Certified Variable Reordering in Cardinality Encodings

Joseph E. Reeves[1], Marijn J. H. Heule[1] and Randal E. Bryant[1]

[1]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA

Abstract

Cardinality constraints in Boolean satisfiability (SAT) problems are typically encoded as sets of clauses in conjunctive normal form, which serves as the standard input format of SAT solvers. The ordering of data variables within a cardinality constraint can have a substantial impact on the quality of the resulting encoding and on solver performance, motivating the development of various pre-processing and reordering techniques. In this work, we address the scenario in which a cardinality constraint has already been encoded into clauses. We introduce a proof-producing reencoding method that systematically transforms an existing encoding into a new encoding with reordered data variables, while producing a certificate that establishes satisfiability equivalence between the original and transformed formulas. To illustrate the practical advantages of our approach, we evaluate it using a family of vertex cover benchmarks specifically designed with adversarial variable orderings.

Keywords

Cardinality Constraint Encodings, Clausal Proofs, SAT Solving

1. Introduction

For more than two decades, Conjunctive Normal Form (CNF) has served as the standard input format of Boolean satisfiability (SAT) solvers. Nevertheless, SAT problems originate from a diverse array of domains, including mathematical discovery [1, 2, 3], optimization [4, 5], and both software [6] and hardware [7, 8] verification, and are frequently formulated using high-level constraints. Consequently, the development of novel and improved encoding techniques remains an active and ongoing area of research [9, 10, 11].

Traditionally, different types of encodings have been evaluated primarily in terms of their size, specifically the number of *auxiliary variables* and clauses introduced. Propagation power has also served as a comparative metric [12]. Recent works on the encoding of *cardinality constraints* have shifted focus to the meaning of auxiliary variables within an encoding. This demonstrated that the order of data literals may be more impactful than the particular type of encoding [13, 14]. At the same time, modern SAT workflows increasingly rely on proof certificates to validate solver results and pre- and inprocessing steps. Consequently, transformations applied after encoding should also be certifiable by independently checkable proofs.

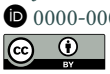
A *cardinality constraint* comprises a set of *data literals* $(\ell_1, \ldots, \ell_s)$, a comparison operator $(>, \geq, <, \leq)$, and a bound k. For example, the AtMostK cardinality constraint $\ell_1 + \ell_2 + \cdots + \ell_s \leq k$ is true if at most k of the literals are true. The order of data literals within the constraint will affect the meaning of auxiliary variables within an encoding. This changes how a solver reasons. Consider the two representations of a Sequential Counter encoding [15] in Figure 1. Given the clause $(\overline{\ell}_2 \vee \overline{\ell}_4)$, unit propagation could infer the literal $\overline{s}'_{2,2}$ on the second encoding, whereas no analogous reasoning is possible on the first encoding. Thus, grouping related data literals together enables a solver to reason more effectively about those groups through the auxiliary variables, particularly when the groups are interconnected by other constraints within the formula.

When a problem is initially presented in a cardinality format, data literals may be ordered prior to encoding, for example, using a proximity-based metric [13]. However, practical scenarios arise in which a constraint may already be encoded with a suboptimal ordering. This may occur for several reasons: (i) an optimal ordering is unknown prior to solving; (ii) the problem is sufficiently large that identifying an

Pragmatics of Sat, July 19, 2026, Lisbon, Portugal

jereeves@andrew.cmu.edu (J. E. Reeves); marijn@cmu.edu (M. J. H. Heule); randy.bryant@cmu.edu (R. E. Bryant)

0000-0002-4585-0565 (J. E. Reeves); 0000-0002-5587-8801 (M. J. H. Heule); 0000-0001-5024-6613 (R. E. Bryant)



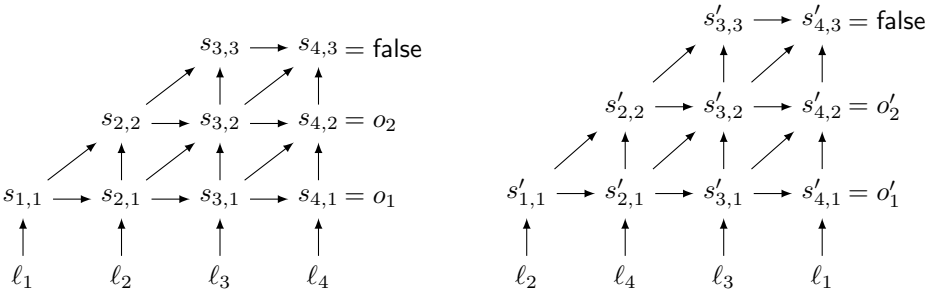


Figure 1: The Sequential Counter on four literals representing $\ell_1+\ell_2+\ell_3+\ell_4 \leq 2$ (left) and $\ell_2+\ell_4+\ell_3+\ell_1 \leq 2$ (right). See Section 2.2 for a detailed description of the Sequential Counter.

optimal ordering is computationally expensive; or (iii) the constraint is generated dynamically during solving. Although methods exist for reordering data literals within AtMostOne encodings [16, 14], these techniques do not extend to general AtLeastK encodings.

In this work, we assume that a cardinality constraint is already encoded, that we know its set of clauses, and that we know a better encoding using reordered data literals; we then focus on producing a proof that derives the second encoding from the first. In practice, this original encoding may be supplied by the user or discovered using a constraint extraction tool, and the reencoding may be constructed using the proximity-based metric. We introduce two proof-producing methods for transforming an AtLeastK encoding into a Sequential Counter encoding with reordered data literals, certifying that the transformation preserves satisfiability. The first uses a SAT solver to generate a proof; it is general but does not scale to large constraints. The second employs a series of adjacent swaps with manual proof construction; it produces more structured proofs and thereby accommodating substantially larger constraints. We evaluate the reencoder on a family of vertex cover problems and find that the overhead of variable reordering and proof checking is negligible relative to the gains from an optimized encoding.

2. Background

2.1. Boolean Satisfiability and Clausal Proofs

We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula F is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal ℓ is either a variable x (positive literal) or a negated variable $\overline{x}$ (negative literal). The *polarity* of a literal indicates whether it is positive or negative. We may refer to clauses as sets of literals and formulas as sets of clauses.

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

A *unit* is a clause containing a single literal. *Unit propagation* applies the following operation until a fixed point: take all units α in a formula F and remove from F clauses containing a literal in α, remove from clauses all literals negated in α. In cases where unit propagation yields the empty clause ($\bot$) we say it derived a *conflict*. A formula is pure when it contains only pure literals. A clause C is *blocked* in formula F if there exists a literal $\ell \in C$ such that for all clauses $D \in F_{\overline{\ell}}$ there exists an $\ell' \in D$ where $\overline{\ell'} \in C$ and $\mathsf{var}(\ell) \neq \mathsf{var}(\ell')$. A blocked clause is redundant and can be added or removed from F while preserving satisfiability.

A *clausal proof* is a sequence of clauses; for example, the sequence $F, C_1, C_2, \ldots, C_m$ is a clausal

proof of C_m from F. Each subsequent clause addition step must meet the criteria of the specified proof system. The case of $C_m = \bot$ serves as a refutation for F.

The Resolution Asymmetric Tautology (RAT) proof system captures extended resolution and can be used to express most common inprocessing techniques [17]. RAT with deletions (DRAT) proof logging is the most widely supported format. Typically, a solver emits a DRAT proof, which is then *trimmed* by a tool such as DRAT-TRIM [18]. 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 [19].

2.2. Cardinality Constraint Encodings

A Boolean AtLeastK cardinality constraint has the form $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$ and is satisfied by a partial assignment if the sum of the true literals is at least the *bound* k. The *size* of the cardinality constraint is the number of literals (s) it contains. Variables occurring in the cardinality constraint are *data variables* (referred to as *data literals* if their polarities are relevant), and fresh variables used in a clausal encoding are *auxiliary variables*. For order-sensitive encodings such as the Sequential Counter, these auxiliary variables represent partial sums over prefixes of the data literals, so their semantics depend directly on the chosen ordering.

An encoding of $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$ is *consistent* if assigning any $s - k + 1$ literals to false will always result in a conflict by unit propagation and *arc-consistent* [12] if it is consistent and unit propagation will assign all unassigned literals to true if exactly $s - k$ literals are assigned to false. Several tools, e.g., PySAT [20], provide APIs to generate common types of encodings including the Totalizer [21], Sorting Network, pigeon-hole, and BDD-based [22, 23]. In this work, we focus on the Sequential Counter by Sinz [15].

The Sequential Counter, e.g., Figure 1, uses a grid of auxiliary variables $s_{i,j}$ with $1 < i < s$ and with $1 < j < s$ 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. A simplified version of the encoding uses only the first $k + 1$ rows, and removes auxiliary variables $s_{i,j}$ if $i \leq k$ and $j > i$ or $i > (s - k)$ and $j \leq (i - (s - k))$. In the AtMostK encoding, auxiliary variables interact locally in the grid, e.g., $s_{i,j} \rightarrow s_{i+1,j}$ and $(s_{i-1,j} \wedge \ell_i) \rightarrow s_{i,j+1}$ and the bound $\leq k$ is enforced with the unit $\overline{o}_{k+1}$.

The AtLeastK encoding of the Sequential Counter propagates information in the reverse direction using the implications $(s_{i-1,j} \vee s_{i,j-1}) \leftarrow s_{i,j}$ and $(s_{i-1,j} \vee \ell_i) \leftarrow s_{i,j}$, and enforcing the bound with the unit o_k. The ExactlyK encoding includes clauses for both directions and both units. Note, some optimized versions of the encoding use slightly different sets of clauses and have no units [16].

2.3. Variable Ordering Techniques

The Natural ordering is given by increasing variable names, which are integer values identifying each variable in the formula. This method typically reflects the underlying structure of the problem, since most formula generators enumerate variable names in a systematic order. 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.

The Proximity ordering is more complex, capturing the spatial relationship of variables within a formula [13]. It achieves this by performing 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 are interpreted as variables. The Proximity algorithm from [13] 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. If two variables have the same score, choose the variable encountered first in the search.
2. Append v to the ordering.
3. 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.

4. If all variables in cardinality constraints are processed, return the ordering; otherwise, return to step 1.

This method orders variables close together if they co-occur in related parts of the formula, potentially introducing auxiliary variables in the resulting encoding that can be used to reason about partitions of the formula. Computing the Proximity can be costly due to score updates, since a single clause can be traversed multiple times.

Example 1

Consider the following formula

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

Proximity will first select the most occurring variable x_2. Then, scores are updated for variables occurring in the clause with x_2. 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. No scores are updated because x_1 does not occur in a clause with an unprocessed variable. The third variable selected is x_3 (seen before x_4) with a score of $1/3$ and finally x_4, which yields: $x_2 + x_1 + x_3 + \overline{x}_4 \leq 2$.

3. Proof Generation for Variable Reordering

In this section, we present two methods for generating proofs to transform an encoding through variable reordering. Let F denote a formula containing a clausal encoding (Φ) of the cardinality constraint $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$. Variable reordering takes a new ordering, $\omega = (r_1, r_2, \ldots, r_s)$, of the data literals and produces a Sequential Counter encoding (Φ') of $\ell_{r_1} + \ell_{r_2} + \cdots + \ell_{r_s} \geq k$ along with a clausal proof Ψ showing that $F \rightarrow \Phi'$. Together with the original formula, this proof certifies that replacing the old encoding by the reordered one preserves satisfiability.

Guarantee. Given an input CNF formula F containing an encoding Φ, the reencoding procedure produces a CNF formula F' in which Φ is replaced by Φ' and a proof certificate showing that F is satisfiable if and only if F' is satisfiable.

3.1. SAT-based Proof Generation

When constructing a new Sequential Counter encoding with fresh auxiliary variables, all clauses are blocked except the unit that enforces the bound. Clauses defining auxiliary variables are added by column, starting from the first row (e.g., $s_{1,1}, s_{2,1}, s_{2,2}, s_{3,1}, \ldots$). We will denote the unit enforcing the bound as $b^{\Phi'}$, and the encoding clauses without the unit ($\Phi' \setminus (b^{\Phi'})$) as Φ'^{-b}.

The unit $b^{\Phi'}$ can be derived by solving the formula $G = \Phi \wedge \Phi'^{-b}$, under the assumption $\overline{b}^{\Phi'}$. This represents the unsatisfiable formula $(\ell_1 + \ell_2 + \cdots + \ell_s \geq k) \wedge (\ell_{r_1} + \ell_{r_2} + \cdots + \ell_{r_s} < k)$. The proof generated by a SAT solver would be inserted between the definition clauses Φ'^{-b} and the unit $b^{\Phi'}$.

The formula G becomes harder as the variable orderings between Φ and Φ' differ more significantly. This can lead to timeouts for relatively small constraints (see Section 5.2). In contrast, G remains relatively easy when both Φ and Φ' use the same ordering but different Sequential Counter encoding implementations. For instance, if Φ is implemented with the PySAT Sequential Counter, we can introduce our own Sequential Counter encoding as Φ' and solve G to generate a proof deriving Φ' from Φ. We use this method to transform an input encoding into our implementation of the Sequential Counter, allowing us to perform a structured LRAT transformation from a known starting point. In principle, this will work for any starting encoding, e.g., going from a Totalizer encoding to our Sequential Counter encoding, but the more different the encodings the more difficult it is to solve G.

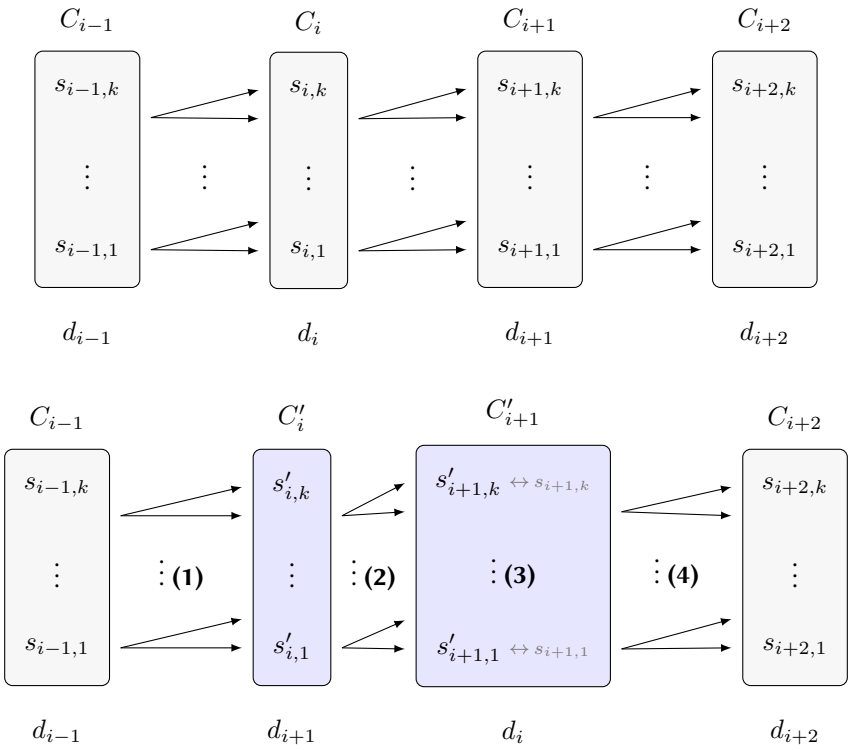


Figure 2: The Sequential Counter on four adjacent columns (top), and the new encoding when swapping d_i and d_{i+1} (bottom). Fresh columns C'_i and C'_{i+1} (blue) replace the originals. In C'_{i+1}, gray suffixes show the equivalences $s'_{i+1,j} \leftrightarrow s_{i+1,j}$ from step **(3)**. In each column gap, the arrows represent the dependencies of the definitions; step numbers **(1)**, **(2)** add new variables and their defining clauses, whereas step number **(4)** only adds clauses.

3.2. Swapping Adjacent Data Variables

Rather than constructing the new encoding Φ' in a single step, we incrementally modify the Sequential Counter by introducing new columns two at a time through a series of adjacent variable swaps. The SAT-based approach is general and simple to implement, but may produce large proofs. The swap-based approach yields more structured and typically smaller proofs, and scales better in practice.

Let d_i represent the data literal in column i and C_i represent the set of auxiliary variables in column i. The procedure for swapping two adjacent data variables d_i, d_{i+1}, shown in Figure 2 proceeds as follows:

1. Add new definition clauses in column i using d_{i+1} as the data literal and fresh auxiliary variables in C'_i. If $i > 0$, these definitions will depend on unchanged variables from C_{i-1}. These definition clauses are blocked.
2. Add new definition clauses in column $i+1$ using d_i as the data literal and fresh auxiliary variables in C'_{i+1}. These definitions will depend on the new variables in C'_i. These definition clauses are blocked.
3. Add the equivalences $s'_{i+1,j} \leftrightarrow s_{i+1,j}$, for $s_{i+1,j} \in C_{i+1}$ and $s'_{i+1,j} \in C'_{i+1}$. These clauses are RAT, and each RAT step can be manually inserted into the proof.
4. Add the clauses $s'_{i+1,j} \vee D$ for $(s_{i+1,j} \vee D) \in \Phi$ and $s_{i+1,j} \in C_{i+1}$. These clauses are RUP due to the equivalences derived in the previous step. These clauses tie the new variables in C'_{i+1} to the unchanged variables in C_{i+2}. If $i+1 = s$, only the unit enforcing the bound is added.
5. Delete all clauses containing variables from $C_i \cup C_{i+1}$.

In summary, this process eliminates two adjacent columns from the Sequential Counter and replaces them with two new columns in which the underlying data variables have been swapped. This approach is analogous to the movement of definition variables in QBF encodings [24]. Notably, if more than two

columns are introduced simultaneously, some of the equivalences in step 3 are no longer RAT, and their derivation would require a stronger proof system or possibly additional steps.

3.3. Swapping-based Proof Generation

In the overall swap-based reencoding procedure, first, the SAT-based proof generation transforms Φ into our Sequential Counter encoding with the original variable ordering. Then, a modified bubble sorting algorithm uses adjacent swaps to produce the reordered encoding. At each iteration i, the i^{th} element in ω is located in the current encoding and bubbled to its final position using adjacent swaps.

The reencoding procedure exhibits worst-case complexity $\Theta(kn^2)$, with $\Theta(n^2)$ swaps from bubble sort and $\Theta(k)$ operations to interchange columns during each swap. The size of the formula is $\Omega(kn)$ because it already contains an encoding, so the procedure is no worse than quadratic in the size of the original formula. Note, it is possible to avoid the worst-case number of swaps by reversing ω. Although Sequential Counter is not a symmetric encoding, prior experiments suggest that reversing an effective ordering can still yield positive results. In addition, it may be possible to reorder the Totalizer encoding with $\Theta(kn \log n)$ operations, but performing reordering in the Totalizer systematically remains an open problem.

4. Vertex Cover with Adversarial Variable Ordering

Given a graph G with vertices V and edges E, a *vertex cover* of G is a set of vertices S such that for all edges $(i, j) \in E$, either $i \in S$ or $j \in S$. The problem of finding a vertex cover with at most k vertices can be encoded into CNF using variables v_i for $i \in V$, where v_i is true iff vertex $i \notin S$, and constraints:

- $(\overline{v}_i \vee \overline{v}_j)$ for $(i, j) \in E$
- $v_1 + v_2 + \cdots + v_n \geq (n - k)$

We implemented a vertex cover generator to produce problems for which the ordering of the cardinality constraint has a major impact on the performance. Our generator takes as input the number of vertices and constructs the graph as follows:

1. Designate $1/10$ of the vertices as hubs (H) and the remaining as periphery vertices (P)
2. Add a cycle between the hubs
3. Add edges between two hub vertices h_i, h_j with probability $0.3 \cdot e^{-(|i-j|/2)}$
4. Add a cycle between the periphery vertices
5. Add edges between two periphery vertices p_i, p_j with probability $0.08 \cdot e^{-(|i-j|/4)}$
6. Add edges between a hub vertex h_i and periphery vertex p_j with probability $0.5 \cdot e^{-(|9i-j|/5)}$ to connect the periphery vertices that are close to "their" hub.

The graph is constructed with an inner hub cycle and an outer periphery cycle, and edges between hub and periphery nodes are more likely to occur when the nodes are proximate. This structural design can be exploited by an adversarial ordering. The adversarial ordering is generated in two phases. In the first phase, each hub node h_i is assigned to a position $i \times |V|/(2|H|)$ if i is even and $i \times |V|/(2|H|) + |V|/2$ if i is odd. Subsequently, the hubs are processed by descending order of degree, and for each hub, all neighboring periphery vertices are ordered and assigned positions via the same interleaving strategy.

In the second phase, pairs of vertices are randomly selected and swapped if the resulting swap increases the overall span. Let p_i be the position of vertex i in the ordering. Then the span of an ordering is the sum of distances $|p_i - p_j|$ for each $(i, j) \in E$. Increasing the span ensures nodes close together in the graph are farther apart in the ordering. In our experiments, $2{,}000$ swaps were attempted for each graph. An adversarial ordering is shown in Figure 3 along with an ordering generated from the Proximity technique.

We utilize an integer linear programming (ILP) solver to find the minimum vertex cover τ, and set $k = \tau - 1$ to make the problem unsatisfiable. Finally, the cardinality constraint is encoded using the adversarial order and a Sequential Counter.

5. Experimental Evaluation

All experiments were performed in the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [25]. We used a 5,000 second timeout for solving and a 5,000 second timeout for proof checking, and 64 experiments were run in parallel per node, so each process held approximately 4GB of memory. We report solving time, reencoding time, proof size, and proof-checking time to separate the benefit of the reordered encoding from the cost of certification.

5.1. Adversarial Vertex Covers

In this section, we evaluate the effectiveness of our reencoding technique for solving adversarial vertex cover problems. We use instances ranging in size from 40 to 300 vertices, and show the results in Figure 4. The reencoding configuration generates a target ordering using the Proximity method, then uses the swap-based reencoding technique to generate a reordered encoding, and finally solves the resulting problem with CaDiCaL version 3.0.0. The default configuration solves the vertex cover problem with the adversarial encoding using CaDiCaL.

The adversarial ordering significantly increases the difficulty of the vertex cover problem, even for relatively small graphs with 100 vertices. In contrast, an optimized variable ordering can lead to substantial performance improvements. Notably, for larger graphs, overall solving time is not the primary bottleneck in the reencoding process. For example, in the case of a graph with 300 vertices, the reencoding procedures require 250 seconds in the initial SAT call to transform the PySAT encoding into our implementation of the Sequential Counter encoding, 190 seconds to perform 22,311 swaps, and subsequently 7.5 to solve the resulting reencoded formula. Finally, the complete LRAT proof was checked in 105 seconds. These measurements show that proof sizes and checking times remain manageable for the instances considered.

5.2. Proof Generation Scaling

In this section, we evaluate the scaling of the SAT-based and Swap-based reencoding procedures. The instances are PySAT Sequential Counter encodings with sizes ranging from 40 to 200 data variables, each with a bound of $1/4$ the size. The target ordering is constructed with two parameters: *swap window* (W) a fraction that determines how far two variables can be swapped ([1,$W \times Size$]), and *number of swaps* (we present data for instances with 100 swaps each). For each swap, a distance ($dist$) is selected randomly from [1,$W \times Size$], a variable d_i is selected from [1,$Size - d$], and d_i, d_{i+dist} are swapped in the new ordering.

Results for the runtimes and LRAT proof sizes (by number of proof lines) are shown in Figure 5. The swap-based method completes all reencodings in under 100 seconds, whereas the SAT-based method

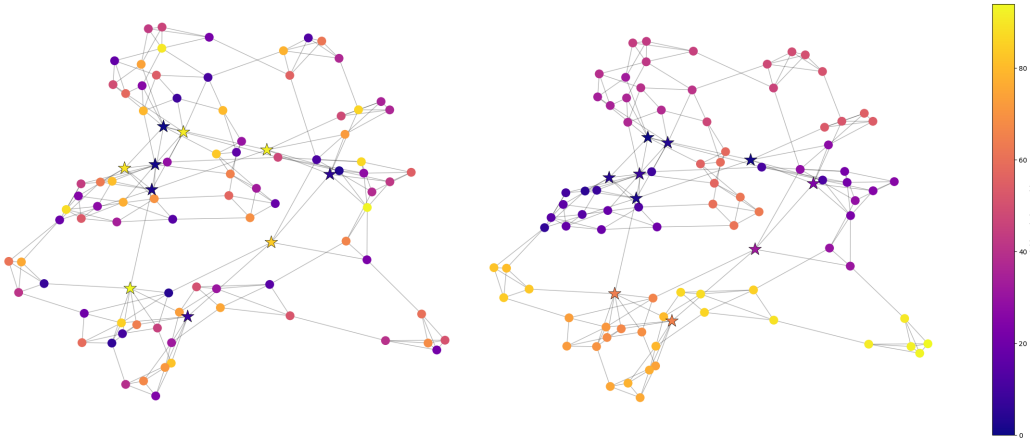


Figure 3: A graph with 95 nodes, 10 of which are hubs (stars). Coloring represents a node's position in the order from 1 (dark purple) to 95 (light yellow), with an adversarial ordering (left) and proximity ordering (right).

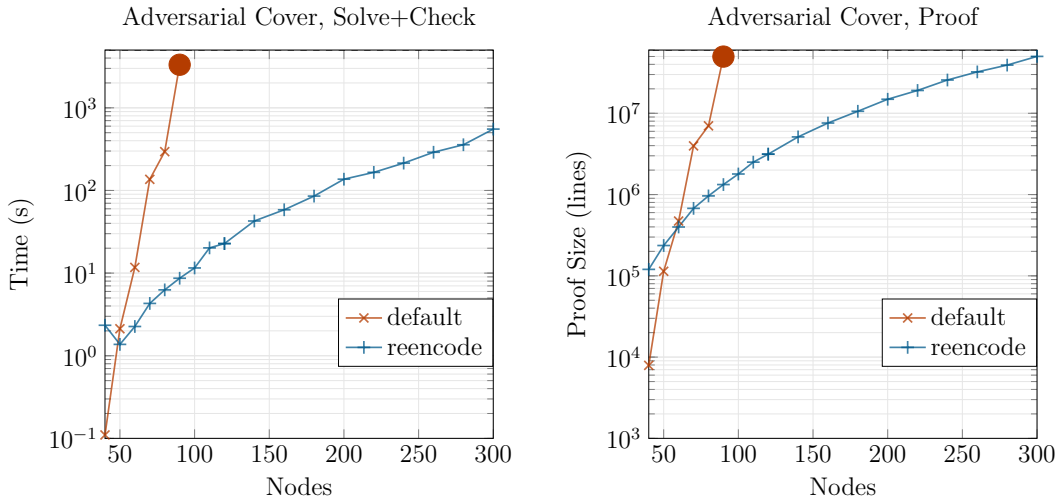


Figure 4: Solving and checking time (left) and proof sizes (right) on adversarial vertex cover problems with CaDiCaL (default) and swap-based reencoding technique with the Proximity ordering then CaDiCaL (reencode).

exceeds the 5,000 second timeout for small encodings involving numerous swaps. Moreover, the swap window significantly impacts the runtime of the SAT-based approach, in contrast to the more predictable behavior of the swap-based approach.

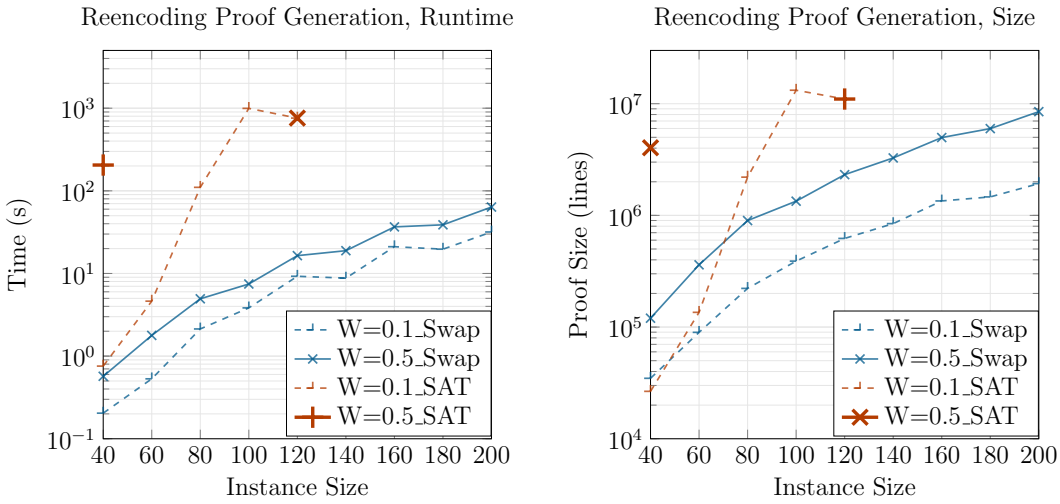


Figure 5: Runtime and proof size for the swap-based and SAT-based reencoding procedures. W is the swap window, 100 swaps are performed, the size is the number of data literals in the constraint, and the bound of the constraint is $1/4$ the size. Bold point marks the last completed execution.

6. Conclusion and Future Work

In this work, we presented a technique for transforming an encoding of a cardinality constraint into a new Sequential Counter encoding with reordered data variables. The technique uses manual LRAT proof production for efficient proof generation and checking, so the optimized encoding can be adopted without sacrificing the certifiability of the original workflow. We demonstrated the potential impact of variable reordering, considering a family of vertex cover problems with adversarially generated orderings. Unlike prior work that optimizes variable orderings before encoding, our method operates directly on CNF formulas and preserves correctness via proof certificates.

Several avenues exist for extending this work. Although the Totalizer encoding often outperforms the Sequential Counter, we have focused on the latter because it lends itself to manual proof production

through adjacent swaps. Each auxiliary variable within the Sequential Counter is defined in terms of auxiliary variables from the previous column and one data literal, which makes it possible to derive the equivalences needed to interchange two adjacent columns. By contrast, the auxiliary variables in the internal nodes of the Totalizer are more deeply nested relative to the data literals, and the method for replacing them in a reencoding remains a non-trivial open question.

Additionally, the reencoding technique could be incorporated as an inprocessing strategy, though further investigation is needed to determine optimal triggering conditions and to assess how target orderings might be refined using solver state information.

Acknowledgments

The work has been supported by the National Science Foundation (NSF) under grant 2551599.

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly in order to: Grammar and spelling check, and improve writing style. Further, the author(s) used Claude to generate images for Figures 2 and 3. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] M. J. H. Heule, O. Kullmann, V. 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)*, Springer, 2016, pp. 228–245.

[2] M. J. H. Heule, Schur number five, in: AAAI Conference on Artificial Intelligence, AAAI Press, 2018.

[3] M. J. H. Heule, M. 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.

[4] R. Brayton, A. Mishchenko, ABC: an academic industrial-strength verification tool, in: Computer Aided Verification (CAV), Springer-Verlag, 2010.

[5] J. Yang, Y. A. Kharkov, Y. Shi, M. J. H. Heule, B. 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)*, Schloss Dagstuhl, 2024, pp. 29:1–29:18.

[6] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, B. Parno, Verus: A practical foundation for systems verification, in: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24, ACM, 2024, p. 438–454.

[7] K. Jia, M. Rinard, Efficient exact verification of binarized neural networks, in: Advances in Neural Information Processing Systems (NeurIPS), volume 33, Curran Associates, Inc., 2020, pp. 1782–1795.

[8] J. Yang, Y. K. Tan, M. Soos, M. O. Myreen, K. 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)*, Schloss Dagstuhl, 2025, pp. 32:1–32:22.

[9] I. P. Gent, P. Nightingale, A new encoding of alldifferent into sat, in: International Workshop on Modelling and Reformulating Constraint Satisfaction, volume 3, 2004, pp. 95–110.

[10] J. Marques-Silva, I. Lynce, Towards robust cnf encodings of cardinality constraints, in: C. Bessière (Ed.), Principles and Practice of Constraint Programming (CP), Springer, 2007, pp. 483–497.

[11] G. V. Bard, N. T. Courtois, C. Jefferson., 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, 2007.
[12] I. P. Gent, Arc consistency in SAT, in: European Conference on Artificial Intelligence, 2002.
[13] J. E. Reeves, J. Filipe, M.-C. Hsu, R. Martins, M. J. H. Heule, The impact of literal sorting on cardinality constraint encodings, in: AAAI Conference on Artificial Intelligence, volume 39, AAAI Press, 2025, pp. 11327–11335.
[14] A. Sheng, J. E. Reeves, M. J. H. Heule, Reencoding unique literal clauses, in: Theory and Applications of Satisfiability Testing (SAT), volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl, 2025, pp. 29:1–29:21.
[15] C. 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)*, 2005, pp. 827–831.
[16] J. E. Reeves, M. J. H. Heule, R. E. Bryant, From clauses to klauses, in: Computer Aided Verification (CAV), volume 14681 of *Lecture Notes in Computer Science (LNCS)*, Springer, 2024, pp. 110–132.
[17] M. Järvisalo, M. J. H. Heule, A. Biere, Inprocessing rules, in: International Joint Conference on Automated Reasoning (IJCAR), volume 7364 of *Lecture Notes in Computer Science (LNCS)*, Springer, 2012, pp. 355–370.
[18] N. Wetzler, M. J. H. Heule, W. 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)*, 2014, pp. 422–429.
[19] Y. K. Tan, M. J. H. Heule, M. 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)*, 2021, pp. 223–241.
[20] A. Ignatiev, A. Morgado, J. 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)*, 2018, pp. 428–437.
[21] O. Bailleux, Y. 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)*, Springer, 2003, pp. 108–122.
[22] S. Jabbour, L. Sais, Y. Salhi, A pigeon-hole based encoding of cardinality constraints, Theory and Practice of Logic Programming 13 (2013).
[23] N. Eén, N. Sörensson, Translating pseudo-Boolean constraints into SAT, Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 1–26.
[24] J. E. Reeves, M. J. H. Heule, R. 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)*, 2022, pp. 76–93.
[25] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, N. A. Nystrom, Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research, ACM, 2021, pp. 1–4.

7. Online Resources

The benchmark generator, reordering tool, and experimental data are available at

- GitHub