Reencoding Unique Literal Clauses

Aeacus Sheng
Department of Philosophy, Carnegie Mellon University

Joseph E. Reeves
Computer Science Department, Carnegie Mellon University

Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University

Abstract

We present a lightweight reencoding technique that augments propositional formulas containing implicit or explicit exactly-one constraints with auxiliary variables derived from the order encoding. Our approach is based on the observation that many formulas contain clauses where each literal appears only in that clause, and that these *unique literal clauses* can be replaced by the corresponding sequential counter encoding of exactly-one constraints, which introduces the same variables as the order encoding. We implemented the reencoding in the state-of-the-art SAT solver CaDiCaL with support for proof logging and solution reconstruction. Experiments on SAT Competition benchmarks demonstrate that our technique enables solving dozens of additional formulas. We found that shuffling a formula before reencoding harms performance. To mitigate this issue, we introduce a method that sorts literals within clauses based on the formula structure before applying our reencoding. The same technique also predicts whether reencoding is likely to yield improvements.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability solving, auxiliary variables, graph coloring

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.14

Supplementary Material *Source Code and Data*: https://github.com/jreeves3/ulc-cadical

Funding Supported by the National Science Foundation (NSF) under grant CCF-2415773.
Joseph E. Reeves: Supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

Acknowledgements We thank Sonya Simkin for her contributions during the early stages of this research.

1 Introduction

When encoding combinatorial problems into propositional logic, it is often necessary to encode the selection of an object from a set. The *direct encoding* handles this with one propositional variable per object, which is `true` if that object is selected, along with a clause enforcing that one of these variables must be `true`. A potential disadvantage of the direct encoding is that it does not contain variables to facilitate reasoning about multiple objects simultaneously. For example, it could be beneficial to have a variable that indicates whether an object in the first half of the set is selected (using some ordering). If this variable is assigned to `false`, then the solver knows that the selected object needs to be in the second half. Other encodings, such as the *sequential counter encoding* [28] and the *order encoding* [30], include such variables. This paper presents a lightweight reencoding method that incorporates these variables by replacing specific clauses to enhance solver performance across various benchmarks.

It is well-known that auxiliary variables in proofs could reduce their size exponentially [31].

The impact of auxiliary variables in formulas has been less studied. However, the variables that we introduce are used in the order encoding, which improves solver performance on a range of problems [1,8,10,24,29]. Moreover, the award-winning CSP solver Sugar employs the order encoding [30]. Introducing new variables tends to reduce propagation speed, thereby offsetting potential gains. Hence, ideally, the order variables should only be introduced when they provide a tangible benefit.

While automatically reencoding problems can be challenging, direct encodings for object selection constraints are typically easy to detect: the literal that expresses that an object is selected occurs only once in the formula. We exploit this property by searching for unique literal clauses (ULCs), that is, clauses containing only literals that do not appear elsewhere in the formula. We will show later that ULCs express either implicit or explicit ExactlyOne constraints. Whenever we find a ULC, we can reencode the involved part of the formula to turn the direct encoding into the sequential counter or order encoding.

To make this transformation effective, two additional steps are required. First, many formulas contain pairs of ULCs with complementary literals. We resolve these pairs to obtain longer ULCs and to simplify the reencoding process. Second, it is critical to *sort* the literals in ULCs in a way that respects the structure of the formula. We introduce the concept of *aligning* ULCs so that the newly introduced order variables have a meaningful interpretation. We observed that when ULCs cannot be aligned, the reencoding can be harmful.

Existing work on reencoding propositional formulas focuses on reducing the size of the formula by introducing new variables [13,20,26]. In contrast, our reencoding increases the size of the formula, although it may be reduced because certain clauses become redundant. The closest related work is by Manthey and Steinke [21], who also propose transforming the direct encoding into the sequential counter encoding. They detect direct encodings of explicit ExactlyOne constraints rather than ULCs to guarantee a reduction in formula size. Due to a small-scale evaluation that showed limited impact on performance, it is unclear whether their approach is effective.

Experiments show that our reencoding strategy, implemented in the state-of-the-art solver CaDiCaL [5], solves substantially more formulas from the 2022 SAT Competition Anniversary Track. The introduction of meaningful order variables is the key component of our technique.

2 Encodings

In this section, we describe the three encodings of the ExactlyOne constraint that are relevant to this paper. The constraint ExactlyOne $(\ell_1, \ldots, \ell_k)$ requires that exactly one of the literals $\ell_1, \ldots, \ell_k$ is assigned to true.

2.1 Direct Encoding

The direct encoding of ExactlyOne $(\ell_1, \ldots, \ell_k)$ splits the constraint into AtLeastOne $(\ell_1, \ldots, \ell_k)$ and AtMostOne $(\ell_1, \ldots, \ell_k)$, where the former is just a clause and the latter blocks any pair of these literals from both being true.:

$$(\ell_1 \vee \cdots \vee \ell_k) \wedge \bigwedge_{1 \leq i < j \leq k} (\overline{\ell}_i \vee \overline{\ell}_j)$$

Note that the number of binary clauses is quadratic in k.

If the literals $\ell_1, \ldots, \ell_k$ do not occur in other clauses, the binary clauses can be omitted while preserving satisfiability. In that case, we call it an implicit ExactlyOne constraint. If all binary clauses are present, we call it an explicit ExactlyOne constraint.

2.2 Sequential Counter Encoding

The sequential counter encoding by Sinz [28] facilitates a compact representation of cardinality constraints. The method works for arbitrary cardinality constraints and uses auxiliary variables $s_{i,j}$ expressing that out of the first i literals, exactly j are true. Since we focus on the ExactlyOne constraint, we only use variables with $j = 1$ and therefore drop the second index. For this restricted case, the encoding uses the following definitions: $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$ for $1 < i < k$ and $s_1 \leftrightarrow \ell_1$. Additionally, to enforce that at most one can be true, the clauses $\overline{s}_{i-1} \vee \overline{\ell}_i$ are included for $1 < i < k$. Finally, at least one of them must be true, which is encoded as follows: $\overline{s}_{k-1} \to \ell_k$.

▶ **Example 1.** Consider the constraint ExactlyOne (ℓ_1, ℓ_2, ℓ_3). The sequential counter encoding uses the following clauses:

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

For our reencoding technique, we will replace a clause $(\ell_1 \vee \cdots \vee \ell_k)$ if it represents an implicit or explicit ExactlyOne constraint with the sequential counter encoding shown above. Note that the reencoding replaces one clause by roughly $3k$ clauses. However, clauses of the form $(\overline{\ell}_i \vee \overline{\ell}_j)$ become redundant and will therefore be removed if present.

2.3 Order Encoding

The order encoding [30] uses order variables $o_{\leq i}$ with $1 < i < k$ to express if one of $\ell_1, \ldots, \ell_i$ is `true`. To state that ℓ_i is `true` using only order variables, $o_{\leq i}$ must be `true`, while $o_{\leq i-1}$ must be `false`. The case for ℓ_1 is special: ℓ_1 is `true` if and only if $o_{\leq 1}$ is `true`. When encoding ExactlyOne, the case ℓ_k is also special: ℓ_k is `true` if and only if $o_{\leq k+1}$ 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`.

The order encoding eliminates the x_i variables by replacing all their occurrences with $o_{\leq i}$ variables. More specifically,

- all literals ℓ_1, $\overline{\ell}_1$, ℓ_k, and $\overline{\ell}_k$ are replaced by $o_{\leq 1}$, $\overline{o}_{\leq 1}$, $\overline{o}_{\leq k-1}$, and $o_{\leq k-1}$, 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.

▶ **Example 2.** For the order encoding of a graph-coloring problem with graph $G = (V, E)$ and k colors, let $v_{\leq i}$ denote whether vertex v has color at most i. The clauses are:

- For each $v \in V$: $(\overline{v}_{\leq 1} \vee v_{\leq 2}) \wedge (\overline{v}_{\leq 2} \vee v_{\leq 3}) \wedge \cdots \wedge (\overline{v}_{\leq k-2} \vee v_{\leq k-1})$
- For each $(u, v) \in E$: $(\overline{u}_{\leq 1} \vee \overline{v}_{\leq 1}) \wedge (u_{\leq 1} \vee \overline{u}_{\leq 2} \vee v_{\leq 1} \vee \overline{v}_{\leq 2}) \wedge \cdots \wedge (u_{\leq k-1} \vee v_{\leq k-1})$

Note that for larger k, the clauses of length four will outnumber the binary clauses.

Transforming the direct encoding of ExactlyOne into the sequential counter encoding is simple: replace the clauses that encode the constraint. The remaining part of the formula is unaltered. Now, observe that transforming from the sequential counter encoding into the order encoding is also easy: eliminate the original variables using variable elimination (also known as DP resolution). To see this, note that the s_i variables of the sequential counter encoding map to the $o_{\leq i}$ variables of the order encoding.

3 Reencoding

In this section, we describe the theoretical foundation of our reencoding method. Our method focuses on introducing meaningful auxiliary variables that help the solver. We aim to achieve this by allowing the solver to reason about multiple objects simultaneously. We first describe the pattern that we use to detect a reencoding opportunity. Afterward, we merge these patterns when possible to increase effectiveness. Finally, we sort the literals to provide more meaningful variables.

3.1 Unique Literal Clauses

The key building block of our reencoding method is the notion of unique literal clauses, which is defined below.

▶ **Definition 3.** *A clause C is a* unique literal clause (ULC) *with respect to a formula Γ if none of the literals $\ell \in C$ occurs in $\Gamma \setminus \{C\}$.*

Given a formula Γ, the set of ULCs can be computed in linear time in the size of Γ by first computing which literals occur only once in Γ and then determining which clauses only contain such literals.

▶ **Lemma 4.** *Given a formula Γ and a ULC $C \in \Gamma$, if Γ is satisfiable, then it can be satisfied with exactly one literal in C assigned to true.*

Proof. Let τ be a satisfying assignment of Γ, then τ must satisfy at least one literal in C. Then, we flip assignments to variables in C until there is only one satisfied literal in it. Since each literal in a ULC is unique in Γ, each flip cannot falsify any satisfied clause. ◀

An alternative argument showing Lemma 4 is that for a ULC $(\ell_1 \vee \cdots \vee \ell_k)$, the formula can be extended with all binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ for $1 \leq i < j \leq k$ using blocked clause addition [19]. These binary clauses enforce that at most one literal of the ULC can be `true`.

To transform the direct encoding into the sequential counter encoding, ULC $(\ell_1 \vee \cdots \vee \ell_k)$ and all present binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ for $1 \leq i < j \leq k$ are replaced by the definitions of the auxiliary variables, $s_1 \leftrightarrow \ell_1$ and $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$ with $1 < i < k$, together with the at-most-one clauses $(\overline{s}_{i-1} \vee \overline{\ell}_i)$ with $1 < i < k$ and the at-least-one clause $(s_{k-1} \vee \ell_k)$.

Alternatively, one could reencode the direct encoding into the order encoding. This would eliminate the ULC $(\ell_1 \vee \cdots \vee \ell_k)$ and replace the literals $\overline{\ell}_1, \ldots, \overline{\ell}_k$ by literals of new variables $o_{\leq 1}, \ldots, o_{\leq k-1}$ that are inspired by the order encoding:

- $\overline{\ell}_1 \equiv \overline{o}_{\leq 1}$
- $\overline{\ell}_i \equiv o_{\leq i-1} \vee \overline{o}_{\leq i}$ for $1 < i < k$
- $\overline{\ell}_k \equiv o_{\leq k-1}$

Again, binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ in Γ would become redundant after the reencoding and should be discarded. An optional extension of the reencoding is the addition of the order clauses $(\overline{o}_{\leq i-1} \vee o_{\leq i})$ for $1 < i < k$. Typically, adding these order clauses improves performance.

3.2 Clashing ULCs

To maximize the effectiveness of the reencoding, the ULCs should be as large as possible. Also, we want to reencode *all* ULCs above a size threshold. Since original variables occur positively and negatively in the definition clauses of the sequential counter encoding, reencoding one

ULC may break the ULC property of another clause. We achieve both objectives by resolving "clashing" ULCs.

▶ **Definition 5.** *Two unique literal clauses C and D are said to* clash *(on a literal ℓ) if there exist $\ell \in C$ and $\overline{\ell} \in D$. We also say that they clash on the variable v underlying ℓ.*

▶ **Definition 6.** *A formula Γ is* clash-free *if none of its ULCs clash.*

We can extend Lemma 4 to show that if a satisfiable formula is clash-free, then every ULC can be satisfied on a single literal.

▶ **Lemma 7.** *Let Γ be a clash-free formula. If Γ is satisfiable, then it can be satisfied with exactly one literal in each ULC assigned to true.*

Proof. Pick a satisfying assignment τ of Γ. For each ULC, augment τ as in the proof of Lemma 4. Since the ULCs do not clash, flipping assignments to literals in them does not falsify other ULCs. ◀

Note that resolving two clashing ULCs either results in a tautology (if there are multiple clashing pairs) or a new ULC. So, if a formula has a pair of clashing ULCs, we can either just eliminate the pair (if the resolvent is a tautology) or replace the pair with a new one while eliminating a variable.

▶ **Lemma 8.** *Eliminating all clashing variables among the ULCs in a formula Γ is confluent, and the unique fixpoint is a clash-free formula.*

Proof sketch. There exists a unique partition of the clashing ULCs such that for each partition, it holds that i) each ULC does not clash with any ULC in another partition, ii) clashing ULCs are in the same partition, and iii) each ULC clashes with at least one other ULC in its partition. If a partition with n ULCs has more than $n-1$ clashing pairs, then variable elimination will remove all ULCs in it. On the other hand, if a partition with n ULCs has $n-1$ clashing pairs, then variable elimination (which in this case is just resolution) will result in a unique ULC consisting of all the literals that are not clashing. Because clashes do not occur between partitions, the resulting formula is clash-free. ◀

Our reencoding method begins by making the input formula clash-free through variable elimination, as described in the proof. Apart from eliminating clashes, variable elimination increases the size of ULCs. The larger the ULC, the more likely the auxiliary variables will be beneficial, as they allow the solvers to reason about more objects at the same time.

3.3 Aligning ULCs

In this section, we assume that a formula is clash-free. Any formula that is not clash-free can be easily turned into one by applying the method described in Section 3.2. While all literals are symmetric in the direct encoding, their order matters when performing our reencoding. Since ULCs are clauses, the literals can be permuted in any way to optimize performance. We consider ULCs as sequences of literals in this section. The following example shows that the order of literals in ULCs could impact the effectiveness of our reencoding.

▶ **Example 9.** Consider a graph-coloring problem with k colors. Let the graph contain a 4-clique among the vertices u, v, w, x. The direct encoding contains the ULCs $(u_1 \vee \ldots \vee u_k)$, $(v_1 \vee \ldots \vee v_k)$, $(w_1 \vee \ldots \vee w_k)$, $(x_1 \vee \ldots \vee x_k)$. The variables u_i denote that vertex u has

color i. Reencoding these ULCs using the natural ordering of the literals creates auxiliary variables that can be interpreted as $u_{\leq i}$, $v_{\leq i}$, $w_{\leq i}$, $x_{\leq i}$ with $i \in \{1, \ldots, k-1\}$.

Now, let's consider the assignment that makes $u_{\leq 3}$, $v_{\leq 3}$, $w_{\leq 3}$, $x_{\leq 3}$ false, which makes variables $u_1, u_2, u_3, v_1, v_2, v_3, w_1, w_2, w_3, x_1, x_2, x_3$ false by unit propagation. Using $k = 6$, a solver can quickly determine that this assignment cannot be extended to a satisfying assignment as at least four distinct colors are required for the vertices u, v, w, x. Hence, the solver can learn the clause $(u_{\leq 3} \vee v_{\leq 3} \vee w_{\leq 3} \vee x_{\leq 3})$. Note that a similar clause with original variables would be much longer: $(u_1 \vee u_2 \vee u_3 \vee v_1 \vee v_2 \vee v_3 \vee w_1 \vee w_2 \vee w_3 \vee x_1 \vee x_2 \vee x_3)$.

Alternatively, consider the case of a different order of literals in ULCs. For example, using $k = 6$, they could be $(u_4 \vee u_2 \vee u_5 \vee u_1 \vee u_6 \vee u_3)$, $(v_2 \vee v_1 \vee v_4 \vee v_3 \vee v_6 \vee v_5)$, $(w_1 \vee w_3 \vee w_2 \vee w_6 \vee w_5 \vee w_4)$, $(x_6 \vee x_2 \vee x_4 \vee x_3 \vee x_1 \vee x_5)$. After reencoding, the auxiliary variables have no meaning. Moreover, assigning the corresponding four auxiliary variables to false will falsify the first three literals in each clause to false by unit propagation. Now it is still possible to assign the vertices u, v, w, x to four distinct colors, so no (short) clause can be learned. Therefore, order of literals in ULCs is crucial for learning the short clause.

To allow the solver to make good use of the auxiliary variables, we want to order the literals in the ULCs so that the solver can learn more useful short clauses like the one above. This motivates our definition of *aligned* ULCs.

▶ **Definition 10.** *Let Γ be a formula whose unique literal clauses are $\{C_1, \ldots, C_m\}$. We say that Γ is aligned if for every pair of binary clauses $(\overline{a} \vee \overline{b}), (\overline{c} \vee \overline{d}) \in \Gamma$ 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.*

As we will show in the experimental evaluation, aligning ULCs increases the effectiveness of our reencoding method. Moreover, when formulas are not aligned or unalignable, the reencoding can become ineffective or even harmful to performance.

▶ **Definition 11.** *A formula is* independent *if it is trivially aligned, that is, it does not contain any binary clauses between its ULCs. Suppose Γ is not independent. Then it is* alignable *if there exists a permutation of the literals in its ULCs such that it becomes aligned, and* unalignable *otherwise.*

Whether a formula is alignable is computable in linear time in the size of the formula. We will discuss this in more detail in Section 4. Since we can efficiently determine whether a formula is alignable, we can run this check first and only reencode alignable formulas.

3.4 Exclusive Literal Clauses

Manthey and Steinke discuss transforming the direct encoding of ExactlyOne constraints [21]. They look for the following pattern: a clause $(\ell_1 \vee \cdots \vee \ell_k)$ together with all binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ with $1 \leq i < j \leq k$. Note that they do not require literals $\ell_1, \ldots, \ell_k$ to occur uniquely in the formula, in contrast to ULCs. On the other hand, ULCs do not rely on the presence of binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$. Thus, each method targets a different pattern. The concept of exclusive literal clauses generalizes both methods.

▶ **Definition 12.** *Given a formula Γ and a clause C. Let $U(C, \Gamma)$ be the set of literals in C that do not occur in $\Gamma \setminus \{C\}$ and $X(C, \Gamma) = C \setminus U(C, \Gamma)$. A clause C is an* exclusive literal clause (XLC) *with respect to Γ if $(\overline{\ell} \vee \overline{\ell'}) \in \Gamma$ for all $\ell, \ell' \in X(C, \Gamma)$.*

So, a ULC is an XLC in which all literals are unique, while Manthey and Steinke detect XLCs with no unique literals. For the purpose of experiments, it is convenient to consider XLCs that are not ULCs separately, and we call them *proper* XLCs.

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 13.** 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 XLC. However, doing so can be very costly in practice, as many clauses may contain the resolution variable. We observed that there are many formulas from SAT Competitions for which resolving away all pairs of clashing XLCs resulted in an enormous blowup of the formula. For these reasons, we will not resolve clashing XLCs in our experiments, and we apply only the sequential counter encoding in our experiments, without trying to eliminate into the order encoding. Definitions regarding alignment extend straightforwardly to formulas with XLCs.

3.5 Proof Production

Modern SAT solvers support proof logging, so the results of each run can be verified [14]. All techniques used in our reencoding technique can be compactly expressed in the DRAT proof system [32]. The first step of our method, described in Section 3.2, makes the formula clash-free. Each step either resolves two ULCs or eliminates a pure literal. 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. The procedure for XLC is similar but somewhat more complicated.

1. Make the ExactlyOne constraint explicit. Before reencoding a ULC $(\ell_1 \vee \cdots \vee \ell_k)$, we need all the binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ with $1 \leq i < j \leq k$ to be present in the formula. Each of the missing ones can be added using blocked clause addition.
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 $1 < i < k$. 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 $\ell_1, \ldots, \ell_k$ 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 $1 < i < k$, while for the at-least-one part we add the clause $(s_{k-1} \vee \ell_k)$. All these clauses have the reverse unit propagation (RUP) property w.r.t. the formula and can be added in arbitrary order [11].
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 Implementation

We implement XLC reencoding inside the state-of-the-art solver CaDiCaL [5]. 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 [27]. We perform reencoding after lucky search so that trivial problems are still solved quickly.

1. Classify XLCs
2. Perform variable elimination with clashing ULCs
3. Optionally, align all XLCs
4. Add sequential counter encoding
5. Remove tautologies and clauses that became redundant
6. Optionally, perform variable elimination to create the order encoding

First, we classify clauses as XLCs. Each clause in the formula is considered during this stage, and the clause size cutoff is only enforced during reencoding in step 4. We compute occurrence counts for each literal in the formula Γ, then classify clauses as an ULC, XLC, or neither. In the first check, the clause is considered a ULC if $\mathsf{occ}(\ell, \Gamma) = 1$ for all $\ell \in C$. If this check fails, then in the second check, the clause is classified as an XLC if all of the binary clauses between the negations of all pairs of non-unique literals in C exist in the formula. To perform this check efficiently, we use a lookup table that stores the binary clauses for each literal. Finally, if neither check succeeds, the clause is neither a ULC or XLC and therefore is not a candidate for reencoding.

Second, we perform resolution on ULC clashes. 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 clauses 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 two forms: (1) the resolvent is a tautology in which case we delete it; (2) 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 clash with an existing ULC. We do not perform variable elimination on clashing XLCs that are not ULCs to avoid a blowup in the formula size.

Third, we optionally align XLCs. The alignment algorithm is described in more detail below in Section 4.1. Alternatively, the user may specify the natural or a shuffled ordering for literals within XLCs. The natural ordering is the ordering of literals by variable names (ids in the DIMACS format). This ordering of literals within each XLC will determine which clauses are generated in the sequential counter encoding. This step is crucial, even without alignment enabled 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 3.5 do not exist in the formula, in which case we can add them or add them directly or can add a more compact set of clauses [12].

Fourth, we reencode each XLC by adding the sequential counter encoding to the formula.

We generate new variables for the encoding starting from one plus the maximum variable 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.

Fifth, 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.

Sixth, 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 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 upon solve completion. 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 [17]. 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.

4.1 Alignment of XLCs

This section presents a procedure for aligning the XLCs within a formula. This amounts to assigning each literal occurring in an XLC an integer alignment value and then using these values to sort the literals within the XLCs.

To perform alignment, we first construct a graph G where nodes are literals and the graph contains an undirected edge between two literals ℓ and ℓ' iff $(\overline{\ell}' \vee \overline{\ell}) \in \Gamma$ and both ℓ and ℓ' occur in distinct XLCs. Next, we sort the XLCs from largest to smallest, sort the literals within each individual XLC by the natural ordering, and initialize the alignment of all literals to 0. Then, for each $C \in XLCs$ and for each $\ell \in C$: if ℓ is not aligned, ℓ is assigned $alignment + 1$ and all unaligned literals in the connected component of G containing ℓ are aligned with $alignment + 1$. We increment $alignment$, then continue to the next literal. After aligning all literals in C, the clause is sorted based on the alignment values. If any pair of literals in C shares the same alignment value, the problem is said to be unalignable. This will happen if two literals from an XLC occur in the same connected component in G. In this case, the literals are sorted lexicographically by alignment and then by natural ordering. If no alignment occurs and the formula is independent (i.e., G is empty), then each clause retains its natural literal ordering.

▶ **Example 14.** Consider the following formula:

$$C_1 : (\ell_1 \vee \ell_2 \vee \ell_3) \wedge C_2 : (\ell_4 \vee \ell_5 \vee \ell_6) \wedge (\overline{\ell}_1 \vee \overline{\ell}_6) \wedge (\overline{\ell}_2 \vee \overline{\ell}_4) \wedge \Gamma$$

Assume Γ does not contain a binary clause with literals from C_1 and C_2, so they are both ULCs. We first consider the literals from C_1 in natural order. After ℓ_1 is given alignment 1, ℓ_6 also receives this alignment because of the binary clause $(\overline{\ell}_1 \vee \overline{\ell}_6)$. ℓ_2 is given alignment 2, along with ℓ_4. ℓ_3 is given alignment 3. Now, C_1 is sorted by alignment to $\ell_1 \vee \ell_2 \vee \ell_3$. When C_2 is processed, both ℓ_4 and ℓ_6 are skipped because they already have alignments. ℓ_5 is given alignment 4, and the clause is sorted as $\ell_6 \vee \ell_4 \vee \ell_5$. If the binary clause $\overline{\ell}_2 \vee \overline{\ell}_5$ would have been in Γ, the formula becomes unalignable because ℓ_4 and ℓ_5 would receive alignment 2.

The alignment procedure uses a linear number of steps to assign alignment values, processing each literal within an XLC once and processing all binary clauses within the formula at most once. There are multiple possible alignments for any set of alignable XLCs. For instance, in Example 14, ℓ_5 can be placed at the beginning of C_2 and the XLCs would still be aligned. Specifically, after assigning each literal an alignment value, those values can then be ordered in some way, and that ordering can be used to sort the clauses. In addition, the alignment procedure will always produce an aligned formula if the formula is alignable. This is the case because our procedure aligns all variables within a connected component before moving on to the next connected component, ensuring that the property in Definition 10 is not violated.

The independent and unalignable problems may also benefit from ordering. For example, the `shift1add` family benchmarks each contain only a single ULC (after resolving clashing ULCs) and are thus independent, but in experiments we found by accident that different orderings can lead to 10× difference in solving time. Recent work sorts literals in a single cardinality constraint [25], and a similar approach may be beneficial here. Furthermore, for unalignable problems, partial alignment may still be possible. We focus on the alignable case

Table 1 Number of formulas reencoded, along with their formula types, average preprocessing time, number of formulas that take longer than 20 seconds, the medium number of encoded XLCs, and the medium of the maximum sizes of encoded XLCs. ULC-Clash is ULC detection without resolution on clashing literals. XLC shows results for formulas where at least one proper XLC was detected and reencoded. XLC-full is full detection, combining ULC and proper XLC. SBVA is only run on formulas for which an XLC was detected, and the reencoded count shows the number of those formulas for which SBVA performed any reencoding.

	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	–	–

as these benchmarks tend to yield the most positive results, and leave ordering of independent and unalignable formulas for future work.

5 Results

We ran our implementation to scan and reencode formulas in the Anniversary track of SAT Competition 2022 [4] for ULCs and XLCs, and the data are in Table 1. Of the 5,355 benchmarks, our method reencoded 1,739 ($\sim$ 32%) formulas with at least one XLC, with 1,014 ($\sim$ 19%) of them having ULCs, of which 459 ($\sim$ 9%) are alignable. These formulas are encoded after resolving clashing ULCs and filtering out clauses of size 4 or less, as explained in Section 4. Notice that by resolving clashing ULCs, we obtain more formulas containing ULCs above the set size threshold. This suggests that our method can sometimes find implicit large ExactlyOne constraints that the user is not aware of when encoding a problem. We also tested if the state-of-the-art reencoding technique SBVA (structured bounded variable addition) [13] would reencode these formulas, and found that it does affect most of them, so our experiments include a performance comparison between our method and SBVA, which will be discussed later in this section. An observation to be made now is that our method is very efficient, taking less than 5 seconds on average to reencode these formulas, while SBVA took 30 seconds on average.

All experiments were carried out at the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [7], running solvers on the 1,014 benchmarks with ULCs. We used a 5,000 second timeout for solving, and 64 experiments were run in parallel per node, so each process held approximately 4GB of memory. 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 [13]. All runtimes include reencoding time.

We used the following solving configurations for reencoding ULCs: original encoding, SBVA, sequential counter encoding with aligned, natural, or shuffled sorting, and order encoding with aligned sorting. Resolution was performed on clashing unique literal clauses before applying sequential counter and order encodings. Table 2 presents the performance of these configurations on all formulas that contain ULCs, divided into three classes: aligned, independent, and unalignable. Each class is further divided into satisfiable (SAT) and unsatisfiable (UNSAT) instances. Both the number of solved instances and the average solve time (where unsolved formulas contribute twice the time limit, denoted by Par2) are

Table 2 Performance of ULC reencoding across different configurations. *SBVA uses original encoding for the 123 formulas it does not reencode.

	Alignable (459)				Independent (216)				Unalignable (339)			
	# Solved		Avg. PAR2		# Solved		Avg. PAR2		# Solved		Avg. 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

reported. Best performing configurations in each column are highlighted in bold, and a VBS (virtual best solver, which picks the best solver from available configurations) configuration is included for reference.

Reencoding ULCs with our method provides a significant speedup on solving alignable formulas, even when compared with SBVA. In particular, the sequential counter encoding with alignment yields the best overall results. We visualized the impact of our aligned sequential counter ULC reencoding on solver performance using scatter plots, comparing it against configurations on different formula classes. In each plot, every point represents one benchmark formula, comparing the runtime or conflicts of two solver configurations. Points below the diagonal indicate instances where the tested configuration yields improvement against the baseline configuration (lower values on the y-axis), while points above the line indicate slower performance. We distinguish satisfiable (SAT) and unsatisfiable (UNSAT) instances by marker style in each plot to analyze performance trends across these classes. We now discuss each comparison in detail, highlighting the key patterns and what they reveal about the reencoding method.

We also ran experiments for reencoding XLCs with the aligned sequential counter encoding on formulas where at least one proper XLC was detected and reencoded. On these formulas, reencoding had limited impact, and data will be provided in Section 5.5.

5.1 Alignable Formulas

The left plot of Figure 1 compares solver runtimes on alignable formulas, contrasting our method's aligned sequential counter encoding against the original encoding (baseline). For alignable formulas, we observe a clear performance improvement with the aligned reencoding, particularly on UNSAT instances. Most UNSAT points lie well below the diagonal, often on the axes, indicating that adopting our method can solve more UNSAT formulas using much less time. The SAT instances exhibit mixed results, but still, our method solves more SAT formulas and has an overall advantage. We have verified all the proofs and satisfying assignments produced by our method on the original formulas in this plot to make sure these speedups are not due to conceptual or implementation errors.

To understand the importance of aligning variables in ULCs, we also evaluated a shuffled reencoding configuration, where the literals in each ULC are randomized (and not aligned) before reencoding. The right plot of Figure 1 compares this configuration against the original encoding on the same set of alignable benchmarks. In stark contrast to the left one,

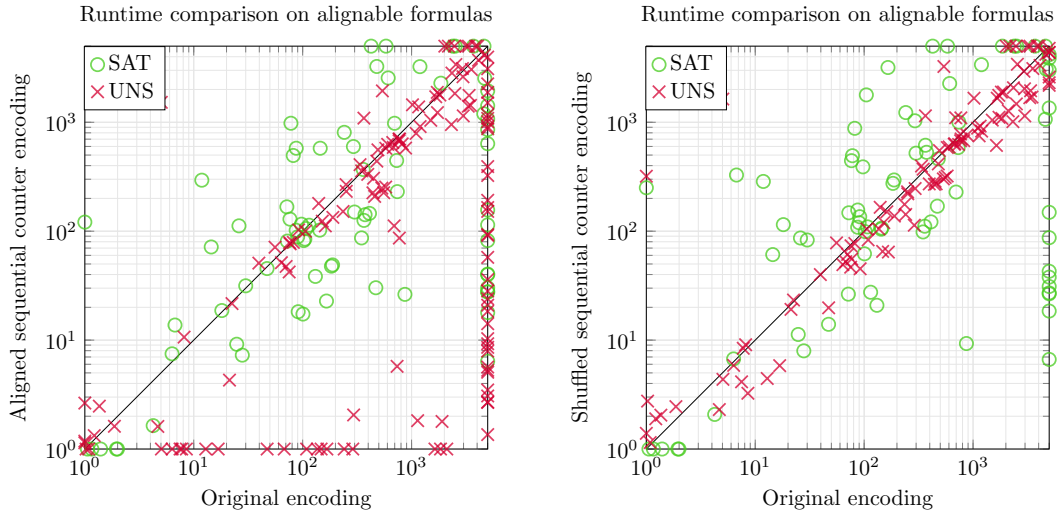

Figure 1 Sequential counter encoding on alignable instances: Aligned vs. original (left) and shuffled vs. original (right).

shuffling largely negates the performance gains of our reencoding method. Most points in the plot lie near the diagonal, indicating no change over the baseline. While some formulas experience degraded performance, several satisfiable instances still benefit substantially from the reencoding, even after shuffling. However, the overall outcome is that shuffling offsets the advantage of our reencoding method.

Table 3 shows solver performance on the subset of benchmarks explicitly labeled with "Shuffled" or "Random". Sequential counter and order encodings with aligned sorting provide significant benefits, notably improving the number of solved UNSAT instances (from 55 solved by the original encoding to 73 with the order encoding). These results confirm the importance and effectiveness of alignment.

5.2 Independent and Unalignable Formulas

Experiments on shuffled alignable formulas demonstrate that when the formulas are not aligned, our method does not consistently provide benefits. We therefore expect the same when our method is applied to independent and unalignable formulas. Figures 2 examine

Table 3 Results on aligned with Shuff or Rand in benchmark name (SAT: 63, UNS: 76, UNK: 74)

	# Solved		Avg. PAR2	
	SAT	UNS	SAT	UNS
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

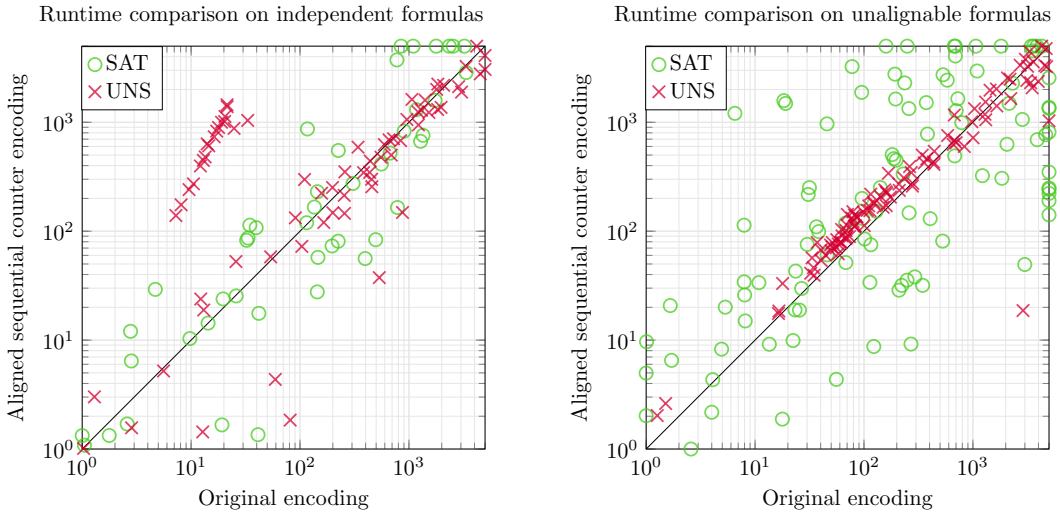

Figure 2 Runtime comparison between the aligned sequential counter encoding and the original encoding on independent formulas (left) and unalignable formulas (right). The cluster of UNSAT formulas in the top left of the independent plot are the shift1add formulas mentioned in Section 4.

the solver's performance on these instances, comparing the reencoded formula to the original. In both classes, most UNSAT instances lie close to the diagonal, and SAT instances are spread out, with no clear performance gains observed. The results suggest that reencoding ULCs tends to provide no benefit, occasionally even hurting performance. As mentioned in Section 4, future work on ordering literals in independent and unalignable formulas may yield better results with our reencoding.

5.3 Order Encoding

Our reencoding method can transform the direct encoding of ExactlyOne constraints to the sequential counter or the order encoding. The only difference between the two is that for the latter, variables from the direct encoding are eliminated using variable elimination. Bounded variable elimination [9], a technique used in all state-of-the-art SAT solvers, eliminates variables if that results in a reduction in the number of clauses. After reencoding ULCs, this is the case for all variables from the direct encoding, so solvers could eliminate them. However, they may not be eliminated because the solver can choose to eliminate some of the order variables instead. After eliminating some order variables, the variables from the direct encoding can no longer be eliminated without increasing the size of the formula.

Figure 3 shows the comparison between the sequential counter encoding and the order encoding on aligned formulas. When we compare the runtime (left), both encodings perform quite similarly. This is even more apparent when comparing the number of conflicts (right). If we only focus on the number of solved benchmarks, then the order encoding solves a couple of extra UNSAT formulas, while the sequential encoding solves a couple of extra SAT formulas. The results suggest that the solver does not always eliminate the original variables in ULCs, and sometimes it is better to keep them.

Figure 4 shows the runtime and conflict comparison between the sequential counter encoding and the order encoding on unalignable and independent formulas. The left plot shows that the sequential counter encoding is noticeably faster than the order encoding. This contrasts with the aligned formulas, where the two methods performed similarly. The right

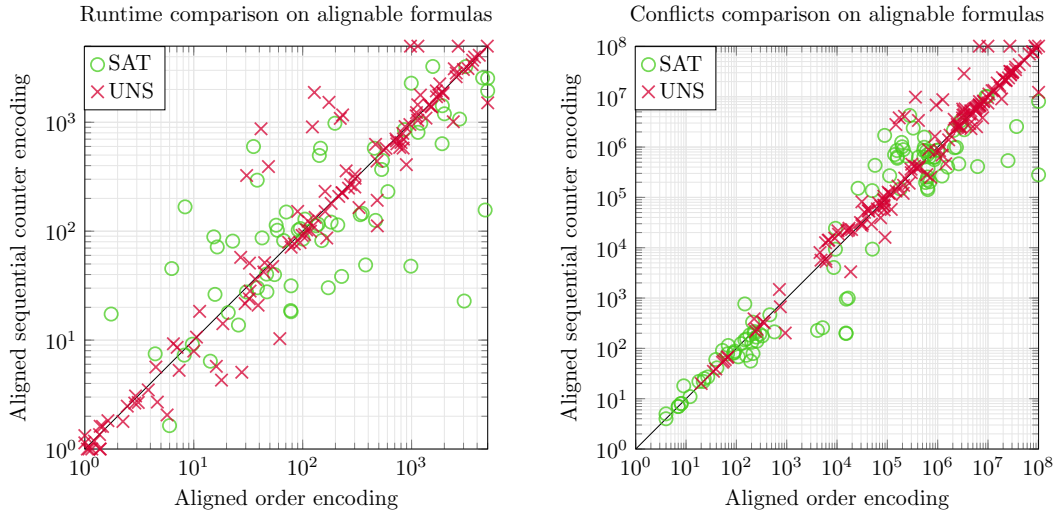

Figure 3 A comparison in runtime (left) and the number of conflicts (right) between the sequential counter encoding and the order encoding on alignable formulas with ULCs.

plot suggests that the worse performance of the order encoding is due to slower propagation rather than increased conflicts.

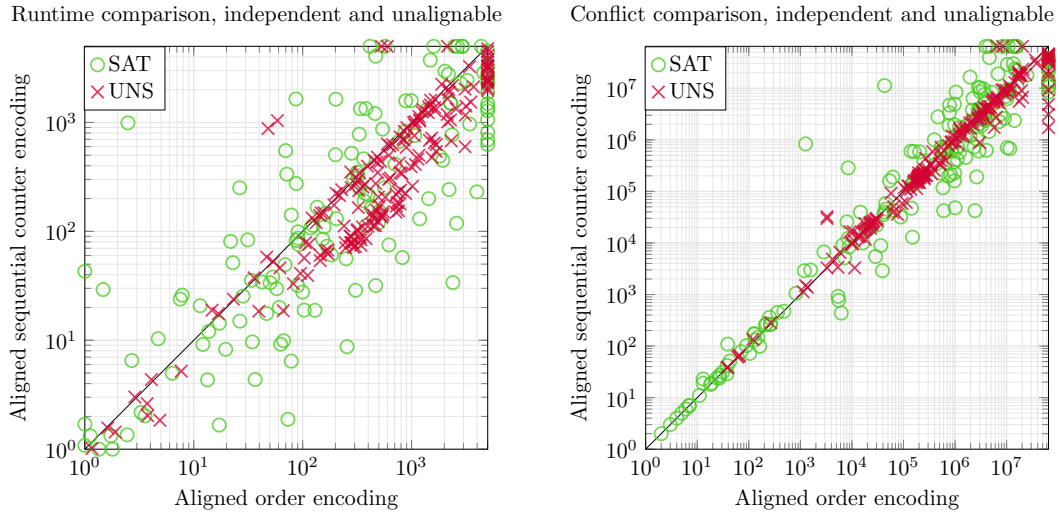

Figure 4 Comparison between runtime (left) and conflicts (right) on unalignable and independent formulas. The reduced runtime on unsatisfiable formulas is due to slower propagation.

5.4 Bounded Variable Addition

Bounded variable addition (BVA) is a reencoding technique that searches for sets of clauses that can be reencoded to a smaller set of clauses by introducing a new variable [20]. In most cases, BVA will more compactly encode the set of binary clauses. BVA was recently improved by taking the structure of the formula into account during reencoding. This version is known as structured BVA (SBVA) [13]. The combination of SBVA and CaDiCaL won the main track of SAT Competition 2023.

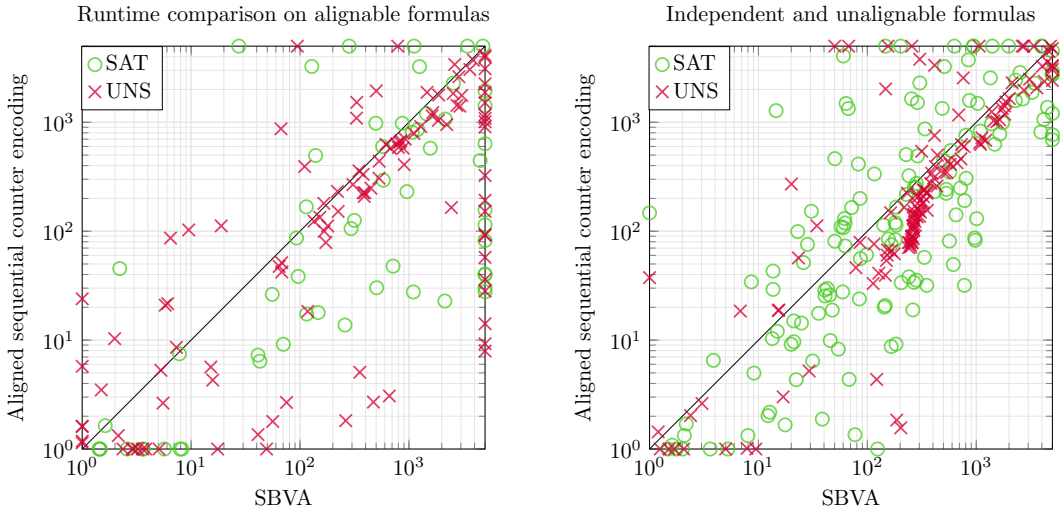

Figure 5 A runtime comparison between aligned sequential counter encoding and SBVA on alignable formulas (left) and unalignable and independent formulas (right). The runtime includes the reencoding time. Formulas for which SBVA does not perform reencoding are filtered out.

We compared our reencoding method with SBVA as preprocessor for CaDiCaL. The results are shown in Figure 5. Our sequential counter encoding with alignment can solve dozens of alignable formulas that cannot be solved when using SBVA, suggesting that they reencode the formulas differently. Note that a large majority of SAT formulas are below the diagonal, showing that our reencoding works much better on both SAT and UNSAT formulas.

On unalignable and independent formulas, the picture is more mixed. Although most points are below the diagonal, there are more formulas that time out after applying our reencoding while they can be solved when using SBVA.

5.5 Reencoding XLCs

Table 4 and Figure 6 show the performance of sequential counter encoding on formulas containing proper XLCs. Our reencoding consistently helps solve UNSAT formulas while making SAT formulas slower to solve. The impact of the reencoding is not as significant as on formulas containing only ULCs, especially when looking at alignable formulas. XLC reencoding solved two more UNSAT instances (134 vs. 132) but one fewer SAT instance (118 vs. 119), with a modest reduction in average UNSAT solving time (PAR2 reduced from 723 to 525), and a slight increase in average SAT solving time (PAR2 increased from 318 to 397).

Although the improvement is small compared with reencoding the ULCs in alignable formulas, reencoding XLCs still provides an overall performance gain. Moreover, the performance is consistent across alignable, independent, and unalignable formulas. Reencoding XLCs always helps solve a couple more UNSAT formulas, even on independent and unalignable formulas. This was not the case when reencoding the formulas with only ULCs, as our reencoding generally slightly hurts performance on non-alignable formulas.

5.6 Benefited Benchmark Families

We briefly describe some specific families of alignable formulas that our method for reencoding ULCs drastically improves solver performance on.

Table 4 Performance of XLC reencoding

	Alignable (305)				Independent (187)				Unalignable (333)			
	# Solved		Avg. PAR2		# Solved		Avg. PAR2		# Solved		Avg. 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

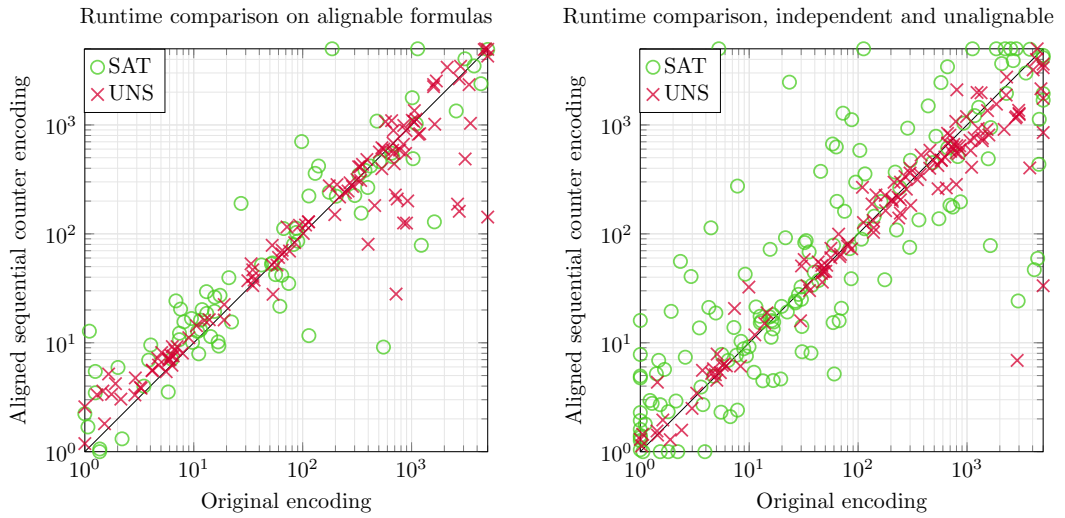

Figure 6 Performance comparison between the sequential counter and the original encoding on formulas with at least one proper XLC. Left, the alignable formulas and right, the unalignable and independent formulas.

- **Pigeonhole** Pigeonhole formulas try to put n pigeons into m holes, consisting of many ULCs (every pigeon must be in one hole). It is one of the famous problems that is easy for humans and hard for SAT solvers, and reencoding ULCs really helps. For example, both the original encoding and SBVA timed out on the formula `php-018-014.shuffled`, which is a shuffled formula that tries to put 18 pigeons in 14 holes, while after reencoding it was solved in 192 seconds. Similar speed boosts 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 them. For example, our method solved `queen14_14.col.14` [15] in 9 seconds and `le450_15b.col.15` in 28 seconds, while SBVA and original encoding timed out on both of them.
- **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 [22]. 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 [6]. 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 drastically improved solver performance on these formulas. For example, `vlsat2_30744_3925645.dimacs` [3], 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 [2]. The formula contains many unalignable ULCs encoding the permutations.

5.7 Graph coloring

Based on the strong results on graph-coloring problems, we tested whether our version of CaDiCaL enhanced with reencoding boost SAT-based graph-coloring approaches. For this purpose, we used the CliColCom framework [16], which represents the state-of-the-art in graph coloring and includes several techniques to improve performance for this application, such as symmetry breaking. We ran CliColCom on the DIMACS graph coloring suite [18]. Using CaDiCaL version 2.1.0 (unmodified), the framework could solve 90 benchmarks within an hour, while using the modified CaDiCaL (same version, but with our reencoding technique) allowed solving two more benchmarks. The additional benchmarks are: `queen9_9` (solved in 116 seconds) and `queen10_10` (solved in 2103 seconds). The runtime on most graphs was similar. An exception is `abb313GPIA`, which was solved in 2.12 seconds using the modified version and in 150.11 seconds using the unmodified CaDiCaL. Note that this graph was not solved with CaDiCaL version 1.5.2 used in the CliColCom paper [16].

6 Related Work

Encodings of ExactlyOne constraints have been studied for over 20 years. A seminal early work by Ansótegui and Manyá [1] already uses order variables. Our direct, sequential counter, and order encodings correspond to their standard mapping, regular mapping, and full regular mapping, respectively. They evaluate different encodings on mostly random problems, including graph coloring using the Chaff and Siege solvers. The encodings with order variables generally result in stronger performance, although Chaff is faster using the direct encoding on sparse graphs. A later study didn't observe significant differences in solver performance using various encodings [23]. Our evaluation is the first to involve a massive benchmark suite and concludes that the order variables can boost performance.

A more recent study zooms in on the effectiveness of a range of encodings on graph-coloring problems [10]. That study shows that variants of the order encoding are the most effective on the classical suite of graph-coloring problems [18]. Most instances of that suite are relatively easy, as the chromatic number equals the clique number, so if you find the largest clique, the lower bound call becomes trivial after symmetry breaking. The order encoding is especially effective on the `queen` graphs and `abb313GPIA`, which are exactly the graphs that can be solved using CliColCom when using our reencoding technique.

Other related works deal with reencoding formulas [13, 20, 21]. These works focus on reducing the number of clauses by introducing new variables. In practice, this comes mostly down to constructing a more compact representation of the binary clauses. In contrast, our

reencoding may increase the number of clauses, especially in case of formulas with implicit ExactlyOne constraints. The work by Manthey and Steinke [21] is the closest to ours as it also targets ExactlyOne constraints, although they limit their technique to explicit ExactlyOne constraints. Their evaluation showed little benefit of performing reencoding, which is likely due to the small scale of the experiments and ignoring the implicit ExactlyOne constraints.

7 Conclusions

We presented a new reencoding technique for propositional formulas. While all existing work on reencoding focuses on reducing the size of formulas, our technique increases the number of variables and may also increase the number of clauses. The experimental results demonstrate that our method increases the number of solved formulas from SAT Competitions, providing concrete evidence that our reencoding method can significantly enhance solver performance and even outperform state-of-the-art reencoding techniques on the targeted class of formulas. Importantly, our implementation scans and recovers the target structure in a formula before reencoding, allowing us to predict the performance of our method and determine whether to apply it. Future work on formula structures could leverage our method to a broader range of formulas.

We conjecture that the new variables help guide the solver. However, we could not show that the new variables could exponentially reduce the size of the smallest resolution proof. Answering this question is part of future work.

Our implementation of reencoding logs DRAT proofs, allowing us to formally verify the results presented in the experimental evaluation. In some cases, DRAT proof validation was expensive. To address this issue, CaDiCaL provides LRAT proof logging for all implemented reasoning techniques to create efficiently checkable proofs. In future work, we plan to support LRAT proof production.

References

1 Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, pages 1–15, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

2 Calin Anton and Lane Olson. Generating satisfiable SAT instances using random subgraph isomorphism. In Yong Gao and Nathalie Japkowicz, editors, *Advances in Artificial Intelligence, 22nd Canadian Conference on Artificial Intelligence, Canadian AI 2009, Kelowna, Canada, May 25-27, 2009, Proceedings*, volume 5549 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2009.

3 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, Finland, 2021.

4 Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2022. Accessed: 2025-03-13. URL: http://hdl.handle.net/10138/359079.

5 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024.

6 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, Berlin, Heidelberg, 2021. Springer-Verlag.

7 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. Association for Computing Machinery, New York, NY, USA, 2021.

8 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*, AAAI'94, page 1092–1097. AAAI Press, 1994.

9 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 *LNCS*, pages 61–75. Springer, 2005.

10 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, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

11 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf.

12 Isaac Grosof, Naifeng Zhang, and Marijn J. H. Heule. Towards the shortest drat proof of the pigeonhole principle. In *Workshop on Pragmatics of SAT (PoS')*, 2022. URL: https://api.semanticscholar.org/CorpusID:251040741.

13 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, pages 11:1–11:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl.

14 Marijn J. H. Heule. *Proofs of Unsatisfiability*, chapter 15, pages 635 – 668. IOS Press, 2021.

15 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, Finland, 2018.

16 Marijn J. H. Heule, Anthony Karahalios, and Willem-Jan van Hoeve. From Cliques to Colorings and Back Again. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:10, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2022.26.

17 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 355–370, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

18 David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

19 O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96–97:149–176, 1999.

20 Norbert Manthey, Marijn Heule, and Armin Biere. Automated reencoding of Boolean formulas. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *HVC 2012*, volume 7857 of *LNCS*, pages 102–117. Springer, 2012.

21 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.

22 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, New York, NY, USA, 2001. Association for Computing Machinery.

23 Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. Empirical study on sat-encodings of the at-most-one constraint. In *The 9th International Conference on Smart Media and Applications*, SMA 2020, page 470–475, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3426020.3426170.

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

25 Joseph E. Reeves, Joao 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)*. AAAI Press, 2025.

26 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses [inline-graphic not available: see fulltext]. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, page 110–132, Berlin, Heidelberg, 2024. Springer-Verlag.

27 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. doi:10.1016/0020-0190(95)00019-9.

28 Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

29 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.

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

31 G. S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_28.

32 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, Cham, 2014. Springer International Publishing.