The Impact of Literal Sorting on Cardinality Constraint Encodings

Joseph E. Reeves[1], **João Filipe**[1,2], **Min-Chien Hsu**[1], **Ruben Martins**[1], **Marijn J. H. Heule**[1]

[1]Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
[2]University of Amsterdam, The Netherlands
{jereeves,minchieh,rubenm,mheule}@andrew.cmu.edu, j.sa@uva.nl

Abstract

The effectiveness of satisfiability solvers strongly depends on the quality of the encoding of a given problem into conjunctive normal form. Cardinality constraints are prevalent in numerous problems, prompting the development and study of various types of encoding. We present a novel approach to optimizing cardinality constraint encodings by exploring the impact of literal orderings within the constraints. By strategically placing related literals nearby each other, the encoding generates auxiliary variables in a hierarchical structure, enabling the solver to reason more abstractly about groups of related literals. Unlike conventional metrics such as formula size or propagation strength, our method leverages structural properties of the formula to redefine the roles of auxiliary variables to enhance the solver's learning capabilities. The experimental evaluation on benchmarks from the maximum satisfiability competition demonstrates that literal orderings can be more influential than the choice of the encoding type. Our literal ordering technique improves solver performance across various encoding techniques, underscoring the robustness of our approach.

Code — https://github.com/jreeves3/LiteralSorting

Introduction

Over the past two decades, Boolean satisfiability (SAT) solving, in particular conflict-driven clause learning (CDCL) (Marques-Silva, Lynce, and Malik 2021), has been applied to problems across many domains including planning (Rintanen, Heljanko, and Niemelä 2006), quantum circuit synthesis (Yang et al. 2024), cryptography (Soos, Nohl, and Castelluccia 2009), hardware (Biere et al. 1999) and software verification (Kroening and Tautschnig 2014), and combinatorial mathematics (Heule 2018). The effectiveness of CDCL solvers on such a wide-ranging set of problems is surprising, especially given that these problems must all be reduced to the same simple input format, conjunctive normal form (CNF), erasing all high-level, problem-specific information.

Encoding a problem into CNF requires the introduction of *auxiliary variables*, that is, new variables introduced into the formula that abstract information, to prevent an exponential blow-up in size. Auxiliary variables do much more than keep the formula small: They represent abstractions over the original problem variables that can be leveraged during clause learning to help a solver find short proofs.

Thus, good encodings, with the right abstractions, are crucial to the performance of modern CDCL solvers. Previous work on developing new types of encodings for various high-level constraints, such as all-different (Gent and Nightingale 2004), cardinality (Marques-Silva and Lynce 2007), and XOR constraints (Bard, Courtois, and Jefferson. 2007), has focused on optimizing structural features, including the size of the encoding, i.e., the number of clauses and auxiliary variables introduced, and their propagation power.

In this paper, we shift the focus from optimizing structural features of a specific encoding towards finding the right literal ordering for any encoding. We sort together related literals within cardinality constraints so that the auxiliary variables in the encoding summarize the related groups. We find that improving the meaning of auxiliary variables is often more important than choosing the best encoding type.

Cardinality constraints are one of the most important high-level constraints in SAT, appearing frequently (Reeves, Heule, and Bryant 2024), often as resource bounds for optimization problems. For example, the cardinality constraint $x_1 + x_2 + \cdots + x_n \leq k$ is satisfied if at most k of the *data literals* $x_1, x_2 \ldots, x_n$ are satisfied. Cardinality constraints have been the subject of encoding research for decades. The most common encodings, totalizer (Bailleux and Boufkhad 2003; Ogawa et al. 2013; Morgado, Ignatiev, and Marques-Silva 2015), sorting network (Batcher 1968; Eén and Sörensson 2006), cardinality network (Asín et al. 2009; Asín et al. 2011; Abío et al. 2013; Karpinski and Piotrów 2019), and sequential counter (Sinz 2005) break the cardinality constraint into subproblems via trees, networks, or grids, and are often compared by the encoding size and performance (Nguyen et al. 2021; Martins, Manquinho, and Lynce 2011; Karpinski and Piotrów 2019). For each of these encodings, the order of data literals will impact the meaning of auxiliary variables within the encoding.

Example 1 Consider the following formula with one cardinality constraint that has not been encoded yet:

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

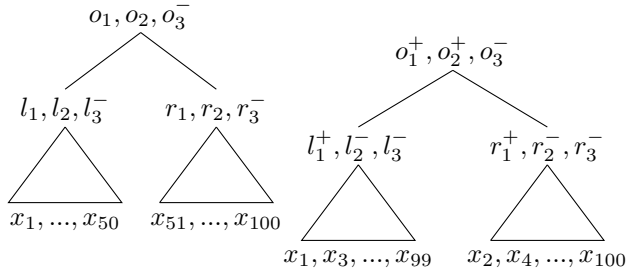

Figure 1: Two ktotalizer encodings for the cardinality constraint in Example 1 with two literal orderings: $x_1 + x_2 + \cdots + x_{100} \leq 2$ (a) and $x_1 + x_3 + \cdots + x_{99} + x_2 + x_4 + \cdots + x_{100} \leq 2$ (b). The auxiliary variables l_i, r_i, o_i are counters for true input data literals, so l_3, r_3, o_3 are set to false ($^-$), to enforce the bound of at most two. Additional values, true denoted by ($^+$), can be derived in (b) via probing.

In Figure 1 (b), a solver can learn the units l_1 and r_1 through failed literal probing (Freeman 1995): Assigning l_1 to false would propagate all the input literals for the left subtree to false, causing a conflict with the odd-literal clause. Similarly, r_1 can be learned. This propagates o_1 and o_2 to true and then l_2 and r_2 to false, since the sum of the children cannot exceed 2. In short, the auxiliary variables l_1 and r_1 allow the solver to reason about the entire set of data literals in either subtree and cheaply derive units. In Figure 1 (a), l_1 and r_1 summarize different sets of data literals preventing reasoning about the clauses. Leaving the solver to reason over the entire ktotalizer encoding with no units learned.

This example sheds light on the importance of a good literal ordering, and more importantly, that such an ordering may be nonintuitive, for example, the ordering of even integers then odd integers. We propose several automated techniques that use the structure of a formula to sort literals within a cardinality constraint, modifying the meaning of auxiliary variables without changing the size of the encoding. In an experimental evaluation of benchmarks from the maximum satisfiability evaluation (MaxSAT) (Järvisalo et al. 2023), we find that literal sorting significantly improves solver performance for all common encoding types for cardinality constraints. Furthermore, we find that selecting the best ordering is often more important than selecting the best encoding type to improve performance.

Background

Boolean Satisfiability We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula F is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal is either a variable x (positive literal) or a negated variable $\overline{x}$ (negative literal). An *assignment* α is a mapping from variables to truth values 1 (*true*) and 0 (*false*). Assignment α *satisfies* a positive (negative) literal if α maps $\mathsf{var}(x)$ to true (α maps $\mathsf{var}(\overline{x})$ to false, respectively), and *falsifies* it if α maps $\mathsf{var}(x)$ to false (α maps $\mathsf{var}(\overline{x})$ to true, respectively). An assignment satisfies a clause if the clause contains a literal satisfied by the assignment, and satisfies a formula if every clause in the formula is satisfied by the assignment. A formula is *satisfiable* if there exists a satisfying assignment, and *unsatisfiable* otherwise.

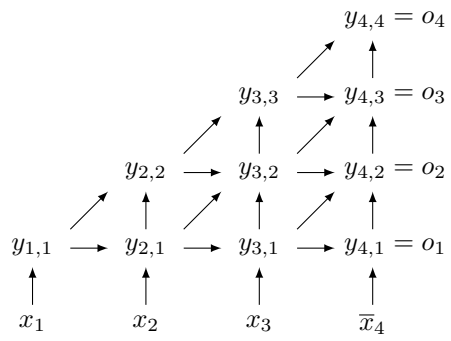

Figure 2: Sequential counter of Example 2.

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

Cardinality Constraints A *cardinality constraint* consists of a set of *data literals*, a comparison operator ($>, \geq, <, \leq$), and a bound k, e.g., the at-most-k (AMK) cardinality constraint $x_1 + x_2 + \cdots + x_n \leq k$ is satisfied if at most k of the literals are satisfied. *Literal sorting* is the process of selecting the order in which data literals appear in a cardinality constraint, independent of the *variable naming*, i.e., the explicit integer naming of variables.

To keep the formula size small, cardinality constraints are not encoded directly into CNF. Instead, an encoding abstracts the cardinality constraint into layers of subproblems, introducing auxiliary variables to summarize the information output from each layer. Encoding types are differentiated by their size (clauses and auxiliary variables), the structure of the abstractions, and their propagation power. A clausal encoding of $x_1 + x_2 + \cdots + x_s \leq k$ is *consistent* if assigning any $k+1$ literals to true always results in a conflict by unit propagation, and *arc-consistent* (Gent 2002) if additionally unit propagation assigns all unassigned literals to false if exactly k literals are assigned to true. The most common cardinality constraint encodings are the totalizer (Bailleux and Boufkhad 2003), sorting network (Batcher 1968), cardinality network (Asín et al. 2009), and sequential counter (Sinz 2005) and each is arc-consistent. We refer the reader to the citations for specific properties.

Example 2 Consider the following cardinality constraint:

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

The sequential counter, e.g., Figure 2, uses auxiliary variables $y_{i,j} (1 \leq i \leq n, 1 \leq j \leq n)$ in a grid to count the

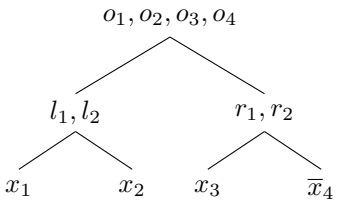

Figure 3: Totalizer of Example 2.

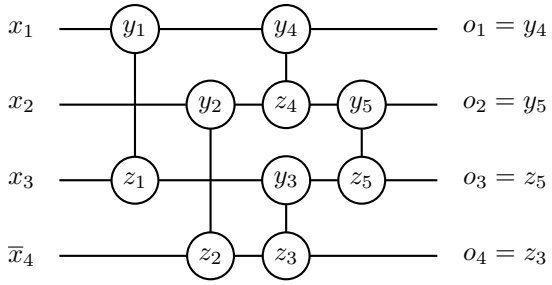

Figure 4: Sorting network of Example 2.

true data literals, where $y_{i,j}$ is true if at least j of the first i data literals are true. Literals in the last column, $y_{n,j}$, 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 2 removing $y_{4,4}$. Auxiliary variables interact locally in the grid, e.g., $y_{i,j} \rightarrow y_{i+1,j}$ and $(y_{i-1,j} \wedge x_i) \rightarrow y_{i,j+1}$. The sequential counter is asymmetrical, with auxiliary variables on the left-hand side summarizing information from fewer data literals. For example, in Figure 2, a solver can reason about the pair x_1, x_2 via the auxiliary variables $y_{2,1}$ (at least one of the pair is true) and $y_{2,2}$ (both are true), but no two auxiliary variables allow similar reasoning about the pair x_3, $\overline{x}_4$.

The totalizer, e.g., Figure 3, uses a binary tree to incrementally count the number of true data literals at each level. Data literals form the leaves, and each node has auxiliary variables representing the unary count from the sum of its children counters. For example, in Figure 3, variable o_3 is true if either pairs l_1, r_2 or l_2, r_1 are true. The bound k is enforced by adding the unit $\overline{o}_{k+1}$. The modulo totalizer (mtotalizer) (Ogawa et al. 2013) uses a quotient and remainder at each node to reduce the number of auxiliary variables required to count the sum. The encoding can be simplified by only encoding the count up to $k+1$ at each node (kmtotalizer) (Morgado, Ignatiev, and Marques-Silva 2015). Unlike the sequential counter, the totalizer splits the subproblems symmetrically, so a solver can reason about x_3 and $\overline{x}_4$ via the auxiliary variables r_1 (at least one of the pair is true) and r_2 (both are true). The further apart literals are in the ordering, the more levels of abstraction are present in their shared counters. For example, x_1 and $\overline{x}_4$ do not share a counter until the root, two levels of abstraction away from the data literals. Furthermore, a node's counters can only be used to reason about all of the data literals below it together, e.g., o_1 means at least one of the four data literals is true. With the right ordering, reasoning over a large set of data literals at once can be advantageous, as seen in Example 1.

The sorting network, e.g., Figure 4, 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. 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.

Shuffling data literals in the figures above will not change their size, but will change the meaning of auxiliary variables. The CARDENC module in PySAT (Ignatiev, Morgado, and Marques-Silva 2018) provides an API to encode cardinality constraints into CNF using the specified encodings.

Literal Sorting Methods

In this section, we present several methods for sorting literals within a cardinality constraint. In order of complexity, the methods are Natural, Random, Occur, Proximity, PAMO, and Graph. Given a SAT problem defined as a set of clauses and cardinality constraints, each method produces a single ordering on all variables occurring in cardinality constraints, and this ordering is used to sort the literals (interpreted as variables in the ordering) within the cardinality constraints. However, the approach of sorting literals will modify the meaning of auxiliary variables without affecting the number of variables or clauses in the encoding and can be used with off-the-shelf encoding APIs. We apply each sorting method to the cardinality constraint in Example 3.

Example 3 Consider the following formula

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

The Natural ordering is the simplest method, given by increasing variable names, which are integer values identifying each variable in the formula: $x_1 + x_2 + x_3 + \overline{x}_4 \leq 2$. This method typically reflects the underlying problem's structure, as most formula generators enumerate variable names in a logical way. For example, a list of variables for a problem over a grid would naturally be enumerated either row-wise or column-wise. Note that this may be different from the order in which literals appear in constraints.

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.

Occurrence and Proximity Orderings

Next, we consider orderings that are derived from the clausal structure of the input problem. The Occur 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.

The Proximity method orders variables both by occurrence and proximity. It can be extended by detecting at-most-one (AMO) constraints from the clauses in the problem. We detect AMO constraints with more than 5 variables using the BDD-based Guess&Verify tool (Reeves, Heule, and Bryant 2024), which works on all commonly used types of AMO encoding. The Proximity method performs a BFS-like search over the clauses, using variable scores, initialized to 0, to select the next variable. We do not consider polarity, so literals inside a clause or AMO constraint are interpreted as variables. The Proximity algorithm is presented below:

1. Select the unprocessed variable v (i.e., not part of the ordering yet) with the highest score. If the highest score is 0, select the most occurring unprocessed variable.
2. Append v to the ordering.
3. If AMO detection is enabled, for each AMO constraint K that contains v, increment the scores of unprocessed variables occurring in K by $len(K)^2$, where the length of an AMO 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 $1/len(C)$ 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.

As long as the formula is fully connected, only the first variable selected in step 1 will be picked based on the occurrence count and the rest will be selected based on score. At a high level, variables occurring in short clauses together will be ordered close together, with an even stronger bias towards variables that occur in large AMO constraints together. So, the score function differentiates variables with the small $1/len(C)$ increments, and orders variables with the large $len(C)^2/len(K)^2$ increments. To make the selection deterministic in case two variables have the same score, we prefer the variable that was seen earliest in the search. The main idea behind this method is that grouping literals that occur close together in the formula will localize the meaning of auxiliary variables in the cardinality constraint encoding, giving a solver the ability to reason about independent sections of the formula. Further, if variables occur together in an AMO constraint grouping them together makes it possible to reason about the entire AMO constraint, e.g., by learning unit auxiliary variables in the encoding stating at-most-one of the grouped data literals is true, and the same applies for binary clauses (an AMO constraint on the two literals negated). Computing the Proximity can be costly due to score updates. Given n variables, in the worst case, each clause is traversed n times (once each iteration) and each clause contains n literals, giving $O(n^3)$ complexity. The algorithm exits once all variables in cardinality constraints have been added to the ordering, saving time if the formula has a large number of variables but only a small subset occurring in cardinality constraints.

Example 4 Applying the Proximity algorithm without AMO detection to the clauses in Example 3, the first variable selected is x_2 since it occurs the most. 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 $1/3$. 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 $1/3$ and finally x_4, which yields: $x_2 + x_1 + x_3 + \overline{x}_4 \leq 2$.

Graph-based Ordering

Finally, we consider orderings extracted from graphs constructed from the literals and clauses of the problem (Ansótegui, Giráldez-Cru, and Levy 2012). The Variable Incidence Graph is an undirected, unweighted graph $G = (V, E)$, where V denotes the set of nodes representing each variable of the problem, and E denotes the set of edges. Each edge (i, j) connects two nodes if the corresponding variables, regardless of their polarity, share a clause.

We use the Louvain Community Detection algorithm (Traag, Waltman, and Van Eck 2019). 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 sets of communities 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. We believe that this detailed partitioning could potentially lead to more effective variable orderings, as it allows for a more targeted approach to handling different parts of the problem. Furthermore, since the variables within each community are ordered by their original naming, a higher number of communities will also lessen the reliance on the original ordering. Among these sets of communities, we then choose the one with 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.

Example 5 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 sets of communities, as they contain the highest number of communities. Next, we would choose the third set, as the community sizes show a smaller deviation from the average size. Lastly, we concatenate the variables from all communities, yielding: $x_1 + x_3 + x_2 + \overline{x}_4 \leq 2$.

Experimental Evaluation

We ran experiments on StarExec (Stump, Sutcliffe, and Tinelli 2014). The specs can be found online (StarExec 2024). Each experiment had 32 GB of memory and an 1,800 second timeout. We present average PAR-2 scores: the average runtime with timeouts counted twice. A runtime is the sum of literal sorting, clausal encoding, and solving.

The main literal sorting configurations are the following: Natural, Random (from 5 random permutations we reported the best (BestRandom) and worst (WorstRandom) times for each formula), Occur, Proximity, and Graph (computing communities up to 50 times with a 300 second timeout). Proximity can be extended with the AMO detection (PAMO) restricted by a 50 second timeout. Additionally, we ran Natural for 100 seconds then restarted with PAMO if the formula was unsolved (Natural+PAMO) and ran PAMO for formulas with fewer than one million clauses and occurrence otherwise (PAMO+Occur).

Given a problem represented as a set of clauses and cardinality constraints, the tool chain sorts literals in the cardinality constraints, encodes the cardinality constraints into CNF using PySAT, then runs the CDCL SAT solver CaDiCaL (Biere et al. 2020) on the resulting CNF formula.

MaxSAT Competition Benchmarks

Our evaluation uses MaxSAT benchmarks from the 2023 competition unweighted track (Järvisalo et al. 2023). Each MaxSAT formula contains a set of hard clauses and soft units. The goal is to find the optimum (minimum) number of soft units that must be falsified while satisfying all hard clauses. MaxSAT problems can be converted to SAT by combining the hard clauses with one, often large, cardinality constraint stating at most k soft units are falsified. We can generate a satisfiable SAT problem by making the cardinality constraint's bound the optimum, and an unsatisfiable SAT problem by making the bound the optimum minus one. We consider problems with known optimums and bounds greater than 1 and less than one minus the number of soft units, ensuring that the resulting cardinality constraint is not a clause. This gives 398 satisfiable and 398 unsatisfiable SAT problems.

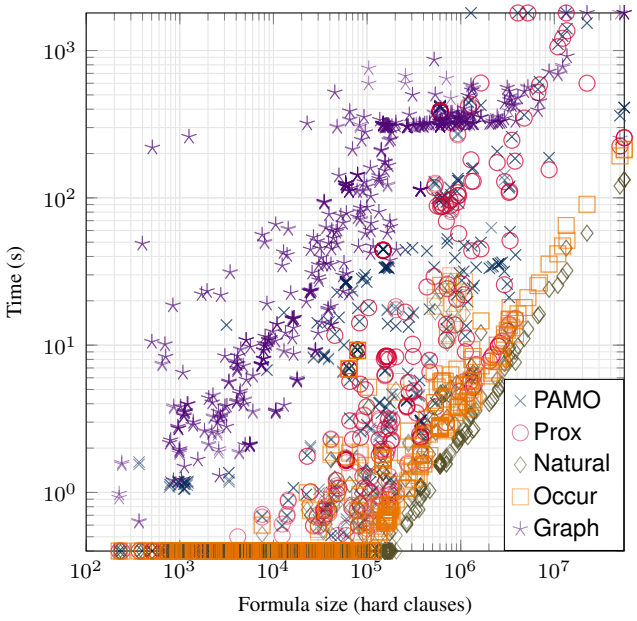

Figure 5: Preprocessing time that includes literal sorting and clausal encoding.

These benchmarks are a suitable choice for evaluating cardinality constraint encodings. The cardinality constraints are crucial and must be utilized in solver reasoning, as they enforce nontrivial (optimal) bounds. This significance allows us to compare the impact of different encodings more accurately. In contrast, cardinality constraints in other benchmarks may be less relevant to solving the problem, and performance differences between encodings could be influenced by other factors like branching heuristics, introducing noise into the comparison. Furthermore, the MaxSAT Competition provides a diverse set of 30 benchmark families, each with approximately 10 instances, highlighting the robustness of our techniques.

Figure 5 shows the preprocessing times on different-sized formulas. The Natural ordering is given by the variable names, so the preprocessing time amounts to parsing and encoding the problem into CNF. The parsing time scales linearly with the number of hard clauses, but the encoding time depends on the size of the cardinality constraint, as determined by the number of soft units and the optimum. Random (not displayed) is nearly identical to Natural, with a negligible cost for generating a random permutation for literal sorting. Counting occurrences requires a single pass over the formula, creating a slight slowdown between Occur and Natural. Proximity and Graph have larger overheads. Proximity can take more than 100 seconds for formulas over 100,000 clauses, and 1,000 seconds for formulas over 1,000,000 clauses. The difference between PAMO and Proximity is more pronounced on smaller formulas when AMO detection with a 50 timeout outweighs the cost of Proximity. SAT solvers will forego preprocessing, execute for some time, then run preprocessing as inprocessing, to avoid the preprocessing overhead on quickly solved prob-

Encoding	Nat	P+O	Prox	Occur	BRand
kmtotalizer	635	**670**	655	588	561
mtotalizer	623	**653**	643	572	547
cardinality network	608	**639**	629	565	544
sorting network	602	**645**	631	561	532
sequential counter	597	**617**	611	563	539

Table 1: MaxSAT 2023 instances solved on the generated SAT problems with various literal sortings for the optimal bound constraint: Natural (Nat), PAMO+Occur (P+O), Proximity (Prox), Occur,and BestRandom (BRand)

Ordering	Solved		Par2 (s)	
	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
Proximity	343	312	591	856
Graph	332	312	762	949
Natural	334	301	635	916
Occur	317	271	792	1189
BestRandom	313	248	818	1388
WorstRandom	284	244	1106	1434

Table 2: kmtotalizer with additional literal sorting methods. Solved instances and Par2 scores are presented for satisfiable (SAT) and unsatisfiable (UNSAT) formulas. VBS is the virtual best solver, i.e., a solver that picks the best literal sorting method for each formula.

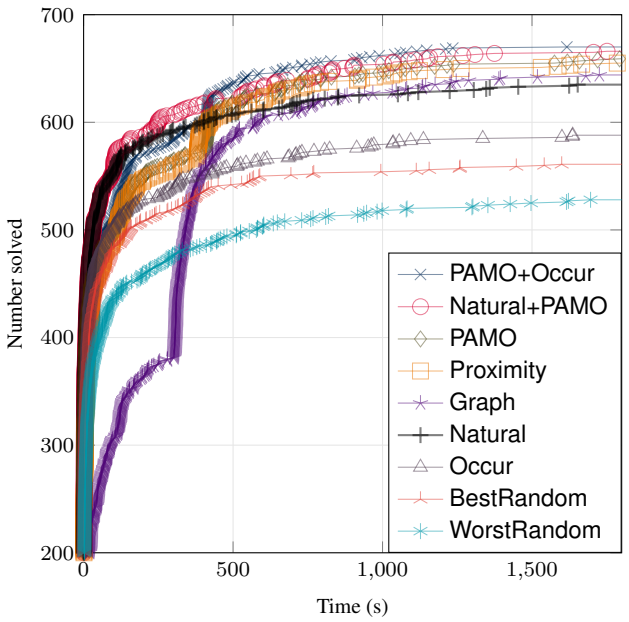

Figure 6: The kmtotalizer encoding with all literal sorting methods on all MaxSAT formulas. Runtime includes the preprocessing and solving time.

lems. This motivated the Natural+PAMO approach, but for some large formulas not solved in 100 seconds the Proximity computation after the restart may timeout. To address this, PAMO+Occur uses the cheap Occur method for larger formulas. For many large formulas, the Graph approach cannot compute the community structure 50 times, instead exiting at 300 seconds or more if a single execution of community detection exceeds 300 seconds.

Table 1 shows the impact of literal sorting on the solving time for the five encoding types. When considering the Natural ordering, the kmtotalizer solves the most formulas; however, when using the PAMO+Occur ordering the mtotalizer, cardinality network, and sorting network solve more instances than the Natural kmtotalizer. In other words, choosing the best encoding type, kmtotalizer, is less impactful than choosing the best literal ordering, PAMO+Occur, with the other encoding types. The Occur ordering does not perform as well as the Natural, so a cheap heuristic that ignores the structure of the formula is not good enough to match default performance. Furthermore, even when selecting the best runtime from five random permutations Random performs the worst, suggesting a bad ordering cannot be adequately compensated for by a good choice of encoding type. The sequential counter not only performs the worst with Natural, but has the smallest performance gain with PAMO+Occur. The sequential counter's asymmetry may be stifling the impact of literal ordering.The sorting network has the biggest increase from Natural to PAMO+Occur. The intermediary auxiliary variables (outputs of swap gates) have unclear meaning, but the hierarchical nature of the sorting network allows the solver to reason over layers of abstractions. Surprisingly, the sorting network overtakes the cardinality network with PAMO+Occur, indicating that simplifications in the cardinality network (fewer clauses and fewer auxiliary variables) might limit a solver's learning capabilities. This is not the case for the totalizer, with the simplified kmtotalizer outperforming the mtotalizer.

Table 2 shows the fine-grained results for all literal sorting methods when using the kmtotalizer encoding. The Par2 scores for PAMO+Occur show that a good ordering affects both satisfiable and unsatisfiable formulas. The gap between PAMO+Occur and the VBS implies that some problems require different orderings, and PAMO+Occur is not one-size-fits-all. The Par2 includes preprocessing, so the overhead of Proximity (see Table 5) is worth the cost to improve the solving time. On the other hand, the Graph approach solves more instances than Natural but has a larger Par2 score due in large part to the preprocessing time.

Five configurations in Figure 6 solve more instances than Natural. Natural+PAMO is consistently better than Natural by solving easy problems in the first 100 seconds with Natural and solving harder problems with PAMO. The other four configurations have a preprocessing overhead for easier formulas, with the Proximity configurations taking around 400 seconds and the Graph approach taking around 800 seconds to meet Natural. Note, the Par2 scores for the Proximity approaches are still lower than Natural, so their slow start pays

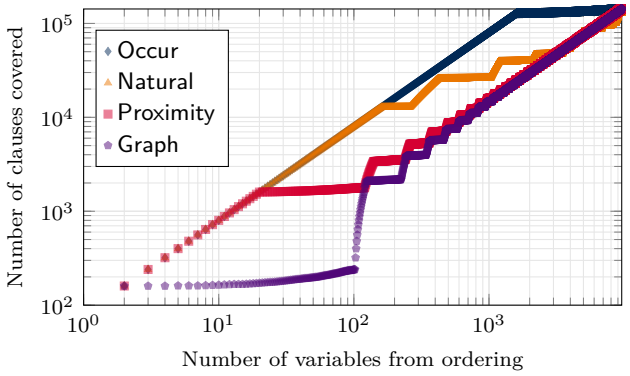

Figure 7: Clause coverage for unsatisfiable extension-enforcement-extension-enforcement-strict-com formula. Solving times: Occur (timeout), Natural (timeout), Proximity (9), and Graph (308).

off with many more instances solved. Occur performs closer to the best Random than Natural, highlighting the need for complex sorting methods like Proximity.

Clause Coverage

One possible explanation for the effectiveness of the Proximity and Graph orderings is that they split the formula into equally-sized sections such that introduced auxiliary variables have similar levels of meaning across the formula. To quantify this, we introduce the notion of *clause coverage*. A clause C is covered by a set of variables S if all of the variables in C are in S. We track the number of clauses covered by the first i variables in the generated ordering.

We consider a formula from the extension enforcement family (Niskanen, Wallner, and Järvisalo 2018). The Proximity and Graph methods consistently outperform Natural and Occur on this family. In Figure 7, the clause coverage for Proximity and Graph is more consistent than Natural and Occur, which both cover more clauses with the first half of their ordering. The imbalance may inhibit the solver's learning capabilities, with auxiliary variables from the first half of the encoding representing information that is too coarse-grained (large parts of the formula) and auxiliary variables from the second half representing information that is too fine-grained (small parts of the formula).

Related Works

Many types of cardinality constraint encoding have been introduced in the past two decades (Batcher 1968; Eén and Sörensson 2006; Sinz 2005; Bailleux and Boufkhad 2003; Ogawa et al. 2013; Morgado, Ignatiev, and Marques-Silva 2015; Asín et al. 2009; Asín et al. 2011; Abío et al. 2013; Karpinski and Piotrów 2019; Jabbour, Sais, and Salhi 2013). They are generally evaluated based on their size (number of clauses and auxiliary variables), propagation (arc-consistency), and performance on different benchmark sets. Most research focuses on reducing the size of an encoding, as this tends to be more performant (Asín et al. 2011). Some limited approaches have implemented lazy encodings that introduce auxiliary variables during solving (Abío and Stuckey 2012). For pseudo-Boolean constraints, ordering the literals by coefficient can lead to smaller encodings (Eén and Sörensson 2006). However, this technique is directed at reducing the size and cannot be used for cardinality constraints where all coefficients have value 1. In our work, we do not change the size but instead focuses on shuffling literals within the constraints to modify the meaning of auxiliary variables and improve the effectiveness of clause learning.

The theoretic importance of auxiliary variables is well-known, with small proofs existing for the pigeon-hole problem using new variables (Cook 1976). Recently, SAT preprocessing tools have tried introducing auxiliary variables to improve performance (Haberlandt, Green, and Heule 2023; Reeves, Heule, and Bryant 2024), and their success motivated our exploration into the impact of auxiliary variable meanings within cardinality constraints.

Graph-based algorithms, especially community detection algorithms, have been used to understand the community structure of industrial SAT benchmarks (Ansótegui, Giráldez-Cru, and Levy 2012), the relationship between community structure and performance (Newsham et al. 2014), to explain the usefulness of different CDCL heuristics (Ansótegui et al. 2015), and in MaxSAT to partition the MaxSAT formula into subformulas (Martins, Manquinho, and Lynce 2013; Neves et al. 2015). In this paper, we find another use of graph representations to sort the literals in cardinality constraints and show that despite the performance cost of running the community detection algorithm multiple times, it can still lead to some improvements.

Conclusion

The surprising success of CDCL SAT solvers across many problem domains is in large part due to the introduction of auxiliary variables in clausal encodings, which allow the solver to reason over abstractions. In this work, we proposed several automated methods for sorting the literals within cardinality constraints, shuffling the literals within the clausal encoding and, therefore, changing the meaning of auxiliary variables. Our Proximity methods significantly improved solver performance on all encoding types we studied, and we found that a good literal ordering was often more important than a good choice of encoding type.

Acknowledgements

This work is supported by the U.S. National Science Foundation under grant CCF-2415773.

Joseph E. Reeves is 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).

João Filipe is co-financed by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under the Visiting Students Program.

References

Abío, I.; Nieuwenhuis, R.; Oliveras, A.; and Rodríguez-Carbonell, E. 2013. A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints. In Schulte, C., ed., *Principles and Practice of Constraint Programming (CP)*, volume 8124 of *Lecture Notes in Computer Science*, 80–96. Springer.

Abío, I.; and Stuckey, P. J. 2012. Conflict Directed Lazy Decomposition. In *Principles and Practice of Constraint Programming (CP)*.

Ansótegui, C.; Giráldez-Cru, J.; and Levy, J. 2012. The Community Structure of SAT Formulas. In Cimatti, A.; and Sebastiani, R., eds., *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, 410–423. Springer.

Ansótegui, C.; Giráldez-Cru, J.; Levy, J.; and Simon, L. 2015. Using Community Structure to Detect Relevant Learnt Clauses. In Heule, M.; and Weaver, S. A., eds., *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *Lecture Notes in Computer Science*, 238–254. Springer.

Asín, R.; Nieuwenhuis, R.; Oliveras, A.; and Rodríguez-Carbonell, E. 2009. Cardinality Networks and Their Applications. In Kullmann, O., ed., *Theory and Applications of Satisfiability Testing (SAT)*, 167–180. Springer.

Asín, R.; Nieuwenhuis, R.; Oliveras, A.; and Rodríguez-Carbonell, E. 2011. Cardinality Networks: a theoretical and empirical study. *Constraints An Int. J.*, 16(2): 195–221.

Bailleux, O.; and Boufkhad, Y. 2003. Efficient CNF Encoding of Boolean Cardinality Constraints. In Rossi, F., ed., *Principles and Practice of Constraint Programming (CP)*, 108–122. Springer.

Bard, G. V.; Courtois, N. T.; and Jefferson., C. 2007. Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers. Cryptology ePrint Archive, Paper 2007/024.

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

Biere, A.; Cimatti, A.; Clarke, E. M.; and Zhu, Y. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*.

Biere, A.; Fazekas, K.; Fleury, M.; and Heisinger, M. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In Balyo, T.; Froleyks, N.; Heule, M.; Iser, M.; Järvisalo, M.; and Suda, M., eds., *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, 51–53. University of Helsinki.

Cook, S. A. 1976. A Short Proof of the Pigeon Hole Principle Using Extended Resolution. *SIGACT News*, 8(4): 28–32.

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

Freeman, J. W. 1995. *Improvements to Propositional Satisfiability Search Algorithms*. Ph.D. thesis, University of Pennsylvania, USA.

Gent, I. P. 2002. Arc Consistency in SAT. In *European Conference on Artificial Intelligence*.

Gent, I. P.; and Nightingale, P. 2004. A new encoding of alldifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, volume 3, 95–110.

Haberlandt, A.; Green, H.; and Heule, M. J. H. 2023. Effective Auxiliary Variables via Structured Reencoding. In Mahajan, M.; and Slivovsky, F., eds., *Theory and Applications of Satisfiability Testing (SAT)*, volume 271 of *LIPIcs*, 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Heule, M. J. H. 2018. Schur number five. In *Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*, 428–437.

Jabbour, S.; Sais, L.; and Salhi, Y. 2013. A Pigeon-Hole Based Encoding of Cardinality Constraints. *Theory and Practice of Logic Programming*, 13.

Järvisalo, M.; Berg, J.; Martins, R.; and Niskanen, A. 2023. MaxSAT Evaluation 2023 Benchmarks. https://maxsat-evaluations.github.io/2023/benchmarks.html. Accessed: 2024-08-13.

Karpinski, M.; and Piotrów, M. 2019. Encoding cardinality constraints using multiway merge selection networks. *Constraints An Int. J.*, 24(3-4): 234–251.

Kroening, D.; and Tautschnig, M. 2014. CBMC–C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 389–391. Berlin, Heidelberg: Springer Berlin Heidelberg.

Marques-Silva, J.; and Lynce, I. 2007. Towards Robust CNF Encodings of Cardinality Constraints. In Bessière, C., ed., *Principles and Practice of Constraint Programming (CP)*, 483–497. Springer.

Marques-Silva, J.; Lynce, I.; and Malik, S. 2021. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, 133 – 182. IOS Press.

Martins, R.; Manquinho, V. M.; and Lynce, I. 2011. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *Tools with Artificial Intelligence (ICTAI)*, 313–320. IEEE Computer Society.

Martins, R.; Manquinho, V. M.; and Lynce, I. 2013. Community-Based Partitioning for MaxSAT Solving. In Järvisalo, M.; and Gelder, A. V., eds., *Theory and Applications of Satisfiability Testing (SAT)*, volume 7962 of *Lecture Notes in Computer Science*, 182–191. Springer.

Morgado, A.; Ignatiev, A.; and Marques-Silva, J. 2015. MSCG: Robust Core-Guided MaxSAT Solving: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9: 129–134.

Neves, M.; Martins, R.; Janota, M.; Lynce, I.; and Manquinho, V. M. 2015. Exploiting Resolution-Based Representations for MaxSAT Solving. In Heule, M.; and Weaver, S. A., eds., *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *Lecture Notes in Computer Science*, 272–286. Springer.

Newsham, Z.; Ganesh, V.; Fischmeister, S.; Audemard, G.; and Simon, L. 2014. Impact of Community Structure on SAT Solver Performance. In Sinz, C.; and Egly, U., eds., *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *Lecture Notes in Computer Science*, 252–268. Springer.

Nguyen, V.-H.; Nguyen, V.-Q.; Kim, K.; and Barahona, P. 2021. Empirical Study on SAT-Encodings of the At-Most-One Constraint. In *Smart Media and Applications (SMA)*, 470–475. ACM.

Niskanen, A.; Wallner, J. P.; and Järvisalo, M. 2018. Extension Enforcement under Grounded Semantics in Abstract Argumentation. In Thielscher, M.; Toni, F.; and Wolter, F., eds., *Principles of Knowledge Representation and Reasoning (KR)*, 178–183. AAAI Press.

Ogawa, T.; Liu, Y.; Hasegawa, R.; Koshimura, M.; and Fujita, H. 2013. Modulo Based CNF Encoding of Cardinality Constraints and Its Application to MaxSAT Solvers. In *Tools with Artificial Intelligence (ICTAI)*, 9–17.

Reeves, J. E.; Heule, M. J. H.; and Bryant, R. E. 2024. From Clauses to Klauses. In Gurfinkel, A.; and Ganesh, V., eds., *Computer Aided Verification (CAV)*, volume 14681 of *Lecture Notes in Computer Science*, 110–132. Springer.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12): 1031–1080.

Sinz, C. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 3709 of *LNCS*, 827–831.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing (SAT)*, 244–257. Springer.

StarExec. 2024. StarExec. https://starexec.org/starexec/public/about.jsp. Accessed: 2024-08-14.

Stump, A.; Sutcliffe, G.; and Tinelli, C. 2014. StarExec: A Cross-Community Infrastructure for Logic Solving. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *LNCS*, 367–373. Springer.

Traag, V.; Waltman, L.; and Van Eck, N. 2019. From Louvain to Leiden: guaranteeing well-connected communities. Sci. Rep. 9, 5233.

Yang, J.; Kharkov, Y. A.; Shia, Y.; Heule, M. J. H.; and Dutertre, B. 2024. Quantum Circuit Mapping Based on Incremental and Parallel SAT Solving. In *Theory and Applications of Satisfiability Testing (SAT)*.