How to Discover Short, Shorter, and the Shortest Proofs of Unsatisfiability: A Branch-and-Bound Approach for Resolution Proof Length Minimization

KONSTANTIN SIDOROV*, KOOS VAN DER LINDEN, GONÇALO HOMEM DE ALMEIDA CORREIA, MATHIJS DE WEERDT, and EMIR DEMIROVIĆ, Delft University of Technology, The Netherlands

Background: Modern SAT solvers are indispensable reasoning tools that are widely applied in verification, planning, and combinatorial design. Beyond answering satisfiability, they also produce unsatisfiability proofs, which are valuable for certification and analysis. Since there are no guarantees that their length is close to optimal, however, those proofs are insufficient to make claims about the gap between the solver proofs and the shortest possible proof. Understanding this gap is important for analyzing solver behavior, as it can be seen as a proxy for evaluating the room for improvement of the solver for the input formula in question.

Objectives: We study the problem of finding the shortest resolution proofs for unsatisfiable formulas. Our goal is to understand how far solver-generated proofs deviate from the optimum, and to develop algorithms that can discover proofs substantially shorter than the solver-produced proofs.

Methods: We propose a novel branch-and-bound algorithm for resolution proof length minimization. To facilitate it, we introduce a *layer list* representation that eliminates all symmetries from clause permutations, thereby improving upon an earlier SAT-encoding approach to proof length minimization. Further, we accelerate the search by integrating pruning techniques based on proof length lower bounds and clause subsumption.

Results: Our algorithm consistently reduces solver proof lengths by 25–50% for synthetic instances and by 15—50% for formulas from the SAT Competition editions from 2002 to 2025, as well as halves the proof lengths for half of the available formulas from planning problems. As an exact method, it solves twice as many minimally unsatisfiable instances as the state-of-the-art approach and is faster by orders of magnitude on the overlapping instances.

Conclusions: These results show that solver-generated proofs can often be substantially shortened—not by removing redundant claims, but by an entirely different chain of derivations—and that resolution proof length minimization is more tractable in practice than earlier approaches suggested. Our work highlights proof length minimization as a useful perspective for analyzing solver behavior and opens avenues for applying similar techniques to stronger proof systems.

JAIR Associate Editor: Sasha Rubin

JAIR Reference Format:
Konstantin Sidorov, Koos van der Linden, Gonçalo Homem de Almeida Correia, Mathijs de Weerdt, and Emir Demirović. 2026. How to Discover Short, Shorter, and the Shortest Proofs of Unsatisfiability: A Branch-and-Bound Approach for Resolution Proof Length Minimization. *Journal of Artificial Intelligence Research* 86, Article 54 (August 2026), 39 pages. DOI: 10.1613/jair.1.22128

*Corresponding author.

Authors' Contact Information: Konstantin Sidorov, ORCID: 0009-0009-0655-4200, k.sidorov@tudelft.nl; Koos van der Linden, ORCID: 0009-0001-4015-0594, j.g.m.vanderlinden@tudelft.nl; Gonçalo Homem de Almeida Correia, ORCID: 0000-0002-9785-3135, g.correia@tudelft.nl; Mathijs de Weerdt, ORCID: 0000-0002-0470-6241, m.m.deweerdt@tudelft.nl; Emir Demirović, ORCID: 0000-0003-1587-5582, e.demirovic@tudelft.nl, Delft University of Technology, Delft, The Netherlands.

DOI: 10.1613/jair.1.22128

1 Introduction

The last two decades have seen remarkable advances in propositional satisfiability (SAT), making SAT solvers indispensable across domains such as verification (Prasad et al. 2005), AI planning (Ernst et al. 1997), and combinatorial designs (Zhang 2021). Despite this success, the process of understanding, tuning, and improving solvers remains largely heuristic and empirical. Solver performance is often explained in terms of well-crafted engineering choices, while the underlying complexity of their behavior remains difficult to analyze in a principled way.

A promising perspective to explain solver performance comes from looking at unsatisfiability proofs (M. J. H. Heule 2021), which capture the reasoning steps made by a solver on unsatisfiable instances. These proofs can be viewed as execution traces: proofs that are short in the number of resolutions correspond to solver runs with few steps and, ideally, short running times. This naturally raises a fundamental question: how far are solver-generated proofs from the *shortest* possible ones?

Posing a question of finding the shortest unsatisfiability proof like this translates naturally as a combinatorial optimization problem. More precisely, the search space is now specified as the set of all valid proofs for a given formula, and the objective is to minimize the number of steps of a proof; one can stretch this analogy further by interpreting the optimal proof—or, in fact, any valid bound on the objective—as the theoretical limit on the proof length.

Addressing this question is more than an abstract curiosity, as it connects directly to understanding solver behavior. For example, in the mixed-integer programming (MIP) community, researchers have already begun studying *optimal* branch-and-bound trees (Dey et al. 2024), using them to derive insight into solver behavior—in particular, the impact of branching choices. The analogy to SAT is immediate: if optimal branch-and-bound trees illuminate the structure of MIP solving, then shortest proofs may do the same for SAT.

Existing work already hints that solver-generated proofs are far from optimal. Proof trimming approaches, such as DRAT-trim (Wetzler et al. 2014), routinely reduce proof length, showing that solvers often produce redundant derivations. Yet this raises a further question: how much shorter can proofs become if one explicitly *optimizes* for length, rather than trimming after the fact? As it turns out, there indeed is substantial room for improvement, as Figure 1 illustrates. This figure shows proof lengths for random unsatisfiable 3-CNF formulas produced by *CaDiCaL* (Biere et al. 2024), a state-of-the-art SAT solver, and trimmed with LRAT-trim (Pollitt et al. 2023), and contrasts them with the shortest possible proof length, obtained by our method, which we introduce in this work. We observed that in a quarter of the analyzed SAT formulas, the CaDiCaL proof is at least 50% longer than the optimal one, *even after trimming* the redundant steps out of it. This also holds for some formula classes encountered in applications: as our results show, the proof length for planning formulas in many cases can at least be halved compared with the proof returned by CaDiCaL after trimming.

Unfortunately, much as this perspective is appealing, there is a steep computational cost attached to it. More specifically, the optimization problem of finding the *shortest* proof is intractable: under the assumption that $P \neq NP$, this problem is not solvable in time polynomial with respect to the *shortest proof* length (Atserias and Müller 2020), which itself can be exponential with respect to the formula size (Haken 1985). Worse still, assuming stronger and widely believed conjectures from computational complexity, the same problem is not even solvable in time sub-exponential in the proof length (Atserias and Müller 2020). As a result, any practical approach to proof construction must accept that global proof-length optimality is unattainable in general and that attention must be restricted to well-controlled settings.

We therefore focus in this work on one simple yet non-trivial setting: discovering shortest *resolution* proofs. While resolution does not circumvent the inherent intractability of proof length minimization, it provides a clean and foundational framework in which the structure of proofs can be studied precisely. Moreover, resolution underlies several prior computational approaches to shortest-proof analysis, notably including the work of Peitl

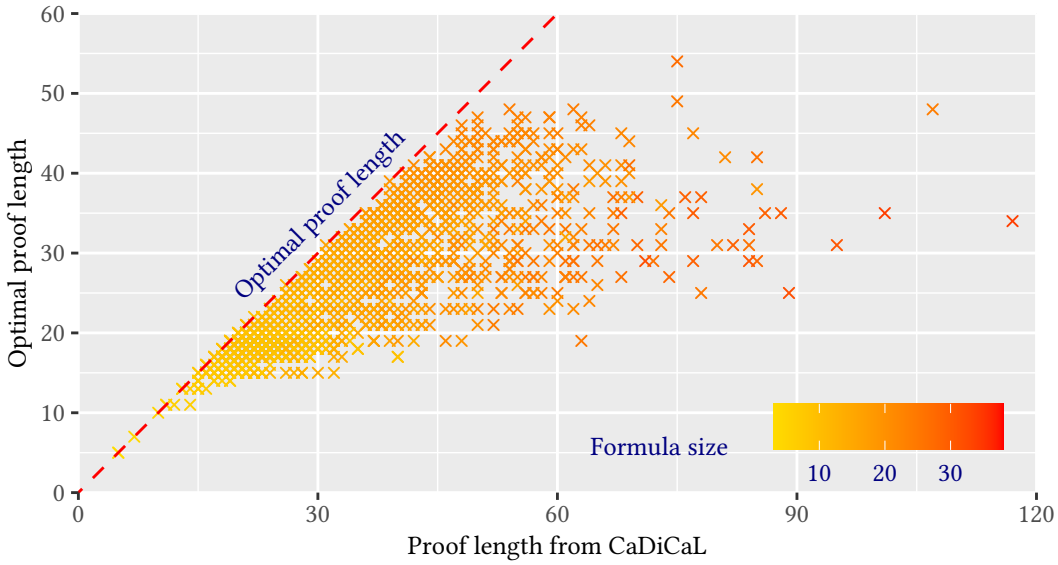


Fig. 1. CaDiCaL proof length comparison against the shortest proofs on random unsatisfiable 3-CNFs.

and Szeider (Peitl and Szeider 2021), which we refer to as the *encoding approach*. They address the problem of discovering shortest resolution proofs for a given formula by constructing a SAT encoding whose satisfying assignments correspond to valid resolution proofs of a fixed length. By querying a SAT solver and incrementally increasing the target length, the approach can either find a proof or certify its non-existence. When combined with symmetry-breaking constraints, this method has been successfully used to establish lower bounds on the minimum resolution proof length for several families of formulas.

However, the aforementioned approach suffers from three significant limitations. First, as the search space for this problem grows with the size of the formula and the length of the proof, the empirical evaluation of this approach suggests that it is feasible *only for formulas with up to a dozen clauses*. Second, this approach enumerates lower bounds on the proof length until the proof of the target length is discovered, which is then declared optimal; in particular, *at no time before termination is there any "short" proof* that could be returned, for example, when the allotted time runs out. Last, *the search space of this problem is highly symmetric*, and while Peitl and Szeider use symmetry-breaking constraints to remedy this issue and show substantial gains for the proof length minimization problems, we point out that this does not resolve all permutation symmetries.

In this paper, we address all three limitations by proposing a novel branch-and-bound algorithm for minimizing the resolution proof length, thereby making a step forward from the encoding approach from earlier work. The key contributions supporting our approach are a representation of the proof that breaks *all* symmetries stemming from clause permutations, and a procedure for deriving lower bounds on the proof length that generalizes the strategy employed by the encoding approach for the case of non-MUS formulas.

Combined in a single workflow, our approach is typically capable of reducing the proof length of a state-of-the-art solver by 25% to 50% on synthetic formulas (e.g., 3-CNFs, graph coloring formulas), and by 15% to 50% on UNSAT formulas from various editions of the SAT Competition, more specifically, on all formulas from SAT Competitions between 2002 and 2025 that CaDiCaL has shown to be UNSAT within fifteen seconds. For the competition formulas, we were consistently able to discover proofs that are shorter than the trimmed proofs

for a few special formula classes. In addition, our algorithm substantially reduces the time to optimality, as suggested by the comparisons with the encoding approach, where our approach solves twice as many instances and terminates faster by at least an order of magnitude in the majority of solved instances.

The remainder of the paper is structured as follows. Section 2 introduces the key concepts of propositional logic and the necessary context about the resolution proof system. We review the state-of-the-art approaches to proof length minimization, as well as further context about the proof complexity, in Section 3. The main novel contribution of this paper is introduced in Section 4, which describes the branch-and-bound procedure for discovering the shortest resolution proof of an unsatisfiable formula, as well as the optimizations used within it to improve the search time. We evaluate the performance of our approach, both for the time to optimality and for the proof length, in Section 5. Since our approach is limited to resolution proofs—and therefore does not cover state-of-the-art solver techniques that rely on stronger proof systems, such as RUP—we frame our approach as the stepping stone towards more expressive reasoning on proofs and discuss the conceptual limitations of our approach in Section 6. Finally, we conclude and discuss further research directions in Section 7.

2 Preliminaries

In this section, we establish the main concepts from propositional logic and SAT solving that we need to motivate the problem we address and the approach we propose. Section 2.1 contains the foundational definitions of propositional logic, such as the definition of an unsatisfiable formula. In Section 2.2, we introduce the notion of the resolutional proof system and restate the most important facts involving it.

2.1 Propositional Satisfiability

We start by introducing propositional formulas in conjunctive normal form, which is a common expectation in modern SAT solvers. Given a Boolean variable x, a *literal* is either the variable x itself or its negation $\bar{x}$. A *clause* ω is a disjunction of a (possibly empty) set of literals: $\omega = \{\ell_1, \ldots, \ell_k\}$; a *formula* F is a conjunction of a set of clauses: $F = \{\omega_1, \ldots, \omega_m\}$. To streamline the notation, we also use a more compact notation for clauses that juxtaposes the literals; for example, the terms $\{x, \bar{y}\}$ and $x\bar{y}$ correspond to the same clause.

Given a formula F over variables $x_1, \ldots, x_n$, a *model* $\boldsymbol{x}$ is a mapping from variables to Boolean values. A model *satisfies* a clause $\ell_1 \cdots \ell_k$ if at least one of the literals evaluates to true, that is, there is an index j such that either $\ell_j = x_t$ and $\boldsymbol{x}_t = 1$ or $\ell_j = \bar{x}_t$ and $\boldsymbol{x}_t = 0$. By extension, a model satisfies a formula F if it satisfies all its clauses. If a model does not satisfy a clause or a formula, we say that the model *falsifies* the clause or formula. Finally, we call a formula *satisfiable* (SAT) if there is a model that satisfies it; otherwise, if all models falsify a formula, we call it *unsatisfiable* (UNSAT).

Note. We assume that clauses do not contain repeated variables, because the logical expression $x \vee \bar{x}$ is tautologically true. Therefore, any clause with opposite literals can be removed from the formula without loss of generality: if a model satisfies the remaining clause, it also satisfies the entire formula.

Note. An empty clause $\bot$ is falsified by any model as there are no literals to evaluate to true; by extension, a formula F that contains an empty clause is falsified by any model.

Determining the SAT/UNSAT status of a given formula is an NP-complete problem (Cook 1971; Levin 1973), which also implies the following asymmetry between SAT and UNSAT statuses: if a formula F is reported to be SAT, this can be verified independently given a satisfying model, however, no similarly compact certificate could support the claim that F is UNSAT. There are a few less compact ways to do it; the next section introduces the certification approach studied in this paper.

2.2 Resolutional Proof System

As stated in the introduction, the UNSAT claim is not trivial to verify. A common way of addressing this is by using the resolution rule, which is a rule for deriving an implied clause from two given clauses (which themselves may have been derived earlier); in particular, the UNSAT claim in that context corresponds to deriving the empty clause.

Definition 1. Given clauses $A = C' \vee x$ and $B = C'' \vee \bar{x}$, a *resolution* rule is an inference rule that produces a clause $C' \vee C''$—also denoted as $A \diamond B$—from A and B:

$$\frac{C' \vee x, \quad C'' \vee \bar{x}}{C' \vee C''}.$$

The resulting clause $A \diamond B$ is called the *resolvent*, and the variable x is referred to as *pivot variable* (or pivot literal, which in this case points to the same entity).

Note. We can assume that A and B do not contain any other pivot literals other than x and $\bar{x}$ because otherwise, the resolvent would be trivially true, and any proof using such a clause can be rewritten without it. For example, $xy \diamond \bar{x}\bar{y} = y\bar{y}$, where x is the pivot variable, is tautologically true. In particular, if any two clauses can be resolved, there is no ambiguity about the pivot choice.

Theorem 1. *The resolution rule is* sound *in the sense that any model satisfying clauses A and B also satisfies the resolvent clause $A \diamond B$, and is refutation complete (Robinson 1965) in the sense that an empty clause can be derived from a formula if and only if it is UNSAT.*

For the purposes of our work, soundness means that if it is possible to derive $\bot$ from a formula F only by applying the resolution rule, then F is unsatisfiable, because any satisfying assignment of F also has to satisfy $\bot$, which is not possible. Refutation completeness is the opposite property: *if a formula F is unsatisfiable*, then there is a sequence of resolution steps on F that eventually derives $\bot$.

Definition 2. Given an UNSAT formula F, a (resolution) *proof* (of unsatisfiability) is a sequence of M clauses $(Q_1, \ldots, Q_M)$ such that $Q_M = \bot$ and for all $1 \leq i < M$ either $Q_i \in F$ (in which case Q_i is referred to as an *axiom*) or $Q_i = Q_j \diamond Q_k$ for some $j, k < i$. The proof *length* is the number of clauses M in it.

Example 1. Consider an UNSAT formula $F = \{x\bar{y}, \bar{x}, y\}$. The following sequence of resolution steps derives an empty clause:

$$\underbrace{(x\bar{y} \diamond \bar{x})}_{=\bar{y}} \diamond y.$$

In our notation, this corresponds to a proof of length five with the following order of clauses: $x\bar{y}, \bar{x}, \bar{y}, y, \bot$. Note that the axiom clause y can be placed in any position in the proof (except the final one) without violating the proof definition; for example, the sequence $x\bar{y}, \bar{x}, y, \bar{y}, \bot$ that swaps y and $\bar{y}$ still conforms to the proof definition: y is an axiom, and placing a clause $\bar{y}$ on a later position does not invalidate the proof.

Another way to encode the proof structure is to track the clauses derived in the proof and the mapping between the resolvents and their premises. This observation suggests the following equivalent definition of a proof:

Definition 3. Given an UNSAT formula F, a *proof DAG* is a directed acyclic graph with the following structure:

- Vertices correspond to clauses over the variables of F and include the clauses of F and the empty clause.
- Any source vertex is a clause in F, and any other vertex has two incoming edges.
- If $\hat{\omega}$ has two incoming edges from ω' and ω'', then $\hat{\omega} = \omega' \diamond \omega''$.

Proposition 1. *Given a proof $(Q_1, \ldots, Q_M)$ of an UNSAT formula F, there exists a proof DAG with vertices $\{Q_1, \ldots, Q_M\}$. Conversely, given a proof DAG of an UNSAT formula F with vertices V, there exists a proof $(Q_1, \ldots, Q_M)$ such that $\{Q_1, \ldots, Q_M\} = V$.*

The resolution proof system is important for proof logging applications as it closely matches the inferences produced by SAT solvers. For example, *unit propagation*, a rule commonly used in SAT solvers that prescribes to assign literal ℓ to be true if there is a clause $C \vee \ell$ such that all literals in C are falsified, corresponds to a sequence of resolutions of clauses that were used in the propagation:

Example 2. Consider the use of unit propagation in Example 1. Starting with an empty model, suppose that the unit propagation discovers the following propagation sequence:

(1) Clause $\bar{x}$ is unit, assign $x \leftarrow 0$.
(2) Clause $x\bar{y}$ is unit,[1] assign $y \leftarrow 0$.
(3) Clause y is unit; as assigning $y \leftarrow 1$ is no longer possible, halt the procedure and declare the formula UNSAT.

We can traverse this sequence of inferences in reverse, and that corresponds to the proof in Example 1. Observe that the final conflict corresponds to the derivation $y \diamond \bar{y} = \bot$. Going a step back, we recognize that $\bar{y}$ was derived using $x\bar{y}$. To complete the reasoning, we need to trace how the $\bar{x}$ was assigned, which happened in the first step via the clause $\bar{x}$. We can summarize this by saying that $\bar{y} = x\bar{y} \diamond \bar{x}$, and, in turn, $y \diamond (x\bar{y} \diamond \bar{x}) = \bot$, which is exactly the proof introduced in Example 1.

Given an UNSAT formula, there are usually many different valid proofs of that fact. The discussion above indicates that the shorter proofs correspond to solver executions making fewer steps and are thus preferable, as they correspond, under simplifying assumptions, to faster executions of a SAT solver. We reflect this observation in the following definition, which introduces the problem we address in the remainder of this paper.

Definition 4. Let F be an UNSAT formula, and $\texttt{Proofs}[F]$ be the set of all its proofs in the sense of Definition 2 that have no unused clauses other than $\bot$. Then the *proof length minimization* problem is the optimization problem of form $\min \{\#P \mid P \in \texttt{Proofs}[F]\}$,[2] and we refer to the proof $P^* \in \texttt{Proofs}[F]$ with the smallest number of clauses as the *optimal proof* of formula F.

Note. We assume the non-existence of unused clauses in the proofs P that we consider while solving the proof length minimization problem. If that is not the case for some proof P', that is, there is an unused clause $\omega \in P'$, then discarding ω from P' yields a proof that is shorter than P', which makes P' irrelevant for the problem of finding the *shortest* proof.

In this work, we are mostly interested in the optimization version of this problem. However, it can also be stated as a decision problem of establishing whether a given UNSAT formula F has a proof with at most k clauses. This formulation of the proof length minimization problem was studied in proof complexity literature, and it is known that there are no algorithms polynomial with respect to k that solve this problem unless P $\neq$ NP (Alekhnovich et al. 2001).

3 Related Work

In this section, we give an overview of earlier work that approached the task of discovering smaller proofs from a variety of angles. We start from the earliest line of work that focused on the local rewriting of resolution proofs (Section 3.1), follow up with proof trimming approaches that sought to discard unused parts of the proof

[1] $\bar{y}$ is the only unassigned literal; the rest of the clause has been falsified.
[2] Throughout the text, we use the notation $\#X$ for the cardinality of a set X.

(Section 3.2), and conclude with the most recent works that attempted to approach this problem holistically, as a single—albeit exceedingly complex—optimization problem (Section 3.3). We introduce these three lines of work in chronological order of their appearance in the SAT solving domain; however, we also cite similar proposals for other combinatorial search paradigms, where relevant, which do not necessarily follow the chronological order.

3.1 Proof Rewriting

The utility of smaller resolution proofs was already recognized in a variety of early works on SAT proof logging (Bar-Ilan et al. 2009; Gershman et al. 2006). Unlike the current state of the art, early approaches to proof logging directly constructed resolution proofs; so the focus of reducing the proof length was on local rewriting rules. Some examples include the Recycle-Units heuristic by Bar-Ilan et al. (Bar-Ilan et al. 2009) and the Trimmer heuristic by Gershman et al. (Gershman et al. 2006).

Later, these ideas were generalized into a *local rewriting* framework that proposed a set of valid rewriting rules for resolution proofs (Rollini, Bruttomesso, and Sharygina 2011; Rollini, Bruttomesso, Sharygina, and Tsitovich 2014) and was packaged in the Skeptic system (Boudou et al. 2014). These techniques have also found their application beyond SAT: for example, Rollini et al. evaluated their local rewriting framework both on SAT and SMT benchmarks and showcased several applications for both paradigms (Rollini, Bruttomesso, and Sharygina 2011), and similar rewriting approaches were proposed for first-order theorem provers (Gorzny and Woltzenlogel Paleo 2015).

Another related line of work explored proof rewriting as a *local search*, such as the RANGER approach (Prestwich and Lynce 2006) which mutates the clause set until the empty clause is reached. Similar developments appear in the field of mixed-integer programming: for example, Muñoz et al. (Muñoz et al. 2023) described how to reduce the size of branch-and-bound trees by trimming irrelevant subtrees or discovering a branching direction improving the dual bound.

Methodologically, rewriting approaches remain inherently local: they can improve a proof via local transformations, but cannot overhaul large parts of the proof in a single step.

3.2 Proof Trimming

Another prominent line of work stems from an observation that a proof may contain irrelevant inferences. Backward checking in DRUP-trim (M. J. H. Heule, Hunt, et al. 2013) processes proofs in reverse order, simultaneously trimming and verifying them. However, for the RUP proof system, just as is the case for the resolution proofs, there is a range of simple problems such that any RUP proof has exponential length. Thus, the paradigm of trimming was extended to more expressive *interference-based systems* (M. Heule and Kiesl 2017) such as DRAT (Wetzler et al. 2014), enabling certification of advanced solver techniques by allowing clauses that are not true for all satisfying models, as long as they do not change the SAT/UNSAT status of the formula.

Interference-based systems introduce additional complexity: proofs are non-monotone, redundancy rules become increasingly involved, and checker implementations must resolve subtle semantic issues (Altmanninger and Rebola Pardo 2020; Rebola-Pardo and Biere 2018). Frameworks such as GRAT (Lammich 2017) address this by shifting complexity back to the proof generator via explicit witness traces. Further extensions (e.g., DPR and DSR (C. R. Codel et al. 2024; M. J. H. Heule, Kiesl, et al. 2017)) and the discussion around them illustrate that the design space of expressive proof systems and their trimming procedures is still actively evolving, and that a universally accepted solution is yet to emerge (Rebola-Pardo 2023).

The idea of removing the parts of a proof that are not relevant to the conclusion of the proof is known under various names in other fields. For example, the VIPR certification approach for mixed-integer programming (Cheung et al. 2017) mentions the "tightening" step that serves the same purpose in the VIPR toolchain as trimming in SAT proof verification. In quantified Boolean formula solving, a similar idea is known as a "dependency scheme"

and serves to eliminate unnecessary variable dependencies that inflate proofs (Blinkhorn and Beyersdorff 2017), a problem present in standard Q-resolution where each existential variable is assumed to depend on *all* preceding universal variables, often leading to artificially long derivations.

The fundamental methodological gap in proof trimming is that it is inherently subtractive: it can only remove redundancy, not introduce structural efficiency. Since these methods operate solely by identifying unneeded steps in a fixed trace, their performance is bounded by the statements logged by the solver. In particular, if the solver navigates into a complex, exponentially long derivation when a simple shortcut exists, trimming can prune the branches of that derivation, but it cannot bridge the gap to a better strategy.

3.3 Explicit Proof Length Minimization

The rewriting and trimming techniques discussed above operate as post-processing steps on a *fixed*, typically solver-generated proof. As a result, they are inherently constrained by the reasoning strategy embodied in that proof: while they can remove redundancies or locally simplify derivations, they cannot replace an inefficient proof structure with a fundamentally shorter one. Overcoming this limitation requires shifting the perspective from repairing a given trace to constructing an optimal proof directly. This amounts to treating proof discovery as a combinatorial optimization problem, where valid proofs form the feasible set and proof length (e.g., the number of inference steps) is the objective. The following subsection reviews approaches that adopt this viewpoint.

Mencía and Marques-Silva (Mencía and Marques-Silva 2019) were, to the best of our knowledge, the first to study the resolution proof length minimization as a combinatorial optimization problem. They observed that the "meta-encoding" introduced by Prestwich and Lynce (Prestwich and Lynce 2006) can be used, with certain performance optimizations, to either discover a resolution proof of length at most s or *prove that no such proof exists*, inviting the construction of another meta-encoding for a larger value of s.

This work also recognized that the proof length minimization problem is highly symmetric—for example, reordering two resolution steps also produces a correct proof unless it breaks in-between steps—and proposes an optimization based on the notion of the "level" in the proof DAG as introduced in Definition 3. Their definition of the clause level is the distance from the axioms in the proof; that is, axioms have level zero, their immediate implications have level one, and so on (cf. the layer list structure in Definition 5). However, they only use it to propose an optimization for the level-one clauses; more specifically, they observe that at least half of its derivations do not have level one.

Further work by Peitl and Szeider (Peitl and Szeider 2021), which we refer to as the encoding approach, also proposed to construct a propositional formula that is true if and only if a given formula F has a proof of length s. However, they took a more holistic approach to improving the search performance by canonicalizing the topological ordering on the underlying proof DAG in order to break underlying symmetries.

More precisely, the encoding approach imposes symmetry-breaking constraints that admit exactly one ordering for any proof DAG, a technique also known as LexTopSort (Fichte et al. 2020). This canonical ordering accepted by LexTopSort is defined using a fixed literal order: among all clauses eligible at each step (those whose premises have already appeared), the lexicographically smallest clause[3] is chosen next. The approach has two advantages: it reduces the search from all proofs to the exponentially smaller space of proof DAGs, and it admits a polynomial-size propositional encoding (Fichte et al. 2020).

However, breaking only symmetries stemming from reordering the same proof DAG does not eliminate all equivalent search states, as proofs with the same sets of derived clauses can be encoded by different DAGs. In other words, *state-of-the-art techniques for proof length minimization do not eliminate all permutational symmetries*, that is, the search states corresponding to the same set of derived clauses but disagreeing on how exactly they

[3] Given a fixed ordering of literals.

are derived. Appendix A provides an explicit example of an UNSAT formula and two proof DAGs on the same set of clauses such that no earlier approach, including symmetry-breaking based on LexTopSort, eliminates either.

Another trait of our work that differentiates it from the encoding approach is that we do not use the SAT solver for any purpose other than to generate a valid proof of the original formula. In particular, this decouples our approach from the encoding considerations, potentially making it applicable for the proof systems with inferences that cannot be compactly represented in propositional form.

Beyond SAT, the idea of treating the search for a valid proof of the smallest size as an optimization problem is also present in other paradigms for optimization and search. For example, Flatt et al. reformulated the problem of finding the smallest proof of congruence of two SMT terms as an integer program (Flatt et al. 2022). Also, Dey et al. (Dey et al. 2024) formulated the problem of finding the smallest branch-and-bound tree for 0—1 integer programs as a dynamic program, and compared the optimal tree sizes for various classes of problems against the strong branching variable selection. Another variation on this idea is an approach for constructing compact tree proofs of optimality in constrained shortest path problems (Sidorov et al. 2024); the distinctive trait of that work is that it focuses on reasoning in an *ad-hoc* proof system designed for interpretability, as opposed to a proof system encoding the solver inference steps.

Therefore, while explicit proof length minimization can, in principle, achieve globally shortest proofs, current encoding-based approaches scale only to small instances due to rapidly growing search spaces and residual symmetries. This insight motivates our proposed approach, which is described in the next section.

4 A Branch-and-Bound Approach to Proof Length Minimization

This section introduces our methodology for discovering the shortest proofs of UNSAT formulas; we structure our narrative as follows:

- Section 4.1 introduces our novel *layer list representation* and shows via Theorem 2 that this representation breaks all permutation symmetries.
- Section 4.2 gives a high-level overview of our *branch-and-bound* approach to discovering shortest proofs.
- Section 4.3 describes the *branching* scheme used in our tree search.
- Section 4.4 shows that discarding subsumed clauses from the formula does not change the optimal proof, an idea which we capture in the *frontier set* definition and use for search tree pruning.
- Section 4.5 completes the description of our approach by establishing the *lower bound* on the proof length.

4.1 Layer List Representation

As argued in the related work survey (page 8), state-of-the-art symmetry breaking, specifically DAG symmetry breaking, misses opportunities to reduce the search space. More specifically, it discriminates proofs having the same clauses but deriving them differently. With that in mind, we start the presentation of our approach by defining an alternative proof representation that breaks *all* permutation symmetries.

Definition 5. Given an UNSAT formula F, the *layer list* $(L^0, L^1, \ldots, L^n)$ on F is the sequence of sets of clauses (referred to as *layers*) satisfying the following properties:

(1) *Initialization.* The first layer $L^0 \subseteq F$ (further the *axiom layer*) is a subset of the input formula.
(2) *Termination.* The final layer is a singleton set containing the empty clause: $L^n = \{\bot\}$.
(3) *Consistency.* All clauses in every layer, except the axiom layer, are obtained by resolving a clause from the preceding layer with a clause from any earlier layer:

$$\forall 1 \le j \le n, \omega \in L^j \implies \omega \in \left\{ \omega' \diamond \omega'' \,\middle|\, \omega' \in L^{j-1}, \omega'' \in L^0 \cup \cdots \cup L^{j-1} \right\}.$$

(4) *Take-it-or-leave-it property.* All clauses, including axioms, are derived in the earliest possible layer:

$$\forall 1 \leq k < j \leq n, \omega \in L^j \implies \omega \notin F \cup \left\{ \omega' \diamond \omega'' \;\middle|\; \omega' \in L^{k-1}, \omega'' \in L^0 \cup \cdots \cup L^{k-1} \right\}.$$

We refer to the total number of clauses across all layers $\sum_{j=0}^{n} \#L^j$ as the *length* of the layer list.

Note. Both the consistency and take-it-or-leave-it property operate on clauses that can be produced as a resolution of *two* clauses. In particular, a clause $\omega_1 \diamond \omega_2 \diamond \omega_3$ with $\omega_1, \omega_2, \omega_3 \in L^0 \cup \cdots \cup L^{j-1}$ is ineligible to be added to L^j (unless the same clause can also be constructed as a resolvent of two clauses), and this clause is also not excluded from further layers by the take-it-or-leave-it property (under the same condition).

The first three conditions of Definition 5 replicate the proof DAG definition (Definition 3), as they assert that the layers constitute a topological ordering of a proof DAG. In particular, any layer list can be mapped to a proof by writing out the axioms, then writing the second layer clauses in any order, then writing out the third layer clauses, and so on until the final layer, which concludes the proof. The take-it-or-leave-it property implements the desired symmetry breaking by mandating that any clause used in the proof is derived in the earliest available layer, as shown in the following theorem.

Theorem 2 (Layer list uniqueness)**.** *Consider an UNSAT formula F and a finite set of clauses P that can be ordered into a proof of satisfiability of F. Then there is exactly one layer list $(L^0, L^1, \ldots, L^n)$ that derives the clauses from P and only them:*

$$L^0 \cup L^1 \cup \cdots \cup L^n = P. \tag{1}$$

Proof. To prove the theorem statement, we need to show that (a) there exists a layer list satisfying the theorem requirements, and (b) this layer list is unique. We start with showing the *existence*; to this end, we place the clauses in the next layer by filtering out used clauses and retaining only the resolvents of a clause in the preceding layer with a clause on one of the earlier layers. More formally, consider the following sequence of clause sets:

$$\begin{aligned}
L^0 &\coloneqq F \cap P,\\
L^1 &\coloneqq (P \setminus L^0) \cap \{\omega' \diamond \omega'' \mid \omega', \omega'' \in L^0\},\\
L^2 &\coloneqq (P \setminus (L^0 \cup L^1)) \cap \{\omega' \diamond \omega'' \mid \omega' \in L^1, \omega'' \in L^0 \cup L^1\},\\
&\cdots\\
L^k &\coloneqq (P \setminus (L^0 \cup \cdots \cup L^{k-1})) \cap \{\omega' \diamond \omega'' \mid \omega' \in L^{k-1}, \omega'' \in L^0 \cup \cdots \cup L^{k-1}\}\, \forall k \geq 1.
\end{aligned}$$

Since $L^k \subseteq P$ for $k \geq 0$ and all constructed sets are pairwise disjoint, that is, $L^j \cap L^k = \emptyset$ for $j \neq k$, we can conclude that this sequence only has a finite amount of non-empty sets, because P is a finite set subsuming all of the sets in the sequence. Let n be the index of the last non-empty set, that is, $L^n \neq \emptyset$ and $L^{n+k} = \emptyset$ for all $k > 0$. We show that $(L^0, L^1, \ldots, L^n)$ satisfies the requirements of the theorem, that is, it satisfies the layer list definition and contains exactly the clauses from P. We structure the remainder of the existence argument as follows: first, we show that Eq. (1) entails the layer list definition, and second, we show that Eq. (1) itself holds, that is, that all clauses in P are covered.

To demonstrate why the *layer list definition follows from Eq.* (1), consider the properties of Definition 5 separately. First, the initialization property ($L^0 \subseteq F$) is satisfied by construction, and the consistency property is ensured by only considering the resolvent clauses in the definitions of L^j for $j > 0$. Next, the take-it-or-leave-it property holds, because if $\omega \in L^j$ for $j \geq 1$ and k is an index such that $1 \leq k < j$, then it is not in F (because then it would be in L^0, and L^k does not contain L^0 by construction), and also

$$\omega \notin \left\{ \omega' \diamond \omega'' \;\middle|\; \omega' \in L^{k-1}, \omega'' \in L^0 \cup \cdots \cup L^{k-1} \right\},$$

because that would imply $\omega \in L^k$, which would contradict $\omega \in L^j$, since L^j has no elements from preceding sets, including L^k.

Since F is UNSAT, we know that $\bot \in P$, which by Eq. (1) implies that $\bot \in L^k$ for some k; to verify the termination property, we need to show that $k = n$ and L^k has no clauses other than $\bot$. If $k < n$, then the subsequent layers contain a redundant clause. That means by Eq. (1) that P itself has a redundant clause, which is not possible, because by Definition 4, all non-empty clauses are used in proofs. Otherwise, $k = n$, and consider a set of clauses $E = L^n \setminus \{\bot\}$; if $E = \emptyset$, then $L^n = \{\bot\}$ (as required), otherwise, the clause in E that occurs the latest in the proof P is redundant.

Now, we show that Eq. (1) holds, that is, that *all clauses in P are covered* by $L^k, 0 \le k \le n$. For a proof by contradiction, let P' be the (non-empty) set of clauses that have not found their way to the L^k sets:

$$P' := P \setminus (L^0 \cup \cdots \cup L^n).$$

Further, we assume that P' does not contain axioms from F, because that would violate $L^0 = F \cap P$. Since clauses in P can be ordered into a proof, we can assume that P is a *sequence* of clauses satisfying the proof definition.

Let $\omega \in P'$ be the clause that appears *first* among the clauses in P' in the proof P. By proof definition, $\omega = \omega' \diamond \omega''$, and since both premises are mentioned earlier in the proof than ω, neither of them is in P' – or, equivalently, both of them can be found in $L^0 \cup \cdots \cup L^n$. But that is a contradiction, because if $\omega' \in L^p$ and $\omega'' \in L^q$ (and assuming without loss of generality that $p \ge q$), then we have $\omega \notin L^{p+1}$, yet $\omega \in P \setminus (L^0 \cup \cdots \cup L^p)$ and $\omega \in \{\omega' \diamond \omega'' \mid \omega' \in L^p, \omega'' \in L^0 \cup \cdots \cup L^p\}$.

We complete the proof by showing the *uniqueness*, in other words, we show that if $(L^0, \ldots, L^n)$ and $(M^0, \ldots, M^{n'})$ are layer lists that derive the same set of clauses, then $n = n'$ and $L^k = M^k$ for all k. If that is not the case, we consider the earliest disagreeing layer $L^k \neq M^k$ with $L^0 = M^0, \ldots, L^{k-1} = M^{k-1}$. Assume without loss of generality that there is a clause ω such that $\omega \in L^k$ but $\omega \notin M^k$. Given that both proofs contain the same clauses, ω can be found in a later layer of M even though it is either an axiom (if $k = 0$) or is implied by the layers $M^0, \ldots, M^{k-1}$, violating the take-it-or-leave-it property for M. □

To illustrate the layer list structure and the reasoning behind the uniqueness claim, we propose the following example.

Example 3. Consider an UNSAT formula $F = \{y\bar{t}, xyzt, \bar{x}y, \bar{y}z, \bar{z}t, \bar{x}\bar{t}, x\bar{y}\}$ and a proof encoded by the sequence starting with the seven axioms and deriving clauses $\bar{x}z, yzt, \bar{x}t, yt, y, \bar{x}, \bar{y}, \bot$. The following sequence of sets complies with the layer list definition:

$$\begin{aligned} L^0 &= \{y\bar{t}, xyzt, \bar{x}y, \bar{y}z, \bar{z}t, \bar{x}\bar{t}, x\bar{y}\} \\ L^1 &= \{\bar{x}z, yzt\} \\ L^2 &= \{\bar{x}t, yt\} \\ L^3 &= \{y, \bar{x}\} \\ L^4 &= \{\bar{y}\} \\ L^5 &= \{\bot\}. \end{aligned}$$

The first three properties of a layer list can be validated by observing that $L^0 = F$ and the remaining layers are constructed from the previous ones by resolving clauses mentioned earlier (for example, $\bar{x}z = \bar{x}y \diamond \bar{y}z$ and $\bar{x}t = \bar{x}z \diamond \bar{z}\bar{t}$). The take-it-or-leave-it property also holds, but requires more effort to verify.

For the third and fourth layers, it can be validated by recognizing that a unit clause ℓ can be derived either as $\ell v \diamond \ell\bar{v}$ or by $\ell v \diamond \bar{v}$ and acknowledging that neither is possible without using the preceding layer. For example,

the only clauses from $L^0 \cup L^1$ that contain $\bar{x}$ are $\bar{x}y$, $\bar{x}z$, and $\bar{x}\bar{t}$; since no two clauses can be resolved, the clause $\bar{x}$ cannot appear on layer 1 or earlier.

For the second layer, we can observe that yt has to have a premise with literal t; the only clauses fitting this description in L^0 are $xyzt$, which cannot be a premise of yt, and $\bar{z}t$, which would require a non-axiom yz as the other premise. Therefore, yt cannot be obtained purely from axioms, satisfying the take-it-or-leave-it property.

To summarize, in this section, we have introduced a representation of proofs that does not reproduce the same set of clauses more than once and is also much simpler to generate procedurally. The next subsection proceeds to exploit this insight.

4.2 Branch-and-bound Search, High-level Description

We now proceed to use the layer list structure to define a tree search procedure for enumerating all unsatisfiability proofs. Layer list uniqueness (Theorem 2) implies that enumerating all proofs—modulo clause permutations—is equivalent to enumerating all layer lists. Definition 5 of the layer list suggests that it can be done by the same construction as used to show existence in Theorem 2:

- Enumerate all possible sets for L^0, that is, all UNSAT subsets of the input formula F.
- For each such L^0, enumerate all possible sets for L^1 – those are the sets of clauses formed as resolvents of L^0 *that respect the take-it-or-leave-it property*, which in this case means only deriving non-axiom clauses using the clauses from L^0, that is, $L^1 \cap F = \emptyset$.
- For each such (L^0, L^1), enumerate all possible sets for L^2 – those are the sets of clauses formed as resolvents of L^1 and $L^0 \cup L^1$ that, again, respect the take-it-or-leave-it property.
- Proceed to do so until $\bot$ can be constructed in a single resolution step, which concludes the layer list.

We observe that this procedure does not require tracking each layer *separately*. More specifically, to generate all valid sets for L^{k+1}, it is sufficient to know (a) which clauses were introduced in the *previous* layer L^k, (b) which clauses were introduced in the *union* of all previous layers $L^0 \cup \cdots \cup L^k$, and (c) which clauses could have been added into earlier layers, but were not – and would thus violate the take-it-or-leave property if added next. Given that, the task of enumerating layer lists starting from $(L^0, L^1, \ldots, L^k)$ can be seen as the task of enumerating layer list suffixes given the *union* of all but the last layer $L^0 \cup \cdots \cup L^{k-1}$ and the last layer L^k.

However, translating this enumeration directly into a tree search has a practical problem: as the number of clauses that can be added to the layer L^{k+1} is quadratic with respect to $\#\left(L^0 \cup \cdots \cup L^k\right)$, the branching factor of that search tree, exponential with respect to the number of possible clauses, is going to be prohibitive. Therefore, we do not directly branch on all possible subsets for L^k, but construct these sets iteratively by adding or ignoring one candidate clause at a time.

Given that, we now provide the definition of a *subproblem* in our tree search procedure.

Definition 6. Given an UNSAT propositional formula F, a *subproblem* $\mathcal{P}$ is a tuple containing:

- a set of *previous layer clauses* $\texttt{Previous}[\mathcal{P}]$,
- a set of *current layer clauses* $\texttt{Current}[\mathcal{P}]$,
- a set of *next layer clauses* $\texttt{Next}[\mathcal{P}]$,
- a set of *forgotten clauses* $\texttt{Forgotten}[\mathcal{P}]$,
- and, for convenience, we define $\texttt{Known}[\mathcal{P}] = \texttt{Previous}[\mathcal{P}] \cup \texttt{Current}[\mathcal{P}] \cup \texttt{Next}[\mathcal{P}]$.

The individual parts of this definition translate to the previous narrative as follows: Current layer clauses of the subproblem store the last fully constructed layer L^k. Previous layer clauses of the subproblem store the union $L^0 \cup \cdots \cup L^{k-1}$ of all preceding layers. Next layer clauses store a *subset* of the next layer; this storage is used for enumerating the clauses used in the next layer L^{k+1}.

Last, forgotten clauses are clauses not added on earlier occasions, despite the opportunity to do so. Unlike the other parts of the subproblem definition, $\texttt{Forgotten}[\cdot]$ does not translate to any part of a resulting layer list. Instead, the purpose of the $\texttt{Forgotten}[\cdot]$ is to enforce the take-it-or-leave-it property by (i) storing clauses satisfying the consistency property for preceding layers while being absent from the proof and (ii) disregarding them during the construction of the following layers.

To navigate between the notions of a subproblem and a layer list, we also introduce the definition that encodes the concept of a "feasible solution" of a subproblem.

Definition 7. Let F be an UNSAT propositional formula, and $L = (L^0, L^1, \ldots, L^n)$ is a layer list for its proof of unsatisfiability. We say that L is a *compatible proof* for a subproblem $\mathcal{P}$ with a non-empty $\texttt{Current}[\mathcal{P}]$ if:

- $\texttt{Current}[\mathcal{P}]$ is equal to one of the layers L^k,
- the union of all preceding layers $L^0 \cup \cdots \cup L^{k-1}$ is equal to $\texttt{Previous}[\mathcal{P}]$,
- the next layer L^{k+1} is a superset of $\texttt{Next}[\mathcal{P}]$,
- no clauses from $\texttt{Forgotten}[\mathcal{P}]$ occur in the layer list.

If, on the other hand, $\mathcal{P}$ has an empty $\texttt{Current}[\mathcal{P}]$, we say that L is a compatible proof for it if $L^0 \supseteq \texttt{Next}[\mathcal{P}]$ and no clauses from $\texttt{Forgotten}[\mathcal{P}]$ occur in the layer list.

Note. It is possible that $\texttt{Next}[\mathcal{P}] \neq \emptyset$ but $\texttt{Current}[\mathcal{P}] = \emptyset$. This corresponds to the search state where no layer of the layer list has been fixed—in particular, *the axiom layer* has not been fixed—but *some* of the axioms have already been selected. Just as with the other introduced clauses, they will eventually be pushed into $\texttt{Current}[\cdot]$ and $\texttt{Previous}[\cdot]$ of the further subproblems produced by branching; see Section 4.3 for a more detailed description of the branching strategy.

Figure 2 illustrates the definitions of the subproblem and the compatible proof, with the following example elaborating further on the interactions between the subproblem components.

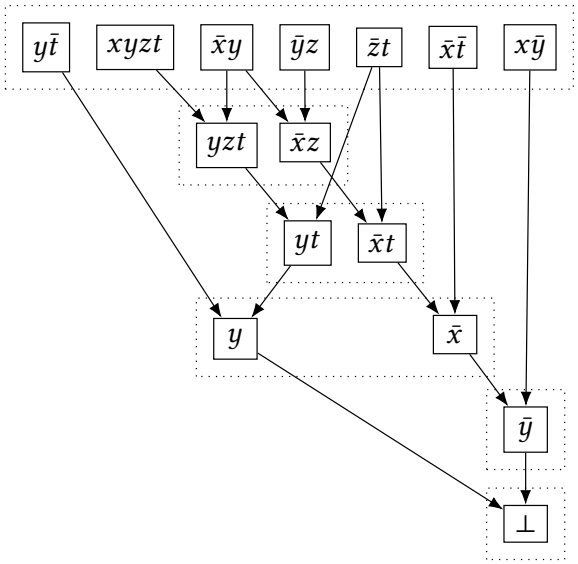
Example 4. In the subproblem from Figure 2b, the definition components are as follows:

- $\texttt{Previous}[\mathcal{P}] = F \cup \{yzt, \bar{x}z\}$ stores the union of the formula $F = L^0$ and the layer L^1 from Figure 2a,
- $\texttt{Current}[\mathcal{P}] = \{yt, \bar{x}t\}$ stores L^2 as the last fully constructed layer,
- $\texttt{Next}[\mathcal{P}] = \{\bar{x}\}$ stores the $\bar{x}$ clause from L^3 (encoding the decision to include it in the proof) but not the y clause; note that this does not imply any decision made on the y clause.
- $\texttt{Forgotten}[\mathcal{P}] \supseteq \{zt, yz, \bar{y}t, xzt\}$; we do not write out the full set for the sake of brevity, but the three listed clauses are in the forgotten set, as they can be decomposed into a resolution of two earlier clauses from $\texttt{Previous}[\mathcal{P}]$; for example, $zt = yzt \diamond \bar{y}z$, and $xzt = xyzt \diamond \bar{y}z$. Hence, $y \in \texttt{Forgotten}[\mathcal{P}]$ encodes the decision to *not* include this clause in the proof, whereas $y \notin \texttt{Forgotten}[\mathcal{P}]$ in this context means that this clause is not *yet* decided to be in the proof.

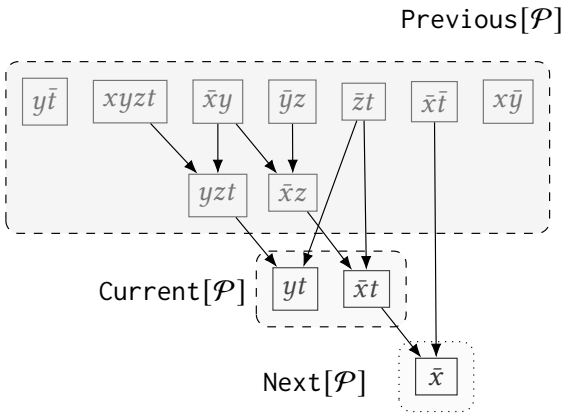
Algorithm 1 formalizes the insight developed in this subsection as a single search procedure. The algorithm works in a branch-and-bound fashion by maintaining a queue of unexplored subproblems and an incumbent shortest proof. The queue is initialized with the *root subproblem* $\mathcal{P}_{root}$ such that

$$\texttt{Previous}[\mathcal{P}_{root}] = \texttt{Current}[\mathcal{P}_{root}] = \texttt{Next}[\mathcal{P}_{root}] = \texttt{Forgotten}[\mathcal{P}_{root}] = \emptyset.$$

Each iteration takes an unexplored subproblem off the queue, prunes it if possible, updates the incumbent if needed, and splits it into new subproblems. In particular, as it normally happens with branch-and-bound algorithms, our approach is an *anytime* approach, meaning that stopping it at any point after starting the loop over subproblems results in a valid resolution proof from the incumbent L^* and a valid lower bound on the proof length.



(a) A layer list L; the arrows encode the resolutions deriving the non-axiom clauses and are included purely for illustrative purposes.



(b) A subproblem $\mathcal{P}$ such that L is a compatible proof for $\mathcal{P}$. The second-to-last layer of the L is a superset of the next set $\texttt{Next}[\mathcal{P}]$ of the subproblem.

Fig. 2. An illustration of the concepts related to the subproblem definition.

As is common in branch-and-bound procedures, we need some `Partition` procedure to partition a subproblem into smaller subproblems without discarding the optimal layer list and a `Bound` procedure to provide a valid lower bound for a proof length of a compatible proof of a subproblem.[4] Additionally, we use a `Complete` procedure that provides a valid but not necessarily shortest proof starting from the clauses of the current subproblem. In the next proposition, we specify the conditions on these procedures sufficient for the whole algorithm to be correct; the next subsections explain each of these subroutines in detail.

Proposition 2. *Algorithm 1 terminates and returns the shortest proof of an unsatisfiable propositional formula F if the following assumptions hold for any subproblem $\mathcal{P}$:*

1. *For any compatible proof for $\mathcal{P}$ there is a compatible proof for $\mathcal{P}'$ of at most the same length for some $\mathcal{P}' \in \texttt{Partition}(\mathcal{P}, F)$.*
2. *Any compatible proof for $\mathcal{P}$ has length at least $\texttt{Bound}(\mathcal{P})$.*
3. *$\texttt{Complete}(\cdot)$ returns a valid resolution proof of unsatisfiability of its input formula.*

Note. $\texttt{Complete}(\cdot)$ does not interact with the take-it-or-leave-it property and the notion of the forgotten set; in particular, this procedure constructs only a *proof* and not a *layer list*. This is sufficient to ensure the search completeness, because the subproblems containing the optimal proof are not going to be pruned until the optimal proof is constructed in some $\texttt{Complete}(\cdot)$ call, provided that the first two conditions are satisfied.

[4]The previous version of the paper also considered the use of *dominance* pruning, which established an order $\mathcal{P} \preceq \mathcal{P}'$ between subproblems $\mathcal{P}, \mathcal{P}'$ such that for any compatible proof of $\mathcal{P}'$, there is a compatible proof of $\mathcal{P}$ with the same or smaller number of clauses. However, that version proposed the following *incorrect* version of the dominance rule: subproblem $\mathcal{P}$ dominates another subproblem $\mathcal{P}'$ when (a) any axiom used in the proof prefix of $\mathcal{P}$ is also used in $\mathcal{P}'$, (b) any clause that can be derived in $\mathcal{P}'$ can also be derived in $\mathcal{P}$, (c) any frontier clause of $\mathcal{P}'$ is also a frontier clause of $\mathcal{P}$, (d) any forgotten clause of $\mathcal{P}$ is also forgotten in $\mathcal{P}'$, and (e) $\mathcal{P}$ contains no more resolution steps than $\mathcal{P}'$.

Algorithm 1: Branch-and-bound algorithm for resolution proof length minimization

Data: Unsatisfiable propositional formula F.
Result: The shortest resolution proof L^* of unsatisfiability of F.

```
// Subproblem queue; initialized with the root subproblem
Q ← {P_root};
// Incumbent proof
L* ← Complete(F);
while Q ≠ ∅ do
    // Take the next subproblem from the queue
    P ← Pop(Q);
    // Prune if no compatible proof can improve the incumbent
    if Bound(P) ≥ #L* then
        continue;
    end
    // Partition the current subproblem
    for P_sub ∈ Partition(P, F) do
        Q ← Q ∪ {P_sub};
        // Complete the new subproblems to full proofs, update the incumbent if possible
        L_sub ← Complete(Known[P_sub]);
        if #L_sub < #L* then
            L* ← L_sub;
        end
    end
end
return L*;
```

Before proceeding to a more specific discussion, we first explain the design decisions of the more straightforward parts of this algorithm. First, we implement the queue Q as a priority queue that pops the subproblem with the most optimistic lower bound, thereby making this approach a best-first search strategy; ties are broken in favor of more recent subproblems.

The `Complete` function triggers a SAT solver on the `Known`$[\mathcal{P}]$ as an input formula, extracts its proof in LRAT format (Cruz-Filipe et al. 2017), and decodes each derived clause into a propagation sequence by traversing the dependent clauses back-to-front. Additionally, we vary the random seed of a solver; we observe that this is helpful to increase the variety of proofs constructed during search, which, in particular, increases the likelihood of discovering a short proof.

The remainder of Section 4 finishes the algorithm description by establishing the subroutines compliant with Proposition 2.

4.3 Subproblem Partitioning

In this subsection, we leverage our subproblem representation to design a branching strategy that generates at most one subproblem per resolvent of a clause in `Current`$[\cdot]$ and `Current`$[\cdot] \cup$ `Previous`$[\cdot]$ while addressing the problem of enumerating the exponential number of sets of possible resolution clauses. To illustrate how we approach this, we first introduce an example:

Example 5. Consider the subproblem from Example 4 and observe that we can list the clauses that could be in $\texttt{Next}[\mathcal{P}]$, except for the clause $\bar{x}$ that is already pinned to $\texttt{Next}[\mathcal{P}]$, by evaluating pairwise resolutions between the $\texttt{Current}[\mathcal{P}]$ clauses and $\texttt{Current}[\mathcal{P}] \cup \texttt{Previous}[\mathcal{P}]$ clauses. In our case, this gives us a candidate list $y, zt, xt, \bar{y}t$.[5]

However, the previous example also establishes that the clauses zt and $\bar{y}t$ are in the forgotten set and hence must be deleted. The interpretation we can give here is that those clauses could have been derived on earlier layers and therefore, were derived in other parts of the search tree, making the decisions about them in the current subproblem pointless. We are left with clauses y, xt; suppose now that this is an *ordered* list.

All compatible proofs of $\mathcal{P}$ can be split into three pairwise disjoint groups:

- $y \in \texttt{Next}[\mathcal{P}]$. This corresponds to deriving y in the layer L^3 and postponing the decision about the xt clause. These decisions can be encoded by adding y to the next set, as illustrated in Figure 3a.
- $y \notin \texttt{Next}[\mathcal{P}], xt \in \texttt{Next}[\mathcal{P}]$. This corresponds to adding xt to the layer L^3 and never touching y. These decisions can be encoded by adding y to the forgotten set and xt to the next set, as illustrated in Figure 3b.
- $y, xt \notin \texttt{Next}[\mathcal{P}]$. This corresponds to deriving only $\bar{x}$ in the layer L^3 and never touching y, xt. These decisions can be encoded by adding y, xt to the forgotten set and shifting previous/current/next sets as illustrated in Figure 3c.

The takeaway from this example is that once we list all clauses $\{\omega_1, \ldots, \omega_n\}$ that can be derived in the $\texttt{Next}[\cdot]$ set of a subproblem $\mathcal{P}$, we can split all compatible proofs into the compatible proofs of the subproblems created by the following procedure:

- Add ω_1 into $\texttt{Next}[\mathcal{P}]$ to obtain $\mathcal{P}'_1$, the subproblem corresponding to all compatible proofs deriving ω_1.
- Add ω_1 into $\texttt{Forgotten}[\mathcal{P}]$ and add ω_2 to $\texttt{Next}[\mathcal{P}]$ to obtain $\mathcal{P}'_2$, the subproblem corresponding to all compatible proofs *not* deriving ω_1 but deriving ω_2.
- Add ω_1, ω_2 into $\texttt{Forgotten}[\mathcal{P}]$ and add ω_3 to $\texttt{Next}[\mathcal{P}]$ to obtain $\mathcal{P}'_3$, the subproblem corresponding to all compatible proofs deriving neither ω_1 nor ω_2 but deriving ω_3.
- Create the subproblems $\mathcal{P}'_k$ for all k between 1 and n; each of those subproblems captures the compatible proofs that do not derive any clause from $\{\omega_1, \ldots, \omega_{k-1}\}$ but derive ω_k.
- Finally, obtain the subproblem $\mathcal{P}^{commit}$ by adding all derivable clauses to $\texttt{Forgotten}[\mathcal{P}]$, moving all clauses from $\texttt{Next}[\mathcal{P}]$ to $\texttt{Current}[\mathcal{P}^{commit}]$ and all clauses from $\texttt{Current}[\mathcal{P}]$ to $\texttt{Previous}[\mathcal{P}^{commit}]$. This subproblem corresponds to all compatible proofs that derive *no* clauses from the set $\{\omega_1, \ldots, \omega_n\}$.

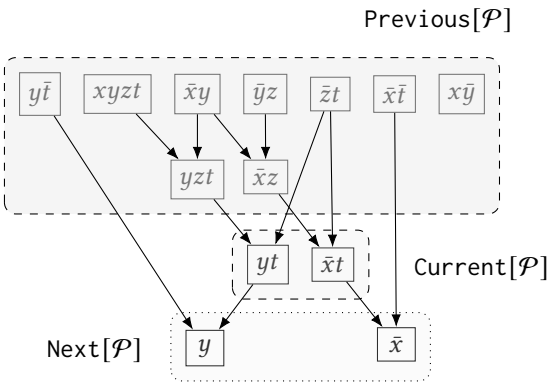
This procedure can be formalized as a valid branching strategy in the sense of Proposition 2.1, as well as to cover the special case of $\texttt{Current}[\mathcal{P}] = \emptyset$, where the set of axioms used in the proof is yet to be fixed. The statements of such branching rules and their proofs can be found in Appendix B.1.

However, while this branching strategy provides a mechanism for the complete enumeration of proofs, the branching factor it introduces is one plus the number of clauses that can be derived in a given subproblem – which, in turn, is at worst case quadratic with respect to the total number of derived clauses in the subproblem. The next section proposes a simple strategy for reducing the resulting branching factor in practice.

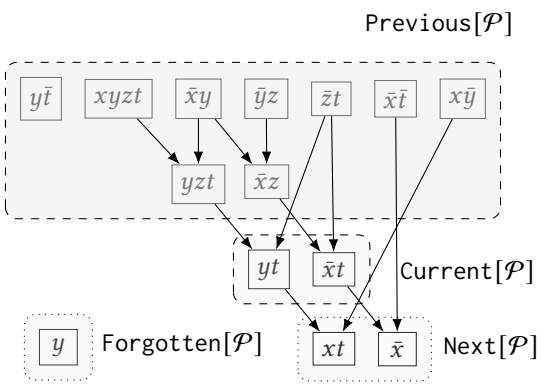
4.4 Frontier Pruning

In this subsection, we introduce a mechanism for simplifying a subproblem without loss of optimality, which is based on the idea that clauses subsumed by some other known clause are not helpful for deducing unsatisfiability. We start with embedding this notion in the following definition.

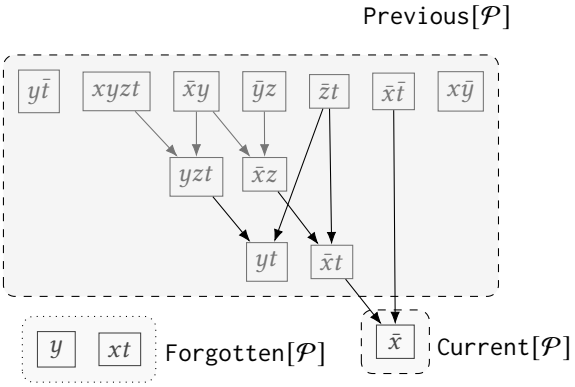
[5]Excluding the clauses already in the proof.



(a) Subproblem that derives y in the next layer.



(b) Subproblem that derives xt but not y in the next layer.



(c) Subproblem that does not derive new clauses in the next layer. Note that this subproblem shifts the current and next clause sets, so $\texttt{Next}[\mathcal{P}] = \emptyset$, and $\bar{y} = x\bar{y} \diamond \bar{x}$ can be added as the next clause in the subproblems stemming from further branching.

Fig. 3. Subproblems that cover all the compatible proofs of the subproblem in Figure 2b.

Definition 8. A *frontier* is a set of clauses $\texttt{Frontier}[F]$ of an unsatisfiable propositional formula F that are not subsumed by another clause of that formula:

$$\texttt{Frontier}[F] \coloneqq \{\omega \in F \mid \nexists \omega' \in F : \omega' \subset \omega\}.$$

The relations $\in_{\mathcal{F}}$ and $\subseteq_{\mathcal{F}}$ are shorthands for membership and inclusion in the frontier:

$$\omega \in_{\mathcal{F}} F :\Leftrightarrow \omega \in \texttt{Frontier}[F],$$
$$\Omega \subseteq_{\mathcal{F}} F :\Leftrightarrow \Omega \subseteq \texttt{Frontier}[F].$$

If a proof of an UNSAT formula F only uses clauses of $\texttt{Frontier}[F]$, then this proof is called a *frontier proof*.

In other words, a frontier excludes obviously redundant clauses (more specifically, the ones implied by some other clause). In particular, if F is UNSAT, so is $\texttt{Frontier}[F]$, and any proof of $\texttt{Frontier}[F]$ is also a proof of the original formula F; we use this observation to simplify the input formula before running the branch-and-bound

search. More importantly, *using non-frontier clauses is pointless* in the sense that they could be swapped out with clauses from the frontier of $\texttt{Previous}[\mathcal{P}] \cup \texttt{Next}[\mathcal{P}]$ to derive a stronger statement. To make this point clearer, consider the following example:

Example 6. Consider again the subproblem shown in Figure 4; the frontier clauses of $\texttt{Known}[\mathcal{P}] = \texttt{Previous}[\mathcal{P}] \cup \texttt{Current}[\mathcal{P}]$ are displayed with a thicker border. Note that the "frontier clause" $\in_{\mathcal{F}}$ property is not monotonic along the proof DAG: a frontier clause, such as $\bar{z}t$, may be used to derive a clause superseded by a newer clause, which is what happens with $\bar{x}t$ and $\bar{x}$.

Now consider the candidates for the next layer suggested by the branching rule Lemma 5, more specifically, the derivation $xy = xt \diamond y\bar{t}$. While this clause is valid with respect to our branching strategy, its derivation uses a non-frontier clause $y\bar{t}$. If we instead use the frontier clause y, then we can omit one inference (as this clause is already derived) and obtain a stronger clause in the sense that if y is true, then $y\bar{t}$ also has to be true.

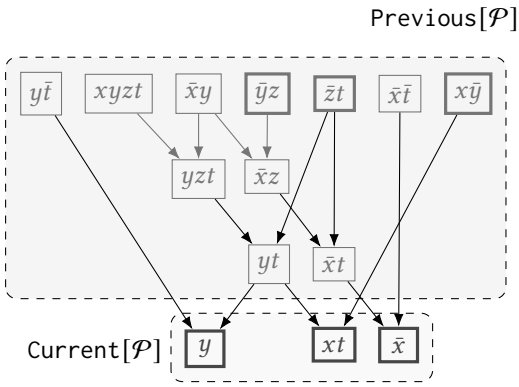


Fig. 4. The subproblem for formula F with frontier clauses highlighted with a thicker border.

The last example suggests that any proof with non-frontier clauses can be rephrased only with frontier clauses without making the proof longer – and maybe even making it shorter. The next theorem formalizes this insight and shows by induction over the proof length that every subsequent step in the original proof can be mirrored—or even improved by removing literals—in a proof of the frontier formula; the proof can be found in Appendix B.2.

Theorem 3 (Frontier sufficiency)**.** *If F has a proof with m resolution steps, then it also has a* frontier *proof with at most m resolution steps.*

Corollary 1. *The branching strategy from Lemma 5 is correct when the underlying clause set is restricted to resolving the clauses from frontiers of the previous and current sets:*

$$\{\omega_1, \ldots, \omega_m\} = \{\omega = \omega' \diamond \omega'' \mid \omega' \in_{\mathcal{F}} \texttt{Current}[\mathcal{P}], \\ \omega'' \in_{\mathcal{F}} \texttt{Current}[\mathcal{P}] \cup \texttt{Previous}[\mathcal{P}], \\ \omega \notin \texttt{Forgotten}[\mathcal{P}], \omega \notin \texttt{Known}[\mathcal{P}]\}.$$

Corollary 1 implies, in particular, that the branching applied only to the frontiers remedies the problem demonstrated by Example 6. In that case, xy can no longer be derived, as it uses the non-frontier clause; the branching strategy only creates the commit subproblem, reflecting that there is nothing meaningful to derive in the next layer.

We can combine this reasoning with a trivial observation that any clause has to be used somewhere later in the proof to derive the following pruning rule:

Corollary 2. *Consider a subproblem $\mathcal{P}$ with a clause $\omega \in \mathsf{Known}[\mathcal{P}]$ such that $\omega \notin_{\mathcal{F}} \mathsf{Current}[\mathcal{P}] \cup \mathsf{Previous}[\mathcal{P}]$, and suppose ω can be omitted from $\mathcal{P}$ without invalidating it. Then $\mathcal{P}$ can be discarded from the branch-and-bound queue without loss of optimality.*

4.5 Lower Bounding the Proof Length with UNSAT Clause Subsets

To complete the definition of the branch-and-bound procedure of Algorithm 1, we need to specify a valid subroutine $\mathsf{Bound}(\mathcal{P})$ for deriving proof length lower bounds for arbitrary subproblems $\mathcal{P}$. We start by restating the lower bound on the proof length of a *minimally unsatisfiable formula* proposed in Lemma 4 of the encoding approach (Peitl and Szeider 2021):

Lemma 1. *Suppose that F is a minimally unsatisfiable formula (i.e., excluding any clause from F makes it satisfiable). Then any proof of F has at least $2\#F - 1$ clauses: all $\#F$ formula clauses and at least $\#F - 1$ derived clauses.*

The key idea of our bounding approach is to generalize this bound to UNSAT formulas that are not necessarily *minimally* unsatisfiable. To this end, we define the problem of finding the smallest minimal unsatisfiable subset as follows:

Definition 9. Suppose that F is an UNSAT formula and $U \subseteq F$ is a set of *fixed* clauses. Then the *smallest minimal unsatisfiable subset* is defined as the set that has the smallest cardinality among the unsatisfiable subsets of F containing U as a subset:

$$\mathrm{SMUS}(F; U) := \operatorname*{arg\,min}_{U \subseteq S \subseteq F, S \text{ is UNSAT}} \#S.$$

The next lemma presents the resulting expression; the proof can be found in Appendix B.2.

Lemma 2. *Any compatible proof of a subproblem $\mathcal{P}$ derives at least*

$$\#\,\mathrm{SMUS}(\mathsf{Known}[\mathcal{P}]; \emptyset) - 1 = \min\{\#S - 1 \mid S \subseteq_{\mathcal{F}} \mathsf{Known}[\mathcal{P}], S \textit{ is UNSAT}\}$$

clauses not present in $\mathcal{P}$.

This idea can be strengthened using Corollary 2 to only consider clause subsets S that include all unused clauses:

Lemma 3. *Let $\mathcal{P}$ be a subproblem and $U \subseteq_{\mathcal{F}} \mathsf{Known}[\mathcal{P}]$ be the set of clauses that have not been used as a premise of a resolution step. Consider a compatible proof L that does not use all of the clauses from U. Then there is another subproblem $\mathcal{P}'$ with a compatible proof L' having fewer clauses than L.*

Combining the last two observations provides the following tighter version of the SMUS lower bound:

Corollary 3. *Given a formula F, let $\mathcal{P}$ be a subproblem with $\mathsf{Current}[\mathcal{P}] \neq \emptyset$ and $U \subseteq \mathsf{Known}[\mathcal{P}]$ being the set of clauses that were not used so far. Then the expression*

$$\mathsf{Bound}(\mathcal{P}) = \#\mathsf{Known}[\mathcal{P}] + \#\,\mathrm{SMUS}(\mathsf{Known}[\mathcal{P}]; U) - 1 \tag{2}$$

satisfies Proposition 2.2 as a lower bound on the length of compatible proofs.

To illustrate how the bound in Eq. (2) works, both with and without strengthening by unused clauses, we analyze the example subproblem from Figure 4:

Example 7. The first term $\#\mathsf{Known}[\mathcal{P}]$ counts clauses already added, fourteen of them for $\mathcal{P}$ from Figure 4; the non-trivial part of bounding the proof length for this subproblem is in the remaining terms that estimate how many clauses are *yet* to be added. It is easy to see that $\mathrm{SMUS}(\mathsf{Known}[\mathcal{P}], \emptyset) = \{\bar{x}, y, x\bar{y}\}$—this is an UNSAT formula, and $\mathsf{Known}[\mathcal{P}]$ clearly has no UNSAT subformula with two clauses—which gives a lower bound of $14 + 3 - 1 = 16$ clauses on any proof compatible with $\mathcal{P}$.

However, we can also assume that all unused clauses $U = \{xt, \bar{x}, y\}$ will eventually be used. Since U is SAT, any MUS containing U has at least four clauses; in fact, this is the exact SMUS size, because $\{\bar{x}, y, xt, x\bar{y}\} = \text{SMUS}(\texttt{Known}[\mathcal{P}], U)$ is UNSAT. This gives a tighter lower bound of $14+4-1 = 17$ clauses on any proof compatible with $\mathcal{P}$ that also cannot be discovered in another subproblem.

Unfortunately, determining the SMUS$(\cdot,\cdot)$ quantity is itself a non-trivial task; in fact, it is known to be Σ_2^P-complete (Liberatore 2005). Therefore, we also need an efficient procedure for determining SMUS lower bounds; we proceed to use them transitively in Eq. (2). To this end, we use the Forqes package by (Ignatiev et al. 2015); the defining trait of this approach is that it is a dual optimization algorithm in the sense that it enumerates lower bounds on the SMUS cardinality until it discovers the unsatisfiable set having exactly that length. Therefore, we run this approach with a time limit of one second and introduce the best discovered lower bound instead of the SMUS terms in Eq. (2).

However, for subproblems with small frontier sets, this approach introduces an overhead from repeated calls to the SMUS bounding subroutine. While the slowdown of a *single* bounding call is not substantial, the *cumulative* effect of thousands of bounding calls from different subproblems, in our experience, was prohibitive even for the small formulas. To remedy this, we run *another* branch-and-bound algorithm for finding the SMUS with the same time limit if the SMUS subroutine input has at most M_{switch} clauses, where M_{switch} is a tunable parameter.

More precisely, given a formula F with at most M_{switch} clauses, we compute a lower bound on $\#\,\text{SMUS}(F, U)$ with a branch-and-bound algorithm stated in Algorithm 2. In this bounding procedure, we represent subproblems by the set of *formula* clauses F' and the set of *fixed* clauses U' such that $U \subseteq U' \subseteq F' \subseteq F$, and the root subproblem corresponds to the inputs of a SMUS$(\cdot,\cdot)$ call. The branching step corresponds to taking a non-fixed clause ω and creating two subproblems as follows:

- retain all the formula clauses, add ω to the fixed set;
- remove ω from the formula clauses and add all critical clauses to the fixed set; a critical clause here is a clause such that removing it from the formula clauses makes the remaining set satisfiable.

However, this procedure alone does not solve the problem of bounding the # SMUS quantity, because non-trivial subproblems still need a bound of their own. One valid naïve bound is the number of fixed clauses $\#U'$. To explain how we refine this pruning strategy, consider the following example:

Example 8. Consider an arbitrary UNSAT 3-CNF, that is, an UNSAT problem F such that all clauses have exactly three literals. Then we can show that *any UNSAT subset $U \subseteq F$ has at least eight clauses*, in other words, that $\#\,\text{SMUS}(F, \emptyset) \geq 8$. To see why, observe that if F uses n variables, then any clause $\omega \in F$ is false for exactly $2^{n-3} = \frac{1}{8} \times 2^n$ models, as falsifying all three literals of ω is necessary and sufficient to falsify ω itself, without any constraints on the remaining $(n-3)$ variables. But that means that the subformula with k clauses can falsify at most $\frac{k}{8} \times 2^n$ models — whereas any UNSAT formulas falsify all 2^n models, which can only be the case if $k \geq 8$.

We generalize this reasoning to account for fixed clauses and clauses of various lengths to construct the following SMUS bounding subroutine:

Lemma 4 (model-covering lower bound)**.** *Let F be an UNSAT formula with a set of fixed clauses $U \subseteq F$ such that $\sum_{\omega \in U} 2^{-\#\omega} < 1$, and suppose that its non-fixed clauses are ordered by ascending clause length, that is,*

$$F \setminus U =: \{\omega_1, \ldots, \omega_m\}, \#\omega_1 \leq \cdots \leq \#\omega_m.$$

Then $\#\,\text{SMUS}\,(F; U) \geq \#U + k$, where $k \in [1, m]$ is the smallest index satisfying the inequality

$$\sum_{\omega \in U} 2^{-\#\omega} + \sum_{j=1}^{k} 2^{-\#\omega_j} \geq 1. \tag{3}$$

Algorithm 2: Branch-and-bound algorithm for computing a lower bound on # SMUS.

Data: Unsatisfiable propositional formula F.
Data: A set of fixed clauses $U \subseteq F$.
Result: A number ℓ such that $\ell \leq \#\,\mathrm{SMUS}(F, U)$.
// Subproblem queue; initialized with the root subproblem
// Subproblems are encoded by their available clauses F' and fixed clauses U'
$\mathcal{Q} \leftarrow \{(F, U)\}$;
// Best known lower and upper bounds
$\ell \leftarrow \mathtt{LowerBound}(F, U)$;
$u \leftarrow \#F$;
while $\mathcal{Q} \neq \emptyset$ *and compute budget not exceeded* **do**
  // Update the global lower bound
  $\ell \leftarrow \min\{\mathtt{LowerBound}(F', U') \mid (F', U') \in \mathcal{Q}\}$;
  // Take the next subproblem from the queue
  $(F', U') \leftarrow \mathtt{Pop}(\mathcal{Q})$;
  // Prune if all MUSes in this subproblem have at least u clauses
  if $\mathtt{LowerBound}(F', U') \geq u$ **then**
    continue;
  end
  // Split into two subproblems: fix an arbitrary clause in one, discard it in another
  if $F' = U'$ **then**
    continue;
  end
  $\omega \leftarrow$ a clause from $F' \setminus U'$;
  $F_{crit} \leftarrow \{\omega' \mid F' \setminus \{\omega, \omega'\}$ is SAT$\}$;
  // Update the subproblem queue and the upper bound
  for $(F_{sub}, U_{sub}) \in \{(F' \setminus \{\omega\}, U' \cup F_{crit}), (F', U' \cup \{\omega\})\}$ **do**
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(F_{sub}, U_{sub})\}$;
    if $u > \#F_{sub}$ **then**
      $u \leftarrow \#F_{sub}$;
    end
  end
end
return ℓ;

Note. In the vocabulary of Algorithm 2, Lemma 4 means that $\mathtt{LowerBound}(F, U) = \#U + k$ with k defined by Eq. (3) is a valid implementation of pruning by lower bound.

5 Experimental Evaluation

As our approach can be seen as both the procedure that eventually discovers the shortest proof and as an anytime optimization algorithm operating on resolution proofs, we are able to put forward two distinct experimental hypotheses for both of those viewpoints as follows:

- Our approach leverages the layer list representation to reduce the time to arrive at the shortest proofs.
- Our approach can enumerate the sets of implied clauses to give enough information to CaDiCaL to discover shorter proofs.

In this section, we conduct an extensive empirical evaluation of our approach, addressing both our hypotheses: we compare the produced proof length against trimmed proofs from CaDiCaL[6] on competition formulas and the time to optimality against the encoding approach on small synthetic formulas. Our evaluation supports both hypotheses: for the smaller formulas, our approach is capable of discovering optimal proofs faster (by orders of magnitude in many cases), whereas, for more realistic formulas, our approach discovers substantially shorter proofs than CaDiCaL can discover on its own.

There are two important caveats that are appropriate to place here. First, SAT solvers are designed to *quickly* produce proofs of unsatisfiability rather than producing *short* proofs; while the solver runtime and the proof lengths are closely related, shorter proofs do not automatically imply lower runtime. Second, CDCL solvers are more adequately modeled by the RUP proof system rather than resolution; while the difference between the two is at most polynomial in terms of the formula size, it does introduce some distortion into how proofs are evaluated. In particular, this impedance is the reason why we have to resort to manually processing CaDiCaL proofs, as described below in the paragraph about the reference proof construction. For a more complete discussion of the limitations stemming from choosing the resolution proof system, see also Section 6.1.

The implementation of our approach with the infrastructure for running the experiments is available in Online Appendix 1.[7] We ran our experiments on the *DelftBlue* (Delft High Performance Computing Centre 2024) supercomputer, with each experimental run being allocated a single core of an Intel Xeon E5-6248R 24C 3.0GHz processor and 16GB of RAM with a time limit of one hour.

5.1 Algorithm Configuration

Aside from the design choices described in Section 4, we make several other heuristic decisions for the algorithm components listed below:

Clause ordering strategy We order the clauses that can be derived in the branching strategy by descending clause length. We also considered the opposite order, but we observed that the descending clause length results in substantially smaller search trees and lower time to optimality. We believe this happens because ordering clauses by descending clause length puts the longest clauses—which are the weakest in the sense that they invalidate the fewest models—in the forgotten sets of most of the generated subproblems, and therefore does not waste effort on considering them any further.

SMUS switch-point value M_{switch} As mentioned in Section 4.5, we invoke Algorithm 2 instead of Forqes if the input has at most M_{switch} clauses; we use $M_{switch} = 36$, but we did not observe any meaningful difference from using slightly higher or lower values.

Last, depending on the use case, we use two different implementations for the `Complete` subroutine of Algorithm 1. In principle, this step can be implemented with any SAT solver that can emit a resolution proof. More precisely, we consider *optimality-focused* runs, where we seek to produce the shortest proof, and conclude that no shorter proofs exist, as fast as possible, and *length-focused* runs, where we look for the shortest possible proof without optimizing for faster time to solve the instance. The configurations that we use for these use cases are as follows:

[6] We used plain-text proofs in order to simplify the post-hoc processing described later in this section.
[7] Available at https://dx.doi.org/10.5281/zenodo.18852372.

- Optimality-focused configuration runs a custom implementation of the DPLL procedure (Davis, Logemann, et al. 1962; Davis and Putnam 1960) in `Complete`$(\cdot)$ calls, as we observe that it introduces less overhead on proofs with a few dozen clauses.
- Length-focused configuration runs CaDiCaL to obtain a proof in `Complete`$(\cdot)$ calls, leveraging its heuristics for fast unsatisfiability detection, which we expect to correlate with the production of shorter proofs.

5.2 Dataset Description

We run our evaluation on two collections of UNSAT formulas available in Online Appendix 2:[8]

- *Synthetic* instances are the formulas obtained from the procedure described hereafter in such a way that we can control the formula size.
- *Competition* instances are the 878 instances from all editions of SAT Competition from 2002 to 2025 (Biere 2025; C. Codel et al. 2025) that have been proven unsatisfiable by CaDiCaL within 15 seconds. We made this choice of benchmark formulas to assemble a collection of realistic resolution proofs, where most proofs can be reasonably processed by our approach and would thus give an adequate representation of its performance.

The synthetic part of the evaluation dataset comprises the following instances:

- *Pigeonhole principle* formulas for $1 \leq n \leq 6$ holes and $m = n + 1$ pigeons.
- *Parity principle* formulas for $(2n - 1)$ elements with $1 \leq n \leq 6$.
- *Ordering principle* formulas for $1 \leq n \leq 6$ elements.
- *Random 3-CNFs* with the number of variables n varying from 5 to 15 and the number of clauses m varying from $4n$ to $6n$.
- *Subset cardinality* formulas for random bipartite graphs with the part size n varying between 5 and 15.
- *Graph coloring* formulas encoding k-colorability of random $2(k - 1)$-regular graphs with k varying from 3 to 5 and number of vertices varying from $2k$ to $4k$.
- *Graph coloring with clique* formulas are the same but with a random $(k + 1)$-clique introduced in the graph, thereby guaranteeing the unsatisfiability of a formula.
- *Saturated minimally unsatisfiable* formulas obtained with the encoding approach for at most 9 clauses.

Our motivation behind this choice of instances was to include instance classes that have many small UNSAT formulas, are simple enough to generate, and are diverse enough to evaluate our approach in different situations.

To broaden the diversity of the benchmark distribution, we extend the synthetic instance set with minimally unsatisfiable subformulas (MUSes) derived from the original formulas. This extension is motivated by the empirical observation that MUS instances tend to be harder for SAT solvers than their parent formulas, suggesting that they represent a substantially different and more challenging distribution. Moreover, MUS inputs carry additional structural information for proof length minimization, as every clause must necessarily participate in any refutation.

Concretely, for each synthetic instance, we generate a MUS variant by invoking the MUSer2 package (Belov and Marques-Silva 2012) on the input formula. Instances that are already MUS in the original collection are not re-generated in order to avoid double-counting results.

For constructing the reference proofs, we use a procedure similar to the `Complete` subroutine described in Section 4.2, namely, we produce the proofs with the CaDiCaL 2.0 solver in UNSAT configuration, while requesting it to produce the proofs in LRAT format. The choice of the proof format was dictated by the fact that it contains not only derived clauses but also *the clauses used during the unit propagation*, which is essential for tracing the resolution sequence leading to a given learned clause. After that, we trim the proof emitted by CaDiCaL using

[8] Available at https://dx.doi.org/10.5281/zenodo.18851925.

LRAT-trim (Pollitt et al. 2023); this step provides an LRAT proof without redundant inferences, but it still needs to be encoded as a resolution proof.

To compute the trimmed proof length, we parse the proof, ignore the deletion lines, and simulate the resolution steps needed to derive a clause by going through the propagated clauses in reverse order, which is guaranteed to correspond to valid resolution steps due to the LRAT format structure. We also track the clauses derived in resolution steps *and skip the previously visited clauses*, regardless of whether they were mentioned in the proof file.

5.3 Evaluation on Synthetic Formulas

First, we discuss the synthetic instances depending on a single parameter; we report the results for pigeonhole principle formulas as a representative example, with the results for the others provided in Appendix C. Table 1 compares the proof lengths produced on these formulas by CaDiCaL with the proofs we have discovered with our approach in length-focused configuration, highlighting that even for these well-known formulas, there is some room for better search strategies. On the other hand, these results also show that the approach we propose is memory-intensive, as it ends up consuming substantially more memory than needed either to run a solver or to store its proof. (See also the discussion of Figure 9 on page 27.) In fact, our approach runs out of memory for $n = 3$, but not for any higher values; our explanation is that the pigeonhole principle formula with $n = 3$ is hard enough to warrant a heavy search, but simple enough for Algorithm 1 to post enough subproblems into the queue to exceed the memory limit.

Table 1. Proof lengths for pigeon-hole principle formulas. Values with asterisks are additionally proven to be optimal; OOM stands for "out of memory" and means that the run has exceeded its memory limit.

Number of pigeons n	1	2	3	4	5	6
Proof length, our approach	5*	19*	69	278	1368	10005
Proof length, CaDiCaL + LRAT-trim	5	19	74	334	1671	14439
Peak memory usage, megabytes	154	154	OOM	7423	7891	3158

Among the other synthetic formulas, the only classes that have been solved to optimality are saturated MUSes and random 3-CNFs. We thus proceed to compare the performance of our approach on these instances against the encoding approach. To this end, we compare (a) the sets of instances solved by either approach and (b) the time it took both approaches to solve an instance to optimality within the time limit.

For the number of solved instances, we observe that there is only *one* instance that our approach has failed to solve to optimality, but was solved by the baseline. On the other hand, our approach has solved 428 extra MUS instances on top of 396 MUS instances solved by both approaches. (The comparison on non-MUS instances is much more pronounced, with 18 instances solved by both approaches and 607 instances solved only with our approach.) Figure 5 shows the relative time to solve an instance for the samples solved by both approaches with respect to the number of clauses in a formula. The only instances where our approach had to take more time correspond to runs where both approaches finished within ≈ 10 seconds; for the remaining instances, differences in several orders of magnitude are not uncommon.

We have investigated the factors influencing the time to optimality, and our results suggest an exponential trend depending on the formula size and on the *MUS bound gap*, which for the case of MUS formulas is defined as $\#P^* - (2\#F - 1)$ for a MUS formula F with the shortest proof of unsatisfiability P^*; this quantity is the gap between the shortest proof and the MUS lower bound from Lemma 2. Figure 6 supports the claim about the MUS bound gap; as can be seen, the solving time scales exponentially with the increasing gap to the MUS bound.

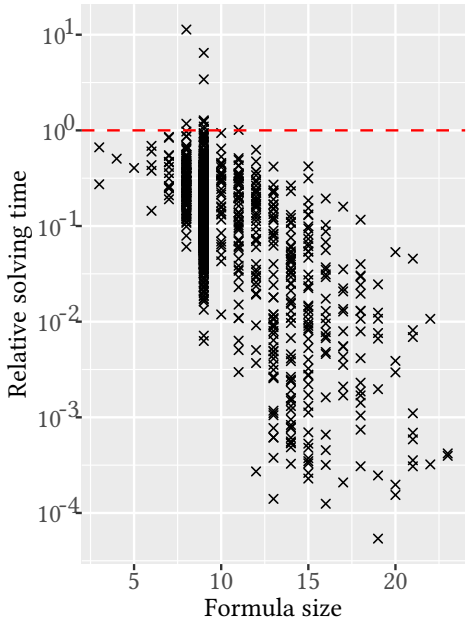


Fig. 5. Distribution of the ratio of solving time with our approach to the baseline solving time; lower is better. Data points correspond to the synthetic MUS instances solved by both approaches.

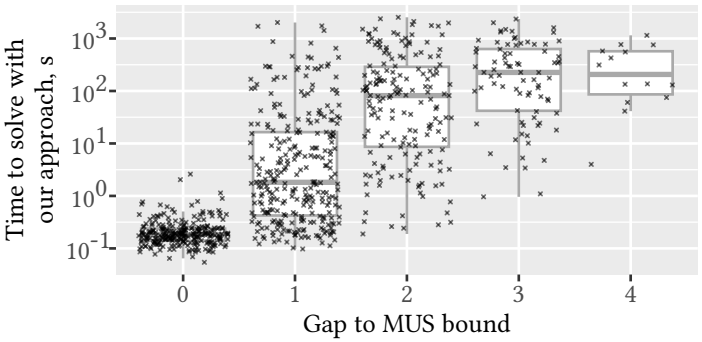


Fig. 6. Distribution of the solving time of our approach by gaps to the MUS bound. A horizontal jitter is added for clearer visualization.

We also observed that the instances with zero MUS bound gap are typically solved in an amount of time not dependent on the formula size. This is explained by the observation that the search is halted as soon as the best proof is discovered, regardless of the search tree structure; incidentally, this also justifies randomization in proof completion. On the other hand, the instances with a non-zero MUS gap can only be considered solved after all *subproblems* with the bound equal to the root subproblem bound are exhausted.

We have also compared the proof lengths across the synthetic formulas between our approach and CaDiCaL; Figure 7 summarizes this comparison. Our approach is consistently able to reduce the trimmed proofs by CaDiCaL by 25%, with some of the proofs being halved; the comparison is particularly striking for subset cardinality formulas, where our approach consistently finds reductions of at least 40%.

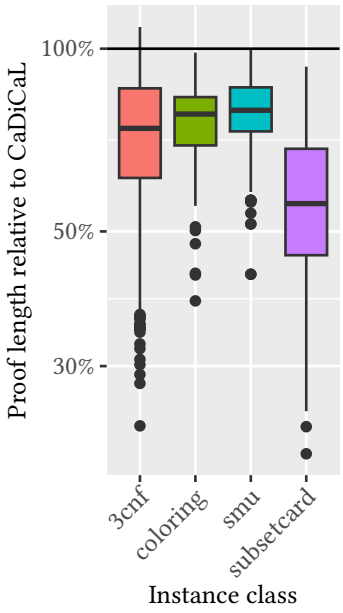


Fig. 7. Comparison of proof lengths produced by our approach with CaDiCaL proofs. Pigeonhole principle, parity principle, and ordering principle formulas are not shown, as there are only six of them from each class.

5.4 Evaluation on Competition Formulas

Finally, we evaluate the performance of our approach on the UNSAT instances from past editions of the SAT Competition; we use the length-focused version of the approach with several changes:

- We only consider the top ten clauses after applying the ordering criterion in branching (descending clause length), as it is unreasonable to expect the completeness of this procedure.
- We disable the pruning of the subproblems, as the lower-bound pruning is several orders of magnitude away from the discovered proof lengths at best and times out without any meaningful bound at worst.
- We limit the queue size to 10^4, discarding the subproblems with the largest "trivial" lower bound (length of the shortest clause), and earlier subproblems in case of ties.

First, *our approach typically*[9] *reduces the proof length by 15-50%.* This is shown in Figure 8 that compares the lengths of CaDiCaL proofs after trimming with the proof lengths discovered by our approach. The histogram also suggests that our approach can discover much bigger improvements on some formulas; this comparison contains samples with relative length larger than 100%, since our approach is not seeded with exactly the proof obtained by a dedicated CaDiCaL run with trimming. Appendix D provides the full list of formulas on which the

[9]We report the range between the first and third quartiles.

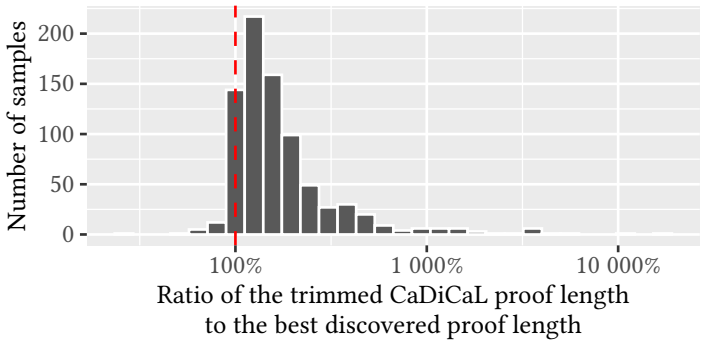


Fig. 8. Distribution of the ratios of proof lengths produced with our approach to CaDiCaL proof length; higher is better.

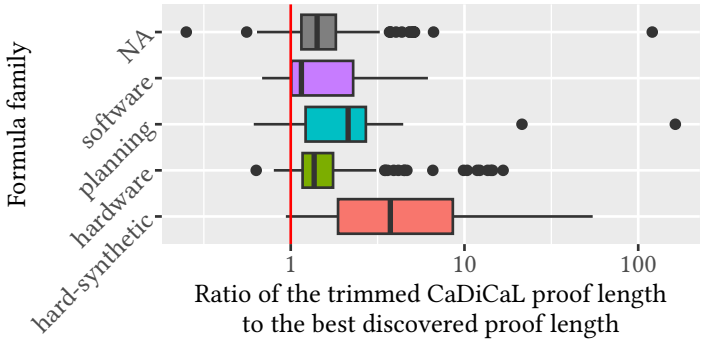


Fig. 9. Distribution of proof length improvements per family type; higher is better.

proof discovered by our approach is at least ten times shorter than its trimmed counterpart; for the purposes of this section, we observe only that a large part of them is classified in the Global Benchmark Database (Iser and Jabs 2024) as a Tseitin formula (Tseitin 1983), a class of formulas extensively studied in proof complexity.

This is not a coincidence: our next observation is that *certain families of UNSAT formulas translate into larger relative improvements* discovered by our approach. Figure 9 demonstrates the distributions of the relative proof length for various problem families; one of the takeaways is that the formulas that are known to be intractable for resolution consistently observe reductions ranging from twofold to ninefold ones, even after changing the focus from the positive outliers to the average case (more specifically, median and quartiles). Interestingly, while formulas related to software or hardware verification only observe improvements of similar scale to the full dataset, this is not the case for *planning* formulas, where the median reduction factor is twofold; in fact, this family also contains the largest reduction we observed: 163-fold[10] reduction on one of the formulas for multi-robot path planning (Surynek 2020). (Since the Global Benchmark Database tracks many formula families, we group them in a way that retains common provenance of the instances without finer details; see Appendix E for the mapping between the families on Figure 9 and the Global Benchmark Database families.)

[10]The trimmed proof contains over four million resolutions, whereas we discovered a proof with less than 25 thousand resolutions.

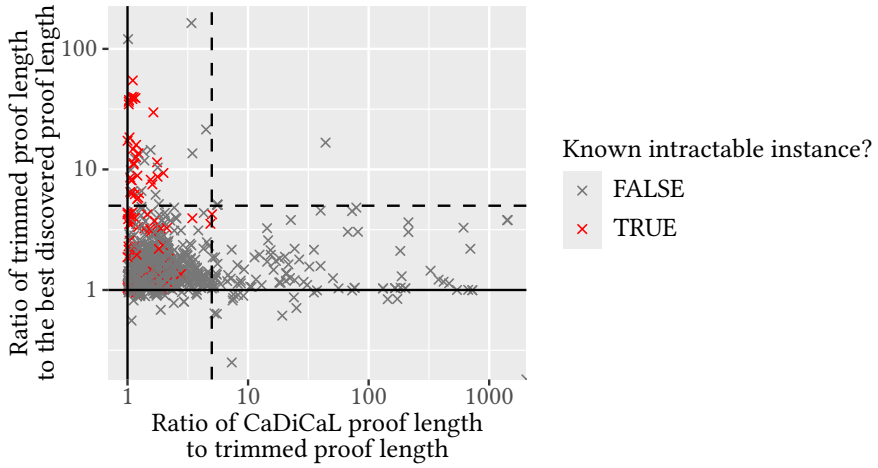


Fig. 10. Comparison of reduction factors from trimming (with respect to the original proof) and our approach (with respect to the trimmed proof). Solid lines correspond to no reduction, dashed lines correspond to five-fold reductions.

Last, we note that *a proof that has been substantially reduced by trimming is very unlikely to be further reduced with our approach, and vice versa.* This observation is supported by Figure 10, which shows for each formula two ratios between proof lengths:

- the ratio of the *original* CaDiCaL proof length to the length of the trimmed CaDiCaL proof, which shows the gain obtained *from* trimming the proof, and
- the ratio of the trimmed CaDiCaL proof length to the proof length discovered by our approach, which shows the gain obtained by our approach *after* trimming the proof.

As can be seen, there are only five cases where (i) trimming has reduced the proof length by at least five times, and (ii) our approach followed up with discovering at least a fivefold reduction *on top of the trimming*. Intractable formulas also appear here as an extreme case, where most of the solver proofs cannot be meaningfully trimmed, but discovering substantially shorter proofs with our approach "from scratch" is a typical behavior.

6 Reflection

Even though we believe our work provides useful insight for the practice of SAT solving, we consider it important to be clear about the areas of solver practice that our approach does not capture, so that the scope of our contribution is not misinterpreted. In this section, we outline the limitations of our work and discuss how the current results may help in addressing these.

6.1 Resolution as a Model of Solver Reasoning

The central focus of this paper is on resolution proofs. This choice is natural because (a) resolution is simpler to define when compared with the state-of-the-art proof formats, and (b) resolution corresponds closely to the inference steps of CDCL solvers without inprocessing. However, resolution is not an adequate model for many stronger techniques employed by state-of-the-art solvers. Common examples of non-resolution inference strategies include:

Satisfiability-preserving techniques The resolution rule is sound in the sense that the resolvent is true for any model satisfying its premises. However, in many instances, both artificially constructed and obtained

from practical problems, the unsatisfiability can be proved more succinctly once it is possible not only to add implied clauses but also transform a formula without loss of satisfiability. The reason for this is that allowing transformations that preserve satisfiability *makes it possible to draw inferences that can invalidate some models*, as long as they do not turn SAT formulas into UNSAT ones, with symmetry breaking (Crawford et al. 1996) serving as a model example of such an inference strategy.

Non-resolutional techniques There is a variety of formula classes, notably including pigeonhole principle formulas (Haken 1985), that are unsatisfiable and admit short "paper-and-pencil" proof of unsatisfiability, but only admit resolution proofs of exponential length (with respect to some size parameter). In computational terms, if a resolutional solver gets to refute such a formula (for example, as a subproblem in a larger formula), it is bound to spend an exponential amount of time to obtain an otherwise trivial UNSAT claim. Given that, modern solvers employ a broad range of inference rules that are exponentially more powerful than resolution, with common examples being pseudo-Boolean reasoning (Vinyals et al. 2018), decision diagram compilation (Sinz and Biere 2006), or Gaussian elimination (Soos et al. 2009).

Another reason why optimizing over resolution proofs is a limitation is that the behavior of a CDCL solver is more adequately described with a RUP (reverse unit propagation) proof system (M. J. H. Heule, Hunt, et al. 2013). These proof systems are equivalent in the following sense: any resolution inference step is also a valid RUP inference, whereas any clause with a RUP property can also be obtained by resolving the formula clause, using each at most once. However, for the purposes of analyzing the CDCL solver behavior, optimization over RUP proofs would be more suitable, as it would minimize the number of *conflict clauses*, rather than the number of *propagations*.

Our results should therefore be interpreted within the scope of solvers where resolution faithfully models inference, and not as a universal description of modern SAT solving practice. That said, we see our work as a stepping stone on the path towards designing proof length minimization approaches for state-of-the-art proof systems, including non-resolutional techniques.

6.2 Proof Formats and Reproducibility

A related limitation concerns the proof formats used in practice. In our experiments, we generate proofs using CaDiCaL, which produces DRAT proofs. Without any further knowledge, this would rule out such a use of CaDiCaL, because DRAT inference rules only provide the satisfiability-preserving guarantees (cf. the discussion of interference-based proof systems on page 7).

This means that it is only possible to use our approach if the SAT solver used in the `Complete`$(\cdot)$ subroutine *only invokes resolutional rules* in its proofs. To ensure this, the codebase distributed with this paper includes a distribution of CaDiCaL v2.0, which does not produce RAT clauses. That said, this limitation highlights the broader challenge of keeping proof-analysis methods aligned with evolving solver technologies.

6.3 Automatizability and Proof Search

Finally, some limitations are imposed by proof complexity rather than the developments of solver technology. Even if short proofs exist, there is no guarantee that they can be found efficiently. The notion of *automatizability* formalizes this gap: a proof system is automatizable if there exists an algorithm that can, given a formula, find a proof whose size is polynomially related to the shortest possible proof, in time polynomial in that size (Bonet et al. 2006). It is known that resolution is not automatizable under a range of plausible complexity assumptions (Atserias and Müller 2020; Mertz et al. 2019).[11] This entails that (a) finding shorter proofs is not merely a matter of designing better algorithms, but is constrained by fundamental barriers in proof complexity, and (b) even when some

[11]These results were predated by an earlier result by Alekhnovich et al. that the stronger version of this problem, namely, approximating up to a *linear* factor, is also intractable (Alekhnovich et al. 2001).

approach for proof length minimization (including ours) suggests the existence of short proofs for a family of formulas, this does not imply that there is an efficient way to reproduce that optimal proof structure within a SAT solver execution.

7 Conclusions

We propose a novel, anytime approach for constructing the shortest resolution proof of a given unsatisfiable propositional formula discovering increasingly tight upper bounds (shorter proofs) and lower bounds (higher under-estimates of the proof length). Our approach has derived substantially shorter proofs on a wide range of instances; it has also managed to prove optimality several orders of magnitude faster than the baseline approach for formulas with up to 45 clauses.

These results are the product of several algorithmic improvements described in the paper. First, we propose a layer list representation that is uniquely defined for any set of clauses constituting a proof; in addition, this representation has a stepwise structure, which serves as the foundation of the branch-and-bound algorithm. To support it, we propose two pruning procedures. The lower bound pruning reasons over MUSes of the current proof prefix, and the frontier pruning discards the subsumed clauses and backtracks once any such clause also happens to be unused in the proof.

Our approach relies on a variety of observations and algorithmic routines that could be further improved with a substantial effect. One of the key improvement points is the bounding, as it relies on a SMUS solver as the source of the lower bounds, which suggests that a tighter integration of the SMUS solver into the branch-and-bound procedure would improve the runtime even further. On the other hand, Lemma 2 itself limits the tightness of a bound: as this bound cannot exceed the size of the formula, it cannot prune subproblems equivalent to, for example, pigeonhole formulas. This indicates two alternative routes for improvement of the lower bound pruning, namely, deriving stronger relaxations than the one offered by Lemma 2 and adapting proof complexity arguments from known problem classes to arbitrary formulas.

This approach can also be applied to proof systems that do not operate on propositional formulas, such as various proof systems for tree-shaped proofs. However, the approach for discovering short proofs in that system, while empirically performant, does not carry any form of optimality guarantees. The main technical challenge in that line of work is in the bounding implementation, as it consumes the majority of time in the experimental runs, as well as takes advantage of the shortest proof structure from Lemma 2.

Acknowledgments

The team of authors would like to thank the JAIR reviewers for sharing in-depth, insightful comments on the earlier version of the text, as well as the attendees of *Pragmatics of SAT 2025*, including Armin Biere, Daniel Le Berre, and Marijn Heule, for a thoughtful Q&A session after the presentation of this work which helped improving the empirical evaluation of our approach.

Konstantin Sidorov is supported by the TU Delft AI Labs program as part of the XAIT lab.

References

M. Alekhnovich, S. Buss, S. Moran, and T. Pitassi. 2001. "Minimum propositional proof length is NP-hard to linearly approximate." *Journal of Symbolic Logic*, 66, 1, 171–191. doi:10.2307/2694916.

J. Altmanninger and A. Rebola Pardo. 2020. "Frying the egg, roasting the chicken: unit deletions in DRAT proofs." In: *Proceedings of the 9th ACM SIGPLAN international conference on certified programs and proofs*. ACM, New York, NY, USA. doi:10.1145/3372885.3373821.

A. Atserias and M. Müller. 2020. "Automating Resolution is NP-hard." *Journal of the ACM*, 67, 5, 1–17. doi:10.1145/3409472.

O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. 2009. "Linear-time reductions of resolution proofs." In: *Hardware and software: Verification and testing* (Lecture notes in computer science). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–128. doi:10.1007/978-3-642-01702-5_14.

A. Belov and J. Marques-Silva. 2012. "MUSer2: An efficient MUS extractor." *Journal on satisfiability, Boolean modeling and computation*, 8, 3-4, 123–128. doi:10.3233/sat190094.

A. Biere. 2025. *SAT Competition Benchmarks 2002 - 2024*. (2025). doi:10.5281/zenodo.15125952.

A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt. 2024. "CaDiCaL 2.0." In: *Computer Aided Verification* (Lecture notes in computer science). Ed. by A. Gurfinkel and V. Ganesh. Vol. 14681. Springer Nature Switzerland, Cham, 133–152. doi:10.1007/978-3-031-65627-9_7.

J. Blinkhorn and O. Beyersdorff. 2017. "Shortening QBF proofs with dependency schemes." In: *Theory and applications of satisfiability testing – SAT 2017* (Lecture notes in computer science). Vol. 10491. Springer International Publishing, Cham, 263–280. doi:10.1007/978-3-319-66263-3_17.

M. L. Bonet, T. Pitassi, and R. Raz. 2006. "On Interpolation and Automatization for Frege Systems." *SIAM Journal on Computing*. doi:10.1137/S0097539798353230.

J. Boudou, A. Fellner, and B. Woltzenlogel Paleo. 2014. "Skeptik: A Proof Compression System." In: *Automated reasoning* (Lecture notes in computer science). Vol. 8562. Springer International Publishing, Cham, 374–380. doi:10.1007/978-3-319-08587-6_29.

K. K. H. Cheung, A. Gleixner, and D. E. Steffy. 2017. "Verifying integer programming results." In: *Integer programming and combinatorial optimization* (Lecture notes in computer science). Ed. by F. Eisenbrand and J. Koenemann. Vol. 10328. Springer International Publishing, Cham, 148–160. doi:10.1007/978-3-319-59250-3_13.

C. Codel, K. Fazekas, M. J. H. Heule, and M. Iser. 2025. *Proceedings of SAT competition 2025: Solver and benchmark descriptions*. Tech. rep. doi:10.34726/10379.

C. R. Codel, J. Avigad, and M. J. H. Heule. 2024. "Verified Substitution Redundancy Checking." In: *2024 formal methods in computer-aided design (FMCAD)*. TU Wien, 186–196. doi:10.34727/2024/isbn.978-3-85448-065-5_24.

S. A. Cook. 1971. "The complexity of theorem-proving procedures." In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, New York, New York, USA, 151–158. doi:10.1145/800157.805047.

J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. 1996. "Symmetry-breaking predicates for search problems." In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning* (KR'96). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 148–159. doi:10.5555/3087368.3087386.

L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt Jr, M. Kaufmann, and P. Schneider-Kamp. 2017. "Efficient certified RAT verification." In: *Automated deduction – CADE 26* (Lecture notes in computer science). Ed. by L. de Moura. Vol. 10395. Springer International Publishing, Cham, 220–236. doi:10.1007/978-3-319-63046-5_14.

M. Davis, G. Logemann, and D. Loveland. 1962. "A machine program for theorem-proving." *Communications of the ACM*, 5, 7, 394–397. doi:10.1145/368273.368557.

M. Davis and H. Putnam. 1960. "A computing procedure for quantification theory." *Journal of the ACM*, 7, 3, 201–215. doi:10.1145/321033.321034.

Delft High Performance Computing Centre. 2024. *DelftBlue*. (2024).

S. S. Dey, Y. Dubey, M. Molinaro, and P. Shah. 2024. "A theoretical and computational analysis of full strong-branching." *Mathematical programming*, 205, 1-2, 303–336. doi:10.1007/s10107-023-01977-x.

M. D. Ernst, T. Millstein, and D. S. Weld. 1997. "Automatic SAT-compilation of planning problems." In: *IJCAI '97: IJCAI-97, proceedings of the fifteenth international joint conference on artificial intelligence*. Ed. by M. E. Pollack. IJCAI, 1169–1177.

J. K. Fichte, M. Hecher, and S. Szeider. 2020. "Breaking symmetries with RootClique and LexTopSort." In: *Principles and Practice of Constraint Programming* (Lecture notes in computer science). Ed. by H. Simonis. Vol. 12333. Springer International Publishing, Cham, 286–303. doi:10.1007/978-3-030-58475-7_17.

O. Flatt, S. Coward, M. Willsey, Z. Tatlock, and P. Panchekha. 2022. *Small proofs from congruence closure*. (2022). doi:10.34727/2022/ISBN.978-3-85448-053-2_13.

R. Gershman, M. Koifman, and O. Strichman. 2006. "Deriving small unsatisfiable cores with dominators." In: *Computer aided verification* (Lecture notes in computer science). Vol. 4144. Springer Berlin Heidelberg, Berlin, Heidelberg, 109–122. doi:10.1007/11817963_13.

J. Gorzny and B. Woltzenlogel Paleo. 2015. "Towards the compression of first-order resolution proofs by lowering unit clauses." In: *Automated deduction - CADE-25* (Lecture notes in computer science). Vol. 9195. Springer International Publishing, Cham, 356–366. doi:10.1007/978-3-319-21401-6_24.

A. Haken. 1985. "The intractability of resolution." *Theoretical computer science*, 39, 297–308. doi:10.1016/0304-3975(85)90144-6.

M. Heule and B. Kiesl. 2017. "The Potential of Interference-Based Proof Systems." In: *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements* (EPiC Series in Computing). Ed. by G. Reger and D. Traytel. Vol. 51. EasyChair, 51–54. doi:10.29007/vr7n.

M. J. H. Heule. 2021. "Proofs of unsatisfiability." In: *Handbook of Satisfiability*. Frontiers in artificial intelligence and applications. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. IOS Press, Amsterdam, Netherlands. Chap. 15, 635–668. doi:10.3233/faia200998.

M. J. H. Heule, W. A. Hunt, and N. Wetzler. 2013. "Trimming while checking clausal proofs." In: *2013 formal methods in computer-aided design*. Ed. by B. Jobstmann and S. Ray. IEEE, 181–188. doi:10.1109/fmcad.2013.6679408.

M. J. H. Heule, B. Kiesl, and A. Biere. 2017. "Short proofs without new variables." In: *Automated deduction – CADE 26* (Lecture notes in computer science). Vol. 10395. Springer International Publishing, Cham, 130–147. doi:10.1007/978-3-319-63046-5_9.

A. Ignatiev, A. Previti, M. Liffiton, and J. Marques-Silva. 2015. "Smallest MUS extraction with minimal hitting set dualization." In: *Principles and practice of constraint programming* (Lecture notes in computer science). Ed. by G. Pesant. Springer International Publishing, Cham, 173–182. doi:10.1007/978-3-319-23219-5_13.

M. Iser and C. Jabs. 2024. *Global Benchmark Database*. (2024). doi:10.4230/LIPICS.SAT.2024.18.

P. Lammich. 2017. "The GRAT Tool Chain." In: *Theory and applications of satisfiability testing – SAT 2017* (Lecture notes in computer science). Vol. 10491. Springer International Publishing, Cham, 457–463. doi:10.1007/978-3-319-66263-3_29.

L. Levin. 1973. "Universal sequential search problems." *Problems of information transmission*, 9, 3, 265–266.

P. Liberatore. 2005. "Redundancy in logic I: CNF propositional formulae." *Artificial intelligence*, 163, 2, 203–232. doi:10.1016/j.artint.2004.11.002.

C. Mencía and J. Marques-Silva. 2019. "Computing shortest resolution proofs." *Portuguese Conference on Artificial Intelligence*, 539–551. doi:10.1007/978-3-030-30244-3_45.

I. Mertz, T. Pitassi, and Y. Wei. 2019. *Short proofs are hard to find*. (2019). doi:10.4230/LIPICS.ICALP.2019.84.

G. Muñoz, J. Paat, and Á. S. Xavier. 2023. "Compressing branch-and-bound trees." In: *Integer Programming and Combinatorial Optimization* (Lecture notes in computer science). Ed. by A. Del Pia and V. Kaibel. Vol. 13904. Springer International Publishing, Cham, 348–362. doi:10.1007/978-3-031-32726-1_25.

T. Peitl and S. Szeider. 2021. "Finding the hardest formulas for resolution." *The journal of artificial intelligence research*, 72, 69–97. doi:10.1613/jair.1.12589.

F. Pollitt, M. Fleury, and A. Biere. 2023. *Faster LRAT checking than solving with CaDiCaL*. (2023). doi:10.4230/LIPICS.SAT.2023.21.

M. R. Prasad, A. Biere, and A. Gupta. 2005. "A survey of recent advances in SAT-based formal verification." *International journal on software tools for technology transfer: STTT*, 7, 2, 156–173. doi:10.1007/s10009-004-0183-4.

S. Prestwich and I. Lynce. 2006. "Local Search for Unsatisfiability." In: *Theory and applications of satisfiability testing - SAT 2006* (Lecture notes in computer science). Ed. by A. Biere and C. P. Gomes. Vol. 4121. Springer Berlin Heidelberg, Berlin, Heidelberg, 283–296. doi:10.1007/11814948_28.

A. Rebola-Pardo. 2023. *Even shorter proofs without new variables*. (2023). doi:10.4230/LIPICS.SAT.2023.22.

A. Rebola-Pardo and A. Biere. 2018. "Two flavors of DRAT." *POS@ SAT*, 94–110. doi:10.29007/NNQS.

J. A. Robinson. 1965. "A machine-oriented logic based on the resolution principle." *Journal of the ACM*, 12, 1, 23–41. doi:10.1145/321250.321253.

S. F. Rollini, R. Bruttomesso, and N. Sharygina. 2011. "An efficient and flexible approach to resolution proof reduction." In: *Hardware and software: Verification and testing* (Lecture notes in computer science). Vol. 6504. Springer Berlin Heidelberg, Berlin, Heidelberg, 182–196. doi:10.1007/978-3-642-19583-9_17.

S. F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich. 2014. "Resolution proof transformation for compression and interpolation." *Formal methods in system design*, 45, 1, 1–41. doi:10.1007/s10703-014-0208-x.

K. Sidorov, G. H. d. A. Correia, M. de Weerdt, and E. Demirović. 2024. "Paths, Proofs, and Perfection: Developing a Human-Interpretable Proof System for Constrained Shortest Paths." In: *Proceedings of the 38th AAAI conference on artificial intelligence*. Ed. by M. Wooldridge, J. Dy, and S. Natarajan. Vol. 38, 20794–20802. doi:10.1609/aaai.v38i18.30068.

C. Sinz and A. Biere. 2006. "Extended resolution proofs for conjoining BDDs." In: *Computer science – theory and applications* (Lecture Notes in Computer Science). Vol. 3967. Springer Berlin Heidelberg, Berlin, Heidelberg, 600–611. doi:10.1007/11753728_60.

M. Soos, K. Nohl, and C. Castelluccia. 2009. "Extending SAT solvers to cryptographic problems." In: *Theory and applications of satisfiability testing - SAT 2009* (Lecture Notes in Computer Science). Vol. 5584. Springer Berlin Heidelberg, Berlin, Heidelberg, 244–257. doi:10.1007/978-3-642-02777-2_24.

P. Surynek. 2020. "Bounded sub-optimal multi-robot path planning using satisfiability modulo theory (SMT) approach." In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 11631–11637. doi:10.1109/iros45743.2020.9341047.

G. S. Tseitin. 1983. "On the complexity of derivation in propositional calculus." In: *Automation of Reasoning*. Ed. by J. H. Siekmann and G. Wrightson. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. doi:10.1007/978-3-642-81955-1_28.

M. Vinyals, J. Elffers, J. Giráldez-Cru, S. Gocht, and J. Nordström. 2018. "In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving." In: *Theory and applications of satisfiability testing – SAT 2018* (Lecture notes in computer science). Vol. 10929. Springer International Publishing, Cham, 292–310. doi:10.1007/978-3-319-94144-8_18.

N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. 2014. "DRAT-trim: efficient checking and trimming using expressive clausal proofs." In: *Theory and applications of satisfiability testing - SAT 2014* (Lecture notes in computer science). Ed. by C. Sinz and U. Egly. Vol. 8561. Springer International Publishing, Cham, 422–429. doi:10.1007/978-3-319-09284-3_31.

H. Zhang. 2021. "Combinatorial designs by SAT solvers." In: *Handbook of Satisfiability*. Frontiers in artificial intelligence and applications. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. IOS Press, Amsterdam, Netherlands. Chap. 21, 819–858. doi:10.3233/FAIA201005.

A Example of Permutational Symmetry Breaking

As mentioned in the related work survey, specifically on page 8 in Section 3.3, state-of-the-art approaches do not break all permutational symmetries. This appendix demonstrates a complete example of a proof structure that is not recognized as symmetric by LexTopSort.

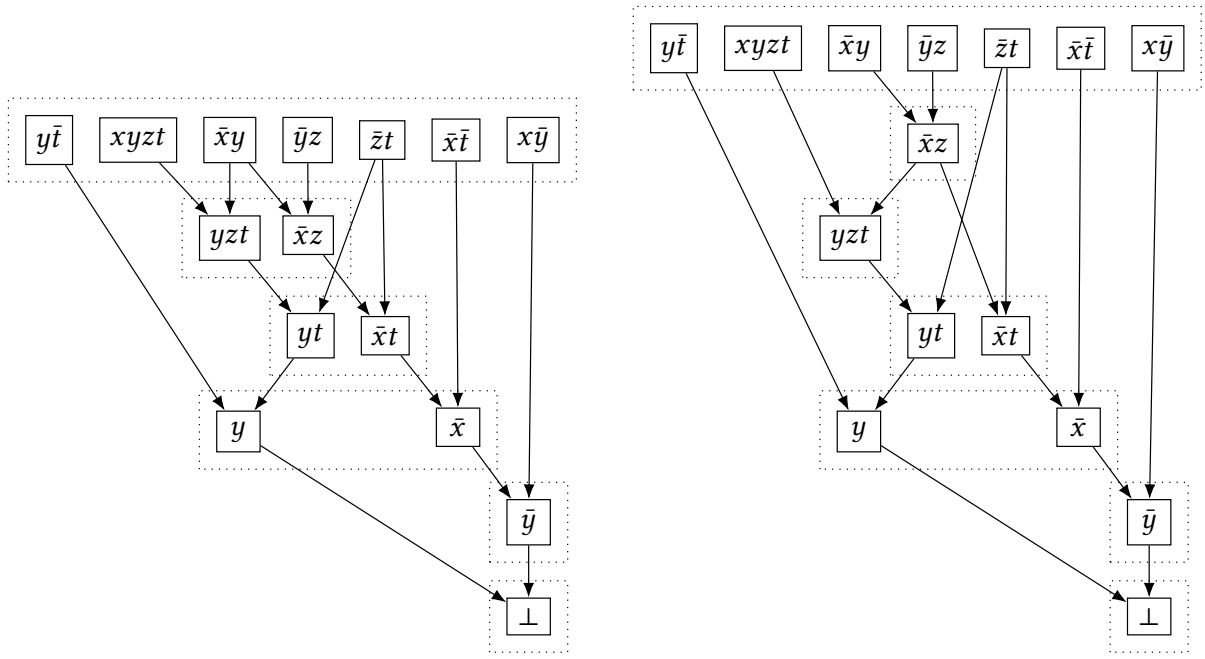
Consider an UNSAT formula

$$F = \left\{ y\bar{t}, xyzt, \bar{x}y, \bar{y}z, \bar{z}t, \bar{x}\bar{t}, x\bar{y} \right\}$$

and a proof encoded by the sequence starting with the seven axioms and deriving clauses

$$\left(\bar{x}z, yzt, \bar{x}t, yt, y, \bar{x}, \bar{y}, \bot \right).$$

This proof can be encoded by *two distinct* DAGs using only the derived clauses represented in Figure 11; the only difference is that the clause yzt is derived as $xyzt \diamond \bar{x}y$ in the left DAG and as $xyzt \diamond \bar{x}z$ in the right DAG.



(a) Proof DAG with $xyzt \diamond \bar{x}y$.

(b) Proof DAG with $xyzt \diamond \bar{x}z$.

Fig. 11. Two proof DAGs with identical vertices but different edge structures. Every row on both figures corresponds to one layer.

Now consider the LexTopSort constraint with lexicographic ordering $x < \bar{x} < y < \bar{y} < z < \bar{z} < t < \bar{t}$; it admits the following two canonical sequences:[12]

- $xyzt, x\bar{y}, \bar{x}y, \bar{x}t, \boldsymbol{yzt}, \ldots$ for the left DAG, and
- $xyzt, x\bar{y}, \bar{x}y, \bar{x}t, \boldsymbol{\bar{y}z}, \ldots$ for the right DAG.

[12]For brevity, we show only the first five clauses of each sequence. The suffixes are irrelevant, since no continuation makes the two sequences coincide.

Therefore, although LexTopSort eliminates symmetries from permuting a single DAG, it cannot remove the redundancy arising from distinct DAGs encoding the same proof.

On the other hand, the layer list definition allows the former structure without allowing the latter one as well: the difference is exposed by the clause $yzt = xyzt \diamond \bar{x}y$, which is not an axiom but can be derived by resolving two axioms. Therefore, the take-it-or-leave-it property pins it to the layer L^1 of the layer list.

B Proofs

This section fills in the proofs of the claims made throughout the paper text; Section B.1 contains the formulations and proofs of branching rules used in our approach, and Section B.2 proves various rules for pruning the search tree.

B.1 Branching Rules

Lemma 5. *Let $\mathcal{P}$ be a subproblem with $\texttt{Current}[\mathcal{P}] \neq \emptyset$, and suppose that $(\omega_1, \ldots, \omega_m)$ is the sequence of clauses such that*

$$\{\omega_1, \ldots, \omega_m\} = \{\omega = \omega' \diamond \omega'' \mid \omega' \in \texttt{Current}[\mathcal{P}], \omega'' \in \texttt{Current}[\mathcal{P}] \cup \texttt{Previous}[\mathcal{P}], \\ \omega \notin \texttt{Forgotten}[\mathcal{P}], \omega \notin \texttt{Known}[\mathcal{P}]\},$$

in other words, the sequence $\omega_j, 1 \leq j \leq m$ lists all derivable clauses and skips forgotten clauses and clauses that are already available in the proof. Then $\texttt{Partition}(\mathcal{P}, \cdot) := \{\mathcal{P}'_1, \ldots, \mathcal{P}'_m, \mathcal{P}^{commit}\}$ complies with Proposition 2.1 for

$$\begin{aligned} \texttt{Previous}[\mathcal{P}'_j] &= \texttt{Previous}[\mathcal{P}], \\ \texttt{Current}[\mathcal{P}'_j] &= \texttt{Current}[\mathcal{P}], \\ \texttt{Next}[\mathcal{P}'_j] &= \texttt{Next}[\mathcal{P}] \cup \{\omega_j\}, \\ \texttt{Forgotten}[\mathcal{P}'_j] &= \texttt{Forgotten}[\mathcal{P}] \cup \{\omega_1, \ldots, \omega_{j-1}\} \end{aligned}$$

and

$$\begin{aligned} \texttt{Previous}[\mathcal{P}^{commit}] &= \texttt{Previous}[\mathcal{P}] \cup \texttt{Current}[\mathcal{P}], \\ \texttt{Current}[\mathcal{P}^{commit}] &= \texttt{Next}[\mathcal{P}], \\ \texttt{Next}[\mathcal{P}^{commit}] &= \emptyset, \\ \texttt{Forgotten}[\mathcal{P}^{commit}] &= \texttt{Forgotten}[\mathcal{P}] \cup \{\omega_1, \ldots, \omega_m\}. \end{aligned}$$

Proof. Observe that

$$\texttt{Known}[\mathcal{P}'_j] = \texttt{Known}[\mathcal{P}] \cup \{\omega_j\}$$

and

$$\texttt{Known}[\mathcal{P}^{commit}] = \texttt{Known}[\mathcal{P}].$$

Suppose that L is a compatible proof of $\mathcal{P}$ that starts with the clauses from $\texttt{Known}[\mathcal{P}]$; the remainder of the proof depends on whether L uses any ω_j clauses.

If no such clauses are produced in L, then L is also a compatible proof for $\mathcal{P}^{commit}$, because extending the forgotten set does not impact any further decisions. However, $\texttt{Known}[\mathcal{P}^{commit}]$ only differs from $\texttt{Known}[\mathcal{P}]$ by pushing some of the clauses "backward". As some prefix of L agrees with $\mathcal{P}$, it must also agree with $\mathcal{P}^{commit}$.

Otherwise, let ω_j be the clause used in L with the smallest index j; by take-it-or-leave-it property, that means that the clauses $\omega_1, \ldots, \omega_{j-1}$ are not used in the proof. Then L' is a compatible proof of $\mathcal{P}_j$, as it starts with the same prefix (up to the current layer) and does not use clauses from the set $\{\omega_1, \ldots, \omega_{j-1}, \omega_j\}$, which is exactly the definition of $\mathcal{P}_j$. □

The branching rule stated in Lemma 5 is only applicable to the case where the axiom layer has been *completed*, which is, for example, not the case for the root subproblem $\mathcal{P}_{\text{root}}$. This case has two differences: first, the set of available clauses is given explicitly by the input formula, and second, the selected set of clauses has to be UNSAT, because for any proof P the formula $F \cap P$ is UNSAT. To account for this, we first enforce adding all critical clauses, that is, clauses whose removal makes the formula SAT:

Lemma 6. *Let $\mathcal{P}$ be a subproblem with $\texttt{Current}[\mathcal{P}] = \emptyset$, and suppose that F' is the set of non-forgotten axioms: $F' := F \setminus \texttt{Forgotten}[\mathcal{P}]$. Then $\texttt{Partition}(\mathcal{P}, F) := \{\mathcal{P}^{\text{crit}}\}$ complies with Proposition 2.1 for*

$$\begin{aligned}\texttt{Previous}[\mathcal{P}^{crit}] &= \emptyset,\\ \texttt{Current}[\mathcal{P}^{crit}] &= \emptyset,\\ \texttt{Next}[\mathcal{P}^{crit}] &= \texttt{Next}[\mathcal{P}] \cup \Omega^{\text{crit}},\\ \texttt{Forgotten}[\mathcal{P}^{crit}] &= \texttt{Forgotten}[\mathcal{P}],\end{aligned}$$

where Ω^{crit} is the set of critical *clauses of F', that is, the clauses ω such that $F' \setminus \{\omega\}$ is SAT.*

Proof. Suppose that Proposition 2.1 does not hold for some subproblem with $\texttt{Current}[\mathcal{P}] = \emptyset$, that is, that there is a proof L that is compatible with $\mathcal{P}$ but not with $\mathcal{P}^{\text{crit}}$. That would imply that L contains all axioms from $\texttt{Next}[\mathcal{P}]$ but does not contain one of the axioms ω from $\texttt{Next}[\mathcal{P}^{\text{crit}}]$. That is only possible if $\omega \in \Omega^{\text{crit}}$, which, in turn, means that L is a proof of the formula $F' \setminus \{\omega\}$; that contradicts the construction of Ω^{crit} as the set of all *critical* clauses. □

Once all critical clauses are pinned, we branch on forgetting the remaining clauses as before.

Lemma 7. *Let $\mathcal{P}$ be a subproblem with $\texttt{Current}[\mathcal{P}] = \emptyset$, and suppose that $\omega_1, \ldots, \omega_m$ is the sequence of axioms that are neither forgotten nor selected:*

$$\{\omega_1, \ldots, \omega_n\} = F \setminus \Big\{\texttt{Forgotten}[\mathcal{P}] \cup \texttt{Next}[\mathcal{P}]\Big\}.$$

Then $\texttt{Partition}(\mathcal{P}, F) := \{\mathcal{P}'_1, \ldots, \mathcal{P}'_m, \mathcal{P}^{commit}\}$ complies with Proposition 2.1 for

$$\begin{aligned}\texttt{Previous}[\mathcal{P}'_j] &= \emptyset,\\ \texttt{Current}[\mathcal{P}'_j] &= \emptyset,\\ \texttt{Next}[\mathcal{P}'_j] &= \texttt{Next}[\mathcal{P}] \cup \{\omega_j\},\\ \texttt{Forgotten}[\mathcal{P}'_j] &= \texttt{Forgotten}[\mathcal{P}] \cup \{\omega_1, \ldots, \omega_{j-1}\}\end{aligned}$$

and

$$\begin{aligned}\texttt{Previous}[\mathcal{P}^{commit}] &= \emptyset,\\ \texttt{Current}[\mathcal{P}^{commit}] &= \texttt{Next}[\mathcal{P}],\\ \texttt{Next}[\mathcal{P}^{commit}] &= \emptyset,\\ \texttt{Forgotten}[\mathcal{P}^{commit}] &= \texttt{Forgotten}[\mathcal{P}] \cup \{\omega_1, \ldots, \omega_m\}.\end{aligned}$$

Proof. As in the proof of Lemma 5, observe that if L is a compatible proof for $\mathcal{P}$, then the axiom layer is going to contain the axioms populating the known set: $L^0 \supseteq \texttt{Known}[\mathcal{P}]$. If $L^0 = \texttt{Known}[\mathcal{P}]$, then it is also compatible with $\mathcal{P}^{\text{commit}}$ because it has the same known set and it does not use clauses from $\texttt{Forgotten}[\mathcal{P}]$. As for the new forgotten clauses $F \setminus \Big(\texttt{Forgotten}[\mathcal{P}] \cup L^0\Big)$, they are not used in L, as all of them are axioms that are not already added in L^0.

Otherwise, let ω_j be the axiom used in L with the smallest index j, which means that the axioms $\omega_1, \ldots, \omega_{j-1}$ are not used in the proof. Then L' is a compatible proof of $\mathcal{P}_j$, as uses the axioms from $\texttt{Next}[\mathcal{P}] \cap \{\omega_j\}$ and does not use clauses from the set $\{\omega_1, \ldots, \omega_{j-1}, \omega_j\}$, which is exactly the definition of $\mathcal{P}_j$. □

B.2 Pruning and Dominance

Proof of Theorem 3. Let $(Q_j = L_j \diamond R_j)_{j=1}^m$ be the suffix proof of F following the introduction of axioms. We describe how to construct a proof $(Q'_j = L'_j \diamond R'_j)_{j=1}^m$ of $\texttt{Frontier}[F]$ such that $Q'_j \subseteq Q_j$. As the last clause Q_K of the original proof is empty, so is the last clause Q'_K of the frontier proof.

Given a clause L_j of the original proof, let $U_j \subseteq L_j$ be a clause from the frontier proof included in the corresponding premise of the original proof. More specifically, $U_j \in_{\mathcal{F}} F$ if $L_j \in F$ and $U_j = Q'_\ell$ if $U_j = Q_\ell$ for $\ell < j$. Similarly, let $V_j \subseteq R_j$ be a corresponding clause for another premise of the same step. (Note that this definition only depends on the clauses from the original formula and earlier derivation steps, which leads to a proof by induction.)

Suppose that the pivot literal is present in both premises U_j and V_j; in that case, any literal in $U_j \diamond V_j$ is contained either in $U_j \subseteq L_j$ or in $V_j \subseteq R_j$, meaning that $U_j \diamond V_j \subseteq L_j \diamond R_j = Q_j$. Thus, in that case setting $(L'_j, R'_j) = (U_j, V_j)$ is sufficient to ensure $Q'_j \subseteq Q_j$.

On the other hand, if the pivot literal is missing in (without loss of generality) U_j, deriving $Q'_j = U_j$ is sufficient to ensure $Q'_j \subseteq Q_j$. If $U_j \in_{\mathcal{F}} F$, then we can introduce it with a bogus derivation $U_j \diamond U_j = U_j$. Otherwise, U_j has already been derived by the j-th step by definition of U-clauses, meaning that we can copy this derivation into the j-th step. □

Proof of Lemma 2. Observe that any compatible proof of $\mathcal{P}$ can be seen as a proof of unsatisfiability of some subset of clauses $S \subseteq_{\mathcal{F}} \texttt{Known}[\mathcal{P}]$, which means that the lowest bound among the bounds for $S \subseteq_{\mathcal{F}} \texttt{Known}[\mathcal{P}]$ is a correct bound for any "remaining" proof length for $\mathcal{P}$. Given that *all* of the clauses in S are used in the proof, we can use the proof of Lemma 1 to show that any such proof has length at least $\#S - 1$. We complete the proof by taking the most optimistic of discovered bounds, which corresponds to minimizing $\#S - 1$ over UNSAT subsets of $\texttt{Frontier}[\texttt{Known}[\mathcal{P}]]$, just as in the lemma statement. □

Proof of Lemma 3. For an unused clause $\omega \in U$, retrace the branching steps needed to reach the subproblem $\mathcal{P}$ and forget ω instead of deriving it. The resulting problem can be used as $\mathcal{P}'$, and omitting the derivation for ω yields its compatible proof L'. □

Proof of Lemma 4. Consider an arbitrary UNSAT subset $G \subseteq F$ containing U, and observe that we can derive the lemma statement as soon as we show that $\#G \geq \#U + k$ holds regardless of the choice of G. Let $H := G \setminus U$ and $h := \#H$, and assume without loss of generality that the clauses of H are ordered by ascending clause length as follows:

$$H = \{\pi_1, \ldots, \pi_h\}, \#\pi_1 \leq \cdots \leq \#\pi_h.$$

Now, let n be the number of variables in G, and consider the following inequality for sets of models θ of G:

$$\sum_{\omega \in G} \#\Big\{\theta : \theta \text{ falsifies } \omega\Big\} \geq \# \bigcup_{\omega \in G} \Big\{\theta : \theta \text{ falsifies } \omega\Big\}. \tag{4}$$

Since the right-hand side of Eq. (4) is the number of models falsifying the UNSAT formula G, it is equal to 2^n; on the other hand, each clause ω is falsified by exactly $2^{n-\#\omega}$ models. We thus obtain the bound $\sum_{\omega \in G} 2^{n-\#\omega} \geq 2^n$, which after dividing by 2^n and splitting the sum over U from the sum over H becomes

$$\sum_{\omega \in U} 2^{-\#\omega} + \sum_{j=1}^{h} 2^{-\#\pi_j} \geq 1. \tag{5}$$

Recall that the clauses $\omega_1, \ldots, \omega_h$ are the shortest h clauses in $F \setminus U$, therefore, $\#\pi_j \geq \#\omega_j$ and $2^{-\#\pi_j} \leq 2^{-\#\omega_j}$ for all j from 1 to h. We use this to bound the sum over H in Eq. (5) from above as

$$\sum_{\omega \in U} 2^{-\#\omega} + \sum_{j=1}^{h} 2^{-\#\omega_j} \geq 1.$$

But this is exactly the expression from Eq. (3), which, as we know from the lemma assumptions, can only be greater than or equal to one if $h \geq k$. □

C Results on Parametric Formulas

Similarly to the pigeonhole principle results reported on page 24, we report in this appendix the same results for parity principle and ordering principle formulas.

Table 2. Proof lengths for parity principle formulas. Values with asterisks are additionally proven to be optimal.

Number of elements n	1	2	3	4	5
Proof length, our approach	11*	81	509	4475	56007
Proof length, CaDiCaL + LRAT-trim	11	80	597	5319	66375
Peak memory usage, megabytes	144	OOM	9409	8756	3238

Table 3. Proof lengths for ordering principle formulas. Values with asterisks are additionally proven to be optimal.

Number of elements n	1	2	3	4	5
Proof length, our approach	5*	16*	39	80	194
Proof length, CaDiCaL + LRAT-trim	5	16	42	79	143
Peak memory usage, megabytes	154	144	OOM	8381	13195

Table 4. Proof lengths for MUSes of ordering principle formulas. Values with asterisks are additionally proven to be optimal.

Number of elements n	1	2	3	4	5	6
Proof length, our approach	5*	16*	40	79	273	254
Proof length, CaDiCaL + LRAT-trim	5	16	67	218	527	1677
Peak memory usage, megabytes	154	144	OOM	7538	8525	9845

D List of Competition Formulas with Large Reductions of Proof Length

This appendix lists the formulas for which our approach yields the largest reductions in proof length. Table 5 summarizes all cases exhibiting reductions by a factor of ten or greater.

Table 5. Competition formulas exhibiting at least tenfold reductions in proof length.

Formula file name	GBD family	Trimmed proof length	Proof length discovered with our approach
rsdecoder6.dimacs.filtered	design-debugging	6275	376
fpga_routing-example2_gr_2pin_w5.shuffled	fpga-routing	48236	3399
fpga_routing-vda_gr_rcs_w7.shuffled	fpga-routing	1089827	88968
fpga_routing-apex7_gr_2pin_w4.shuffled	fpga-routing	2758	233
fpga_routing-term1_gr_2pin_w3.shuffled	fpga-routing	705	68
manol-pipe-g7nidw	hardware-verification	2803242	193209
c10bidw_s	hardware-verification	899913	66187
gt-012.shuffled-as.sat05-1301	ordering-principle	18170	1138
gt-016.shuffled-as.sat05-1291	ordering-principle	34109	2978
mrpp_8x8#24_11	planning	4071528	24888
mrpp_8x8#22_10	planning	110750	5168
ezfact16_10.shuffled	prime-factoring	10480	87
urquhart2_25.shuffled	tseitin-formulas	525088	9597
Urquhart-s3-b3.shuffled-as.sat03-1556	tseitin-formulas	14204699	355322
urqh1c3x3.shuffled-as.sat03-1463	tseitin-formulas	3043357	76581
marg3x4.shuffled-as.sat03-1451	tseitin-formulas	2813454	72032
bevhcube4.shuffled-as.sat03-1426	tseitin-formulas	614203	16022
urqh2x3.shuffled-as.sat03-1471	tseitin-formulas	1195430	31766
Urquhart-s3-b9.shuffled	tseitin-formulas	305889	8399
marg3x3add8ch.shuffled-as.sat03-1448	tseitin-formulas	3423387	98803
mod2-rand3bip-unsat-90-1.shuffled-as.sat05-2594	tseitin-formulas	1008880	33918
Urquhart-s3-b7.shuffled	tseitin-formulas	3219879	174773
marg2x5.shuffled-as.sat03-1443	tseitin-formulas	84324	4881
urqh3x3.shuffled-as.sat03-1476	tseitin-formulas	17700437	1200860
mod2-rand3bip-unsat-105-1.shuffled-as.sat05-2609	tseitin-formulas	1356390	99693
Urquhart-s3-b2.shuffled	tseitin-formulas	884390	78489
urqh1c3x4.shuffled-as.sat03-1464	tseitin-formulas	7579040	691788
x1_24.shuffled	xor-chain	369434	29201

E Mapping Between Formula Families Used in the Paper and Used in the Global Benchmark Database

In the discussion of improvements by formula groups in Figure 9, we use the following mapping from Global Benchmark Database families to our ad-hoc grouping:

Hard synthetic This group comprises `xor-chain`, `tseitin-formulas`, `ordering-principle`, `sgen`, `pigeon-hole`, and `mutilated-chessboard` families.

Planning This group directly corresponds to the `planning` family.

Hardware This group comprises `theorem-proving`, `design-debugging`, `hardware-verification`, `hardware-bmc`, `miter`, and `fpga-routing` families.

Software This group comprises `bitvector`, `bounded-model-checking`, `software-verification`, and `software-bmc` families.

Received 03 March 2026; accepted 10 June 2026