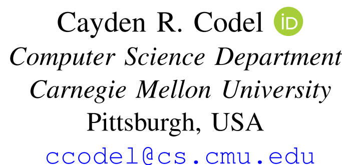
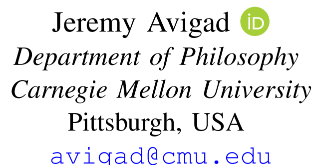
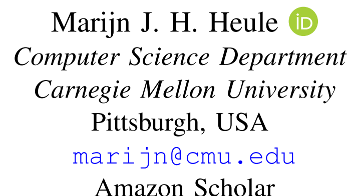
Formal Methods in Computer-Aided Design 2024

Verified Substitution Redundancy Checking


Cayden R. Codel
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
ccodel@cs.cmu.edu


Jeremy Avigad
Department of Philosophy
Carnegie Mellon University
Pittsburgh, USA
avigad@cmu.edu


Marijn J. H. Heule
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
marijn@cmu.edu
Amazon Scholar

***Abstract*—Modern SAT solvers are trustworthy because their results can be expressed in formal proof systems and validated with verified proof checkers. Today, the RAT and PR proof systems are the de facto standard: they capture many reasoning techniques used by SAT solvers, and they are supported by efficient, formally-verified checkers. However, RAT and PR struggle to succinctly express advanced reasoning techniques like symmetry breaking.**

In this paper, we present proof checking tools for the substitution redundancy (SR) proof system, a powerful generalization of PR and RAT. We first highlight three problems with linear-size SR proofs that are not expected to have linear-size PR or RAT proofs. We then present proof formats for SR with and without unit propagation hints, a tool to add those hints, and the first verified SR proof checker. Since SR generalizes other proof systems, our checker has the distinction of supporting the strongest clausal proof system to date. Finally, our experimental results show that SR proofs are much smaller than their RAT counterparts, and that verified SR proof checking is efficient in practice.

I. Introduction

Satisfiability (SAT) solving continues to be a crucial tool for industry and academia. For example, SAT solvers were recently used to resolve open problems in computational geometry [1, 2] and to improve lower bounds for a problem in quantum mechanics [3]. In addition, SAT solvers form the core of SMT solvers, which are queried a billion times a day by various Amazon Web Services applications [4].

SAT solving is *trustworthy* due to the development of verified proof checking. When reporting that a problem has no solutions, modern SAT solvers emit a corresponding *proof of unsatisfiability* expressed in a formal *proof system*. By validating these proofs with verified software, we gain a high degree of confidence in solver results, particularly those backing mathematical theorems and industrial security guarantees.

Proof systems and SAT solvers are complementary, with advances in one driving innovations in the other. For example, the RAT proof system [5] was developed to validate the learned clauses produced by CDCL solvers and some inprocessing techniques. In the other direction, the PR proof system [6], a generalization of RAT, validates short clauses that solvers could not initially derive. But solvers have since caught up: the 2nd- and 8th-place finishers at the 2023 SAT Competition used PR reasoning as a preprocessing step [7, 8]. More generally, stronger proof systems enable solvers to use more-powerful reasoning techniques and produce shorter proofs.

Many thanks to the researchers on the Lean theorem prover Zulip chatroom. Our research was supported by NSF grant CCF-2229099, AFRL agreement FA8750-24-9-1000, and the Hoskinson Center for Formal Mathematics.

In this paper, we develop verified proof checking tools for the substitution redundancy (SR) proof system [9–11] (Section II). SR generalizes PR, such that it can succinctly express a broad range of symmetry-breaking techniques that PR cannot. We show this by highlighting three problems that have SR proofs with size linear in the number of variables and that, as far as we know, do not have linear-sized PR proofs (Section III).

Currently, no solver supports SR reasoning. However, we expect that the availability of our fast, verified SR checker will stimulate the development of SR reasoning and preprocessing techniques, similar to what happened with PR.

To enable SR proof checking (Section IV), we introduce the DSR and LSR proof formats (Section V). Like the formats for RAT and PR, DSR proofs record basic proof steps, while LSR proofs include unit propagation hints that guide proof checking. These formats are backwards compatible with the ones for RAT and PR. Our set of SR proof checking tools (Section VI) includes a tool that converts DSR proofs into DRAT and a tool that converts DSR proofs into LSR by adding hints.

We also present the first verified LSR proof checker (Section VII), giving it the distinction of supporting the strongest clausal proof system to date. It implements several data structures and techniques commonly used in SAT proof checkers, and we discuss how they impacted our formalization. We proved our checker correct with the Lean theorem prover [12].

Finally, our experimental results (Section VIII) show the clear benefits of using a stronger proof system. For example, we found that the SR proofs from our benchmarks had 0.4% as many proof lines as their transformations into RAT. We also show that verified SR proof checking is efficient in practice: our verified checker performs similarly to cake_lpr [13], a fast, verified PR proof checker written in CakeML [14].

II. Substitution Redundancy

We assume that the reader is familiar with SAT.[1] Throughout, let F be a formula in conjunctive normal form (CNF), let C and D be clauses of boolean literals, and let τ be a truth assignment on the boolean variables in F. We write $\overline{\ell}$ for the negation of boolean literal ℓ, and $\neg C := \bigwedge_{\ell \in C} \overline{\ell}$ for the negation of a clause C. A *partial truth assignment* is a non-tautological

[1] For background reading, see Ch. 15 of the Handbook of Satisfiability [15].

https://doi.org/

This article is licensed under a Creative Commons Attribution 4.0 International License

set of literals taken to be true. We abuse notation by writing evaluation under (partial) assignments as $\tau(x)$.

When a SAT solver claims that a formula F is unsatisfiable, it emits a proof of unsatisfiability in a formal proof system. In *clausal proof systems*, each proof step adds a clause C to F such that F and $F \wedge C$ are *equisatisfiable*, meaning that F is satisfiable if and only if $F \wedge C$ is. Such clauses are called *redundant*. The backward direction is trivial, so it suffices to show only the forward direction. Notably, adding C to F may (and often does) reduce the set of satisfying assignments, but the important thing is that satisfiability is preserved. By producing a series of redundant clauses ending in the empty clause $\bot$, a SAT solver can show that F is unsatisfiable.

Unfortunately, it is NP-hard to determine whether an arbitrary C is redundant for F [16]. Therefore, most clausal proof systems instead use a property implying redundancy that is checkable in polynomial time when provided with a *witness*. The SR proof system is based on such a property.

Witnesses for clausal proof systems work as follows. Suppose we are trying to show that C is redundant for F. If F *entails* C, written as $F \vDash C$ and meaning that for any $\tau \vDash F$, we have that $\tau \vDash C$, then certainly F and $F \wedge C$ are equisatisfiable. So assume that τ satisfies F but does not satisfy C. We can show that C is redundant by modifying τ into a new assignment satisfying $F \wedge C$. The witness σ describes this modification, and it is used to form the satisfying truth assignment $\tau \circ \sigma$.

As a motivating example (discussed further in Section III), consider the following proof of unsatisfiability for the pigeonhole formula on n pigeons and $n-1$ holes. To start, we learn that the first pigeon p_1 cannot go in the last hole h_{n-1}, i.e., the unit clause $\overline{x}_{1,n-1}$. If $F \vDash \overline{x}_{1,n-1}$, then we'd be done, so assume that $\tau \vDash F$ but $\tau \vDash x_{1,n-1}$, meaning that τ places p_1 in h_{n-1}. One way of modifying τ to satisfy $\overline{x}_{1,n-1}$ is to swap p_1 with a different pigeon, say, p_n, thus ensuring that p_1 ends up in a different hole while still satisfying the overall formula. We accomplish this by mapping the variables associated with p_1 to the variables associated with p_n, and vice versa, before evaluating them under τ. Functions called substitutions capture this technique, and they serve as the witnesses in SR.

Formally, a *substitution* σ maps boolean variables to either a boolean literal or a truth value. They can satisfy clauses and formulas, written as $\sigma \vDash C$ and $\sigma \vDash F$. They can also reduce them. The *reduction* of C under σ, written as $C_{|\sigma}$, is obtained by mapping σ over its literals and removing those falsified by σ. The reduction of F under σ, written as $F_{|\sigma}$, is obtained by reducing each of its clauses and removing those satisfied by σ. We say that σ *reduces* C if σ does not satisfy C and there is a literal $\ell \in C$ not mapped to itself under σ, i.e., $\sigma(\ell) \neq \ell$. Notably, σ can reduce C even if $C_{|\sigma} = C$, as in the example where $C := x_1 \vee x_2$ and σ maps x_1 and x_2 to each other.

Additionally, we can compose substitutions with truth assignments to form new truth assignments. Define $(\tau \circ \sigma)(x)$ as $\sigma(x)$ if $\sigma(x) \in \{\top, \bot\}$ and as $\tau(\sigma(x))$ otherwise. From this definition, we can derive the core lemma used to prove redundancy from the SR property: if σ does not satisfy C, then $\tau \circ \sigma \vDash C \Leftrightarrow \tau \vDash C_{|\sigma}$. In other words, knowing that τ satisfies $C_{|\sigma}$ is the same as knowing that σ modifies τ into an assignment $(\tau \circ \sigma)$ satisfying C. The lemma follows from the definition of composition: let $\ell \in C$ with $\tau(\sigma(\ell)) = \top$. But since $\sigma \nvDash C$, then $\sigma(\ell) \in C_{|\sigma}$, and so $\tau \vDash C_{|\sigma}$.

The SR property is based on *unit propagation* (UP), which computes in polynomial time the partial assignment implied by the unit clauses in a formula F. Starting from the empty partial assignment τ, UP reduces F with the following rule until fixpoint: if $F_{|\tau}$ contains a unit clause x, then $\tau := \tau \cup x$. If UP finds a unit clause $x \in F_{|\tau}$ with $\tau(x) = \bot$, then we write $F \vdash_1 \bot$, and we say that we have a *UP refutation* for F. Such formulas are unsatisfiable.

We extend this definition to include UP derivations of clauses and formulas. If C is a clause, then $F \vdash_1 C$ if $F \wedge \neg C \vdash_1 \bot$. Likewise, if G is a formula, then $F \vdash_1 G$ if $F \vdash_1 C$ for every $C \in G$. It is well-known that if $F \vdash_1 G$, then $F \vDash G$.

We now define SR. A clause C is *substitution redundant* (SR) [9–11] for a formula F if there exists a substitution σ such that $F \wedge \neg C \vdash_1 (F \wedge C)_{|\sigma}$.[2]

The SR property implies redundancy.

Theorem 1 ([9, 10])**.** *Let F be a CNF formula, and let C be a clause. If C is SR for F, then C is redundant for F.*

Proof. It suffices to show the forward direction. If $F \vDash C$, then we'd be done, so assume that $\tau \vDash F$ and $\tau \vDash \neg C$, and let σ be a substitution satisfying the SR property for C and F. We will show that $\tau \circ \sigma \vDash F \wedge C$, meaning that $\tau \circ \sigma \vDash D$ for every $D \in (F \wedge C)$. If $\sigma \vDash D$, then so would $\tau \circ \sigma \vDash D$ and we'd be done, so assume otherwise. Thus $D_{|\sigma} \in (F \wedge C)_{|\sigma}$. But the SR property tells us that since $\tau \vDash F \wedge \neg C$, we have that $\tau \vDash D_{|\sigma}$, and by the lemma, this implies that $\tau \circ \sigma \vDash D$. ☐

SR generalizes PR [6], which itself generalizes RAT [5].[3] If we restrict the witness to be a partial truth assignment, we obtain PR. If we restrict the witness further to be a partial assignment defined by a single literal, we obtain RAT.

III. Short SR proofs

Many problems, including several that are hard for resolution, have SR proofs with size linear in the number of variables. In this section, we describe SR proofs for three such problems that, as far as we know, do not have linear-sized PR proofs. Our manually-constructed proofs are available at:

https://github.com/marijnheule/sr-proofs.

A. The pigeonhole principle

The pigeonhole principle (PHP) states that if n pigeons are placed in $m < n$ holes, then at least one hole contains multiple pigeons. The unsatisfiable PHP CNF formulas on n pigeons and $n-1$ holes consist of $O(n^2)$ variables $\{x_{p,h}\}$, where $x_{p,h} = \top$ means that pigeon p is in hole h, and $O(n^3)$ clauses,

[2] There are several variants of SR. We use the one due to Gocht and Nordström [10] because it is more general and easier to understand than the original definition due to Buss and Thapen [9], which requires that $\sigma \vDash C$ or that $C_{|\sigma}$ is a tautology, and that $F_{|\tau} \vdash_1 F_{|\sigma}$.

[3] These proof systems, and their extension-free variants, form a proof hierarchy with interesting proof-theoretic properties [9, 11].

encoding that each pigeon must be in at least one hole and that two pigeons cannot both be in the same hole. Resolution proofs of PHP formulas are exponential in n [17]. Extended resolution admits proofs of size $O(n^4)$ [18], while PR admits proofs of size $O(n^3)$ [16].

PHP SR proofs consist of $O(n^2)$ unit clauses [9, 11]. They recursively use the following scheme to show that pigeon p_n must go in hole h_{n-1}. We start by learning that p_1 does not go in the last hole, i.e., the unit clause $\overline{x}_{1,n-1}$. This clause is SR because if the satisfying assignment τ assigns p_1 to h_{n-1}, then we may modify τ with the substitution σ that instead assigns p_n to h_{n-1} by swapping (the variables for) the two pigeons. Formally, $\sigma = \{x_{1,n-1} \mapsto \bot,\ x_{n,n-1} \mapsto \top,\ x_{1,i} \mapsto x_{n,i},\ x_{n,i} \mapsto x_{1,i}\}$ for $i \in \{1, \dots, n-2\}$. Next we learn that p_2 does not go in the last hole by swapping p_2 with p_n, and so on, until only p_n can go in h_{n-1}. Now the problem has been reduced to the PHP formula on $n-1$ pigeons. Repeating this process $n-1$ times results in a refutation.

B. Tseitin formulas for expander graphs

Given a simple, undirected graph with an odd number of black vertices and all others colored white, Tseitin formulas ask whether there exists a subset of the edges S such that every black vertex has odd degree in S and every white vertex has even degree in S. This is not possible by the handshake lemma: the sum of all vertex degrees in S is even, since each edge is counted twice, but the sum of black edges (odd) and white edges (even) must be odd. In the formula, every edge e receives a variable that encodes whether $e \in S$. Figure 1 illustrates the Tseitin constraints for a small graph. Tseitin formulas of expander graphs have exponentially-large resolution proofs [19] and polynomial-sized PR proofs [20].

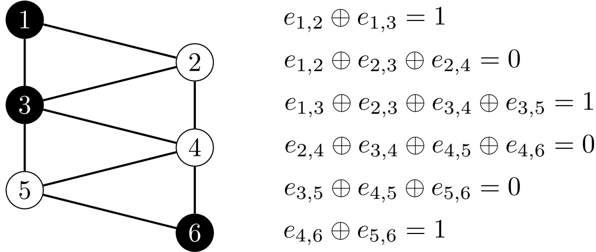


Fig. 1. The Tseitin constraints for a small graph, ordered by vertex. If $e_{i,j} = \top$, then $e_{i,j}$ is in the edge subset S. The symbol $\oplus$ denotes XOR.

SR does better: any Tseitin formula has an SR proof linear in the number of edges using the following derivation. First, we may remove any vertex v incident to only a single edge e because if v is white, then e cannot be in S, so removing them both does not affect any other constraint; and if v is black, then e must be in S, so we may remove them both as long as we flip the color of the neighbor of v, since removing e will change the degree-parity of the neighbor. We keep removing degree-one vertices until a conflict (i.e., a black vertex with no edges) or until every vertex in the graph has degree at least two. Such a graph must have a cycle U. For any satisfying S, we may swap the membership of every edge $e \in U$ in S and get a new satisfying edge set, as doing so maintains the S-degree-parity of every vertex in U. As a result, we may pick an arbitrary edge $e \in U$ and fix $e \notin S$. Repeating this process will eventually result in a conflict.

We illustrate this process with the graph in Figure 1. Consider the cycle $\{e_{1,2}, e_{1,3}, e_{2,3}\}$. We can learn that the unit clause $\overline{e}_{1,2}$ is SR with the witness $\sigma = \{e_{1,2} \mapsto \bot,\ e_{1,3} \mapsto \overline{e}_{1,3},\ e_{2,3} \mapsto \overline{e}_{2,3}\}$. Afterwards, vertex v_1 is adjacent to only the edge $e_{1,3}$. Since v_1 is colored black, $e_{1,3}$ must be in our edge set S, so we may remove v_1 and $e_{1,3}$ to make the graph smaller, as long as we swap the color of v_3 from black to white.

C. Ramsey numbers

Ramsey number $R(k, \ell)$ is the smallest n such that every 2-coloring of the edges of the fully-connected graph on n vertices with the colors red and blue has either a red k-clique or a blue ℓ-clique. The encoding of $R(k, \ell)$ is straightforward: boolean variables $\{e_{i,j}\}_{1 \le i < j \le n}$ represent the color of each edge, where $e_{i,j} = \top$ means the edge is blue, and for each k-clique and ℓ-clique, there is a clause stating that at least one of its edges is blue or red, respectively. An unsatisfiable formula for any n, k, and ℓ proves that $R(k, \ell) \le n$.

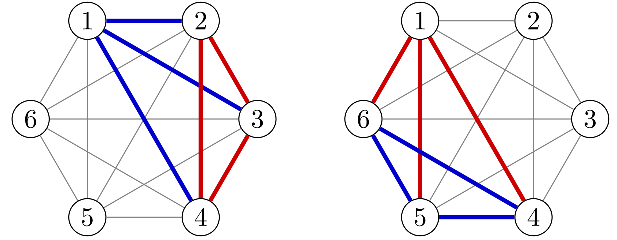


Fig. 2. A short proof of $R(3, 3) \le 6$. After adding clauses that sort the edges of vertex v_1 by color (blue edges first), fixing the edge $e_{1,4}$ to either blue or red forces a red or blue 3-clique, respectively, via unit propagation.

Short SR proofs for small Ramsey numbers can be constructed by sorting edges by color. For example, we can assume that the blue edges for vertex v_1 come first, represented by the clauses $e_{1,j} \vee \overline{e}_{1,j+1}$ for $1 < j < n$. Figure 2 illustrates the refutation derived by adding these four clauses to the formula for $R(3, 3) \le 6$. These binary clauses are SR. For instance, symmetry-breaking clause $e_{1,2} \vee \overline{e}_{1,3}$ has witness $\sigma = \{e_{1,2} \mapsto \top,\ e_{1,3} \mapsto \bot,\ e_{2,4} \mapsto e_{3,4},\ e_{3,4} \mapsto e_{2,4},\ e_{2,5} \mapsto e_{3,5},\ e_{3,5} \mapsto e_{2,5},\ e_{2,6} \mapsto e_{3,6},\ e_{3,6} \mapsto e_{2,6}\}$.

Ramsey number $R(4, 4) = 18$. A resolution proof of $R(4, 4) \le 18$ requires around a billion resolution steps. In contrast, the SR proof uses only 38 clause addition steps. See Section X for the argument.

IV. SR Proof Checking

In this section, we discuss the SR proof checking algorithm and why it checks redundancy, as well as two performance bottlenecks that are addressed in our verification.

The algorithm (Algorithm 1) is divided into two phases. The first phase (Lines 1–4) determines whether $F \vdash_1 C$, which would imply that $F \vDash C$, and thus that C is redundant. By definition, this amounts to showing that $F \wedge \neg C \vdash_1 \bot$. We

start with the partial assignment $\tau := \neg C$ (Line 1) and then try to find a UP refutation for F (Lines 2–4).

Algorithm 1: Validating whether a clause is SR

Input: CNF F, clause C, and witness σ satisfying C
Output: "Yes" if C is SR for F, "No" otherwise.

1 Set $\tau \leftarrow \neg C$
2 **while** there is a $D \in F_{|\tau}$ with $|D| \leq 1$ **do** } UP
3   **if** $|D| = 0$ (i.e., $D = \bot$) **then return** Yes
4   **else** $\tau \leftarrow \tau \cup D$
5 **if** $C = \bot$ **then return** No
6 **for** $D \in F$ **do**
7   **if** $\sigma \models D$ or D is not reduced by σ **then continue**
8   **if** $\tau \models D_{|\sigma}$ **then continue**
9     $\tau' \leftarrow \tau \cup \neg D_{|\sigma}$
10   **while** there is an $E \in F_{|\tau'}$ with $|E| \leq 1$ **do** } UP
11   **if** $|E| = 0$ (i.e., $E = \bot$) **then**
12   **continue** to the next iteration of Line 6
13   **else** $\tau' \leftarrow \tau' \cup E$
14   **return** No
15 **return** Yes

If no UP refutation is found, then τ stores the unit clauses found by UP on F, and we proceed to the second phase, the SR check (Lines 5–15). The empty clause $\bot$ cannot be SR, so we stop if $C = \bot$ (Line 5). Otherwise, we check the SR property. In our proof checking tools, we assume that the witness σ satisfies C, so it suffices to show that $F \wedge \neg C \vdash_1 D_{|\sigma}$ for each reduced clause $D_{|\sigma} \in F_{|\sigma}$.

The actual SR check looks slightly different. Rather than iterate across every $D_{|\sigma}$, we instead iterate across every $D \in F$ (Line 6), but we skip some clauses (Line 7). If $\sigma \models D$, then it is not in $F_{|\sigma}$; and if σ does not reduce D, then since $D \in F$, $F \wedge \neg C \vdash_1 D$ has a trivial UP refutation. In both cases, we can skip D and go to the next iteration of the loop.

That leaves the reduced clauses $D_{|\sigma}$ to be checked. Since τ stores the unit clauses from UP on $F \wedge \neg C$, it suffices to show that $F \wedge \tau \vdash_1 D_{|\sigma}$. If $\tau \models D_{|\sigma}$ (Line 8), then we have a trivial UP refutation. Otherwise, we perform UP with the addition of $\neg D_{|\sigma}$ to τ, forming τ' (Lines 9–13). If UP fails to find a refutation, then we cannot prove that C is SR (Line 14).

There are two potential computational bottlenecks in this algorithm. The first is performing UP. In the worst case, we must reduce every clause in F under τ when looking for unit and empty clauses. Data structures called *watch pointers* [21] are commonly used to efficiently implement this search process. Yet even with watch pointers, a significant amount of SR proof checking time is spent performing UP. One way of making UP more efficient is to be told the series of clauses in F that become unit or empty. And indeed, the hints in the hinted SR proof format are precisely these clauses.

The second bottleneck is creating τ' on Line 9. If τ contains many unit clauses, then we want to avoid copying them into a new τ' object for each loop. One idea is to keep a record of the unit clauses encountered by UP on Lines 10–13, and then undo their effects on τ' afterwards to restore τ. However, this would take time linear in the number of unit clauses. Instead, we want a data structure that can do this in constant time. Such a data structure exists and is commonly used in SAT proof checking tools. We implement it in our SR proof checkers, and we describe how we formalized it in Section VII.

$$
\begin{aligned}
\text{proof} &::= [\text{line}] \\
\text{line} &::= \text{id}, (\text{add} \mid \text{delete}), \texttt{0}, \texttt{\textbackslash n} \\
\text{add} &::= \text{clause}, \langle \text{witness} \rangle, \texttt{0}, [\text{id}], [-\text{id}, [\text{id}]] \\
\text{witness} &::= p : \text{lit}, [\text{lit}], \langle \boldsymbol{p}, [(\textbf{var}, \textbf{lit})] \rangle \\
\text{delete} &::= \texttt{d}, [\text{id}] \qquad \text{clause} ::= [\text{lit}] \\
\text{id}, \text{var} &::= \mathbb{N} \setminus \{0\} \qquad \text{lit} ::= \mathbb{Z} \setminus \{0\}
\end{aligned}
$$

Fig. 3. The formal grammar for the LSR proof format. A list with 0 or more items is written as $[\cdot]$, and an optional object is written as $\langle \cdot \rangle$. Additions to the LSR format from LPR are bolded.

V. The SR Proof Formats

We introduce the proof formats for SR without and with hints, which we call DSR and LSR, respectively.[4] Our formats extend the DPR/LPR proof formats [13], which themselves extend DRAT/LRAT [22, 23]. Our formats are backwards compatible, meaning that any RAT or PR proof is also a valid SR proof.

As with RAT and PR, SR proofs comprise *addition* and *deletion* lines. Each addition line contains a clause C claimed to be redundant. If C is nonempty, then the line may also contain a substitution witness σ satisfying C. If no witness is provided, then σ is defined as $\sigma(p) = \top$, where p is the first literal of C (called the *pivot*), and is the identity everywhere else. That way, σ satisfies C.

In the DSR format, addition lines contain only C and σ, and the checker must run Algorithm 1. But in the LSR format, addition lines also contain *hints*. In LSR, each clause is given a unique numerical identifier starting at 1. Each hint is the ID of a clause that becomes unit or conflict under UP. A list of UP refutation hints are provided for each $D_{|\sigma} \in F_{|\sigma}$. While hints do increase the size of the proof, they generally reduce proof checking times by making UP much more efficient.

Deletion lines specify clauses in F to delete. Deletion does not maintain equisatisfiability, but if the formula after deletion from F is unsatisfiable, then so is F. Deletion speeds up proof checking by reducing the required number of $D_{|\sigma}$ UP refutations to find. It can also increase the set of SR clauses, as it can remove $D_{|\sigma}$ clauses that fail the SR check.

The formal grammar for the LSR format is shown in Figure 3. Figure 4 shows a DIMACS CNF formula and its corresponding LSR proof for PHP with $n = 4$. The parts of an LSR addition line are color-coded. Their order is as follows:

1) A (positive, unique) numerical clause ID. Later LSR lines refer to the added clause by this ID.
2) The (literals of the) candidate redundant clause C.
3) An optional substitution witness σ beginning with p, the first literal of C (the pivot). The witness has two parts,

[4] The names are acronyms. DSR stands for "deletion SR," while LSR stands for "linear SR," where "linear" means that the hints included in the format enable proof checking to take time linear in the size of the proof.

CNF format

```
p cnf 12 22
 1  2  3  0
 4  5  6  0
 7  8  9  0
    ...
```

LSR format

```
23 -10 -10 7 -10 8 11 11 8 9 12 12 9 0  7  9 10 -4 3 -6 -8 -12 13 ... 0
24  -7  -7 4  -7 5  8  8 5 6  9  9 6 0 23  6  8 -3 2 -5 -9 -11 12 ... 0
25  -4  -4 1  -4 2  5  5 2 3  6  6 3 0 23 24  5 -2 1 -6 -7 -11 11 ... 0
26 -11                               0 24 25 15 16 2  3 20            0
27                                   0 23 24 25 26 4 21 22    2  3 14 0
```

Fig. 4. A DIMACS CNF formula (left) and its LSR proof (right). Each line in the CNF is a new clause. LSR addition lines comprise a clause ID (pink), the literals of the clause (green), and an optional substitution witness containing literals mapped to true (orange) and variables mapped to other literals (blue), with another appearance of the pivot literal (black) acting as a separator. The line concludes with UP hints (purple) and reduced-clause UP hints (red). Each reduced clause $D_{|\sigma}$ is identified with a negative ID, and the positive hints that follow are the UP refutation for $D_{|\sigma}$.

separated by another appearance of p:[5] first, a list of literals ℓ that σ maps to true (including p); and second, a list of variable-literal pairs (v, ℓ) setting $\sigma(v) := \ell$. All other variables v are mapped to themselves, i.e., $\sigma(v) := v$.

4) A separating `0`, marking where the hints begin.
5) A list of hints, not necessarily deriving UP refutation, guiding Line 2 of Algorithm 1.
6) A list of hints deriving UP refutation for each reduced clause, guiding Line 10. The reduced clause is identified by the negative of its ID, and the UP hints follow.

If each of the line ID, the hints, and the ending `0` are removed, then the addition line becomes a DSR addition line.

Currently, our checkers assume that the witness σ satisfies the candidate clause C (which is the case for all of the SR proofs that appear in this paper). However, the DSR and LSR formats can also express proofs where σ causes $C_{|\sigma}$ to be a tautology: the proof can simply map the pivot p to itself or to any other literal in the substitution portion. We plan to support this general case in the future.

LSR deletion lines start with a line ID, followed by a `d` and the IDs of clauses to be deleted. Historically, the line ID matches the ID of the most-recently-added clause so that unordered proofs can be sorted. But most modern proof-logging SAT solvers emit ordered proofs, so the line ID is ignored.

DSR does not use clause IDs, so deletion lines specify the (literals of the) clause to be deleted directly. Thus, DSR deletion lines only delete a single clause at a time.

The only addition to the LSR format from LPR is the second part of the substitution. If no variable-literal mappings are provided, then the substitution is a partial truth assignment, which is the type of witness used for PR proofs. It is in this way that the SR formats are backwards compatible.

VI. SR Proof Checking Tools

Absent a dedicated SR proof checker, DSR proofs can be checked by converting them into DRAT and then using conventional DRAT/LRAT checkers. We implemented such a conversion algorithm by extending one that converts DPR proofs into DRAT [20]. The most important change to the algorithm is that it uses *multiple* auxiliary variables to convert a single SR clause addition step into a sequence of DRAT proof steps. More specifically, it introduces a fresh variable (i.e., one not appearing in either the formula or the proof) for each SR addition step and several fresh variables for each variable-literal mapping in the substitution.

Our implementation is open-source and can be found at:

`https://github.com/marijnheule/sr2drat`.

We also developed SR proof checking tools. Following the tradition of drat-trim[6] [22] and dpr-trim[7] [13], we present dsr-trim, a tool for adding hints to DSR proofs to create LSR proofs. The source code is available at:

`https://github.com/ccodel/dsr-trim`.

At the moment, dsr-trim can only perform *forwards checking*, which means that it checks DSR proofs from start to finish and adds hints as it goes. In contrast, drat-trim and dpr-trim both additionally implement *backwards checking*, meaning that they read the entire DSR proof into memory first and then work backwards from the derivation of the empty clause, ignoring unreferenced proof lines. In practice, backwards checking can significantly reduce the size of proofs. Adding backwards checking to dsr-trim is ongoing work.

In addition to dsr-trim, we implemented lsr-check, an unverified LSR proof checker. Despite SR being more complicated than PR/DRAT, lsr-check performs comparably to, and often better than, its sister checkers lrat-check and lpr-check.

Both dsr-trim and lsr-check are configured to parse and/or produce proofs in the ASCII format presented in Figure 4 and in a custom binary format that is faster to parse. Compressed proofs tend to be 50-60% smaller in file size. The `compress`/`decompress` tools included with dsr-trim translate DSR and LSR proofs into and out of this binary format.

VII. The Verified LSR Checker

In this section, we discuss our implementation and verification of an LSR proof checker in the Lean 4 interactive theorem prover [12]. Our checker is open-source and can be found at:

`https://github.com/FormalSAT/trestle`.

[5] The choice to use p as a separator is a historical one. In PR, the pivot appears twice: once in the clause, and once to indicate when the partial assignment begins. For SR, we need a way to indicate when variable-literal mappings occur. Since `0` is a reserved symbol, we use p once again.

[6] `https://github.com/marijnheule/drat-trim`.

[7] `https://github.com/marijnheule/dpr-trim`.

The core of the checker is a function called `checkLine` that runs Algorithm 1 with UP hints (i.e., LSR). Ideally, we would expect its correctness theorem to look like this:

```
checkLine F C line = ok → eqsat F (F ∧ C).
```

And indeed, this is equivalent to what we proved in Lean. But to make our checker efficient, we implemented data structures that cause the correctness theorem to look more complicated:

```
theorem checkLine_ok : models R F C →
  checkLine ⟨R, τ, σ⟩ line = .ok S →
  eqsat F (F ∧ C) := by ...
```

There are three main differences between the correctness theorems. The first is the use of a functional programming idiom similar to the state monad: `checkLine` takes a state triple of a CNF data structure `R`, a partial assignment τ, and a substitution σ, along with the LSR `line`, and it returns an updated $\langle \mathtt{R}, \tau, \sigma \rangle$ state triple `S` and its yes/no result. By passing along `R`, τ, and σ, Lean can allocate the memory for these structures once, rather than at the start of each proof line, which makes the checker more efficient.

The second difference is that the literals of the candidate clause C are stored in `R` rather than in a separate object. This eliminates the need of writing the literals twice: once during parsing, and once when copying them into `R` after checking that C is SR. According to a CPU profiler, this sleight of hand gives a 10% speedup on our longest-running benchmark.

The third and greatest difference is how we implement `R`, τ, and σ. We implement CNF formulas with a data structure we call `RangeArray`. In the correctness theorem, we assume that `R models` formula F and candidate clause C. We implement τ and σ with data structures we call `PPA` and `PS`, standing for "persistent partial assignment" and "persistent substitution." These three data structures enable our checker to efficiently implement UP, but at the cost of a much more complicated proof of correctness.

In total, our verified checker and its supporting theorems and data structures comprise 8k LoC, and the verification took 4 person months. Much of that time was spent adjusting how the checker iterates across data structures in order to make the compiled Lean code performant. For example, implementing reduction (i.e., $C_{|\sigma}$) with an API-breaking, tail-recursive function, as opposed to a `foldlM` in the `Except` monad, gave an immediate speedup of 60% on our longest-running benchmark. We hope that future versions of the Lean compiler will be less picky about generating performant code.

In the rest of this section, we discuss the `RangeArray` and `PPA` data structures, as they represent the most technical portions of our verification. These data structures use techniques common in other SAT solving tools, including dsr-trim.

A. RangeArray

Given a type of boolean literals `ILit`, a straightforward type for CNF formulas is `List (List ILit)`. We use this datatype in our SAT theory, since Lean provides good support for lists. However, this datatype suffers from two drawbacks. The practical drawback is that a nested list unnecessarily fragments the memory for clauses across separately-allocated blocks, leading to additional memory overhead and reduced cache locality. The other drawback is that nested lists cannot easily implement clause deletion. Recall that SR proofs may delete clauses from the formula. Because LSR hints are static IDs, we cannot simply remove deleted clauses from the nested list, as this would shift the indexes of the remaining clauses.

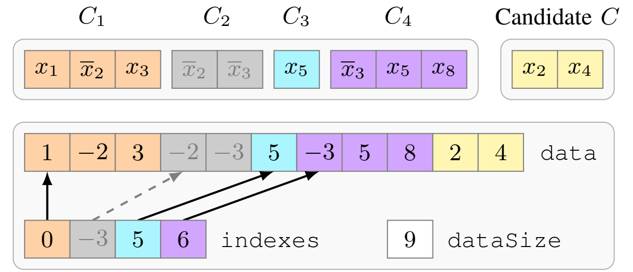


Fig. 5. An example of a `RangeArray` modeling a formula with four clauses and a candidate clause C. All literals are stored in a single array `data`, and clauses are defined based on index "pointers" in `indexes`. The candidate clause is implicitly defined as being the additional literals in `data` beyond the index stored in `dataSize`. The `RangeArray` deletes clauses by marking their index as negative. In the example, clause C_2 is deleted.

One hack is to replace the deleted clause with the binary clause $C_\top := x_1 \vee \overline{x}_1$, since a tautology behaves like $\top$ in our SAT theory. But then the proof checker would be hard-coded to check for deletion by comparing clauses to $C_\top$, which strikes us as inelegant and non-modular. Another solution is to use option types.[8] However, options (in our opinion) clutter up code and proofs, and they add another layer of pointer indirection in compiled code, which leads to slower runtimes.

Instead, we implemented a common data structure for CNF formulas we call `RangeArray`. Figure 5 shows an example. `RangeArray`s flatten the nested list datatype so that all literals lie in a single array `data`, and clauses are defined using index "pointers" stored in a second array `indexes`. Intuitively, the ith clause starts at position `indexes[i]` in `data`, and it has size `indexes[i+1] - indexes[i]`. However, deletion is implemented by setting an index p to $-p$, so calculations involving indexes use their absolute value.

The `RangeArray` has two main benefits. The first benefit is that all literals lie in the same array, so iteration across an entire formula has increased cache locality. The second benefit is the ability to store the candidate clause C in the `RangeArray` during proof checking. We do so by adding the literals of C to `data` without assigning C an index. To differentiate between formula literals and candidate clause literals, we store the total number of formula literals in a variable `dataSize`. Thus, the literals of C lie between `dataSize` and the actual size of `data`. The `commit` operation adds C to the formula by assigning C an index and increasing `dataSize` by $|C|$.

We relate `RangeArray` to our model for CNF formulas and clauses with the `models` predicate. Given a formula `F` and a candidate clause `C`, `models R F C` means that

[8] Values of type `Option V` are either `none` or `some v`, where `v:V`.

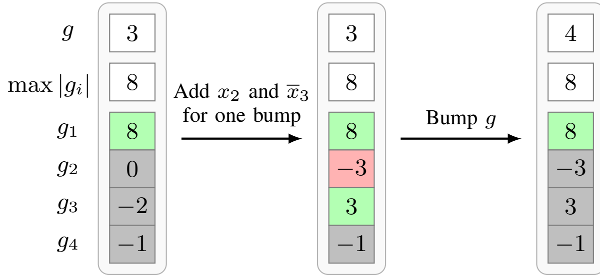


Fig. 6. An example of the PPA data structure in use. If $|g_i| \geq g$, then the sign of g_i determines if $\tau(x_i)$ is true (green) or false (red). Otherwise, $\tau(x_i)$ is undefined (gray). On the left, $\tau := x_1$, and its truth value is set for 6 rounds of UP (6 bumps). In the middle, the unit clauses x_2 and $\overline{x}_3$ are added to τ. On the right, those two truth values are cleared with a bump in $O(1)$ time.

`R` agrees with `F` and `C` on every non-deleted clause, such that `R[indexes[i]+j] = F[i][j]` and `R[dataSize+i] = C[i]`. We prove that `commit` and other operations preserve the `models` predicate in the appropriate way.

B. *Persistent partial assignments*

Partial truth assignments can be implemented with an `Array (Option Bool)`. For any variable v, if `A[v] = none`, then $\tau(v)$ is undefined, and otherwise `A[v] = some b` means that $\tau(v) = b$. However, this implementation would make it inefficient to run Lines 1 and 9 of Algorithm 1: an array of booleans would need to be cleared for each LSR line, and the array would need to be restored from τ' to τ after each reduced-clause check, of which there might be many. All of this copying and clearing would make proof checking intractable.

A common solution to this problem is to use a technique we call *generation bumping* (or *timestamping*), which enables $O(1)$ clearing of UP unit clauses. The idea is for the PPA to store a global *generation number* g and a generation number g_i for each boolean variable x_i. If $|g_i| \geq g$, then the sign of g_i determines if $\tau(x_i)$ is true (+) or false (−). Otherwise, $\tau(x_i)$ is undefined. Incrementing g, called *bumping*, clears the truth values of any x_i with $|g_i| = g$. Setting $g := \max |g_i| + 1$, called *clearing*, clears all truth values.

Proof checkers can use generation bumping because they know in advance the exact number of bumps any particular truth value should be set for. Unit clauses found during UP in the entailment phase (Lines 1–4) must persist in τ for all rounds of UP in the SR phase (Lines 10–13). Since each reduced clause UP refutation is marked in the LSR line, proof checkers can count the expected number of refutations r during parsing, and then set the generation number for unit clauses in τ to $|g_i| := g + r + 1$. In the SR phase, new unit clauses added to τ' have their generation numbers set to g so that they can be cleared afterwards with a single bump. Figure 6 illustrates how this works. Our verification of `checkLine` includes careful bookkeeping to ensure that the unit clauses in τ persist.

Our implementation of PS also uses generation bumping, except that an additional array stores what each variable is mapped to under the substitution.

VIII. Experimental Results

Our experimental results demonstrate the clear benefits of using a strong proof system. We highlight three main results: (1) that our verified LSR checker performs well against cake_lpr [13], a fast, verified LPR checker, (2) that SR proofs are smaller than their RAT counterparts, and (3) that our verified checker incurs reasonable overhead compared to lsr-check, our unverified LSR checker written in C.

Our benchmarks comprise five families of SR formulas: Ramsey instance $R(4,4) \leq 18$, Schur number five [24], a packing problem [1], and PHP and Tseitin formulas. We also include a similar set of five PR families, where instead of packing, Schur, and Ramsey, it has Mycielski [25], mutilated chessboard, and two-pigeons-per-hole (tph) formulas [26].

We ran our experiments on a 2022 M1 Mac Studio with 32 GB of memory and a clock speed of 3.2 GHz. To replicate our results, use the scripts found at:

`https://github.com/ccodel/sr-benchmarking`.

A. *Comparison to cake_lpr*

We first show that our verified Lean checker performs similarly to cake_lpr. Our experiments covered the PR proof families. Figure 7 summarizes our results.

For proofs that took longer than 1 second to verify, our checker took an average of 81.95 seconds, while cake_lpr took an average of 124.86 seconds. The geometric mean of the ratio of Lean / cake_lpr runtimes was 0.718.

For proofs that took less than 1 second to verify, our checker took proportionally longer than cake_lpr. For example, the geometric mean of the ratios of runtimes on these instances was 5.84. A CPU profiler revealed that 75% of the runtime on these instances was spent on Lean's initialization code that is run only once at the start of the program, and so this does not represent a bottleneck for larger proofs.

B. *Comparison of LSR and converted LRAT proofs*

Next, we show that SR proofs are smaller than their DRAT counterparts. For these experiments, we used sr2drat to translate the DSR proofs into DRAT, which were then translated into LRAT by drat-trim. (We convert from SR to DRAT instead of SR to PR because we do not know of a way to use the PR rule when converting from SR.)

The SR proofs are indeed smaller, both in terms of file size and the number of proof lines. Figure 8 shows our results. On average, an LSR proof was 6.2 MB and had 13.2K proof lines, while the translated LRAT proof was 41.2 MB and had 1.03M proof lines. The geometric means of ratios of LSR to LRAT for file size and line count were 0.085 and 0.004, respectively.

Unsurprisingly, the smaller SR proofs were faster to check. Figure 9 shows the runtimes for our Lean checker on the LSR proofs compared to cake_lpr on the converted LRAT proofs. On average, our checker took 1.06 seconds, while cake_lpr took 4.00 seconds. For proofs that took longer than 1 second to check, the geometric mean of the ratio of Lean / cake_lpr proof checking times was 0.255.

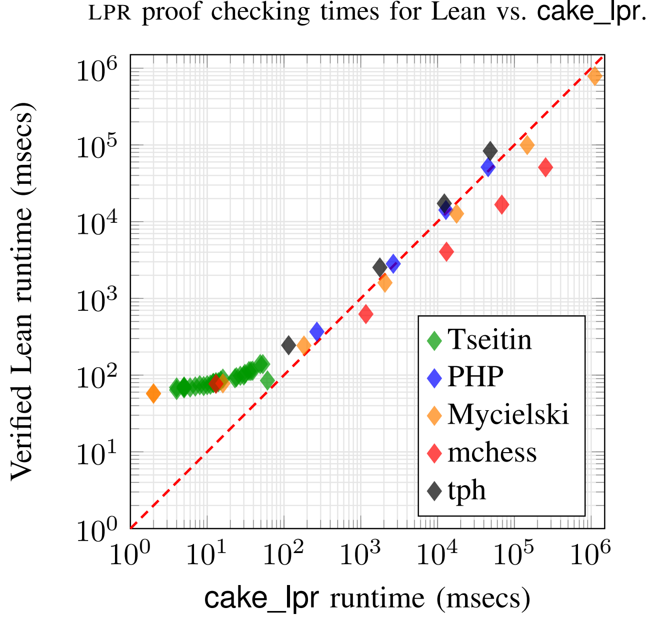


Fig. 7. Comparison of proof checking times for our Lean checker and cake_lpr on the PR proof families. Points below the red $y = x$ line indicate that our checker was faster than cake_lpr.

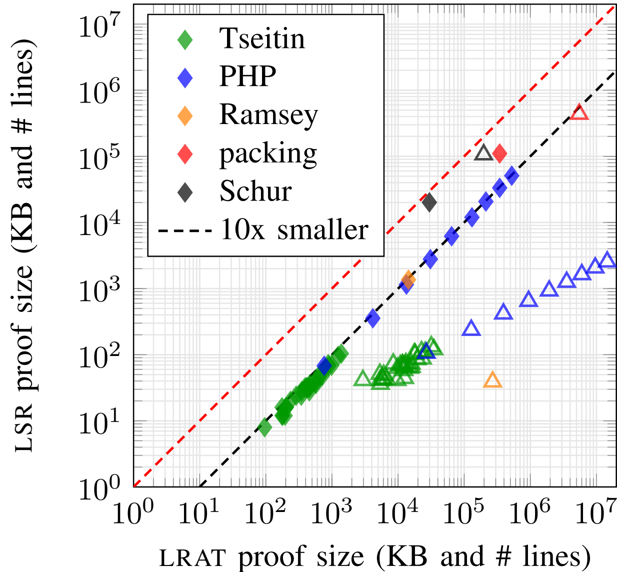


Fig. 8. Proof sizes in terms of KB (♦) and number of proof lines (△) for LSR proofs and their LRAT conversions via sr2drat and drat-trim. Points below the red $y = x$ line indicate that the SR proofs were smaller.

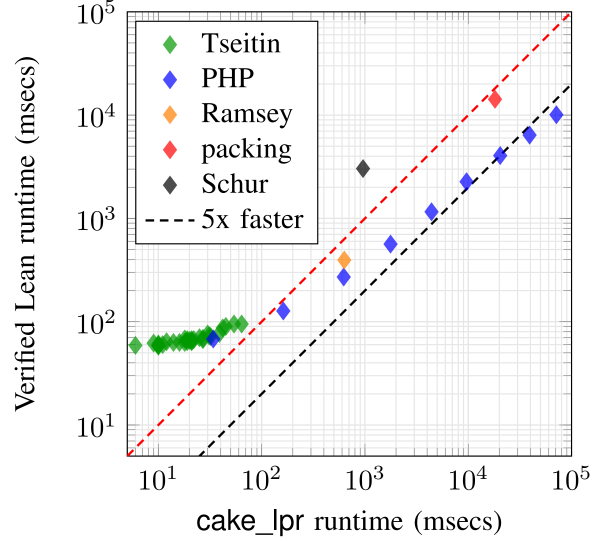


Fig. 9. Comparison of proof checking times for our Lean checker on the LSR proofs and for cake_lpr on the LRAT conversions. Points below the red $y = x$ line indicate that our checker was faster than cake_lpr.

C. *Comparison of verified and unverified checking*

Finally, we report that the added constant factors of our verified proof checker are not excessive. On the LSR proofs, our Lean checker is about 10x slower than our unverified checker lsr-check. Figure 10 summarizes our results. On average, our Lean checker took 1.06 seconds, while lsr-check took 0.10 seconds. For proofs that took longer than 1 second to check, the geometric mean of the ratios of their runtimes was 9.936.

IX. Conclusion and Future Work

In this paper, we presented our tools for verified SR proof checking. SR admits short proofs for many problems, and our experimental results show the clear advantages of using SR over weaker proof systems such as RAT and PR. While no modern SAT solver supports SR reasoning yet, we hope that our tools—including our verified SR proof checker—will support the future development of SR tooling for SAT solving.

There are several avenues for future work. One such avenue is improving dsr-trim and our verified Lean checker. We plan to add backwards proof checking to dsr-trim. In addition, the Lean checker can be made more efficient by minimizing the number of clauses it checks. We have seen that if $D_{|\sigma} = D$, then it can be skipped during the SR phase. By storing the first and last clause containing each literal, we can compute the range of clauses reduced by σ such that clauses outside of this range are not reduced, and thus do not need to be checked. Our lsr-check tool implements this technique. While it only gives modest speedups, we hope that implementing it in Lean will improve the Lean checker's runtime.

Another avenue of future work is in automatically generating symmetry-breaking SR proofs. CDCL SAT solvers tend to struggle on problems with a high degree of symmetry. Adding

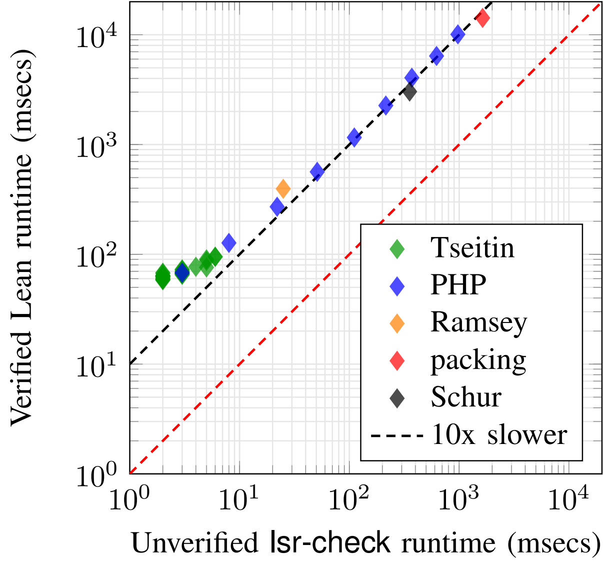


Fig. 10. Comparison of proof checking times for our Lean checker against our unverified checker lsr-check on the SR proof families. Marks on the black dashed line indicate that the Lean checker was 10x slower than lsr-check.

symmetry-breaking clauses via SR proof steps can be a trusted way to improve solver runtimes.

X. Short SR Proof of $R(4,4) \leq 18$

We constructed a short SR proof of $R(4,4) \leq 18$ that consists of only 38 clauses. The proof consists of four phases. In the first phase, we assume WLOG that vertex v_1 is connected to at least nine blue edges and that these blue edges connect v_1 to the vertices $v_2, \dots, v_{10}$. To show this, we sort the edges adjacent to vertex v_1 so that the blue edges appear first. The clauses that express the sorting are of the form $e_{1,i} \lor \overline{e}_{1,i+1}$ with $1 < i < n$. The SR witnesses for these clauses are a permutation of the vertices. At this point, we can still exchange the two colors. We use this to fix the edge $e_{1,10}$ to blue. The result is shown in Figure 11. This phase consists of 17 clause addition steps.

In the second phase, we assume WLOG that v_2 is connected to at least five red edges and that these red edges connect v_2 to the vertices $v_3, \dots, v_7$. In the proof, we sort the edges adjacent to vertex v_2 with the red edges appearing before the blue edges. The clauses that express the sorting are of the form $\overline{e}_{2,i} \lor e_{2,i+1}$ with $2 < i < 11$. If we assign edge $e_{2,7}$ to blue, then unit propagation will result in a conflict as shown in Figure 12. Thus, we may fix $e_{2,7}$, $e_{2,6}$, $e_{2,5}$, $e_{2,4}$, and $e_{2,3}$ to red. This phase consists of 9 clause addition steps.

In the third phase, observe that there cannot be a red or blue 3-clique among the vertices v_3, v_4, v_5, v_6, and v_7, because all of them are connected to v_1 with a blue edge and all of them are connected to v_2 with a red edge. There is a unique red-blue assignment (modulo symmetry) that avoids a red or blue 3-clique among five vertices: a blue 5-cycle and a red 5-cycle. We fix this assignment after sorting the edge for vertex v_3

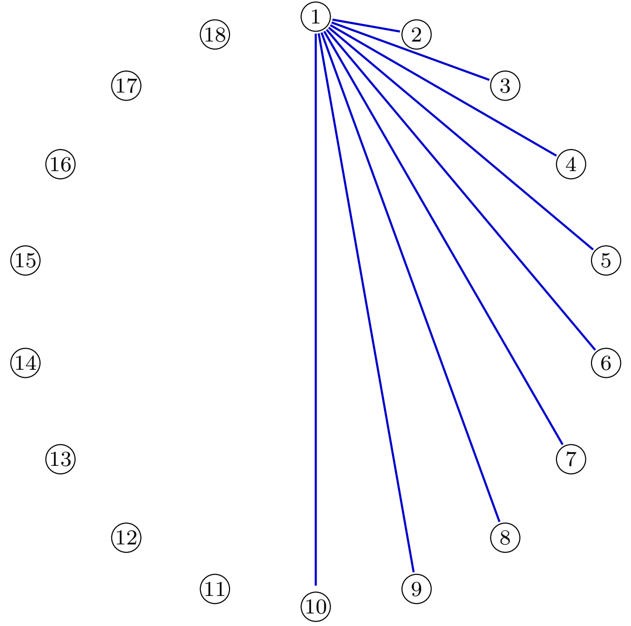


Fig. 11. First phase: sort the edges for vertex v_1 so the blue edges come first, and fix the edge $e_{1,10}$ to blue.

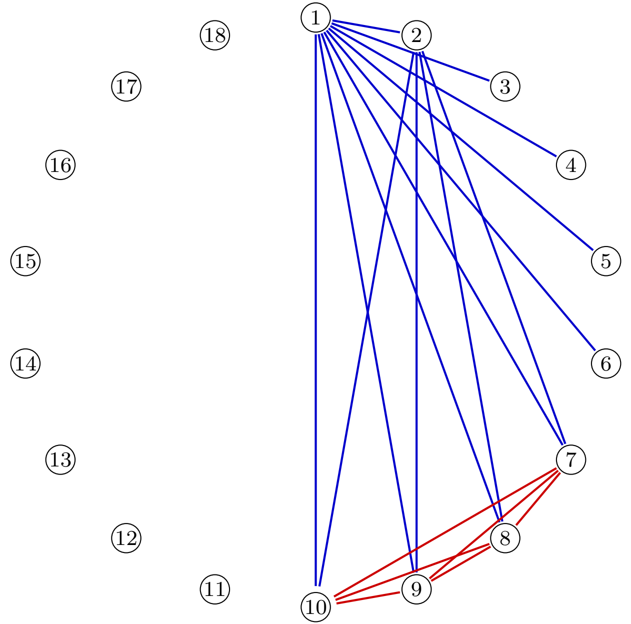


Fig. 12. Second phase: sort the edges for vertex v_2 so the red edges come first, and fix $e_{2,7}$ to blue. This results in a conflict via unit propagation: a red 4-clique v_7, v_8, v_9, v_{10}. As a consequence we can fix $e_{2,7}$ to red.

(blue edges first). The result is shown in Figure 13. This phase consists of 7 clause addition steps.

In the fourth and final phase, we first determine that the edges $e_{2,8}$, $e_{2,9}$, and $e_{2,10}$ must be blue. This is achieved with two clauses. The first clause assumes that $e_{2,8}$ and $e_{3,8}$ are red. This results in a conflict by unit propagation. Afterwards we only assume that $e_{2,8}$ is red. This now results in a conflict by unit propagation as well, as $e_{3,8}$ is forced to be blue. The

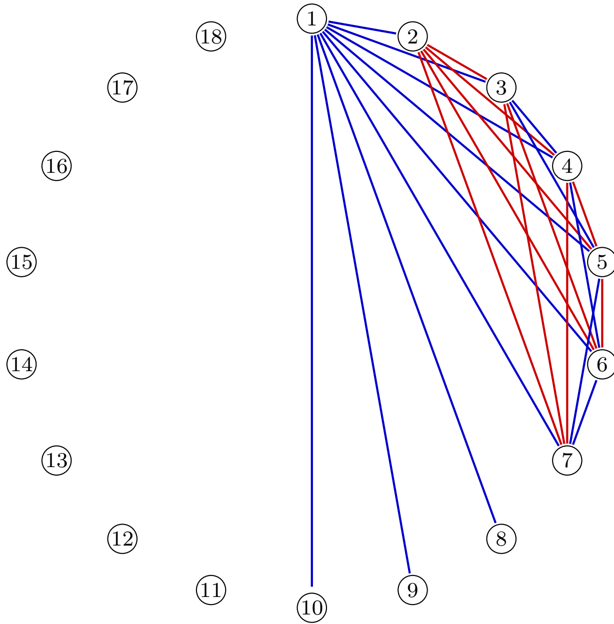


Fig. 13. Third phase: There cannot be a 3-clique in red nor a 3-clique in blue among v_3, v_4, v_5, v_6, and v_7. There is a unique assignment that achieve this. We sort the edges among these vertices and fix that unique assignment.

failed assumption allows us to fix $e_{2,8}$ to blue. See Figure 14.

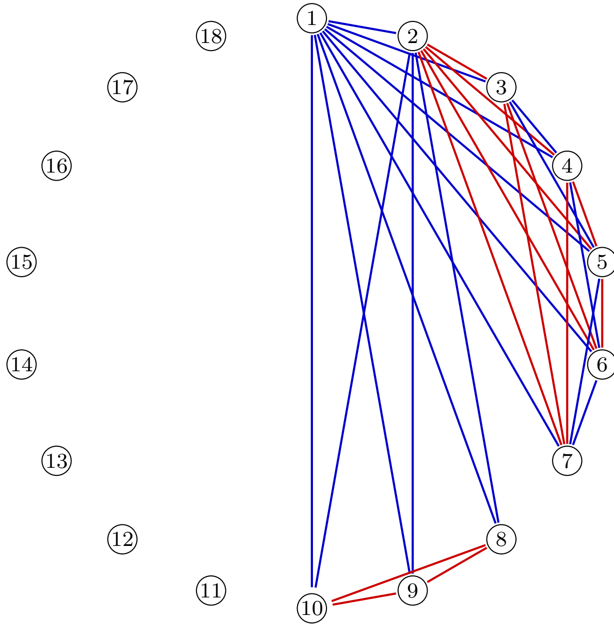


Fig. 14. Fourth phase: determine that $e_{2,8}$, $e_{2,9}$, and $e_{2,10}$ must be blue.

Afterwards, the edges $e_{3,8}$, $e_{3,9}$, and $e_{3,10}$ are sorted (red first). This step is allowed because vertices v_8, v_9, and v_{10} are still interchangeable at this point. Assuming that $e_{3,9}$ is blue results in a conflict by UP, so $e_{3,9}$ (and thus $e_{3,8}$) needs to be red. The final refutation comes from the observation that assuming either $e_{4,8}$ to red or blue results in a conflict by UP. This phase consists of 7 clause addition steps.

References

[1] B. Subercaseaux and M. J. H. Heule, "The packing chromatic number of the infinite square grid is 15," in *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pp. 389–406, 2023.

[2] M. J. H. Heule and M. Scheucher, "Happy ending: An empty hexagon in every set of 30 points," 2024.

[3] Z. Li, C. Bright, and V. Ganesh, "A SAT solver and computer algebra attack on the minimum kochen-specker problem," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 23559–23560, Mar. 2024.

[4] N. Rungta, "A billion SMT queries a day (invited paper)," in *Computer Aided Verification*, pp. 3–18, 2022.

[5] M. Järvisalo, M. J. H. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning*, pp. 355–370, 2012.

[6] M. J. H. Heule, B. Kiesl, and A. Biere, "Strong extension-free proof systems," *Journal of Automated Reasoning*, vol. 64, no. 3, pp. 533–554, 2020.

[7] *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B, 2023.

[8] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, "Preprocessing of propagation redundant clauses," *Journal of Automated Reasoning*, vol. 67, Sep 2023.

[9] S. Buss and N. Thapen, "DRAT and propagation redundancy proofs without new variables," *Logical Methods in Computer Science*, vol. Volume 17, Issue 2, Apr 2021.

[10] S. Gocht and J. Nordström, "Certifying parity reasoning efficiently using pseudo-boolean proofs," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 3768–3777, May 2021.

[11] A. Rebola-Pardo, "Even Shorter Proofs Without New Variables," in *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)* (M. Mahajan and F. Slivovsky, eds.), vol. 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 22:1–22:20, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[12] L. d. Moura and S. Ullrich, "The Lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28*, pp. 625–635, 2021.

[13] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "Verified propagation redundancy and compositional UNSAT checking in CakeML," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 2, pp. 167–184, 2023.

[14] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Principles of Programming Languages (POPL)*, pp. 179–191, Jan 2014.

[15] M. J. H. Heule, *Proofs of Unsatisfiability*, ch. 15, pp. 635–668. Frontiers in Artificial Intelligence and Applications, 2 ed., 2021.

[16] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere,

"PRuning through satisfaction," in *Hardware and Software: Verification and Testing*, pp. 179–194, 2017.

[17] A. Haken, "The intractability of resolution," *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[18] S. A. Cook, "A short proof of the pigeon hole principle using extended resolution," *SIGACT News*, vol. 8, pp. 28–32, oct 1976.

[19] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, pp. 466–483. 1983.

[20] M. J. H. Heule and A. Biere, "What a difference a variable makes," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 75–92, 2018.

[21] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference*, pp. 530–535, 2001.

[22] N. Wetzler, M. J. H. Heule, and W. A. Hunt, "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Theory and Applications of Satisfiability Testing – SAT 2014*, pp. 422–429, 2014.

[23] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction – CADE 26*, pp. 220–236, 2017.

[24] M. J. H. Heule, "Schur number five," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI'18, 2018.

[25] E. Yolcu, X. Wu, and M. J. H. Heule, "Mycielski graphs and PR proofs," in *Proceedings of the 23rd Conference on Theory and Applications of Satisfiability Testing (SAT)*, no. 12178 in Lecture Notes in Computer Science, pp. 201–217, 2020.

[26] A. Biere, "Two pigeons per hole problem," in *Proc. of SAT Competition 2013: Solver and Benchmark Descriptions*, p. 103, 2013.