAT modes. For Magic Squares, Hybrid outperforms ReEncode, and for Max Squares Hybrid outperforms CCDCL. Ideally, the solver would benefit from faster propagation while also deciding on auxiliary variables and incorporating them into learned clauses. This approach works well for the Max Squares problem, with the Hybrid performing much better than CCDCL and almost matching the performance of ReEncode. There are also improvements over ReEncode for the Magic Squares problem but it cannot reach the performance of CCDCL. These results suggest that the combined approach in Hybrid can improve on the worst-case behavior between CCDCL and ReEncode, but still cannot reach the best-case performance of either.

5.2 Conditional Cardinality Constraints

A conditional cardinality constraint has the form:

$$c \implies \ell_1 + \ell_2 + \cdots + \ell_n \geq k$$

We write conditional cardinalities in KNF as:

$$x_5 \implies x_1 + x_2 + x_3 + \overline{x}_4 \geq 2 \qquad \texttt{g 2 -x}_5\ \texttt{x}_1\ \texttt{x}_2\ \texttt{x}_3\ \texttt{-x}_4\ \texttt{0}$$

We refer to c as the conditional literal, and $\ell_1 + \ell_2 + \cdots + \ell_n \geq k$ as the cardinality constraint. A conditional cardinality is evaluated as `true` if the conditional literal c is `false` or the cardinality constraint is `true`, and the conditional cardinality is evaluated as `false` if the conditional literal c is `true` and the cardinality constraint is `false`.

There are two cases for propagation on a conditional cardinality constraint:

- The conditional literal, c, is the propagated literal. Unless the cardinality constraint is unit (the number of literals in the constraint is equal to the bound), we do nothing. This may seem strange because the cardinality constraint may be in conflict or propagating, but that will eventually be handled by the watched literals within the cardinality constraint. However, if the cardinality constraint is unit, we need to check if it is in conflict or can propagate. A unit cardinality constraint is only propagated once the conditional literal is assigned to `true`. If it is in conflict, the conflict clause is the conditional literal and one falsified literal from the cardinality constraint.
- A literal from the cardinality constraint, $\overline{\ell}_i$, is the propagated literal. If the conditional literal c is `false`, we can ignore the cardinality constraint. Otherwise, we perform standard propagation on the cardinality constraint. The reason for the propagated literals is the conditional literal, falsified unwatched literals and $\overline{\ell}_i$. If we find that the cardinality constraint is in conflict, we attempt to propagate c to `false`. If c was already assigned to `true`, the conditional cardinality constraint is in conflict. The conflict clause is the conditional literal, falsified unwatched literals, $\overline{\ell}_i$, and one other falsified watched literal.

Conditional cardinalities have been used to represent machine learning explainability discovering abductive and counterfactual explanations for k-nearest neighbors [16] In addition, similar work has been done for binarized neural network (BNN) verification [60, 115].

The current state of the conditional cardinality support in our solver is targeted at satisfiable instances. Native propagation alone is enough to achieve significant speedups. No mode-switching or specialized inprocessing techniques are required, and the SLS was not updated to support conditional cardinalities. However, for different classes of problems like BNN verification, the goal is to prove formulas are unsatisfiable. For these problems, a combination of native propagation and encoding will likely be the best solving approach, but proof checking becomes more difficult and may require specialized checkers [115].

5.2.1 Point Discrepancy Problem

Given a set of n points, the discrepancy (dK) for any line intersecting at least two points is the absolute difference between the number of points above and below the line. The point discrepancy problem asks if for n points, there exists a graph such that the discrepancy of every line is at least dk.

This can be encoded in CNF using geometric abstractions and pairwise relationships [103]. Conditional cardinalities present a natural way to encode the constraints setting the discrepancy of each line. For every line segment formed from two points x_i, x_j,

$$c \implies a_i + \cdots + a_j + \overline{b}_i + \cdots + \overline{b}_j \geq n-2+k \;\wedge\; \overline{c} \implies \overline{a}_i + \cdots + \overline{a}_j + b_i + \cdots + b_j \geq n-2+k$$

The variables a_i (respectively b_i) are `true` if point i rests above (respectively below) the line, and the conditional literal c is `true` (respectively `false`) if the discrepancy dk is above (respectively below) the line.

The two conditional cardinality constraints can be encoded into CNF by first encoding the two cardinality constraints into sequential counters with output variables o^a_{n-2+k} and o^b_{n-2+k}. Instead of setting the output variables to `true` as would be done in the standard sequential

Table 5.3: Runtimes for solving configurations on the point discrepancy problem with discrepancy 2 and various values of n from 21 to 57. – indicates a timeout at $1{,}800$ seconds.

	number of points						
Configuration	21	27	33	39	45	51	57
NATIVE PROPAGATION	4	1	6	39	45	109	31
CLAUSAL ENCODING	3	90	142	289	121	597	–

counter encoding, the variables are placed in the clause $(o^a_{n-2+k} \vee o^b_{n-2+k})$. This clause ensures that at least one of the cardinality constraints will be satisfied.

It is important to note that a given solution to the formula may not by *realizable*. A realizable solution is one that can produce a valid graph, and the realizability problem is hard. Therefore, it is desirable to generate many solutions for each problem so we can check the realizability of several solutions.

This KNF formulation enables native propagation to reason directly about discrepancy counts, often propagating multiple literals in a single step. As shown in Table 5.3, solving times improve dramatically with native propagation on these conditional cardinality constraints.

5.3 Remarks on Solving

In this chapter, we presented a framework for solving problems with cardinality constraints using native propagation. We described how native propagation can be incorporated into a modern CDCL solver while accounting for the most important inprocessing techniques. We evaluated several configurations for applying native propagation, finding that some combination of native propagation for satisfiable formulas and clausal encoding for unsatisfiable formulas leads to the best performance. We extended native propagation with support for conditional cardinalities and evaluated our tool on the point discrepancy problem, finding that native propagation widely outperforms clausal encodings.

This work culminated in an off-the-shelf KNF solver with native cardinality propagation. The solver has been used in several applications where native propagation produced solutions much faster than the clausal encoding alternative. In the next phase of KNF solving research we would like to better understand the connection between native propagation and encoding, potentially finding ways to encode during solving. Dynamic encoding has the benefit of letting the solver run for some time with native propagation in order to determine which cardinality constraints are being used the most during search. These constraints can then be encoded to increase the chance of finding a short proof. Also, the solver may learn units that shrink cardinality constraints such that the eventual clausal encoding will be smaller and more manageable for the solver. There are two open questions regarding dynamic solving. The first is how to generate proofs for a partial encoding. This would likely require the use of a stronger proof system like cutting planes. The second is what measurements should be used to determine when to encode a cardinality constraint. There are several options including constraint activity, the average number of decisions in the solver, and the average size of learned clauses.

Chapter 6

Parallel Solving

Zachary Battleman, Joseph E. Reeves, and Marijn J. H. Heule. *Problem Partitioning via Proof Prefixes*. In *Theory and Applications of Satisfiability Testing (SAT)*. (2025). [18]

In this chapter we discuss how information from cardinality constraint encodings can be used in parallel SAT solving. The are two main paradigms for parallel solving: clause sharing portfolios in which multiple CDCL solvers are given the same CNF as input and they share learned clauses between each other during the solving process; and, Cube-and-conquer (CnC) [53] in which a formula is divided into subformulas and each is solved by an independent SAT solver. In the remainder of this chapter, we will focus on CnC, though a solver with native cardinality propagation could certainly be added to a clause sharing portfolio.

CnC creates subproblems from an input formula through the use of *splitting variables*. For example, consider a formula F that contains a variable x_1. We could create two subformulas by splitting on x_1 and generating $F \wedge x_1$ and $F \wedge \overline{x}_1$. If both formulas are unsatisfiable, then F is unsatsifiable, and if either formula is satisfiable then F is satisfiable under the found solution. Each of these formulas could be split further by using additional variables. The selection of good splitting variables is crucial to the success of CnC. A good set of splitting variables (also referred to as a *split*) will create a set of formulas that each take approximately the same amount time to solve and can be solved faster than the original formula. This would allow us to solve the formulas in parallel with many independent solvers.

Automated methods for finding splits such as look-ahead techniques have trouble generalizing to a wide range of real-world problems. Indeed, finding splits for solving open problems in discrete math has historically required expert insight and domain knowledge on a problem-by-problem basis [52, 102].

One of the benefits of the KNF input format is that we can encode cardinality constraints however we see fit, then solve the resulting CNF formula. This not only means we can pick a good encoding, but also, we know the semantic meaning of the auxiliary variables introduced by the encoding. We will show how we can select splitting variables from the Totalizer encoding, and that these splits are effective on optimization problems.

6.1 Cube-and-Conquer

Cube-and-Conquer (CnC) is a powerful technique for using multiple cores to solve hard SAT formulas. CnC takes as input a formula F and a set of sub-formulas $\psi_1, \ldots, \psi_n$, such that F is satisfiable if at least one of $F \wedge \psi_i$ is satisfiable, and F is unsatisfiable if all of $F \wedge \psi_i$ are unsatisfiable. The sub-formulas ψ_i are sets of unit clauses, and the set of literals that represent the units are referred to as *cubes*. CnC has each formula $F \wedge \psi_i$ solved by an independent CDCL solver (see Figure 6.1).

Splits can be viewed as a tree where the root and internal nodes are variables with two branches (`true` and `false`). Each leaf represents a cube that can be generated by following the unique path from the root to the leaf, adding the literal from each node (variable) traversed, with the literal being positive if a `true` branch is taken and negative if a `false` branch is taken. A *static* split creates a balanced binary tree based on a set of variables $\{x_1, x_2, \ldots, x_r\}$ where each level of the tree uses one of the variables. This will create cubes representing every combination of polarities on the variables, for a total of 2^r cubes. A *dynamic* split creates an unbalanced tree where different variables might appear on different levels. This can be useful if one polarity of a splitting variable creates a much harder subformula that then needs to be split further.

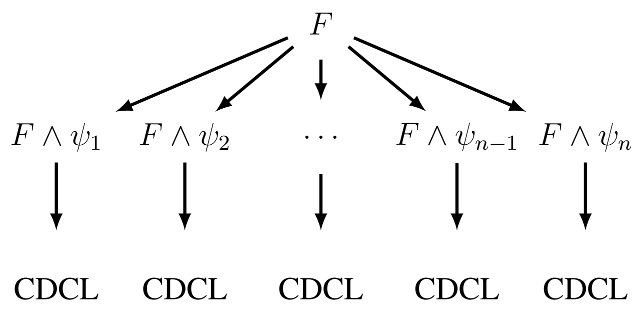


Figure 6.1: Cube-and-conquer heuristically splits a formula F into n subformulas $F \wedge \psi_1$ to $F \wedge \psi_n$ and solves the subformulas using a CDCL SAT solver.

6.2 Splitting the Totalizer Encoding

The Totalizer encoding introduces auxiliary variables for each merge unit that represent the sorted output of the input literals. We will refer to the auxiliary variables within each merge unit as counters. A counter c_i is `true` when at least i of the inputs into the merge unit are `true`, and c_i is `false` when less than i of the inputs are `true`. Note, each merge unit will have its own set of counters, so when we use c_i we will be referring to the counter of a specific merge unit.

We motivate our splitting heuristic with the Totalizer in Figure 6.2. First, let both counters at depth $D1$ be assigned to their respective values. Then, selecting c_2 from the first merge unit of depth $D2$ as a splitting variable will result in the following two cases:

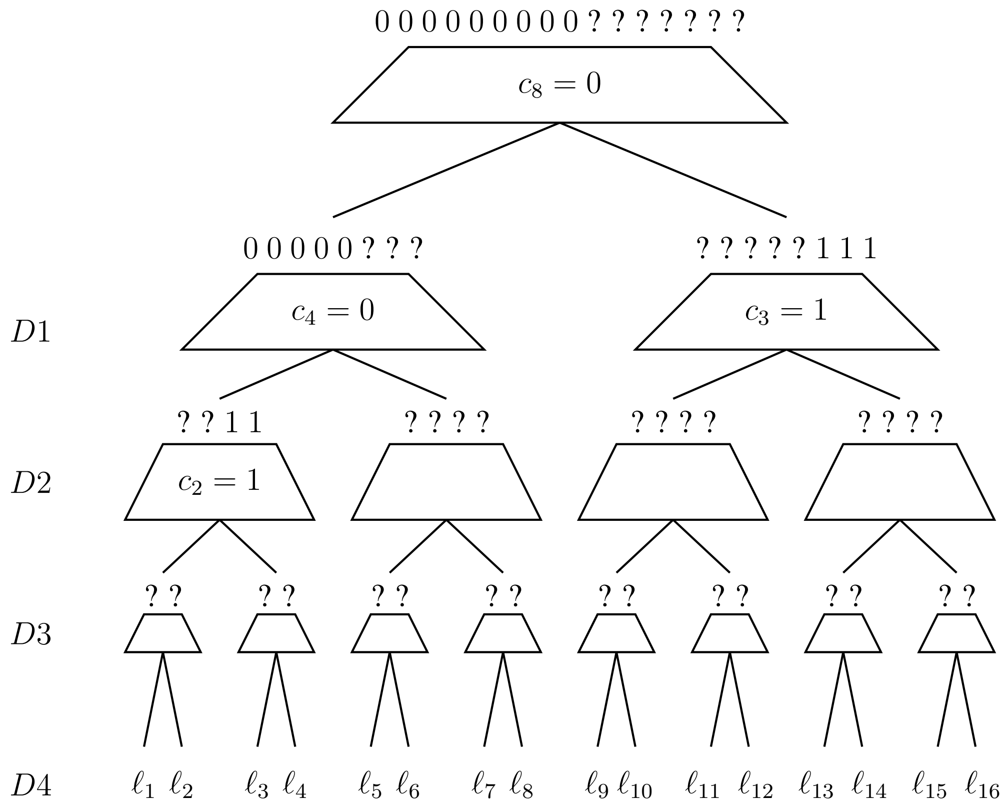

0 0 0 0 0 0 0 0 0 ? ? ? ? ? ? ?
$c_8 = 0$
0 0 0 0 0 ? ? ?
? ? ? ? ? 1 1 1
$D1$
$c_4 = 0$
$c_3 = 1$
? ? 1 1
? ? ? ?
? ? ? ?
? ? ? ?
$D2$
$c_2 = 1$
? ?
$D3$
$D4$
ℓ_1 ℓ_2 ℓ_3 ℓ_4 ℓ_5 ℓ_6 ℓ_7 ℓ_8 ℓ_9 ℓ_{10} ℓ_{11} ℓ_{12} ℓ_{13} ℓ_{14} ℓ_{15} ℓ_{16}

Figure 6.2: Visualization of a Totalizer encoding of $\ell_1 + \ell_2 + \cdots + \ell_{16} \leq 7$ with the bound enforced by $c_8 = 0$ in the root merge unit. We show an example split selecting counters from three merge units.

- If c_2 is `true`, at least two of the data literals from the set $\{\ell_1, \ell_2, \ell_3, \ell_4\}$ are `true`. This also means that at most one data literal from the set $\{\ell_5, \ell_6, \ell_7, \ell_8\}$ is `true` due to the bound on c_4 from the parent merge unit at depth $D1$.
- If c_2 is `false`, at most one of the data literals from the set $\{\ell_1, \ell_2, \ell_3, \ell_4\}$ are `true`. This provides no information about the data literals from the set $\{\ell_5, \ell_6, \ell_7, \ell_8\}$.

Intuitively, splitting on an auxiliary variable at an internal merge unit in the Totalizer will designate how many `true` data literals are among the merge unit's descendants and may potentially constrain its sibling merge units. Therefore, the closer the merge unit is to the root, the more data literals it will impact.

A bad selection can create unbalanced subproblems. For example, picking c_8 from either merge unit at depth $D1$ would create one case that is trivial to solve (c_8 is `true`) and another case with no useful information given by the split (c_8 is `false`). Therefore, we focus our splitting on counters with a large impact that also correlate with the cardinality constraint's bound. To achieve this, we use the ratio $Rk = \frac{k}{r}$ for a cardinality constraint with bound k and r data literals. We select a counter from a merge unit with f inputs as $\mathsf{SelectCounter}(f, Rk) = \lfloor Rk \times f \rfloor$. To select a set of counters from the Totalizer, we use the following procedure:

1. Select a starting depth D and desired number of splitting variables $SplitCnt$.

2. Sort the merge units at depth D by their number of inputs, largest to smallest.
3. For each merge unit in D, select the counter $\mathsf{SelectCounter}(f, Rk) + 1$ for odd nodes and $\mathsf{SelectCounter}(f, Rk)$ for even nodes. Once $SplitCnt$ variables are selected, break.
4. If less than $SplitCnt$ variables have been selected, proceed to depth $D + 1$ and return to step 2 to select more splitting variables.

Consider the Totalizer in Figure 6.2 with $Rk = \frac{7}{16} = 0.4375$. Assume that we want 3 splitting variables, starting from depth $D1$. Starting with the left merge unit which has 8 inputs, the selected counter will be $\mathsf{SelectCounter}(8, 0.4375) + 1 = \lfloor 8 \times 0.4375 \rfloor + 1 = 4$ (c_4). Moving to the sibling merge unit on the right, the next counter selected will be c_3. Next, we descend to depth $D2$, and since all merge units have the same number of inputs we process them in order from left to right. For the first merge unit, $\mathsf{SelectCounter}(4, 0.4375) + 1 = \lfloor 4 \times 0.4375 \rfloor + 1 = 2$ corresponding to c_2.

The reason we alternate between adding one to the counter is to ensure that the sum of the counters within a given depth resembles the original bound of the cardinality constraint. This reduces the number of unhelpful or trivial cubes.

The starting depth is assigned based on the number of desired splitting variables such that all nodes in a given depth are processed if possible. In our experimental configuration we chose 12 splitting variables and therefore started at depth 2, giving 4 variables at this depth and 8 variables at depth 3.

This technique makes no assumptions about the clauses in the formula and forms a static partition solely by selecting auxiliary variables from the Totalizer encoding. It can be adapted to problems containing multiple cardinality constraints by selecting splitting variables from the largest available merge units of the largest cardinality constraints in sequence. This would maximize the impact of the individual splitting variables.

6.3 Alternative Splitting Methods

Proofix [18] finds static splits based on proof prefixes. A proof prefix is the initial portion of a clausal proof generated by running a solver for a short period of time then exiting. A splitting variable can be selected by counting positive and negative occurrences within the clause addition steps of the proof and choosing the variable that occurs most frequently. Given an initial set of splitting variables $\{x_1, \ldots, x_r\}$, Proofix generates proof prefixes from a sample of cubes on $\{x_1, \ldots, x_r\}$ then sums occurrences across proof prefixes to determine the next splitting variable. The sampling strategy helps to avoid an exponential blowup in the number of proof prefixes generated in each step.

March [56] uses look-aheads to find a dynamic split. A lookahead on a literal ℓ examines the formula under an assignment of ℓ. For instance, ℓ can be set to `true`, then unit propagation can be applied to the formula. If there was a conflict, ℓ is a failed literal and $\overline{\ell}$ can be learned. Otherwise, we can quantify how much the formula was *reduced* after unit propagation. There are several metrics that can be used, but a common approach is to compute a weighted count of the clauses that are reduced but not satisfied. The same process can be performed for literal $\overline{\ell}$, and the combined formula reduction will be used to determine if $\mathsf{var}(\ell)$ is a suitable splitting variable.

Table 6.1: Comparison of partitioning methods on combinatorial problems and MaxSAT competition benchmarks. Pre. is the preprocessing time to find a split. totalizer-split does not include a preprocessing time because it finds splits during encoding.

formula	baseline	March			Proofix			totalizer-split	
	1 core	1 core	32 core	Pre.	1 core	32 core	Pre.	1 core	32 core
judge	3,654	4,893	3,162	219	3,437	2,447	19	7,851	4,289
mbd	2,170	2,914	313	287	2,512	409	37	3,784	290
optic	1,236	908	195	23	708	22	45	1,150	135
uaq	2,520	1,408	453	4	970	62	43	1,960	129
mindset	2,162	18,018	1,372	357	16,375	2,252	42	19,002	1,164
max10	6,282	1,939	920	0	7,645	238	29	2,498	269
cross13	$>$ 80,000	?	$>$ 10,000	43	64,610	2,206	71	77,664	3,125
$\mu_5(13)$	2,317	2,526	181	8	1,367	84	80	2,355	543

These look-aheads are performed at each node in the splitting tree, always picking the variable that yields the largest reduction. There are additional types of measurements and look-ahead strategies that can affect the quality of the dynamic splits.

6.4 Results

To account for the harder problems in this section, we ran 32 solver instances in parallel per node with 10,000 second timeouts. Therefore, each solver process held approximately 8GB of memory. We use CaDiCaL as our SAT solver applied to the subformulas after splitting. We also present the runtime of CaDiCaL without splitting to serve as a baseline.

In this section, we evaluate the Totalizer-based splitting (totalizer-split) against proof-based splitting (Proofix) and look-ahead-based splitting (March) on problems containing one large cardinality constraint. As discussed in Section 4.2, MaxSAT benchmarks serve as a good set of KNF problems since the cardinality constraints are often quite large and the cardinality constraint must be used in a solver's reasoning to solve a problem. We use the unsatisfiable version of the MaxSAT problems and encode the KNF into CNF using a Totalizer encoding. Prior to encoding the cardinality constraint, we sort data literals using the proximity method from Section 4.2. This step improves the default solver performance, creating a better baseline for partitioning.

We use the subset of problems from the 2023 MaxSAT competition that take at least 500 seconds to solve with CaDiCaL. The remaining families are judgment aggregation (judge) [61], model-based diagnosis (mbd) [75], approximately propagation complete CNF for an all-different encoding of pigeon hole (optic) [11], user query authorization (uaq) [11], and minimum rule set for labeling data (mindset) [11]. For each family, we selected the hardest problem for CaDiCaL to solve and present the results at the top of Table 6.1.

totalizer-split performs adequately, achieving a $10\times$'s speedup on three of the five problems. Splitting makes the problem harder for the judge benchmark, but the other two splitting techniques also do not perform well on this problem. Furthermore, totalizer-split boasts little to no preprocessing time since the splits are generated as the constraint is encoded. This can help make up for lower quality splits, as seen in the uaq problem. On mindset, the most difficult problem to split, totalizer-split produces the best partitioning for 32 cores, showing that even this simple technique can work well on some problems.

When examining the splitting variables selected by Proofix, we found that it used some auxiliary variables from the Totalizer encoding for optic and mbd, a single auxiliary variable for judge, and no auxiliary variables in uaq and mindset. This reinforces the intuition behind totalizer-split, since the complex proof-based technique selects some of the same auxiliary variables. Furthermore, for problems like uaq and mindset it means that both auxiliary variables and data variables can be used to form good partitions.

In addition to the MaxSAT benchmarks, we consider three combinatorial problems that are much harder for the baseline CaDiCaL to solve. *Max10* is the unsatisfiable version of the max squares benchmark on a 10×10 board.

The crossing number of a graph is the lowest number of edge crossings found when drawing a graph on the plane. The crossing number of the complete graph with 13 nodes is 225. The benchmark *cross13* asks whether there is a drawing of a complete graph on 13 nodes with 224 crossings.

The $\mu_n(k)$ problem asks to find the minimum number of convex n-gons induced by placing k points in the plane with no three points in a line. The $\mu_5(15)$ benchmark asks whether it is possible to construct only 76 convex pentagons in the plane with 15 points, one less than the known minimum of 77.

totalizer-split is slightly worse than Proofix on each of the problems. Notably, totalizer-split can utilize the additional cores and always improve on the CaDiCaL runtime. The best partition for each problem comes from Proofix, and these partitions do not use auxiliary variables from the Totalizer encoding. So, while totalizer-split can be an adequate approach, other variables outside of cardinality constraints may have stronger splitting potential.

6.5 Remarks on Parallel Solving

KNF input gives us the opportunity to encode cardinality constraints ourselves, and by doing so we can leverage the meaning of auxiliary variables we introduce in the encoding during solving. We developed a technique that builds on this idea and selects splitting variables from a Totalizer encoding based on the meaning of the variables. We evaluated the splitting on a set of MaxSAT benchmarks and hard combinatorial benchmarks that each contained one large cardinality constraint. We found that our Totalizer splitting consistently outperformed the previous state-of-the-art splitting tool March on these problems and was close to the performance of the new proof-based splitting technique Proofix.

One question going forward is how to combine Proofix and totalizer-split into a single splitting tool. For some problems, the split found from the auxiliary variables is good enough, but for others a good split must include variables outside the Totalizer encoding. We may be able to speed up the preprocessing time incurred by proof-sampling in Proofix by guiding it with potentially useful auxiliary variables. Alternatively, we could use Proofix to quickly measure the value of a small set of auxiliary variables and use the result to decide whether totalizer-split will be a good choice for a given problem. Furthermore, we would like to incorporate ideas from the totalizer-split into sequential solving, leveraging our understanding of auxiliary variable meanings to update solver heuristics during search.

Chapter 7

Conclusion

This core idea behind this thesis, that KNF should be the new SAT standard, started with our implementation of native cardinality constraint propagation in CADICAL. After minor adjustments to the analysis algorithm and certain inprocessing techniques, we found that native propagation could help a solver discover solutions for certain hard combinatorial problems extremely fast, all the while producing proofs that were immeasurably valuable for debugging. We wanted to apply native propagation to a wider set of benchmarks, so we developed and AtMostOne extraction tool and transformed thousands of CNF formulas from the SAT Competition Anniversary Track into KNF formulas.

We were shocked to find that the vast majority of AtMostOne constraints extracted from the benchmarks used suboptimal encodings, and simply introducing auxiliary variables via reencoding the constraints would drastically change the solver performance on unsatisfiable formulas. It was at this moment that our focus widened, and we saw KNF as more than a means for native propagation but also as a tool for controlling the meaning of auxiliary variables. This led to our first attempt at a general purpose KNF solver that could handle both satisfiable and unsatisfiable formulas. We used native propagation in the solver's SAT mode and only reasoned on clausal encodings in the solver's UNSAT mode, creating something of an internal portfolio that incurred a moderate overhead to perform reasonably well on both satisfiable and unsatisfiable formulas.

It was around this time when the preprocessing technique SBVA hit the annual SAT competition, dominating on unsatisfiable formulas and taking first overall. BVA was introduced more than a decade prior but never took hold in the greater SAT community, that is, until SBVA changed the story. The "structured" in "S"BVA signifies a structured approach to finding patterns and adding auxiliary variables. It modified the original BVA algorithm to use a heuristic that determined which patterns should be reencoded next, and this subtle change was enough to make the technique a champion. The success of SBVA opened our eyes to the importance of *which* auxiliary we introduce and *what* they mean.

Our first attempt at improving the meaning of auxiliary variables came in the form of literal sorting for AtLeastK constraints, motivated by related work on MaxSAT solvers and incremental encodings. It was apparent that changing the ordering of literals could impact which groups auxiliary variables summarized and therefore how a solver might reason. At first, we found many ways to make an ordering worse, reinforcing the idea that ordering can have a large impact on solving time, but we struggled to make an ordering better. Then, we expanded our field of vision and sought out benchmarks from MaxSAT competitions with exceptionally large cardinality constraints. A combination of our AtMostOne extractor and proximity heuristic allowed us to improve encodings across the board simply by changing the order of literals as they were input into the PySAT encoder.

We extended the concept of literal sorting to problems that contained sets of cardinality constraints, specifically ExactlyOne constraints. In reality, we did not go into our study of the extraction and reencoding of ExactlyOne constraints with literal sorting in mind. We had the entire framework developed for a straightforward reencoding but found the performance to be lacking on several sets of shuffled formulas. After many iterations of attempting to "unshuffle" the formulas, we came up with a general approach to align the encodings of every ExactlyOne constraint within the problem. To our surprise, the alignment procedure improved performance on a wide range of problems with no apparent shuffling, showing that the relationships between cardinality constraints can be captured in the auxiliary variables their encodings introduce.

In an orthogonal move, we took our interest of auxiliary variables into the domain of parallel solving. It is easy to see how branch-and-bound can split a problem in two: if a variable x has values from $1 \leq x \leq 6$, solve the problem on one core with $1 \leq x \leq 3$ and on another core with $4 \leq x \leq 6$. However, our approach did more than simply split values in two, it took a hierarchical structure and split on the internal nodes. For a cardinality constraint such as $x_1 + x_2 + \cdots + x_{100} \geq k$, we could create splits that read something like "half of the variables from x_1 to x_{25} are `true`, and a third of the variables from x_{10} to x_{20} are `false`, and a quarter of the variables from x_{50} to x_{100} are `true`." We unlocked these hierarchical splits that summarized nested groups of variables by accessing auxiliary variables at different layers of the Totalizer encoding. It turns out that for a task that is relatively hard to automate, finding good splits is actually quite simple when you have a problem with a large cardinality constraint.

At this point, we have found many more uses for KNF beyond our initial thrust towards native propagation. Whether someone is trying to find solutions quickly, generate short proofs for hard problems, or leverage a multi-core machine, if their problem has cardinality constraints, we have techniques that can help. The KNF input format will help these techniques move into the mainstream, where they can be hidden within solver internals, improving runtimes without a user ever needing to know they exist.

We should not stop here. This thesis has only begun to scratch the surface of KNF solving. We still have much more to explore in terms of auxiliary variable meaning and alignment, the compatibility of native propagation and clausal encoding, and AtLeastK constraint extraction. Furthermore, KNF solving is ripe for expansion into conditional cardinality solving, MaxSAT solving, and local search solving. We see this research agenda as a long-term project, with many known and unknown opportunities waiting to be capitalized on.

7.1 Future Work

7.1.1 Aligning AtLeastK Encodings

In this thesis we described a technique for sorting literals within one large cardinality constraint and separately how to align literals via sorting for many ExactlyOne constraints. These two ideas can be combined for alignment on AtLeastK constraints. The main challenge for AtLeastK constraints is their hierarchical encodings. Alignment between the ExactlyOne constraints is as simple as forming sets that contain literals from different ExactlyOne constraints but no two literals from the same ExactlyOne constraint, then using those sets to sort literals within the individual constraints prior to encoding. For AtLeastK constraints we may want to preserve relationships between literals within the constraint, as well as accomodate relationships between literals across constraints.

Towards this end, we could develop a new Totalizer encoding tool that takes in a set of literals and their relationships and builds a Totalizer that more closely reflects these connections. For example, if three literals are closely related, we can create a merge unit that takes in three literals as input. This would move us away from the balanced tree Totalizer implementations in PySAT, and would be more suited for preserving hierarchal relationships between constraints. Furthermore, we would need to develop a new way to characterize the relationship between constraints, because our alignment procedure would not work when we are also trying to sort literals within a single constraint. Using a combination of our proximity algorithm and graph clustering may help us detect these groupings.

It may be the case that a user has some intuition for which groups of data variables are important. It would be ideal to develop an API that allowed users to specify group relations between variables and then created a clausal encoding that optimized for these groups. Such a tool could be used to iteratively improve encodings with a user in-the-loop.

7.1.2 General Cardinality Constraint Extraction

Extracting high-level constraints from an encoded problem is difficult because the encoding transformation often obfuscates the structure of the original constraint. In our work, when we dealt with AtMostOnes and ExactlyOnes we were able to use syntactic heuristics based on variable polarities. The guess-and-verify approach could effectively classify variables as auxiliary or data based on their occurrences in binary clauses, but this would not work for AtLeastK constraints where auxiliary variables at inner layers in hierarchical encodings appear almost the same as data variables.

One promising path towards a stronger guessing approach is to detect auxiliary variables at ends of an encoding, for example, the auxiliary variables at the root and leaves of a Totalizer encoding. Then, a closure procedure can be performed that expands the encoding from both ends, collecting all of the inner auxiliary variables. In order classify these initial sets of auxiliary variables, we might use a metric like the *blocking count* of a literal, that determines how many clauses would need to be added to transform a literal into a definition. Auxiliary variables in AtLeastK encodings can often have small blocking counts, and this would hopefully distinguish them from data variables. Alternatively, we could use pattern recognition algorithms to attempt to differentiate data variables from auxiliary variables, trained on the set of PySAT encodings.

Verification also poses a problem since the BDD-based approach will have trouble scaling with the size of AtLeastK constraints. We could lower our threshold for trusting that guessed constraints are valid cardinality constraints with two new pseudo-verification techniques. First, we can create a CNF formula that contains the guessed constraint's clauses and a new encoding for a negated version of the guessed cardinality constraint. If the formula is unsatisfiable, then the guessed constraint is redundant with respect to the encoded clauses and can be added to the formula. This however does not imply that we can delete the encoded clauses. Second, we could perform some sort of simulation testing on the guessed constraint, setting different numbers of data literals to `true` and checking that the encoding clauses propagate the expected result.

While the goal of this thesis is to promote the creation of formulas in the KNF format, having this additional general cardinality constraint extraction will allow us to experiment more with existing sets of competition benchmarks. Furthermore, some of the extraction techniques we develop may inform other types of constraint extraction and reencoding, for example, in finding new patterns for SBVA.

7.1.3 MaxSAT with Native Propagation

Our solver with native cardinality constraint propagation has been most effective on large satisfiable problems. This is also true for optimization variants of problems. For example, we found that native propagation was ten times faster at finding functional mappings that optimized the number of transistors, with problems coming from the tool ABC. However, when we consider the set of problem with successively fewer transistors, native propagation exhibits poor performance on the first unsatisfiable problem. That is, the successive problems are satisfiable until the optimal number of transistors is found, then the next problem is unsatisfiable, certifying the optimal bound, and finding a proof for this final problem is hard with native propagation.

It is no surprise that native propagation does not perform well on unsatisfiable problems, but this quirk is not ideal for users that want to certify optimality. It may be possible to incorporate our native propagation solver within MaxSAT algorithms, such as a core-based algorithm, in order to incrementally build a learned clause database that is more suited for the unsatisfiable case. This poses two challenges. The first is to enable incremental solving in our cardinality solver. If we only want to incrementally add cubes (sets of unit assumptions) this is a straight-forward engineering task. However, if we want to add cardinality constraints between incremental calls or update the bounds of cardinality constraints in-place, this would become more complicated. Second, if a cardinality constraint occurs in an unsat core, we will have to decide whether or not we encode the cardinality constraint before constructing an AtMostOne constraint for the core. Answering these question may help us get closer to an optimization variant of our cardinality solver that is good at finding optimal solutions.

7.1.4 LLMs for Automated Encodings

One of the goals of this thesis is to promote the uptake of automated reasoning tools by increasing their usability. We focused on ease-of-use for cardinality constraints, taking a specific set of encoding decisions out of a user's hands. There are many more decisions made during the encoding process. For example, a selection constraint can be represented with a one-hot encoding, the order encoding, or a binary encoding; many high-level constraints such as all-different, pseudo-Boolean, and exclusive or can have multiple encoding choices; and formulas in propositional logic can be transformed into CNF using the Tseytin encoding or Plaisted-Greenbaum encoding, etc. All of these decision points are described across countless research articles, and a new user would have difficulty coming up with an optimal encoding for a novel problem.

One path towards greater usability is to have a tool that takes a set of high-level constraints or a description in a higher-level logic and encodes it automatically. LLMs present an exciting opportunity for realizing this, providing a means for subsuming several encoding choices within a single model. However, the key feature of automated reasoning tools is trustworthy solving with proof production, and this will need to be the focal point of any automated reasoning framework that incorporates an untrusted tool such as an LLM.

For example, one could generate a very simple and straightforward encoding for a problem, then use an LLM to "improve" the encoding (making it faster to solve). A separate tool could produce a derivation from the original encoding to the one produced by the LLM. This framework follows our reencoding framework for cardinality constraints. In this way, the LLM may be untrusted but its output encoding will be proven to be a valid derivation of the original problem and therefore can be used without fear of hallucinations. Such a process would likely require a mechanism for gauging how performant a given encoding might be prior to running a solver on the encoding, and this could benefit from other research that has looked into predicting the hardness of a formula using various machine learning techniques.

Bibliography

[1] Ignasi Abío and Peter James Stuckey. Conflict directed lazy decomposition. In *Principles and Practice of Constraint Programming (CP)*, volume 7514 of *Lecture Notes in Computer Science (LNCS)*, page 70–85, 2012. 2.3.2

[2] Ignasi Abío, Roberto Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A new look at BDDs for pseudo-Boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012. 3.2.3, 3.2.3

[3] Michael Alekhnovich. Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science*, 310(1):513–525, 2004. 2.2.1

[4] Leonardo Alt and Christian Reitwiessner. SMT-based verification of solidity smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 376–388. Springer, 2018. 2

[5] Markus Anders, Sofia Brenner, and Gaurav Rattan. Satsuma: Structure-Based Symmetry Breaking in SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23. Schloss Dagstuhl, 2024. 2.3.2

[6] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *International Conference on Logic Programming (ICLP)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 212–221. Schloss Dagstuhl, 2012. 2.2.4

[7] Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3542 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15. Springer, 2005. 2.2.2

[8] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013. 2

[9] Calin Anton and Lane Olson. Generating satisfiable SAT instances using random subgraph isomorphism. In Yong Gao and Nathalie Japkowicz, editors, *Advances in Artificial Intelligence, Canadian Conference on Artificial Intelligence*, volume 5549 of *Lecture Notes in Computer Science (LNCS)*, pages 16–26. Springer, 2009. 4.3.4

[10] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *Lecture Notes in Computer Science (LNCS)*, pages 167–180.

Springer, 2009. 2.2.3

[11] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2018 Benchmarks. `https://helda.helsinki.fi/items/7c51d9bd-20e8-428f-8926-b2bbf151dc4c`, 2018. Accessed: 2025-04-02. 6.4

[12] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 2833 of *Lecture Notes in Computer Science (LNCS)*, pages 108–122. Springer, 2003. 2.2.3

[13] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science (LNCS)*, pages 16–29. Springer, 2012. 5.1.2

[14] Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, 2021. 4.3.4

[15] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, 2023. 1.4

[16] Pablo Barceló, Alexander Kozachinskiy, Miguel Romero, Bernardo Subsercaseaux, and José Verschae. Explaining k-nearest neighbors: Abductive and counterfactual explanations. *Proceedings of the ACM on Management of Data*, 3, 2025. 2.1.4, 2.4, 5.2

[17] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, AFIPS, page 307–314. ACM, 1968. 2.2.3

[18] Zachary Battleman, Joseph E. Reeves, and Marijn J. H. Heule. Problem partitioning via proof prefixes. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:18. Schloss Dagstuhl, 2025. 1.3, 6, 6.3

[19] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–6, 2010. 2.3.2, 5.1.6

[20] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, and YalSAT entering the SAT competition 2017. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, 2017. 5.1.6

[21] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *Lecture Notes in Computer Science (LNCS)*, pages 405–422, 2015. 2.3.1

[22] 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 *Lecture Notes in Computer Science (LNCS)*, pages 285–301. Springer,

2014. 2.3.3, 3.2, 3.2.4

[23] Armin Biere, Katalin Fazekas, M. Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, 2020. 2.3.1, 4.3.3, 5.1.2

[24] Magnus Björk. Successful SAT encoding techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):189–201, 2011. 2.2.3

[25] Abdelhamid Boudane, Saïd Jabbour, Badran Raddaoui, and Lakhdar Sais. Efficient sat-based encodings of conditional cardinality constraints. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 57, pages 181–195. EPiC Series in Computing, 2018. 2.1.4, 2.2.3

[26] Pierre Bouvier and Hubert Garavel. Efficient algorithms for three reachability problems in safe petri nets. In *Application and Theory of Petri Nets and Concurrency: 42nd International Conference, PETRI NETS 2021, Virtual Event, June 23–25, 2021, Proceedings*, page 339–359. Springer-Verlag, 2021. 4.3.4

[27] Robert Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Computer Aided Verification (CAV)*. Springer-Verlag, 2010. 2, 2.4

[28] Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. *Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research*, pages 1–4. ACM, 2021. 1.3

[29] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. 3.2.3

[30] Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12651 of *Lecture Notes in Computer Science (LNCS)*, pages 76–93, 2022. 3.2.3, 3.2.3

[31] Yi Chu, Shaowei Cai, Chuan Luo, Zhendong Lei, and Cong Peng. Towards More Efficient Local Search for Pseudo-Boolean Optimization. In *Principles and Practice of Constraint Programming (CP)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:18. Schloss Dagstuhl, 2023. 2.1.3

[32] 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. 2

[33] Cayden Codel. Verifying SAT encodings in Lean. Master's thesis, Carnegie Mellon University Pittsburgh, PA, 2022. 5.1.5

[34] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158. ACM, 1971. 2

[35] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976. 2.2.1

[36] William Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-

plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. 2.3.2

[37] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, page 1092–1097. AAAI Press, 1994. 4.3

[38] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999. 3.2.3

[39] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing* (*SAT*), volume 3569 of *Lecture Notes in Computer Science (LNCS)*, pages 61–75. Springer, 2005. 2.3.1, 5.1.3

[40] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electron. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003. 5.1.3

[41] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation,*, 2(1-4):1–26, 2006. 2.2.5

[42] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1291–1299. AAAI Press, 2018. 2.3.2, 5.1.6

[43] Jan Elffers and Jakob Nordström. A cardinal improvement to pseudo-Boolean solving. In *Conference on Artificial Intelligence (AAAI)*, pages 1495–1503. AAAI Press, 2020. 2.3.3, 3.2

[44] Daniel Faber, Adalat Jabrayilov, and Petra Mutzel. SAT Encoding of Partial Ordering Models for Graph Coloring Problems. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:20. Schloss Dagstuhl, 2024. 4.3

[45] Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995. 2.3.1

[46] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *Lecture Notes in Computer Science (LNCS)*, 2006. 2.2.4

[47] Ian P. Gent. Arc consistency in SAT. In *European Conference on Artificial Intelligence*, 2002. 2.2

[48] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25. Dagstuhl - Leibniz-Zentrum für Informatik, 2022. 5.1.5

[49] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19. Schloss Dagstuhl, 2023. 2.2.1, 2.3.1, 4.3.4

[50] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–

308, 1985. 2.2.1, 2.3.3

[51] Marijn J. H. Heule. Schur number five. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 2018. 2

[52] Marijn J. H. Heule and Manfred Scheucher. Happy ending: An empty hexagon in every set of 30 points. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 14570 of *Lecture Notes in Computer Science (LNCS)*. Springer Nature Switzerland, 2024. 2, 6

[53] 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 *Lecture Notes in Computer Science (LNCS)*, 2011. 6

[54] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science (LNCS)*, pages 228–245. Springer, 2016. 2

[55] Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, 2018. 4.3.4

[56] Marijn JH Heule and Hans Van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modelling and Computation*, 2(1-4): 47–59, 2006. 6.3

[57] J. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12: 217–239, 1988. 2.3.2

[58] 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)*, volume 10929 of *Lecture Notes in Computer Science (LNCS)*, pages 428–437, 2018. 2.2.3

[59] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Computer Science (LNCS)*, pages 355–370. Springer, 2012. 2.3.4, 4.3.3

[60] Kai Jia and Martin Rinard. Efficient exact verification of binarized neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1782–1795. Curran Associates, Inc., 2020. 2, 2.1.4, 2.3.2, 5.2

[61] Matti Järvisalo, Jeremias Berg, Ruben Martins, and Andreas Niskanen. MaxSAT Evaluation 2023 Benchmarks. `https://maxsat-evaluations.github.io/2023/benchmarks.html`, 2023. Accessed: 2024-08-13. 6.4

[62] Will Klieber and Gihwon Kwon. Efficient CNF encoding for selecting 1 from n objects. In *Constraints in Formal Verification (CFV)*, page 39, 2007. 2.2.2, 2.2.2

[63] Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. Practically Feasible Proof Logging for Pseudo-Boolean Optimization. In Maria Garcia de la Banda, editor, *Principles and Practice*

of Constraint Programming (CP), volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27. Schloss Dagstuhl, 2025. 2.3.4

[64] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science (LNCS)*, pages 389–391. Springer, 2014. 2

[65] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454. ACM, 2024. 2

[66] Zhendong Lei, Shaowei Cai, and Chuan Luo. Extended conjunctive normal form and an efficient algorithm for cardinality constraints. In Christian Bessiere, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1141–1147. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track. 2.1.3, 2.2.5

[67] Zhendong Lei, Shaowei Cai, Chuan Luo, and Holger Hoos. Efficient local search for pseudo boolean optimization. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 12831 of *Lecture Notes in Computer Science (LNCS)*, pages 332–348. Springer, 2021. 2.1.3

[68] Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artificial Intelligence*, 279(C), feb 2020. 2.3.1, 5.1.3

[69] Jia Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science (LNCS)*, pages 123–140, 2016. 2.3.1

[70] Jordyn C. Maglalang. Native cardinality constraints: More expressive, more efficient constraints, 2019. 2.2.5, 2.3.2, 5, 5.1.1

[71] Norbert Manthey and Peter Steinke. Quadratic direct encoding vs. linear order encoding a one-out-of-n transformation on cnf. Technical Report KRR Report 11-03, Technische Universität Dresden, 2011. 3.3

[72] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *Haifa Verification Conference (HVC)*, volume 7857 of *Lecture Notes in Computer Science (LNCS)*, pages 102–117, 2013. 2.2.1

[73] Joao Marques-Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 4741 of *Lecture Notes in Computer Science (LNCS)*, pages 483–497. Springer Berlin, 2007. 2.2.3

[74] João Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009. 2.3.1

[75] Joao Marques-Silva, Mikoláš Janota, Alexey Ignatiev, and Antonio Morgado. Efficient model based diagnosis with maximum satisfiability. In *International Joint Conference on*

Artificial Intelligence (IJCAI), IJCAI'15, page 1966–1972. AAAI Press, 2015. 6.4

[76] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In *Principles and Practice of Constraint Programming*, volume 8656 of *Lecture Notes in Computer Science (LNCS)*, pages 531–548. Springer, 2014. 2.2.4

[77] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-wbo: A modular MaxSAT solver,. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *Lecture Notes in Computer Science (LNCS)*, pages 438–445. Springer, 2014. 2

[78] 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. 2.2.3, 3.1

[79] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, page 530–535. ACM, 2001. 2.3.1

[80] Gi-Joon Nam, Fadi Aloul, Karem Sakallah, and Rob Rutenbar. A comparative study of two boolean formulations of fpga detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design*, ISPD '01, page 222–227. ACM, 2001. 4.3.4

[81] Van-Hau Nguyen and Son T. Mai. A new method to encode the at-most-one constraint into sat. In *Proceedings of the 6th International Symposium on Information and Communication Technology*, SoICT '15, page 46–53. ACM, 2015. 2.2.2

[82] Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. Empirical study on SAT-encodings of the at-most-one constraint. In *Conference on Smart Media and Applications*, Smart Media and Applications (SMA), page 470–475. ACM, 2021. 2.2.3

[83] Aina Niemetz and Mathias Preiner. Bitwuzla. In *Computer Aided Verification (CAV)*, volume 31965 of *Lecture Notes in Computer Science (LNCS)*, page 3–17. Springer, 2023. 2

[84] 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 *Tools with Artificial Intelligence (ICTAI)*, pages 9–17, 2013. 2.2.3, 2.2.4, 3.1

[85] Chanseok Oh. Between SAT and UNSAT: The fundamental difference in CDCL SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 307–323. Springer, 2015. 2.3.1

[86] Guoqiang Pan and Moshe Y. Vardi. Search vs. symbolic techniques in satisfiability solving. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3542 of *Lecture Notes in Computer Science (LNCS)*, pages 235–250, 2005. 3.2.3

[87] Justyna Petke and Peter Jeavons. The order encoding: from tractable csp to tractable sat. Technical Report RR-11-04, DCS, University of Oxford, 2011. 4.3

[88] Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. Moving definition variables in quantified Boolean formulas. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12651 of *Lecture Notes in Computer Science (LNCS)*, pages 76–93, 2022. 1.4

[89] Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. Preprocessing of propagation redundant clauses. *Journal of Automated Reasoning (JAR)*, 67(3), September 2023. 1.4

[90] Joseph E. Reeves, Benjamin Kiesl-Reiter, and Marijn J. H. Heule. Propositional proof skeletons. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 13993 of *Lecture Notes in Computer Science (LNCS)*, page 329–347. Springer, 2023. 1.4

[91] Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In *Computer Aided Verification (CAV)*, volume 14681 of *Lecture Notes in Computer Science (LNCS)*, pages 110–132. Springer, 2024. 1.3, 2.1.2, 3.2, 3.2.1, 4.1, 5.1

[92] Joseph E. Reeves, João Filipe, Min-Chien Hsu, Ruben Martins, and Marijn J. H. Heule. The impact of literal sorting on cardinality constraint encodings. In *AAAI Conference on Artificial Intelligence*, volume 39, pages 11327–11335. AAAI Press, 2025. 1.3, 4.2

[93] Neha Rungta. A billion SMT queries a day. In *Computer Aided Verification (CAV)*, pages 3–18. Springer, 2022. 2

[94] John S. Schlipf, Fred S. Annexstein, John V. Franco, and R. P. Swaminathan. On finding solutions for extended horn formulas. *Inf. Process. Lett.*, 54(3):133–137, May 1995. 4.3.3

[95] Amar Shah, Twain Byrnes, Joseph E. Reeves, and Marijn J. H. Heule. Learning short clauses via conditional autarkies. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2025. 1.4

[96] Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule. Reencoding Unique Literal Clauses. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:21. Schloss Dagstuhl, 2025. 1.3, 2.1.2, 3.3, 4.3

[97] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 3709 of *Lecture Notes in Computer Science (LNCS)*, pages 827–831, 2005. 2.2.2, 2.2.3

[98] Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental SAT-based method with native boolean cardinality handling for the Hamiltonian cycle problem. In *European Conference on Logics in Artificial Intelligence*, volume 8761, page 684–693. Springer, 2014. 2.2.5

[99] StarExec. Starexec. https://starexec.org/starexec/public/about.jsp, 2024. Accessed: 2024-08-14. 1.3

[100] Jakob Nordström Stephan Gocht, Ciaran McCreesh. Veripb: The easy way to make your combinatorial search algorithm trustworthy. In *From Constraint Programming to Trustworthy AI*, 2020. 2.3.4

[101] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science (LNCS)*, pages 367–373. Springer, 2014. 1.3

[102] Bernardo Subercaseaux, John Mackey, Marijn J. H. Heule, and Ruben Martins. Automated

mathematical discovery and verification: Minimizing pentagons in the plane. In *Intelligent Computer Mathematics*, pages 21–41. Springer Nature Switzerland, 2024. 6

[103] Bernardo Subercaseaux, Ethan Mackey, Long Qian, and Marijn J. H. Heule. Automated symmetric constructions in discrete geometry, 2025. 2.1.4, 2.4, 5.2.1

[104] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009. 3.1, 4.3

[105] Naoyuki Tamura, Mutsunori Banbara, and Takehide Soh. Compiling pseudo-boolean constraints to SAT with order encoding. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 1020–1027, 2013. 4.3

[106] 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 *Lecture Notes in Computer Science (LNCS)*, pages 223–241, 2021. 2.3.4

[107] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys.*, 54(7), July 2021. 2

[108] VA Traag, L Waltman, and NJ Van Eck. From louvain to leiden: guaranteeing well-connected communities. sci. rep. 9, 5233, 2019. 4.2.4

[109] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9:1–12, 2019. 4.2.1, 4.2.3

[110] Gregori S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer, 1983. 2.2.1

[111] 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 *Lecture Notes in Computer Science (LNCS)*, pages 422–429, 2014. 2.3.4

[112] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conference (DAC)*, page 542–545. ACM, 2001. 5

[113] Ed Wynn. A comparison of encodings for cardinality constraints in a SAT solver. *ArXiv*, abs/1810.12975, 2018. 5.1.6

[114] Jiong Yang, Yaroslav A. Kharkov, Yunong Shi, Marijn J. H. Heule, and Bruno Dutertre. Quantum Circuit Mapping Based on Incremental and Parallel SAT Solving. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18. Schloss Dagstuhl, 2024. 2

[115] Jiong Yang, Yong Kiam Tan, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel. Efficient Certified Reasoning for Binarized Neural Networks. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:22. Schloss Dagstuhl, 2025. 2, 2.1.4, 2.3.2, 2.3.4, 5.2

[116] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Fast bit-vector satisfiability.

In *International Symposium on Software Testing and Analysis (ISSTA)*, page 38–50. ACM, 2020. 2