Problem Partitioning via Proof Prefixes

Zachary Battleman
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Joseph E. Reeves
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

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

Abstract

Satisfiability solvers have been instrumental in tackling hard problems, including mathematical challenges that require years of computation. A key obstacle in efficiently solving such problems lies in effectively partitioning them into many, frequently millions of subproblems. Existing automated partitioning techniques, primarily based on lookahead methods, perform well on some instances but fail to generate effective partitions for many others.

This paper introduces a powerful partitioning approach that leverages prefixes of proofs derived from conflict-driven clause-learning solvers. This method enables non-experts to harness the power of massively parallel SAT solving for their problems. We also propose a semantically-driven partitioning technique tailored for problems with large cardinality constraints, which frequently arise in optimization tasks. We evaluate our methods on diverse benchmarks, including combinatorial problems and formulas from SAT and MaxSAT competitions. Our results demonstrate that these techniques outperform existing partitioning strategies in many cases, offering improved scalability and efficiency.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability solving, parallel computing, problem partitioning

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.3

Supplementary Material *Software*: `https://github.com/zaxioms0/proofix`
archived at `swh:1:dir:021ba6e9dc38762dc28c4ecf9df973344138898d`

Funding Supported by the National Science Foundation (NSF) under grant CCF-2415773 and funding from AFRL and DARPA under Agreement FA8750-24-9-1000.
Joseph E. Reeves: Supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

1 Introduction

Satisfiability (SAT) solvers have proven to be invaluable tools for solving large problems of interest to both theorists and industry practitioners. Over the last decade and a half, substantial efforts have focused on parallelizing SAT, leading to two prominent paradigms: clause-sharing portfolios [11] and cube-and-conquer (CnC) [14]. In a clause-sharing portfolio, multiple solver threads are run on the same input formula, typically with slightly varied heuristics, and critical clauses are shared among them. This approach can significantly reduce runtime for problems where sequential SAT methods are already effective, as illustrated by the state-of-the-art cloud solver Mallob [29]. In contrast, a CnC solver partitions the input formula into numerous subproblems that can be solved independently in parallel. This strategy has successfully tackled longstanding open problems in mathematics, including the Pythagorean Triples problem [13], Schur Number Five [12], and the Empty Hexagon problem [15].

Cube-and-conquer comprises two main phases: *cubing* and *solving*. In the cubing phase, a splitting variable is chosen to partition the current formula into two subformulas, one with the variable assigned to `true` and the other with the variable assigned to `false`. Ideally, each split produces two distinctly simpler subproblems. Historically, lookahead techniques proved remarkably effective at selecting splitting variables, often enabling superlinear speedups even on thousands of cores [14]. However, poorly chosen splits can result in substantial redundant work or significant imbalances in subproblem difficulty, which in turn may lead to diminished parallel efficiency or total running times that exceed the time required to solve the original formula.

Although early advances with CnC relied on automated partitioning via lookaheads [14], most of the significant successes in the last five years have depended on expert-crafted manual partitions [6,15,32,34]. Manual approaches were favored primarily due to the prohibitive costs or limited effectiveness of automated partitioning with lookaheads. The manual approaches used a combination of trial-and-error and high-level insight into the problem and its encoding. This prevents many potential users of CnC to solve their problems effectively. We propose two novel partitioning methods to overcome these issues.

Our first cubing approach builds on the information contained in clausal proofs produced by SAT solvers. A *clausal proof* is a sequence of redundant clauses (i.e., clauses whose addition preserves satisfiability) ending with the empty clause to prove unsatisfiability. Proofs have been used to assess and compare the usefulness of learned clauses enabling proof summarization [27], solver heuristic tuning [31], and causal reasoning over combinations of solver heuristics [37]. These works rely on access to complete proofs and strictly evaluate clause usefulness within the proofs. A central insight of this paper is that *prefixes* of clausal proofs can serve as effective stand-ins for complete proofs, and that the variables occurring in these proof prefixes provide a powerful heuristic for guiding partitioning decisions.

Although splitting based on proof prefixes turns out to be effective, it can be computationally expensive. To mitigate this cost, our method relies on *static* partitions rather than the *dynamic* partitions typically used in prior work. In a static partition, the same splitting variable is used at every level of the partition tree. For deeper levels, we compute proof prefixes for only a subset of nodes, then aggregate these prefixes to select the next splitting variable.

Despite the inherent restrictions of a static partition, our results show that it performs well across a broad range of problems, including benchmarks from the SAT Competition. In fact, our tool efficiently generates strong partitions for many formulas on which the state-of-the-art partitioning tool, March [14], fails to produce a result in reasonable time.

Our second cubing approach is for problems containing a set of clauses and one large cardinality constraint. Such problems appear frequently in the constraint optimization setting with the cardinality constraint representing some resource bound. The cubing approach assumes that the problem is presented in a cardinality-based input format. We developed a tool that transforms the problem into CNF by encoding the cardinality constraint with the totalizer [3], and during encoding, produces a set of selected auxiliary variables from the totalizer for splitting. In contrast to the first approach, this technique uses a semantic understanding of auxiliary variables to produce a good problem partition.

Unlike many existing cubing-solvers, these techniques completely compartmentalize the cubing and solving phases, allowing the usage of these techniques on top of any off-the-shelf solver. This compartmentalization assists in the goal of learning information about optimal partitions over a class of problems, rather than specific instances, and applying these findings to solve larger instances of the problem.

1.1 Contributions

This work provides 3 main contributions:

1. We developed a tool that performs partitioning based on proof prefixes during partial application of a solver.
2. We developed a tool that performs cardinality-based partitioning with the k-totalizer encoding, selecting auxiliary variables to split on based on their semantic meanings in the encoding.
3. We performed an experimental evaluation comparing our approaches to the widely used partitioning tools, March. We considered SAT and MaxSAT competition formulas, as well as several combinatorial problems and found that our approaches often performed better than existing tools.

2 Background

2.1 Satisfiability

We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula φ is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal ℓ is either a variable x (positive literal) or a negated variable $\overline{x}$ (negative literal). An *assignment* α is a mapping from variables to truth values 1 (true) and 0 (false). An assignment α *satisfies* a positive (negative) literal ℓ if α maps $\mathsf{var}(\ell)$ to true (false, respectively), and *falsifies* it if α maps $\mathsf{var}(\ell)$ to false (true, respectively).

An assignment satisfies a clause if the clause contains a literal satisfied by the assignment, and satisfies a formula if every clause in the formula is satisfied by the assignment. A formula is *satisfiable* if there exists a satisfying assignment, and *unsatisfiable* otherwise.

2.2 Cardinality Constraints

A cardinality constraint on Boolean variables has the form $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$ and is satisfied by a partial assignment if the sum of the satisfied literals is at least k. The size of the cardinality constraint is the number of literals (s) it contains. Variables occurring in the cardinality constraint are *data variables*, and new variables added in a clausal encoding are *auxiliary variables*. For a general cardinality constraint with $1 < k < s$, unit propagation should lead to a conflict when $s - k + 1$ data literals in the constraint are falsified. Cardinality constraints occur often in SAT problems, and commonly represent the bound in an optimization problem, for example "at least k packages must be delivered using at most j trucks." There are many ways to encode a cardinality constraint as clauses [1, 3, 23, 24, 30]. In this work, we will use the totalizer encoding, explained further in Section 3.2.

2.3 CDCL and Clausal Proofs

To evaluate the satisfiability of a formula, a CDCL solver [22] alternates between conflict-driven search and inprocessing. In search, the solver performs a series of variable decisions [5, 20] and unit propagations. If no conflict is reached, the formula is satisfiable. If the solver encounters a conflict, the solver performs conflict analysis potentially learning a clause. In case this clause is the empty clause, the formula is unsatisfiable. Otherwise, the solver revokes some of its variable assignments ("backjumping") and then repeats the whole procedure. Additionally, modern solvers incorporate pre- and inprocessing techniques that change the formula in some way, usually reducing the number of variables and clauses or shrinking the sizes of clauses. These techniques are intermittently interleaved with search during the solving. Some of the most common inprocessing techniques are bounded variable elimination (BVE) [9], vivification [19], and probing [10]

CDCL solvers produce satisfying assignments for satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. A clause C is *redundant* w.r.t. a formula φ if φ and $\varphi \cup \{C\}$ are *satisfiability equivalent.* The clause sequence $\varphi, C_1, C_2, \ldots, C_m$ is a clausal proof of C_m if each clause C_i $(1 \leq i \leq m)$ is redundant w.r.t. $\varphi \cup \{C_1, C_2, \ldots, C_{i-1}\}$. The proof is a refutation of φ if C_m is $\bot$. Clausal proof systems may also allow deletion.

The strength of a clausal proof system is determined by the syntactic criterion it enforces when checking clause redundancy. The standard SAT solving paradigm CDCL learns clauses that are logically implied by the formula and fall under the *reverse unit propagation* (RUP) proof system. A clause is RUP if unit propagation on the falsified literals of the clause results in a conflict. The *Resolution Asymmetric Tautology* (RAT) proof system generalizes RUP and all commonly used inprocessing techniques can be compactly expressed using RAT steps. Proofs are typically transformed to a format with hints, e.g. LRAT, before being passed to a formally-verified checker like cake-lpr [35].

When a CDCL solver logs a proof, the only distinction made between proof steps is if they are clause additions or clause deletions. Without additional processing there is no way to determine if a clause addition in the proof originated from conflict learning or an inprocessing technique. In this work, we consume proofs externally without modifying the solver, so the proof-based heuristics described in Section 3.1 will consider both learned clauses and inprocessed clauses.

2.4 Cube and Conquer

Cube-and-Conquer (CnC) [14] is a powerful methodology for solving difficult SAT formulas. In typical usage, a formula φ is partitioned into many sub-formulas, $\varphi_1, \ldots, \varphi_n$, such that φ is satisfiable if and only if at least one φ_i is satisfiable. The *cubes* to which the name refers are conjunctions of literals. In particular, if a disjunction of cubes $\psi := \psi_1 \vee \cdots \vee \psi_n$ is a tautology, then

$$\varphi \iff \varphi \wedge \psi \iff \bigvee_i \varphi \wedge \psi_i$$

Thus, the choice of partition from which the technique gets its name is $\varphi_i := \varphi \wedge \psi_i$. With this partition, one can dispatch independent CDCL solvers in parallel, as depicted in Figure 1, and enjoy substantial speedups.

In the best case, each $\varphi \wedge \psi_i$ is of equal difficulty and substantially easier to solve than φ. In the worst case, however, each $\varphi \wedge \psi_i$ can be no easier, and sometimes harder, than just φ. In this case, the wall-clock time is no better than just running φ, and the CPU time can blow up exponentially. As a result, the choice of ψ is of great importance and the focus of this paper. Historically, this has been done with expert insight and domain knowledge on a problem-by-problem basis [15, 32, 34], or with lookahead techniques [14]. In this paper, we will propose several new, automatic methods for finding good choices of ψ.

The state-of-the-art tool for automatically computing partitions is March, which combines the David-Putnam-Logemann-Loveland (DPLL) algorithm [8] with lookaheads. For a general discussion, see the Handbook of Satisfiability [16, 18], while we describe here an exemplary scheme. Given a CNF formula φ, a lookahead on literal ℓ works as follows: First, ℓ is assigned to `true`, followed by unit propagation. Second, in case there was no conflict, the difference between φ and the reduced formula φ' is measured. The quality of look-ahead techniques

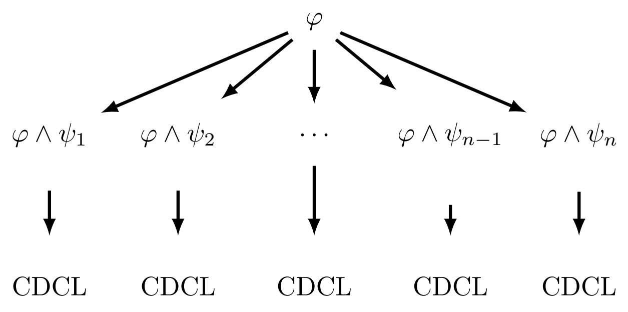

Figure 1 Cube-and-conquer heuristically splits a formula φ into n subformulas $\varphi \wedge \psi_1$ to $\varphi \wedge \psi_n$ and solves the subformulas using CDCL.

depends heavily on the measurement used. A frequently used method weighs the clauses in $\varphi' \setminus \varphi$ (i.e., the clauses that are reduced but not satisfied). Third, all simplifications are reversed to return to φ. If a conflict was detected during the lookahead, then ℓ is forced to `false` and is called a failed literal. The measurements are used to determine the splitting variable in each node of the tree. In general, a variable x is chosen for which both the lookahead on x and $\overline{x}$ result in a large reduction of the formula.

3 Partitioning Techniques

3.1 Proof Prefix Based Splitting

In this section, we describe a partitioning technique based on proof prefixes. A proof, in this context, is a series of clauses that are redundant with respect to the original formula. A formula is unsatisfiable when the empty clause is derived. A *proof prefix* is a sequence of addition steps starting from the beginning of a proof. The key metrics we extrapolate from proof prefixes are variable occurrences, or the number of times a variable appears in a clause addition step. The partitioning technique arises from two observations regarding proof prefixes.

1. Variables which occur frequently in a non-trivial proof prefix will often continue to appear frequently in the remainder of the proof. By non-trivial we mean a proof prefix with a large enough number of steps such that the solver has performed reasoning over several restarts to explore the search space.
2. In problems for which effective partitions are known, splitting variables often occur frequently in a proof generated from the original (unpartitioned) formula.

We ground these intuitions in Figure 2, which shows the variable occurrences in the DRAT proof over time for the $\mu_5(13)$ problem. Despite the proof taking more than 10^7 clause addition steps, the known best variables have risen to near the top by only 10^5. This observation is what allows us to use prefixes as an adequate substitute for the entire proof.

At a high level, we compute proof prefixes, find the most occurring variables in these prefixes, and use theses variables as splitting variables. From the solver's brief search we infer which variables are important in the problem. More concretely, given a formula φ, we run it with an off-the-shelf solver until a desired number of clauses are added to the proof emitted. This is achieved by piping the proof output of the solver, waiting for a desired number of addition steps, and then killing the solver. No modifications to the solver are required. Once the proof prefix is known, we count the variable occurrences in the proof, both positive and negative, and pick the most frequently occurring variable as the next variable to split on.

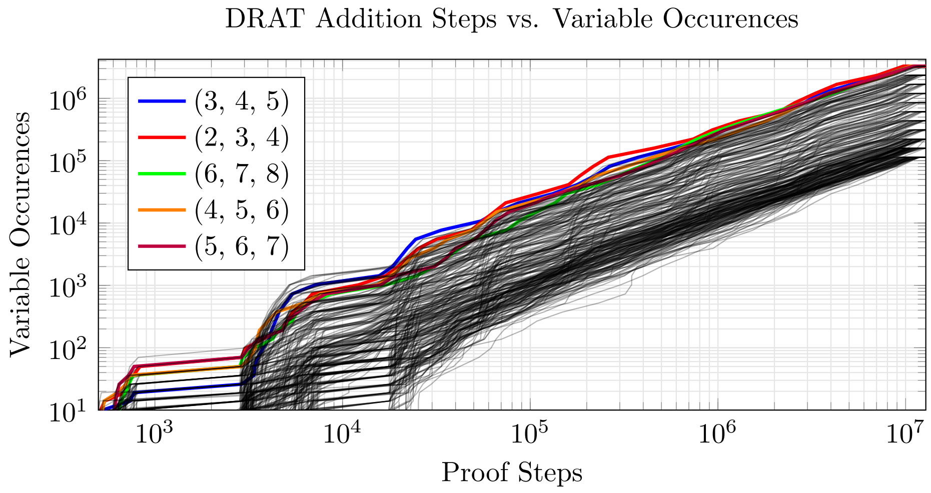

Figure 2 The variable occurrences over the course of a DRAT proof generated by solving the $\mu_5(13)$ problem. This demonstrates that when variables rise to the top, they tend to stay there, and also that the best variables make good splitting choices. In this case, the top 5 variables by the end are exactly the ones found to be good by Subercaseaux et al. [34] and the top 5 variables after 10^5 steps approximate them well enough to create a good partition.

Once a splitting variable x is obtained, we create new formulas, $\varphi \wedge x$ as well as $\varphi \wedge \overline{x}$, and restart the process on both formulas. Naively, generating a complete partition in this manner, where each cube has size d, would require generating $O(2^d)$ proof prefixes, which would be prohibitively expensive. To get around this, we introduce the notion of a *static* partition. A static partition is one where we start with a static set of d splitting variables and then generate cubes with all 2^d combinations of polarities. This can be viewed as a balanced binary tree depicted in the left of Figure 3. In contrast, a *dynamic* partition, depicted in the right of Figure 3, may have unique variables at every vertex in the tree and is not necessarily balanced.

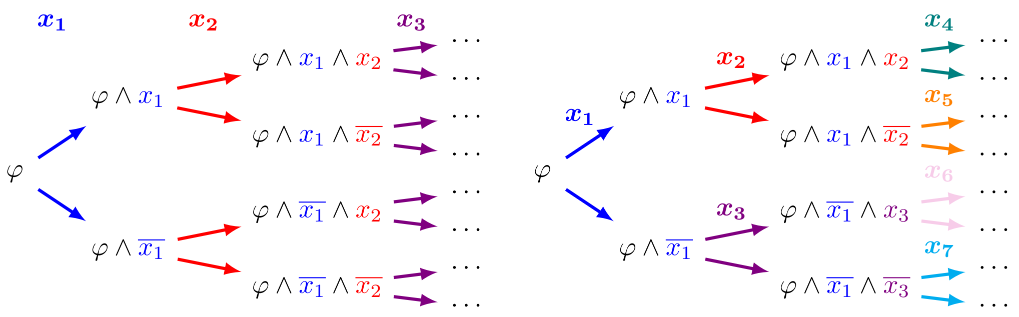
x_1
x_2
x_3
φ
$\varphi \wedge x_1$
$\varphi \wedge \overline{x_1}$
$\varphi \wedge x_1 \wedge x_2$
$\varphi \wedge x_1 \wedge \overline{x_2}$
$\varphi \wedge \overline{x_1} \wedge x_2$
$\varphi \wedge \overline{x_1} \wedge \overline{x_2}$
x_4
x_5
x_6
x_7
$\varphi \wedge \overline{x_1} \wedge x_3$
$\varphi \wedge \overline{x_1} \wedge \overline{x_3}$

Figure 3 Left: A static partition on variables $\{x_1, x_2, x_3, \ldots\}$. Right: A dynamic partition on variables $\{x_1, \ldots, x_7, \ldots\}$.

Given a static partition set of size d, to generate the $d + 1$th variable, we sample a constant number of cubes from the current partition, find the best variable by summing the occurrences across all of the proof prefixes, and use it to extend the set. In this way, we can generate a static partition of depth d by only generating $O(sd)$ proof prefixes, where s is the number of samples per layer. The generation of proof prefixes can be parallelized, so if the number of samples is fewer than the number of cores, which is often sufficient, the cube generation time is a linear function of depth. However, we must synchronize upon solver completion, before processing the proof prefixes. One "layer" of this process is depicted in Figure 4. The specific number of clauses in each proof prefix that produces the best results is formula-dependent, and future work is needed to determine this automatically. However, in our experimental evaluation, we found $100,000$ is an effective cutoff for a wide range of competition formulas, and for many problems this is sufficient. Moreover, the required proof prefix size seems to scale far slower than problem size as demonstrated by the technique's ability to effectively partition very difficult problems with comparatively little preproccesing. Despite the restriction to static partitions, our technique is highly effective across a variety of benchmarks. Moreover, finding an effective static split for a problem has numerous additional benefits, as will be discussed in Sections 4 and 5. This technique was developed into an automated tool, Proofix.

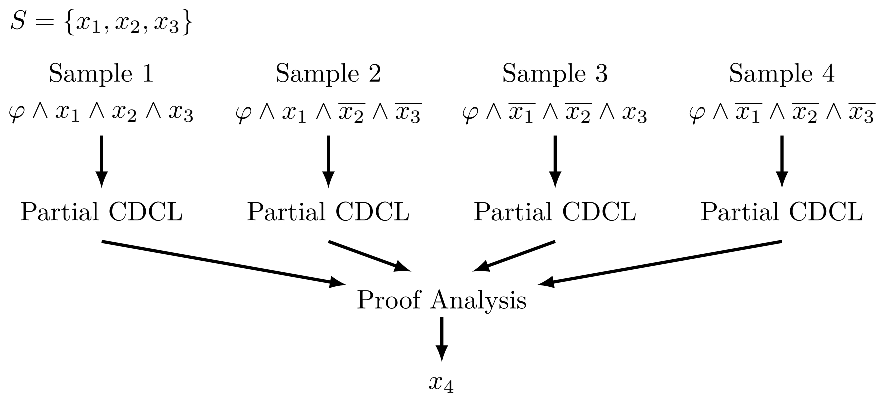
$S = \{x_1, x_2, x_3\}$
Sample 1
$\varphi \land x_1 \land x_2 \land x_3$
Sample 2
$\varphi \land x_1 \land \overline{x_2} \land \overline{x_3}$
Sample 3
$\varphi \land \overline{x_1} \land \overline{x_2} \land x_3$
Sample 4
$\varphi \land \overline{x_1} \land \overline{x_2} \land \overline{x_3}$
Partial CDCL
Partial CDCL
Partial CDCL
Partial CDCL
Proof Analysis
x_4

Figure 4 Example generating x_4 from static partition $\{x_1, x_2, x_3\}$ using 4 samples.

3.2 Totalizer Based Splitting

In this section, we present a partitioning technique based on auxiliary variables from the totalizer encoding. We assume that the problem is given in the cardinality-based input at-least-k conjunctive normal form (KNF [26]), but the cardinality constraint may be encoded as an at-most-k constraint by negating the literals and modifying the bound. This technique makes no assumptions about the clauses in the formula and forms a static partition solely by selecting auxiliary variables from the totalizer encoding.

The totalizer is one of the most widely used classes of cardinality constraint encodings, with incremental variants appearing in modern MaxSAT solvers. The totalizer is structured as a binary tree that incrementally counts the number of true data literals at each node. Data literals form the leaves, and each node has auxiliary variables representing the unary count from the sum of its children counters. For example, in Figure 5 with counters indexed by depth and node id, the counter $c_1^{3,1}$ is set true if either ℓ_1 or ℓ_2 is true, and $c_2^{3,1}$ is true if both ℓ_1 and ℓ_2 are true. Further, if both counters $c_2^{3,1}$ and $c_2^{3,2}$ are true, then the counter $c_4^{2,1}$ is true.

Output variables denoted by o_i at the root of the tree represent the count of true data literals across the entire cardinality constraint. The bound k for an at-most-k constraint is enforced by adding the unit $\overline{o}_{k+1}$. An at-least-k cardinality constraint would enforce the bound with the positive unit o_k. The encoding can be simplified by only encoding the count up to $k+1$ at each node [23], and we adopt this simplification for our encoding.

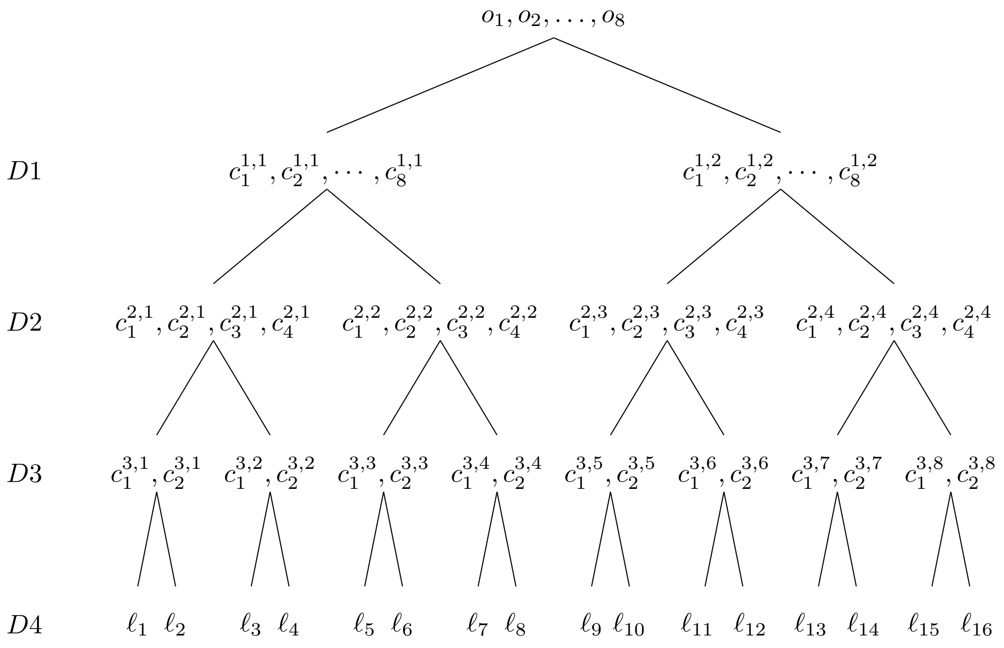
$o_1, o_2, \ldots, o_8$
$D1$
$c_1^{1,1}, c_2^{1,1}, \cdots, c_8^{1,1}$
$c_1^{1,2}, c_2^{1,2}, \cdots, c_8^{1,2}$
$D2$
$c_1^{2,1}, c_2^{2,1}, c_3^{2,1}, c_4^{2,1}$
$c_1^{2,2}, c_2^{2,2}, c_3^{2,2}, c_4^{2,2}$
$c_1^{2,3}, c_2^{2,3}, c_3^{2,3}, c_4^{2,3}$
$c_1^{2,4}, c_2^{2,4}, c_3^{2,4}, c_4^{2,4}$
$D3$
$c_1^{3,1}, c_2^{3,1}$
$c_1^{3,2}, c_2^{3,2}$
$c_1^{3,3}, c_2^{3,3}$
$c_1^{3,4}, c_2^{3,4}$
$c_1^{3,5}, c_2^{3,5}$
$c_1^{3,6}, c_2^{3,6}$
$c_1^{3,7}, c_2^{3,7}$
$c_1^{3,8}, c_2^{3,8}$
$D4$
ℓ_1 ℓ_2 ℓ_3 ℓ_4 ℓ_5 ℓ_6 ℓ_7 ℓ_8 ℓ_9 ℓ_{10} ℓ_{11} ℓ_{12} ℓ_{13} ℓ_{14} ℓ_{15} ℓ_{16}

Figure 5 Visualization of a totalizer for $\ell_1 + \ell_2 + \cdots + \ell_{16} \leq 7$, where $\overline{o}_8$, and both $\overline{c}_8$s would be unit to enforce the bound. Each node has an independent set of auxiliary variable counters c_{cnt}. The nodes are numbered left to right for each depth, and a counter for a given node id at depth dep can be distinguished as $c_{cnt}^{dep,id}$.

To motivate our splitting heuristic, first consider the totalizer in Figure 5. Selecting $c_4^{1,1}$ as a splitting variable will result in the following two cases:

- If $c_4^{1,1}$ is true, at least 4 of the data literals from the set $\ell_1, \ell_2, \ldots, \ell_8$ are true. This also means that at most 3 data literals from the set $\ell_9, \ell_{10}, \ldots, \ell_{16}$ are true due to the bound of at most 7.
- If $c_4^{1,1}$ is false, at most 3 of the data literals from the set $\ell_1, \ell_2, \ldots, \ell_8$ are true. This provides no information about the data literals from the set $\ell_9, \ell_{10}, \ldots, \ell_{16}$.

Intuitively, splitting on an auxiliary variable at an internal node in the totalizer will designate how many true data literals are among its children and potentially its sibling nodes. Furthermore, the closer the auxiliary variable is to the root, the more data literals it will impact. However, a bad selection can create unbalanced subproblems. For instance, if we split on $c_7^{1,1}$, when $c_7^{1,1}$ is true all of the data literals from $\ell_9, \ell_{10}, \ldots, \ell_{16}$ are set false due to the bound. This may create a trivial cube. On the other hand, when $c_7^{1,1}$ is false, we obtain very little information other than that at least one of $\ell_1, \ell_2, \ldots, \ell_8$ is false, and this would likely not benefit the solver. Therefore, we focus our splitting on auxiliary variables with a large impact that also correlate with the cardinality constraint's bound. To achieve this, we use the ratio $Rk = \frac{k}{s}$ for a cardinality constraint with bound k and s data literals. We select a counter from a node (id) with n counters as $\mathsf{SelectCounter}(id) = \lfloor Rk \times n \rfloor$. The selection procedure is as follows:

1. Select a starting depth d and desired number of splitting variables v.
2. Sort the nodes at depth d by number of counters, largest to smallest.
3. For each node (id) in d, select the counter $\mathsf{SelectCounter}(id) + 1$ for odd nodes and $\mathsf{SelectCounter}(id)$ for even nodes.
4. If v variables are selected, break. Otherwise, proceed to depth $d + 1$ and return to step 2 to select the remaining variables.

Consider the totalizer in Figure 5 with $Rk = \frac{7}{16} = 0.4375$. Assume that we want 6 splitting variables, starting from depth 1. Start with node 1 which contains 8 counters. The counter from node 1 will be $\mathsf{SelectCounter}(1) + 1 = \lfloor 8 \times 0.4375 \rfloor + 1 = 4$, which is variable $c_4^{1,1}$. Then move on to node 2 and select $c_3^{1,2}$. Next, move to depth 2, and since all nodes have the same number of counters we process them in order. At node 1 with 4 counters take $\mathsf{SelectCounter}(1) + 1 = \lfloor 4 \times 0.4375 \rfloor + 1 = 2$, which is variable $c_2^{2,1}$. Proceed with the remaining nodes at depth 2 selecting $c_1^{2,2}$, $c_2^{2,3}$, and $c_1^{2,4}$. These 6 variables are then used as the splitting variables to generate a static partition of the formula.

The reason we alternate between $\mathsf{SelectCounter}(id)$ and $\mathsf{SelectCounter}(id) + 1$ is to ensure that the sum of the counts of selected splitting variables across all nodes for a given depth resembles the ratio. This reduces the number of unhelpful or trivial cubes. For example, the selected variables at depth 2: $c_2^{2,1}$, $c_1^{2,2}$, $c_2^{2,3}$, and $c_1^{2,4}$ will sum to 6 in the cube where they are all set to true. If we only selected the c_1 counters, the sum would be 4 and this is far from the bound of 7, and if we only selected c_2 counters, the sum would be 8 and this is trivially unsatisfiable.

The starting depth is assigned based on the number of desired splitting variables such that all nodes in a given depth are processed if possible. In our experimental configuration, we chose 12 splitting variables and therefore started at depth 2, giving 4 variables at this depth and 8 variables at depth 3.

This approach could be adapted to the more compressed modulus totalizer [24], which uses a quotient and remainder at each node to reduce the number of auxiliary variables required to count the sum, by selecting quotient variables at each node. Other hierarchical encodings, for example, those used in pseudo-Boolean solving, would also be candidates for this approach. However, it is not clear how splitting could be achieved with a non-hierarchical encoding like a sequential counter. Also, we do not consider splitting on formulas with many cardinality constraints. It is possible to select a subset of variables from each cardinality constraint encoding, but this would require heuristics for determining the relative importance of each cardinality constraint. Lastly, we focus on unsatisfiable problems with a tight bound, which means that adding 1 to the bound would make the problem satisfiable. If the bound is not tight, we may need to adjust the ratio Rk by sampling for trivial cubes.

4 Explainable Splitting

One of the most common criticisms of large computer-aided proofs is their opacity: many of these SAT-based proofs are massive. For example, the proof of Schur Number Five is roughly 2 petabytes [12], and even CDCL-based proofs for problems that are solvable in seconds can remain impenetrable. It may not be surprising that many basic steps are hard to grasp, but even the high-level structure is completely unclear. However, an effective *static split* can drastically improve our high-level understanding of these proofs. Because our approach extracts a static split directly from CDCL-generated proofs, it exposes the solver's underlying reasoning and highlights the variables it deems most significant.

We have observed that static splits can offer several key advantages:

- *Demystifying solver reasoning.* While the solver's detailed steps may remain intricate, a static split reveals the high-level structure of the proof, making it easier to follow how subproblems branch and simplify.
- *Pinpointing crucial variables.* Intuitively, crucial variables are the ones which always partition a problem into subproblems with equal difficulty. A static split can help identify crucial variables as their introduction would observably split every vertex in half. This contrasts with a dynamic split where it is difficult to identify the crucial variables as they may only occur a few times at unrelated nodes.
- *Facilitating generalization.* Once it is clear which variables matter most, users can readily adapt or replicate that partitioning scheme for related problems.
- *Ease of extension.* Dynamic splits often yield complex trees that are hard to modify. In contrast, a static split is straightforward to expand by adding additional variables that resemble those already used in the partition.

5 Experimental Evaluation

All experiments were performed at the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [7]. We ran 32 solver executions in parallel per node with 10,000 second timeouts. Therefore, each solver process held approximately 8GB of memory.

We compared our splitting techniques with March, the state-of-the-art partitioning tool which uses lookaheads and is not constrained to static partitions. Our experimental comparison focuses exclusively on CnC solvers, primarily due to the fairness of comparison, but also because many of the explainability and usability benefits of our techniques only make sense in the context of CnC partitions, so we felt it warranted to restrict our focus to CnC.

5.1 Stability Under Search Parameters

One of the properties that makes the proof prefix technique powerful is the fact that the preprocessing time required scales far slower than problem difficulty. Indeed, looking at several difficult problems, if Proofix is able to find known good splitting variables, it can do so without needing to search deep into the proof.

The $\mu_n(k)$ problem asks to find the minimum number of convex n-gons induced by placing k points in the plane with no three points in a line. Recently, Subercaseaux et al. [34] conjectured that $\mu_5(n) = \binom{\lfloor n/2 \rfloor}{5} + \binom{\lceil n/2 \rceil}{5}$. The $\mu_5(15)$ benchmark asks whether it is possible to construct only 76 convex pentagons in the plane with 15 points, one less than the known minimum of 77. It should be noted that $\mu_5(15)$ is substantially easier for state-of-the-art MaxSAT solvers with an out-of-the-box `wcnf` formula than for CaDiCaL using the totalizer encoding. This problem takes at least 48 CPU hours.

The $\chi_\rho(\mathbb{Z}^2)$ problem is a heavily optimized formula which was shown to be logically equivalent to asking whether there exists a packing chromatic number of the subset of the plane, $\{(x, y) \in \mathbb{Z}^2 \mid |x| + |y| \leq 15\}$, using 14 colors and center color 6. The "good" variables are precisely the ones corresponding to the "plus" tiling of the graph as described by Subercaseaux et al. [33]. With a manual partition of over $5,000,000$ cubes, this problem was solved in $4,851$ CPU hours.

The 7*gon*-6*hole* problem asks whether every set of 24 points with no 3 three points in a line contains either a convex 7-gon, or a 6-hole (a convex 6-gon with no points inside of it). This problem is estimated to have a single-core runtime of $1,000$ CPU hours and a manual partition was found reducing it to 200 CPU hours (on a single core) [15].

Table 1 Stability of quality of variables under increasing prefix lengths. Each formula was statically partitioned into cubes of depth 15. The number of "good", as determined by their manual partitions in previous works, is shown as well as how many of the variables in the final partition are "good." If there are fewer "good" variables than 15, the score is at most the number of "good" variables.

Formula	# "Good"	# Variables	Prefix Length				
			10^3	10^4	10^5	5×10^5	10^6
$\chi_\rho(\mathbb{Z}^2)$	63	$7,669$	0/15	0/15	12/15	12/15	11/15
$\mu_5(15)$	13	$58,826$	7/15	5/15	4/15	3/13	4/15
7gon-6hole	20	$28,878$	2/15	8/15	13/15	12/15	12/15

Table 1 depicts the tool looking for variables that are known to be good on 3 very difficult problems. Using just 32 samples per variable, on two of them Proofix is able to pick out many known good variables after only searching for a tiny fraction of the time that the problem itself would take. While it is possible that Proofix was able to come up with an alternative set of "good" variables for the $\mu_5(15)$ problem, it more likely provides evidence that for certain proofs the prefix is not an adequate substitute. Understanding why this is the case is the subject of future work.

One caveat to note is that while finding good splitting variables can now be done automatically for many problems, for large problems such as these, it is not necessarily the case that the cubes found will always achieve comparable performance to the ones found manually due to factors such as variable ordering or the specific subset of "good" variables used in the static partition. The partition used to solve the $\chi_\rho(\mathbb{Z}^2)$ problem, for example, used a dynamic partition with an unbalanced binary tree that Proofix cannot capture directly. Future work is required on how to use proof prefixes to efficiently capture dynamic partitions.

5.2 Maximum Satisfiability Problems

In this section, we evaluate the totalizer-based splitting and proof-based splitting on problems containing one large cardinality constraint. These types of problems occur naturally as unweighted maximum satisfiability (MaxSAT) problems. Each MaxSAT problem consists of a set of hard clauses and soft units. The objective is to find the optimum (minimum) number of soft units that must be falsified while satisfying all hard clauses. MaxSAT problems can be converted to SAT by combining the hard clauses with one, often large, cardinality constraint stating at most k soft units are falsified. We can generate an unsatisfiable SAT problem by making the cardinality constraint's bound the optimum minus one, and a satisfiable SAT problem by making the bound the optimum.

In our evaluation, we consider MaxSAT benchmarks from the 2023 competition unweighted track [17]. We generate unsatisfiable SAT problems from formulas with known bounds. The SAT problem is transformed into CNF by encoding the cardinality constraint as a totalizer. The cardinality constraint is transformed from at-least-k to at-most-k if this makes the encoding size smaller. Before encoding the cardinality constraint, we sort data literals using the best literal sorting found in recent work [25]. This step improves the default solver performance, creating a better baseline for partitioning.

We filter out problems with a CaDiCaL solving time of less than 500 seconds, leaving five benchmark families: judgment aggregation (judge) [17], model-based diagnosis (mbd) [21], approximately propagation complete CNF for an all-different encoding of pigeon hole (optic) [2],

Table 2 Comparison of partitioning methods on combinatorial problems and MaxSAT competition benchmarks. Evaluation is in CPU time.

formula	baseline	March			Proofix			Totalizer Splitting	
	1 core	1 core	32 core	Pre.	1 core	32 core	Pre.	1 core	32 core
judge	$3,654$	$4,893$	$3,162$	219	$3,437$	$2,447$	19	7,851	4,289
mbd	$2,170$	$2,914$	313	287	$2,512$	409	37	3,784	290
optic	$1,236$	908	195	23	708	22	45	1,150	135
uaq	$2,520$	$1,408$	453	4	970	62	43	1,960	129
mindset	$2,162$	$18,018$	$1,372$	357	$16,375$	$2,252$	42	19,002	1,164
max10	$6,282$	$1,939$	920	0	$7,645$	238	29	2,498	269
cross13	$> 80,000$	?	$> 10,000$	43	$64,610$	$2,206$	71	77,664	3,125
$\mu_5(13)$	$2,317$	$2,526$	181	8	$1,367$	84	80	2,355	543

user query authorization (uaq) [2], and minimum rule set for labeling data (mindset) [2]. For each family, we selected the hardest problem for CaDiCaL to solve, and present the results at the top of Table 2.

Proofix outperforms the other techniques in both single and 32 core performance on the majority of the problems. Totalizer split performs adequately, which is notable due to its lack of preprocessing time other than time (less than a second) required to encode the cardinality constraint into CNF. Of all the selected problems, only mindset is not successfully partitioned by any technique, with totalizer split producing the best 32 core speedup of only $2\times$.

The preprocessing time of Proofix is consistent across problems because there is only minor variation in the time it takes for the solver instances to produce 100,000 proof steps. On the other hand, March has a large range of preprocessing times. For example, in mbd March spends as much time in preprocessing as totalizer split spends solving the problem on 32 cores.

In addition, we examined the splitting variables selected by Proofix to find out if it used any auxiliary variables from the totalizer encoding. Proofix used some auxiliary variables for optic and mbd, a single auxiliary variable for judge, and no auxiliary variables in the other problems. This suggests that Proofix can discover auxiliary variables when they are more useful, as is the case for mbd based on the superior performance of totalizer split. Furthermore, Proofix achieves this without any additional insight into the problem or variable semantics. Many of the other problems can be effectively partitioned by either auxiliary variables or data variables, as seen by the similar performance of Proofix and totalizer split.

Next we discuss three additional hard combinatorial problems to test the limits of these splitting techniques. We describe the problems and their encodings at a high level below.

The *Max Squares* problem [36] asks whether you can set m cells to true in an $n \times n$ grid such that no set of four true cells form the corners of a square. Problem variables denote whether a cell is in the solution. The problem is encoded with clauses containing 4 literals blocking the 4 corners of each possible square on the grid and one cardinality constraint over all of the problem variables. For a 10×10 grid the optimal value is 61, so a bound of 62 on the cardinality constraint makes the formula unsatisfiable.

The crossing number $cr(G)$ of a graph G is the lowest number of edge crossings of a plane drawing of the graph G. In 1960, Guy conjectured that the crossing number of the complete graph is $cr(K_n) = \frac{1}{4} \cdot \lfloor \frac{n}{2} \rfloor \cdot \lfloor \frac{n-1}{2} \rfloor \cdot \lfloor \frac{n-2}{2} \rfloor \cdot \lfloor \frac{n-3}{2} \rfloor$ [28]. It is known that $cr(K_{13}) = 225$. The benchmark *cross13* asks whether there is a K_{13} drawing with 224 crossings.

On the max10 problem, March exhibits good single-core performance but cannot match the improved 32 core times from both of our techniques. Likewise, March fails to solve the cross13 problem with a 10,000 second cube timeout. For each of these problems Proofix produces a partition with the best 32 core performance, and it does so without using the auxiliary variables from the totalizer encoding. The totalizer split can approximate the performance of Proofix with slightly worse performance on each of the problems. Notably, the preprocessing time for Proofix on the much harder cross13 is lower than for $\mu_5(13)$, displaying the potential of a static splitting technique to scale and achieve a good partitioning with relatively low preprocessing overhead.

5.3 SAT Competition Formulas

To measure the generalizability of Proofix, we compared it to both CaDiCaL [4] and March on SAT competition formulas from the years 2022, 2023, and 2024. We considered problems which were both UNSAT and took CaDiCaL more than $1,000$ seconds.

Proofix was used with settings where it learned the first 10^5 clauses of the proof and generated a static partition of depth 10, resulting in 2^{10} subprobems. Comparing Proofix to CaDiCaL directly, as shown in Figure 6, we see that 63% of competition formulas perform better when first split and run on 32 cores than when run with CaDiCaL alone, including the preprocessing time for splitting. If we disregard preprocessing time, this number increases to 75%. Including preprocessing, there are 26 problems where Proofix performed more than $10\times$ better than CaDiCaL, and even one which performed $100\times$ better.

Unfortunately, March timed out or crashed while generating the cubes on multiple instances. We suspect that this is due to March learning a lot of "local" clauses and exceeding the available memory on formulas with many small clauses. To ensure a fair comparison, and to eliminate many additional cases where March failed to generate a partition before timing out, it was limited to a split of depth at most 10.

As can be seen in Figure 7, Proofix performs favorably compared to March on many problems. Considering only problems where March did not segfault, Proofix performed better than March on 53% of problems, and considering problems where March failed for any reason, that number increases to 61%. While there are a couple of cases where March beats Proofix by a large margin, there are far more where the opposite is true, and even two cases where Proofix produces a partition which is over $1,000\times$ better than March.

Although Proofix always succeeds in generating the cubes, there are 10 cases where at least one cube takes more than $10,000$ seconds. This is in contrast to March, where a cube times out in 5 cases, March itself times out in generating cubes in 8 cases, and it segfaults while generating cubes in 35 cases. The segfaults persisted across multiple compiled March binaries and multiple computers. In addition, it is worth noting that segfaults seem to be roughly correlated with partitioning difficulty, as 6 of the 10 instances for which Proofix times out on a cube also cause segfaults in March.

It can also be seen that while both tools can fail to find a good split in some cases, the amount that Proofix can fail by is far less than March. In the worst case, March's 32 core time is $55,000$ seconds slower than CaDiCaL, whereas Proofix's 32 core time is at most $21,800$ seconds worse. Moreover, there are 9 problems where March produces a partition that has a time on 32 cores worse than CaDiCaL, compared to Proofix for which there are only 4 (conditioned on March generating a partition at all).

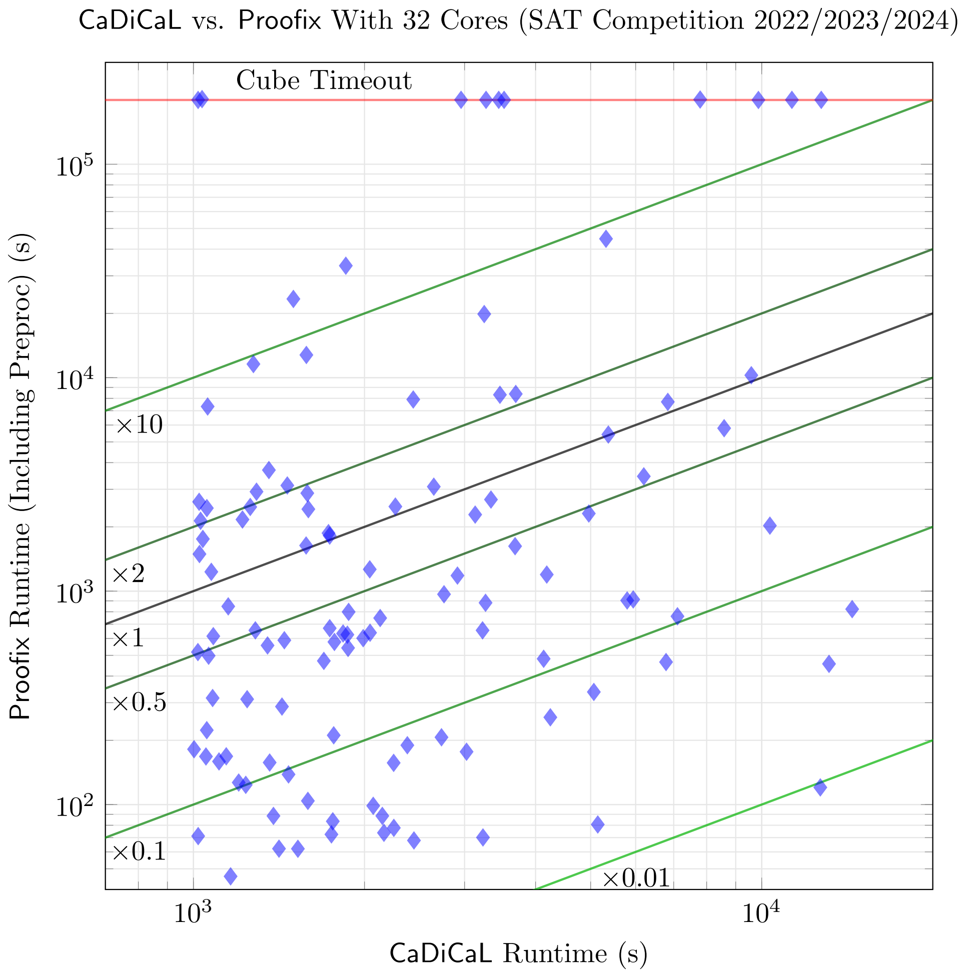

Figure 6 CaDiCaL runtimes vs. Proofix cube generation and runtimes on 32 cores. Lower is better.

6 Applications and Results

We believe that the proof prefix cubing technique is of interest to people solving formulas of all sizes. For competition-sized problems, one way to make use of an n-core machine is to dedicate $n-1$ cores to cubing and solving in parallel, reserving the last core to run an out-of-the-box solver. The last core is used in this manner to account for the case that either the overhead of partitioning outweighs the difficulty of the formula itself or an effective partition cannot be found. If one were to take this approach for the SAT competition problems in 2022, 2023, and 2024, they would see improvements on nearly 65% of the problems from those years. Similar improvements can be seen for MaxSAT, although the formulas tended to be easier overall. However, this is due to the kinds of formulas in the MaxSAT competition, rather than the applicability of Proofix itself. We believe that the main application is toward resolving large combinatorial problems. We repeatedly observe that the partial-proof technique is able to recreate the results of manual effort [15,32,34], and inform optimal partitions automatically. Thus, we believe that Proofix's primary use-case will be returning near-optimal, easily interpretable, static partitions, which can help users quickly identify semantic patterns in cubes in order to solve large problems.

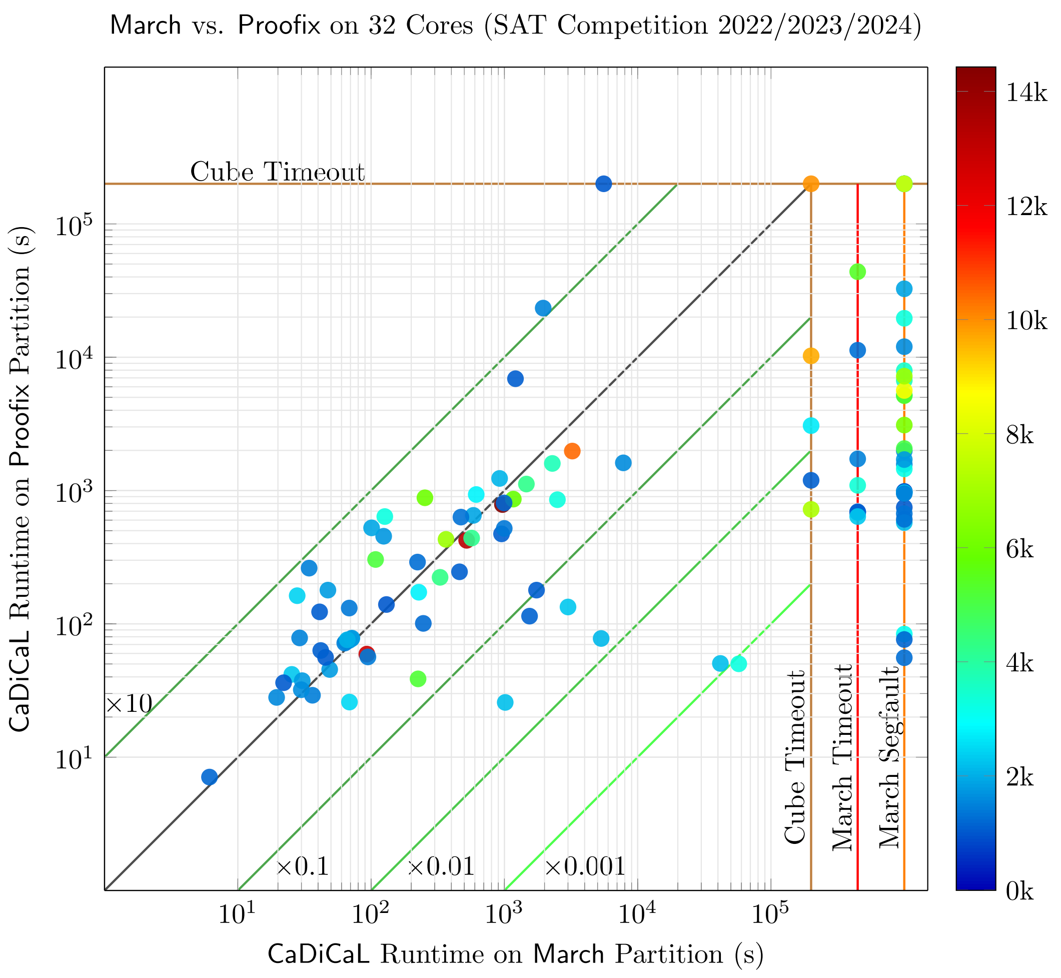

Figure 7 A comparison of March vs. Proofix on SAT competition problems using 32 cores. The color of a point indicates how long the original problem takes CaDiCaL to solve with no partitioning. The diagonal lines compare orders of magnitude of performance between March and Proofix for solving the partition. Several lines are added on the right to denote cases where March failed for one of a few reasons. Being lower on the graph is better.

7 Conclusion and Future Work

In this paper, we presented two novel techniques for automatically partitioning SAT formulas, one based on proof prefixes and the other based on the totalizer encoding. We demonstrated that the limitation to static partitions is not a major setback, and also provides numerous qualitative benefits towards explainability. Finally, we developed tools for both splitting techniques and demonstrated that these techniques outperform the state-of-the-art tool on numerous problems. There are several questions left open for future work:

- What makes some proofs more amenable than others for splitting, and is this property identifiable in a prefix? In other words, can we detect when a prefix is unlikely to yield a good partition?
- Given that we can find good variables for a partition, is there a way to automatically turn them into a dynamic partition, or generalize them from their semantic meaning?
- How do the splitting techniques in this paper compare to clause-sharing solvers, both on competition and difficult combinatorial problems?

References

1 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT)*, pages 167–180. Springer, 2009. doi:10.1007/978-3-642-02777-2_18.

2 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2018 Benchmarks. https://helda.helsinki.fi/items/7c51d9bd-20e8-428f-8926-b2bbf151dc4c, 2018. Accessed: 2025-04-02.

3 Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming (CP)*, pages 108–122, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-45193-8_8.

4 Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. *Proceedings of SAT Competition*, 14:316–336, 2017.

5 Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *LNCS*, pages 405–422, 2015. doi:10.1007/978-3-319-24318-4_29.

6 Joshua Brakensiek, Marijn J. H. Heule, John Mackey, and David Narváez. The resolution of keller's conjecture. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 48–65, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51074-9_4.

7 Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. *Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research*, pages 1–4. Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3437359.3465593.

8 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. doi:10.1145/368273.368557.

9 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

10 Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, USA, 1995.

11 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modelling and Computation*, 6(4):245–262, 2010. doi:10.3233/SAT190070.

12 Marijn J. H. Heule. Schur number five. In *AAAI Conference on Artificial Intelligence*, 2018.

13 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.

14 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

15 Marijn J. H. Heule and Manfred Scheucher. Happy ending: An empty hexagon in every set of 30 points. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 61–80, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57246-3_5.

16 Marijn J. H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, volume 185 of *FAIA*, chapter 5, pages 155–184. IOS Press, February 2009. doi:10.3233/978-1-58603-929-5-155.

17 Matti Järvisalo, Jeremias Berg, Ruben Martins, and Andreas Niskanen. MaxSAT Evaluation 2023 Benchmarks. `https://maxsat-evaluations.github.io/2023/benchmarks.html`, 2023. Accessed: 2024-08-13.

18 Oliver Kullmann. *Fundaments of Branching Heuristics*, volume 185 of *FAIA*, chapter 7, pages 205–244. IOS Press, February 2009. `doi:10.3233/978-1-58603-929-5-205`.

19 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artificial Intelligence*, 279(C), February 2020. `doi:10.1016/J.ARTINT.2019.103197`.

20 Jia Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *LNCS*, pages 123–140, 2016. `doi:10.1007/978-3-319-40970-2_9`.

21 Joao Marques-Silva, Mikoláš Janota, Alexey Ignatiev, and Antonio Morgado. Efficient model based diagnosis with maximum satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI)*, IJCAI'15, pages 1966–1972. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/279`.

22 João Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009. `doi:10.3233/978-1-58603-929-5-131`.

23 Antonio Morgado, Alexey Ignatiev, and Joao Marques-Silva. Mscg: Robust core-guided maxsat solving: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:129–134, December 2015.

24 Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Modulo based cnf encoding of cardinality constraints and its application to maxsat solvers. In *Tools with Artificial Intelligence (ICTAI)*, pages 9–17, 2013. `doi:10.1109/ICTAI.2013.13`.

25 Joseph E Reeves, Joao Filipe, Min-Chien Hsu, Ruben Martins, and Marijn JH Heule. The impact of literal sorting on cardinality constraint encodings. In *Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2025.

26 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From Clauses to Klauses. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification (CAV)*, volume 14681 of *Lecture Notes in Computer Science*, pages 110–132. Springer, 2024.

27 Joseph E Reeves, Benjamin Kiesl-Reiter, and Marijn JH Heule. Propositional proof skeletons. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 329–347. Springer, 2023.

28 R.K.Guy. A combinatorial problem. *Nabla (Bulletin of the Malaysian Mathematical Society)*, 7, 1960.

29 Dominik Schreiber and Peter Sanders. Scalable sat solving in the cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing. `doi:10.1007/978-3-030-80223-3_35`.

30 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 3709 of *LNCS*, pages 827–831, 2005. `doi:10.1007/11564751_73`.

31 Mate Soos, Raghav Kulkarni, and Kuldeep S Meel. : gazing in the black box of sat solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 371–387. Springer, 2019.

32 Bernardo Subercaseaux and Marijn Heule. Toward optimal radio colorings of hypercubes via sat-solving. In Ruzica Piskac and Andrei Voronkov, editors, *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 94 of *EPiC Series in Computing*, pages 386–404. EasyChair, 2023. `doi:10.29007/QRMP`.

33 Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is 15. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–406, Cham, 2023. Springer Nature Switzerland. `doi:10.1007/978-3-031-30823-9_20`.

34 Bernardo Subercaseaux, John Mackey, Marijn J. H. Heule, and Ruben Martins. Automated mathematical discovery and verification: Minimizing pentagons in the plane. In Andrea Kohlhase and Laura Kovács, editors, *Intelligent Computer Mathematics*, pages 21–41, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-66997-2_2.

35 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25(2):167–184, 2023. doi:10.1007/S10009-022-00690-Y.

36 Ed Wynn. A comparison of encodings for cardinality constraints in a SAT solver. *ArXiv*, abs/1810.12975, 2018. arXiv:1810.12975.

37 Jiong Yang, Arijit Shaw, Teodora Baluta, Mate Soos, and Kuldeep S. Meel. Explaining SAT Solving Using Causal Reasoning. In Meena Mahajan and Friedrich Slivovsky, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.SAT.2023.28.