Formal Methods in Computer-Aided Design 2025

Per-Instance Subproblem Generation for Strategy Selection in SMT

Amalee Wilson*, Nina Narodytska‡, Clark Barrett*, and Haoze Wu†
*Stanford University, Stanford, USA {amalee, barrett}@cs.stanford.edu
†Amherst College, Amherst, USA hwu@amherst.edu
‡VMware Research by Broadcom, Palo Alto, USA nina.narodytska@broadcom.com

***Abstract*—In this paper, we investigate customizing the solving strategy for an individual SMT problem, based solely on the problem itself, without relying on any offline strategy tuning. Our key insight is to generate a set of subproblems derived from the original formula, analyze the behavior of candidate solving strategies on these smaller, representative subproblems, and predict which strategy will perform best on the original formula. We demonstrate that performance on the subproblems is frequently indicative of performance on the original formula. Additionally, we introduce a novel subproblem generation procedure that outperforms existing SMT formula partitioning techniques for the proposed workflow. Finally, we show that on a selection of SMT-LIB benchmarks, when our approach can make a prediction, it can reduce the total compute time substantially.**

I. Introduction

Algorithmic tuning is a standard approach for optimizing an SMT solver's performance on a specific problem domain. Traditionally, tuning is regarded as an *offline* process, whereby a collection of related benchmarks is used to evaluate candidate solving strategies (e.g., different solvers, different solver options). The goal is either to find the best overall strategy or to train a strategy selector that predicts the best solving strategy for a given instance. While useful, offline tuning is not always feasible—e.g., when similar benchmarks suitable for tuning are unavailable. Moreover, offline tuning may require significant time and effort, and even when this overhead can be justified by an overall performance gain, dividing the solving procedure into offline and online phases makes the overall process more brittle and less automated. Motivated by these observations, we investigate the following question in this paper: *Can we customize the solving strategy for a given SMT problem based solely on the problem itself, without any offline strategy tuning?*

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0020347. This work was also supported by DARPA under Agreement FA8750-24-9-1000 and by the Stanford Center for Automated Reasoning. Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Our key idea is to view a given SMT problem as a generator of relevant, easier subproblems and to use the generated subproblems to evaluate solving strategies. Existing formula-partitioning methods used in divide-and-conquer-based parallel SMT solving are natural candidates for the subproblem generation step, i.e., they can be used to partition a problem into subproblems from which we can sample for the purpose of tuning. The challenge with this approach, however, is that none of the subproblems are guaranteed to be sufficiently easy to solve. Consequently, tuning on such subproblems could require significant effort, possibly even more than would be required to solve the original problem itself.

To address this challenge, we propose a new subproblem generation technique for producing subproblems of bounded difficulty. Our key observation is that when an SMT solver backjumps during the execution of the CDCL(T) procedure, the subproblem corresponding to the set of decisions made so far at the time of the backjump is known to be solvable. Leveraging this observation, we design a procedure that either solves the full SMT problem or generates easy subproblems (measured by the number of conflicts required to solve the subproblem). Our approach produces subproblems that are significantly easier to solve than those generated by existing formula-partitioning algorithms.

Since the generated subproblems are used to evaluate and choose among candidate solving strategies, a natural follow-up question is: *Does a good solving strategy for the subproblems translate to a good strategy for the original problem?* We empirically demonstrate that the answer is positive for the subproblem-generation procedures we tested.

Building on this result, we propose an online per-instance strategy selection methodology: given an SMT problem, extract a set of easier subproblems, rank candidate solving strategies on those subproblems, and then tackle the original problem based on the ranking. On a selection of SMT-LIB benchmarks across 4 logics (QF_IDL, QF_LIA, QF_LRA, and QF_SLIA), we demonstrate that, when our method can successfully generate a ranking, it can reduce the total computation time over a parallel portfolio approach by an average of 27.8%.

To summarize, our contributions include:

https://doi.org/

- a workflow for performing per-instance strategy selection for SMT completely online, without any offline tuning;
- a novel subproblem generation procedure that can be used to rank solving strategies more efficiently than existing formula-partitioning methods;
- the empirical discovery that strategy performance on subproblems translates to performance on the original problem for various subproblem-generation methods;
- an implementation of the proposed methodology; and
- an experimental evaluation demonstrating that the proposed method can reduce total computation time.

The rest of this paper is organized as follows. Section II and Section III cover preliminary information and related work, respectively. Section IV introduces our strategy selection methodology and the new subproblem generation procedure. Section V details our implementation, and Section VI presents our experimental evaluation. Finally, Section VII concludes the paper and discusses directions for future work.

II. Preliminaries

Logical setting We assume the standard many-sorted first-order logic setting with the usual notions of signature, term, and interpretation. A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations. For convenience, we assume a fixed background theory $\mathcal{T}$ with signature Σ including the Boolean sort Bool. We further assume that all terms are Σ-terms, that entailment ($\models$) is entailment modulo $\mathcal{T}$, equivalence is equivalence modulo $\mathcal{T}$, and that interpretations are $\mathcal{T}$-interpretations. A formula φ is a term of sort Bool and is *satisfiable* (resp., *unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A formula whose negation is unsatisfiable is *valid*. The *satisfiability modulo theories* (SMT) problem is that of determining the satisfiability of an arbitrary formula.

CDCL($\mathcal{T}$)-based SMT solvers solve problems through the cooperation of a Boolean satisfiability (SAT) solver and one or more theory solvers [1]. In this framework, the SAT solver is responsible for constructing a truth assignment M, typically incrementally, that satisfies the Boolean abstraction of the input formula. At any point during the run, the subset of M consisting of decisions made by the SAT solver is called the *decision trail*. As M is built, each new assignment prompts a call to one or more theory solvers to check whether M remains consistent with the corresponding theories. If some theory solver discovers an inconsistency, it returns a *conflict clause* (and optionally new lemmas) to the SAT solver. A conflict clause is a disjunction of literals, valid in the theory, that is falsified under M, while a lemma is any other heuristically-chosen valid formula. Upon receiving a conflict clause, the SAT solver analyzes the conflict and performs a backtrack, removing part of the decision trail and learning the conflict clause to prevent encountering the same conflict again. This iterative process continues until one of two outcomes is achieved: either M is a complete satisfying assignment and no conflicts are detected by the theory solvers, meaning the problem is satisfiable; or, an unrecoverable conflict is derived, and the problem is therefore unsatisfiable.

Strategy selection is the process of choosing a solving strategy for an SMT problem. A solving strategy is a tuple consisting of the solver and a set of options. In the evaluation, the solver is fixed, so we refer to the option sets as strategies. For clarity, we try to avoid using the word "strategy" outside of this context throughout the paper.

Subproblem generation procedures create a subproblem ϕ_i of the form $\phi \wedge s_i$, where ϕ is the original problem and s_i corresponds to a specific part of ϕ's search space. One common approach to creating subproblems is partitioning, such as cube-and-conquer [2] and scattering [3]. When partitioning, the goal is to produce N subproblems, each with difficulty $1/N$. Typically, the disjunction $\phi_1 \vee \cdots \vee \phi_N$ is equisatisfiable with ϕ. In cube-and-conquer partitioning for SMT, N variables in the Boolean abstraction of the SMT problem are selected, and 2^N cubes (i.e., conjunctions of literals) are created. The scattering strategy produces disjoint partitions as follows. The first partitioning formula is a cube C_1. The second uses a new cube, C_2, to create the partitioning formula $\neg C_1 \wedge C_2$, and so on. The final partitioning formula is $\neg C_1 \wedge \cdots \wedge \neg C_{N-1}$. We compare our new subproblem generation procedure to each of these approaches. While it is similar to the cube-and-conquer approach in that each s_i is a cube, our procedure does not attempt to evenly divide the problem.

III. Related Work

Our approach is inspired and informed by several existing lines of work.

1) Subproblem generation: Previously, the main motivation for designing algorithms for generating subproblems from a given SMT problem has been to improve the effectiveness of divide-and-conquer-based SMT solving approaches [4], [5], [6]. In that context, the original problem is partitioned into many subproblems, each of which is solved in parallel. While we show that solver performance on the subproblems generated via existing partitioning algorithms does indeed predict the performance when solving the original problem, we elect not to use these algorithms, because the subproblems they generate are often too difficult. We develop instead a specialized subproblem generation procedure that efficiently generates subproblems that are easy to solve. Note that unlike prior work on subproblem generation, our new procedure does not *partition* the problem (i.e., the disjunction of the generated subproblems is not guaranteed to be equisatisfiable with the original problem).

2) Per-instance strategy selection: Traditional per-instance algorithm selection methods [7], [8], [9], [10] typically utilize machine learning to predict the most effective algorithm for a given problem. These methods assume the availability of a set of training problems, which is used to train an oracle that predicts the best solving strategy for a given problem based on its structural characteristics (e.g., a selected set of problem features used as the input to the oracle). We also aim to customize the solving strategy for a given instance, but we do not assume the availability of training problems. Rather, we generate training problems from the given problem as needed. Our method is also distinct from prior approaches because it does not require distilling a set of features from a problem (i.e., inputs to the strategy-selecting oracle), which can be challenging.

3) Online learning: Our approach can be viewed as a form of online learning, which has been explored in various automated reasoning contexts. The MapleSAT solver [11], [12] was among the first to explore such a direction. For example, in MapleSAT, branching is formulated as a multi-armed bandit problem. The idea is to treat each variable as an arm and maintain its estimated reward throughout the solving. More broadly, algorithms with adaptive components are common in automated reasoning tools (e.g., dynamic local search techniques [13], branching heuristics [14], [15], and restart techniques [16]). In contrast, our method focuses on learning high-level solving strategies instead of low-level heuristics. There has also been work on choosing solving strategies online when solving a sequence of related problems [17], [18]. Our approach goes one step further by conducting online strategy selection for solving a single problem.

The most closely related recent work is on repurposing existing cubing algorithms to generate tuning data [19] for solving SAT and neural network verification problems. This paper explores a similar direction, but it is different in three important ways: (i) we study a new general-purpose SMT solving setting; (ii) instead of relying on existing cubing algorithms for subproblem generation, we design a new, specialized subproblem generation procedure; and (iii) instead of accelerating the solving of cubes in a cube-and-conquer procedure, our approach is aimed at boosting the performance when solving the original problem.

IV. Subproblem Generation for Strategy Selection

In this section, we describe our methodology for using subproblem generation for strategy selection. We start with a top-level overview of the approach (Algorithm 1). We then describe our new technique for subproblem generation (Algorithm 2). We finish the section with an explanation of how we solve subproblems and rank strategies based on the results of solving subproblems. In future sections, we will refer to subproblems generated using Algorithm 2 as *easy-cubes* or *ECubes* to abbreviate.

A. Overview

Algorithm 1 takes as input a formula to be checked for satisfiability. It then creates subproblems and uses them to rank solving strategies for that formula. The algorithm has several parameters, and we discuss specific ways of instantiating them below. We start by walking through the execution of Algorithm 1 step by step. We then discuss specific details and design decisions.

Algorithm 1 Generation of the ranked list of solving strategies for a given problem.

```
Input: $\phi$, an SMT formula
Input: $stratList$, a list of solving strategies
Input: $N \geq 1$, the number of subproblems to solve
Output: $(status, rankedList)$
 1: $res, subprobs \leftarrow makeSubproblems(\phi, N)$
 2: if $res \in \{SAT, UNSAT\}$ then
 3:     return $(res, [\,])$
 4: end if
 5: if $subprobs.size() < N$ then
 6:     return $(FAIL, [\,])$
 7: end if
 8: $res, stratInfo \leftarrow solveSubprobs(subprobs, stratList)$
 9: if $res \in \{SAT, UNSAT\}$ then
10:     return $(res, [\,])$
11: end if
12: $rankedList \leftarrow makeRanking(stratInfo)$
13: return $(RANKED, rankedList)$
```

Algorithm 1 is implemented outside an SMT solver. It takes as input an SMT formula, ϕ, a list of candidate solving strategies, $stratList$, and the number of subproblems to generate, N. On line 1, $makeSubproblems$ is called to generate subproblems from ϕ. The implementation of $makeSubproblems$ requires a subproblem generation procedure and optional code to, for example, increase diversity of subproblems. A new subproblem generation procedure is given in algorithm 2 and described in detail below, and diversification tactics are discussed in section V-B. Because it is possible for the problem to be solved during subproblem generation, $makeSubproblems$ returns a tuple containing both a status and a list of subproblems. If the status is one of SAT or $UNSAT$, this means that ϕ was solved during subproblem generation. In this case, the status is returned with an empty list (lines 2-4). If fewer than N subproblems are generated by $makeSubproblems$, then the status $FAIL$ is returned with an empty list (lines 5-7). Otherwise, $solveSubprobs$ is called, which attempts to evaluate the candidate strategies in $stratList$ on the subproblems in $subprobs$. The specific implementation of $solveSubprobs$ is discussed in Section V. Once again, $solveSubprobs$ returns a tuple, because it is possible (depending, as we explain below, on how $makeSubproblems$ and $solveSubprobs$ are implemented) for the original problem to be solved during the call to $solveSubprobs$. If this happens, then once again, the status is returned with an empty list (lines 9-11). If not, $stratInfo$ is used in the call to $makeRanking$ on line 12 to produce a ranked list of solving strategies according to a ranking heuristic, and this list is returned with the $RANKED$ status.

Notice that there are three situations in which a ranked list cannot be generated: when the problem is solved during subproblem generation, when fewer than N subproblems are generated, and when the problem is solved during subproblem solving (lines 3, 6, and 10, respectively). In the first and last cases, Algorithm 1 solves the problem, so producing a ranking is no longer necessary. However, in the case that not enough subproblems are generated, the method fails, producing neither a result nor a ranking. Each of the functions called in Algorithm 1 can be customized to use a specific subproblem generation procedure, different subproblem solving strategies, or a custom ranking heuristic. We discuss each of them in turn below.

Algorithm 2 Subproblem generation, embedded in the CDCL(T) framework.

```
Input: $\phi$, an SMT formula
Input: $numSubprobs$, the number of desired subproblems
Output: $(status, subprobs)$
 1: $subprobsMade \leftarrow 0$
 2: $subprobs \leftarrow [\,]$
 3: while True do
 4:     $conflict \leftarrow propagate()$
 5:     if $conflict \neq \{\}$ then
 6:         $decisionLevel \leftarrow resolveConflict(conflict)$
 7:         if $decisionLevel < 0$ then
 8:             return $(UNSAT, [\,])$
 9:         end if
10:         if $dumpCondition()$ then
11:             $subprobs \mathrel{+}= \phi \wedge dumpDecisionTrail()$
12:             $subprobsMade \mathrel{+}= 1$
13:             if $subprobsMade == numSubprobs$ then
14:                 return $(UNKNOWN, subprobs)$
15:             end if
16:         else if $abortCondition()$ then
17:             return $(UNKNOWN, subprobs)$
18:         end if
19:         $backtrack(decisionLevel)$
20:     else
21:         if $!makeDecision()$ then
22:             return $(SAT, [\,])$
23:         end if
24:     end if
25: end while
```

B. Subproblem Generation

The goal of subproblem generation is to produce a set of subproblems that are difficult enough to meaningfully distinguish among the different strategies, but easy enough to be solved relatively quickly, so as not to introduce too much overhead. A key contribution of this paper is a new algorithm for achieving this goal. Notice again that this is fundamentally different from the goal of partitioning for divide-and-conquer algorithms, discussed previously.

Algorithm 2 shows our new algorithm. Because it is implemented by instrumenting an SMT solver, we have included simplified pseudocode for a standard CDCL(T) implementation with our changes highlighted in blue. As before, we start by walking through the execution of Algorithm 2.

In addition to the SMT formula ϕ, Algorithm 2 takes $numSubprobs$ as input, which specifies the desired number of subproblems that should be generated. On line 1, $subprobsMade$ is initialized to 0 in order to start tracking the number of subproblems. The list of subproblems is similarly initialized to be empty on line 2. The main changes occur immediately before the call to $backtrack$ on line 17. The inserted code is guarded by a call to $dumpCondition()$ which is a callback function whose role is to check any conditions that must be fulfilled before generating a subproblem. If $dumpCondition()$ returns $true$ (e.g., the solver is sufficiently "warmed up"), then a new subproblem consisting of the conjunction of ϕ and the decisions in the decision trail is added to $subprobs$, and $subprobsMade$ is incremented (lines 11-12). If the number of subproblems is equal to the target specified by $numSubprobs$, then the solver returns $UNKNOWN$ (signaling that the subproblem generation did not solve the problem) and the list of generated subproblems (lines 13-14). Otherwise, if some $abortCondition()$ holds (e.g., some time limit is reached), then $UNKNOWN$ is returned together with whatever subproblems have been generated so far. We discuss specific instantiations of $dumpCondition()$ and $abortCondition()$ in the next section.

Notice that solving is unperturbed unless the requested number of subproblems is created or the abort condition holds. The solver can simply solve the problem as usual and return SAT or $UNSAT$ (lines 8 and 22) if the conditions on lines 13 and 16 are never met. This behavior is important because it allows the instrumented SMT solver to potentially solve the original formula during subproblem generation. Its importance will become clearer in the discussion of implementation details. Briefly, though, the ability to solve the problem avoids wasting time tuning on subproblems for SMT formulas that are easy to solve. Conversely, it is also important to limit the amount of work done (via $abortCondition()$) during subproblem generation so as not to introduce too much overhead for more difficult problems.

The key insight of Algorithm 2 is that a subproblem can be generated from a discarded portion of the search space. At line 10, it is known that the current state of the decision trail results in a conflict. At the same time, the trail has not yet been backtracked, so it can be used to generate a subproblem. Intuitively, the generated subproblem should be easy to solve because we know it leads to a conflict. The subproblem should also require some nontrivial effort from the solver, since a conflict had to be produced.

It is important to note that while the generated subproblems are designed to be efficiently solvable, the actual runtime performance of the candidate solving strategies on those subproblems might vary significantly. In addition to the intrinsic differences in search patterns induced by different solving strategies, SMT solvers are known to exhibit high sensitivity to formula perturbations in some cases [20]. Adding the decision trail to the original problem may, therefore, result in unexpected runtimes for subproblem solving. Despite these caveats, we show in Section VI that, in practice, subproblems generated using Algorithm 2 are frequently both efficient to solve and helpful for selecting good strategies.

Finally, we highlight a few features of Algorithm 2 that contribute to its utility for selecting solving strategies when incorporated into a tuning approach. First, we note that when algorithm 2 is used for *makeSubproblems* in algorithm 1, it can be beneficial to vary the options to provide a greater diversity of subproblems. Additionally, through *numSubprobs*, Algorithm 2 allows flexibility in the number of subproblems that are created. This flexibility can be used to adjust the execution time of subproblem generation to balance it with the other stages of the algorithm. *dumpCondition*() can also be used to help balance execution time, but more generally, it is the main way that this algorithm enables fine-grained control over subproblem generation. Depending on how *dumpCondition*() is defined, it can affect subproblem diversity, the time required to generate subproblems, and other characteristics of the subproblem generation process. We discuss various implementations of *dumpCondition*() in the next section. Lastly, notice that because subproblems are generated by decisions that led to a conflict, they have the additional characteristic that they are always unsatisfiable. Our evaluation shows that tuning on only these unsatisfiable subproblems can be effective. Intuitively, it also makes sense, as unsatisfiable problems require the solver to cover the entire search space, and if the solver can do that efficiently, it should also be able to solve satisfiable instances quickly.

C. Solving Subproblems

The *solveSubprobs* routine takes as input a list of subproblems and a list of strategies. Its goal is to capture information about the performance of the different strategies on the subproblems. There are different ways this can be done. For simplicity, we simply run all subproblems using all strategies using some given time limit. When using Algorithm 2 for subproblem generation, solutions to subproblems can never lead directly to solutions to the original problem because: (*i*) they cannot, by construction, be satisfiable, and (*ii*) they do not fully partition the problem, so even if they are all *UNSAT*, we can't conclude anything about the original problem. However, when the disjunction of the subproblems is equisatisfiable with the original formula, as with formula-partitioning procedures, it is possible to solve the original problem based on the results of the subproblems. In this case, if any subproblem returns *SAT*, we can immediately return with status *SAT*. Similarly, if *all* generated partitions are solved and found to be *UNSAT*, then we can immediately return with status *UNSAT*. When the status of the original problem cannot be deduced based on the result of solving its subproblems, we return the status *UNKNOWN* together with the performance data obtained by running all the strategies on all the subproblems.

D. Subproblem-based Strategy Ranking

Algorithm 1 is parameterized by the *makeRanking* method. We propose ranking solving strategies based on the number of subproblems solved and breaking ties using the total runtime on the subproblems that were solved (a smaller total runtime is better). Note that ranking on time alone could result in solving strategies that solve fewer subproblems being placed higher in the ranked list. Since our priority is solving the original problem, we prioritize solving subproblems in our ranking.

V. Implementation Details

We implemented the subproblem generation method (Algorithm 2) by extending cvc5's internal SAT solver. Our prototype of Algorithm 1 was implemented in Python and makes subprocess calls to cvc5 for subproblem generation (*makeSubproblems*) and solving (*solveSubprobs*). Each strategy in *stratList* is a set of command-line options for cvc5. In this section, we discuss specific design choices we made during the implementation.

A. The Dump and Abort Conditions

The *dumpCondition*() and *abortCondition*() methods in Algorithm 2 are configurable via command-line options in our implementation. *dumpCondition*() implements a delay before subproblem generation is enabled. The motivation is to ensure a sufficient amount of meaningful activity has occurred before producing subproblems. *abortCondition*() also relies on tracking time. In our evaluation, it returns true whenever it has been more than five seconds since the start of the current invocation of Algorithm 2.

B. Making Subproblems

We implement two versions of *makeSubproblems*. The first is based on Algorithm 2 and the other on standard formula-partitioning methods. In the first implementation, Algorithm 2 is invoked repeatedly to generate a small number of subproblems. Each call uses one of the strategies in *stratList* and requests $N/stratList$.size() subproblems. Using the options from each of the strategies while generating subproblems helps generate diverse subproblems while mitigating the risk that subproblems are biased toward any single strategy. To further aid diversity, each call sets the time delay in *dumpCondition*() to a random time between 1.5 and 2.5 seconds. Importantly, only unique subproblems are used for tuning. If subproblem generation ever produces identical subproblems, the duplicate is discarded.

After this first round of generating subproblems, if we do not yet have N subproblems, we enter another loop. In this loop, we randomly pick a strategy and use it to generate $N/stratList$.size()+1 subproblems (we add one to increase the chances of getting new subproblems). As before, duplicates of any previously generated subproblem are removed. Whenever a strategy fails to produce any new subproblems, it is no longer eligible to be picked. The loop continues until N subproblems are generated or until all strategies have been deemed ineligible.

In the second mode where partitioning is used, we use the built-in partitioning capabilities of cvc5. Depending on the run and the partitioning procedure used, the number of subproblems generated may not match N exactly. If fewer

TABLE I: Options used for each logic.

Logic	(Alias) Options
QF_LRA, QF_RDL	(A) --miplib-trick --miplib-trick-subs=4 --use-approx --use-soi --lemmas-on-replay-failure --replay-early-close-depth=4 --replay-lemma-reject-cut=128 --replay-reject-cut=512 --unconstrained-simp
	(B) --no-restrict-pivots --use-soi --new-prop --unconstrained-simp
	(C) defaults
QF_LIA, QF_IDL	(D) --miplib-trick --miplib-trick-subs=4 --use-approx --lemmas-on-replay-failure --replay-early-close-depth=4 --replay-lemma-reject-cut=128 --replay-reject-cut=512 --unconstrained-simp --pb-rewrites --ite-simp --simp-ite-compress --no-use-soi
	(E) --miplib-trick --miplib-trick-subs=16 --use-approx --lemmas-on-replay-failure --replay-early-close-depth=4 --replay-lemma-reject-cut=16 --replay-reject-cut=64 --unconstrained-simp --pb-rewrites --ite-simp --simp-ite-compress --use-soi
	(C) defaults
QF_SLIA	(F) --strings-exp --no-jh-rlv-order
	(G) --strings-exp --strings-fmf --no-jh-rlv-order

than N are generated, we just exit, and Algorithm 1 fails. If more than N subproblems are generated, then N of them are randomly selected. For partitioning runs, CVC5 is limited to 60 seconds and is called using the default options.

C. Solving Subproblems and Ranking

Our implementation of $solveSubprobs$ attempts to solve each subproblem in $subprobs$ with each of the strategies in $stratList$ using CVC5. A solving timeout of 5 seconds is enforced for each attempt at solving a subproblem. The returned $stratInfo$ is a dictionary mapping the solving strategies to a tuple describing how many subproblems were solved using each solving strategy and the cumulative runtime for that solving strategy over the subproblems. Note that having more strategies in $stratList$ may increase the odds of finding the best solving strategy, but it comes at the cost of running each subproblem an additional time. Rankings are calculated in $makeRanking$ based on the ranking heuristic described in Section IV-D.

VI. Experimental Evaluation

In this section, we report on experiments designed to evaluate our approach. The main questions we investigate are:

- How does *Ecubes* compare to subproblem generation methods based on partitioning? [It consistently ranks more problems]
- Does solving strategies' performance on subproblems predict their performance on the original problem? [Yes for all tested subproblem generation methods]
- Can the proposed online strategy selection method reduce computational time? [In many cases]

A. Experimental Setup

We consider quantifier-free benchmarks from the following logics in the SMT-LIB benchmark library: QF_LRA, QF_LIA, QF_RDL, QF_IDL, and QF_SLIA. The first four were selected because they have been studied in previous work on partitioning [4], and specific solving strategies for those benchmarks have been published in the context of portfolio solving [21]. We added QF_SLIA because efficient string solving has been of particular practical interest recently [22].

For candidate solving strategies, we start with expert-selected options for each logic found in CVC5's competition script. These options were expanded by varying the decision heuristic and including default options, which was an effective portfolio technique for QF_LRA, QF_LIA, QF_RDL, and QF_IDL in [21]. We extend the method of varying the decision heuristic for expert-selected options to the QF_SLIA logic. Concretely, we consider the Cartesian product of the option sets shown in Table I and CVC5's three decision heuristics: justification, stop-only, and internal. In total, there are six solving strategies (option sets) for QF_SLIA, and nine for the other benchmark sets. As shown in Figure 1, each of these solving strategies is effective for many problems in each logic, with each strategy being the best single strategy for some portion of the benchmarks.

All experiments were run on a cluster with 48 nodes running Ubuntu 20.04 LTS, each with one AMD Ryzen 9 7950X CPU with 16 cores and just under 128 GB of RAM (127940MB).

We start by walking through a detailed analysis of our methods on the QF_LRA benchmarks (in the next three subsections); we then broaden our scope to include other logics. We focus first on a single logic to keep the analysis simple and understandable, and we choose these benchmarks because the majority of prior work on subproblem generation includes and often focuses on QF_LRA benchmarks.

B. Comparison of Techniques for Subproblem Selection

We compare three subproblem generation methods: The first method is the one based on Algorithm 2 and detailed in Section V-B. Recall that we call this method *easy-cubes*, or *ECubes* for short. The others are partitioning algorithms implemented in CVC5 [4]. One is a cube-and-conquer based approach that builds cubes using literals from the decision trail. We call this the *decision cube* method, or *Dcube* for short. The last method is based on scattering using literals from the decision trail. We call this *Dscatter*.

The metric used for comparison is *rankability*. A problem is rankable for a particular subproblem generation method if running Algorithm 1 using that method does not return $FAIL$ as a status and if the top strategy solves at least three subproblems (solving fewer than three suggests there is not enough data to meaningfully distinguish the strategies).

SMT-LIB contains 1753 QF_LRA problems. Of these, 1728 can be solved by at least one of the solving strategies. And of these, 517 are not trivially solved during subproblem generation (using *Ecubes*). We thus compare our subproblem generation methods on these remaining 517 benchmarks. We ran Algorithm 1 on each benchmark using several variations of our subproblem generation methods using an overall timeout of 1200s for each run.

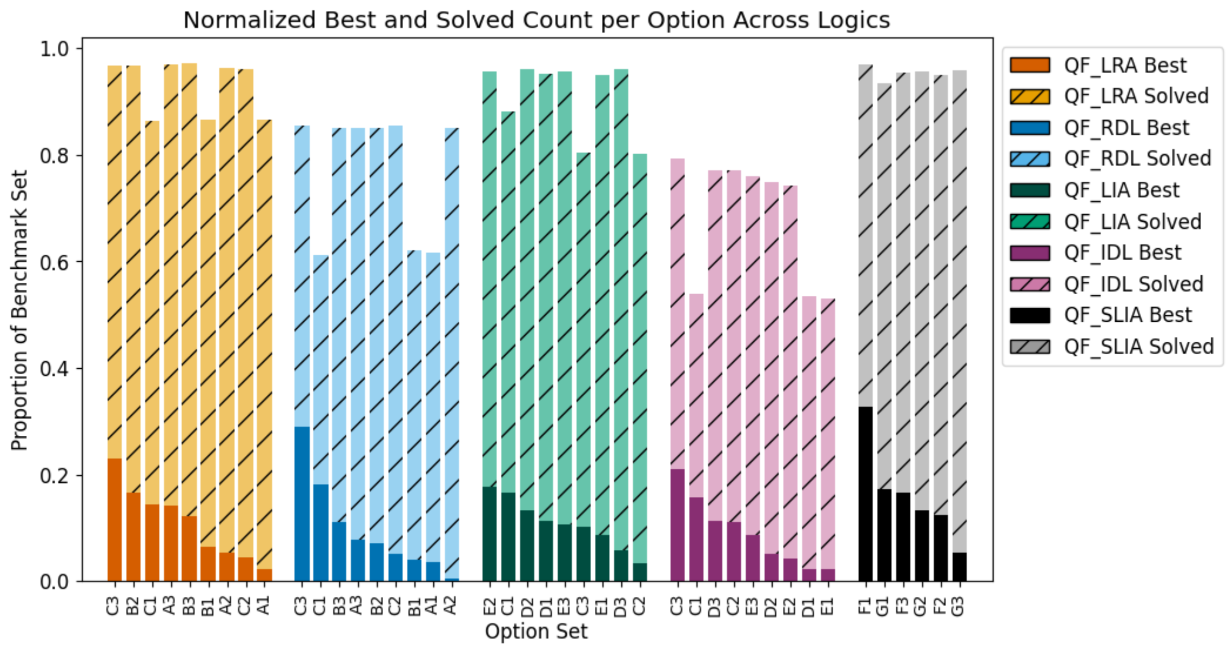

Fig. 1: This graph shows the proportion of problems that each option set can solve in each logic and additionally highlights the proportion of problems for which each option set is the best. Option sets are of the form XN where X is the options alias in Table I and N corresponds to a decision heuristic: justification, stop-only, and internal are aliased as 1, 2, and 3, respectively.

The results are shown in Figure 2. The x-axis shows each subproblem generation procedure we tested. The three main approaches are parameterized by the number of subproblems generated (the first number in parentheses) and the number of subproblems used (the second number). Recall that for partitioning methods, we may generate more subproblems than we use. In this case, we randomly sample from the generated subproblems (results using random sampling are highlighted in the figure as a reminder). For example, Ecube (18) 18 generates 18 easy-cubes and uses all of them, whereas Dcube (256) 27, generates 256 decision cubes and randomly selects 27 for tuning. The y-axis shows the number of problems successfully ranked. The Ecube variations result in the largest number of successful rankings, while Dscatter variants produce the fewest successful rankings. The results show that our new subproblem generation algorithm based on Algorithm 2 is the most effective for ranking our QF_LRA strategies on these benchmarks. Notice that the partition-based techniques struggle because the produced subproblems are too difficult, not because there are too few partitions being generated. For example, Dcube (256) 18 ranks more problems than Dcube (16) 16, even though creating 256 decision-cube partitions is more likely to fail than creating 16.

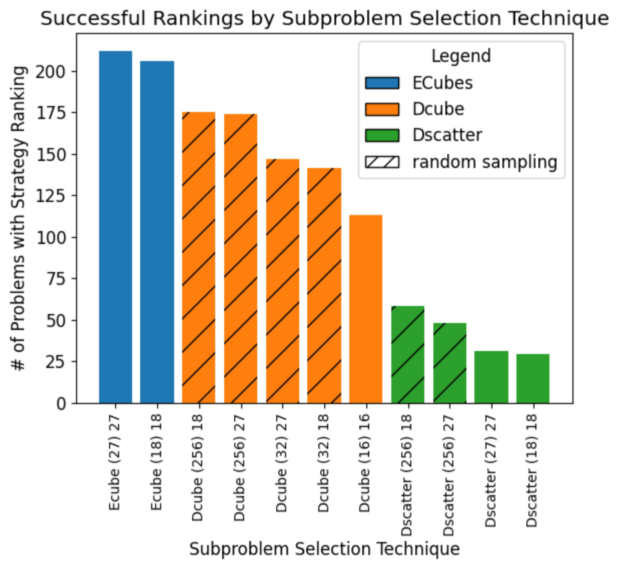

Fig. 2: Bar plot of the number of QF_LRA problems for which a strategy ranking can be produced using various subproblem selection techniques.

We next evaluate the difficulty of the subproblems generated by different methods. Recall that one goal is to produce relatively easy subproblems so that tuning does not take too long. Figure 3 visualizes the time required to solve all of the subproblems generated by each of the top four methods from Figure 2. Recall that we solve each subproblem with all 9 solving strategies, each with a 5s time limit. The horizontal line in the boxplot is the median, the box itself represents the values in the lower to upper quartiles, and the whiskers extend to show the full range of the data. Figure 3 shows that the typical time to solve all subproblems is much lower when the subproblems were generated using the easy-cubes procedure than when they were generated using decision-cube

TABLE II: Comparison of the number of problems solved by top-ranked strategies to the expected number of problems solved by selecting a random solving strategy.

Technique	Common		All Ranked	
	Top Rank Solved	Random Solved	Top Rank Solved	Random Solved
Ecube (18) 18	119/133	103.3/133	181/206	161.2/206
Dcube (256) 18	118/133	103.3/133	152/175	135.7/175
Ecube (27) 27	119/133	103.3/133	184/212	165.9/212
Dcube (256) 27	117/133	103.3/133	143/174	135.9/174

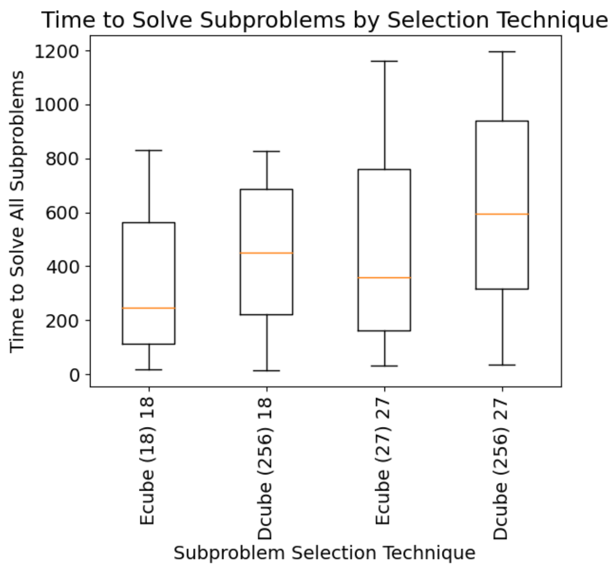

Fig. 3: Box plot showing the time required to solve all subproblems produced with a given subproblem selection technique.

partitioning, with the median of Dcube (256) 18 being almost double that of Ecube (18) 18.

C. Evaluation of the predictive power of rankings

So far, we have seen that, for QF_LRA, the Ecube method not only produces rankings more often, but also does so faster. However, it remains to seen whether these rankings are useful for predicting a good strategy for the original problem. We evaluate this next. Concretely, we evaluate how often the top-ranked solving strategy can solve the original problem.

Table II shows the number of instances that are solved when using the top-ranked strategy for each subproblem selection technique. It also shows the expected number of instances solved by choosing a random strategy. "Common" reports results on the 133 benchmarks ranked by all four methods, whereas "All Ranked" reports results on all benchmarks that each method was able to rank. Regardless of how the subproblems are generated, the ranking successfully selects a good solving strategy with substantially higher accuracy than a random choice. This result shows that tuning on generated subproblems is a promising technique for strategy selection. Since each subproblem generation technique results in a similar number of solved benchmarks, we conclude that the efficiency gain of Ecube (as shown in Section VI-B) does not come at the cost of reduced predictive power.

D. Comparison of online tuning with parallel portfolio

We now determine whether the per-instance strategy selection method could be used to improve solving efficiency in practice. We consider the following approach: first tune using Algorithm 1 and then iteratively attempt to solve the problem using the ranked solving strategies in order with a 20-minute per-strategy timeout. We compare the total computational time of this approach with that of a parallel portfolio solving method, where all strategies are attempted simultaneously (with a 20 minute timeout). The total compute time of our approach is computed as the sum of the time to produce subproblems, the time to solve each of the subproblems with all candidate solving strategies, and the time to try each strategy in the ranked list until the problem is solved. This means assigning a higher rank to a solving strategy that will not solve the problem is penalized because 1200 seconds will be added to the total time for each failed attempt. Note that each of the 6-9 option sets for each logic is the best strategy for some problems, as shown in Figure 1. While including ineffective strategies would clearly reduce the performance of the parallel portfolio, doing so could also sabotage our strategy: every option is considered during tuning, and including ineffective strategies could introduce noise into the ranking heuristic. We use the Ecube (18) 18 subproblem selection technique because it achieves a good balance between ranking many problems effectively and minimizing tuning time, as seen in prior subsections.

Given a set of problems and limited computational resources, online per-instance tuning and the parallel portfolio approach represent two distinct philosophies for utilizing those resources to tackle the problems. A parallel portfolio uses all available workers to attempt the same problem in parallel using different strategies with the hope that one strategy will solve it quickly. In contrast, online tuning solves a given problem with just one strategy at a time, but computational resources are spent on solving subproblems to choosing the strategy order.

We select benchmarks that (i) take at least 90 seconds to solve with the best strategy; (ii) are successfully solved by at least one strategy; and (iii) are successfully ranked using the Ecube (18) 18 method. We focus on harder benchmarks because our method is intended for solving problems where spending the resources required for tuning is justifiable. Note that easy benchmarks are likely to be solved during subproblem generation, so it makes sense to focus on harder problems.

Table III reports the total compute time of the two configurations across benchmark families, with the number of selected instances per family shown in the first column followed by, in parentheses, the number excluded due to each of (i), (ii), and (iii) above, respectively. Even though only three benchmarks remain in QF_RDL after the selection procedure, we include it for transparency.

TABLE III: Comparison of the total compute time required by a parallel portfolio and the Ecube (18) 18 tuning approach on selected problems from different benchmark sets. The aggregate improvement is reported with the per-instance median improvement in parentheses.

Logic Instances (Excluded)	Tactic	CPU Time (s)	Improvement (Median)
QF_IDL 73 (1745/463/247)	Parallel Portfolio	186123	32.5% (52.4%)
	Ecube (18) 18	125609	
QF_SLIA 51 (81661/2281/402)	Parallel Portfolio	87451	9.8% (25.0%)
	Ecube (18) 18	78845	
QF_LRA 23 (1654/25/51)	Parallel Portfolio	65248	50.2% (77.2%)
	Ecube (18) 18	32470	
QF_LIA 15 (12823/334/134)	Parallel Portfolio	47951	18.8% (36.6%)
	Ecube (18) 18	38934	
QF_RDL 3 (209/36/7)	Parallel Portfolio	9337	57.3% (1.5%)
	Ecube (18) 18	3987	

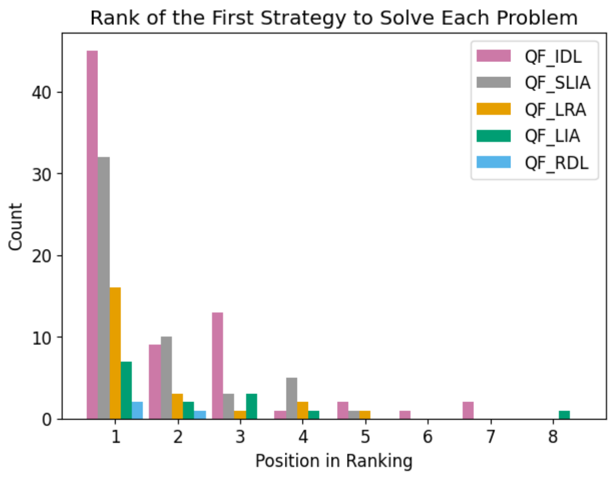

Fig. 4: Histogram of the rank of the first strategy to solve the problem using Ecube (18) 18 for the benchmarks reported in Table III.

Across all considered benchmark sets, our method results in an overall improvement in total compute time, ranging from 9.8% for QF_SLIA to 50.2% for QF_LRA. The per-instance median improvement demonstrates that for the majority of instances, total compute time is substantially reduced using our approach. Figure 4 shows the rank of the first strategy that solved the problem for each logic. The majority are solved by the top ranked strategy, as expected from the results in the previous subsection. Figure 5 shows the per-instance total compute time for the QF_LRA benchmarks. The scatter plot suggests that for these benchmarks, many benchmarks greatly reduce total compute time, while only a few use a modest amount of additional compute.

As shown in Table II, there are cases where our ranking method fails to put a successful strategy at the top, despite its overall effectiveness. Further analysis reveals that for those cases, the performance of the top-ranked strategy during tuning typically does not differ significantly from those of the other

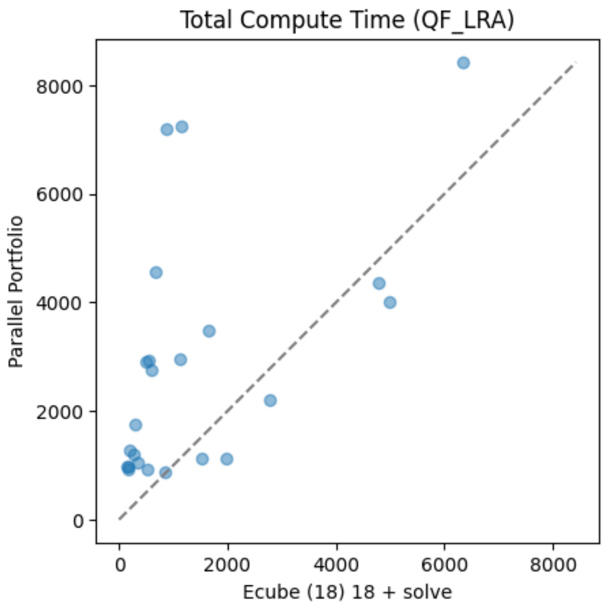

Fig. 5: Scatter plot of the total compute time of Ecube (18) 18 versus parallel portfolio.

TABLE IV: Comparison of the number of problems solved by top-ranked strategies to the expected number of problems solved by selecting a random solving strategy, when using stricter ranking criteria.

Technique	Common		All Ranked	
	Top Rank Solved	Random Solved	Top Rank Solved	Random Solved
Ecube (18) 18	60/62	49.6/62	136/149	117.8/149
Dcube (256) 18	59/62	49.6/62	136/144	112.1/144
Ecube (27) 27	60/62	49.6/62	144/166	130.4/166
Dcube (256) 27	58/62	49.6/62	126/143	111.8/143

strategies. Indeed, when we filter out benchmarks for which all strategies solve the same number of subproblems, the ability of our method to predict a strategy that will solve the original problem jumps—attaining an accuracy of over 90%, as shown in Table IV. Future work could consider how to best leverage this observation. For example, it could be beneficial to generate additional subproblems when the current set fails to produce a meaningful difference in scores for the candidate solving strategies.

VII. Conclusion and Future Work

In this paper, we demonstrated that strategy performance on a given SMT formula can be predicted from strategy performance on its easier subproblems. We showed that the correlation holds for existing subproblem generation methods—formula partitioning methods used for divide-and-conquer-styled parallel solving—as well as for a novel subproblem generation method. The new method can produce a ranking of solving strategies more efficiently than existing methods without sacrificing the quality of the ranking. We then proposed a solving methodology that first performs per-instance strategy selection online and then solves the instance with the ranked strategies. We showed that using this approach can reduce total compute time when compared to a portfolio approach that directly tries all strategies in parallel. Our paper is the first to demonstrate the feasibility of moving meta-algorithmic design *completely* online for SMT solving.

While our new subproblem generation method outperforms existing partitioning-based methods in our setting, there are still many challenging instances for which subproblems cannot be generated, as the complexity of the problem stems from theory reasoning rather than from the Boolean structure, or for which rankings cannot be produced because not enough subproblems are solved. This suggests it could be interesting to develop theory-specific subproblem generation methods or find ways to adjust the difficulty of subproblems by adding additional lemmas. Other future directions include 1) selecting among multiple solvers; 2) exploring a much larger pool of strategies using stochastic optimization techniques; and 3) validating the efficacy of the proposed workflow for other SMT solvers and theories.

References

[1] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to dpll(t)," *J. ACM*, vol. 53, no. 6, p. 937–977, nov 2006.

[2] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Hardware and Software: Verification and Testing*, K. Eder, J. Lourenço, and O. Shehory, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–65.

[3] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving sat in grids," in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–435.

[4] A. Wilson, A. Noetzli, A. Reynolds, B. Cook, C. Tinelli, and C. W. Barrett, "Partitioning strategies for distributed smt solving." in *FMCAD*, 2023, pp. 199–208.

[5] A. E. Hyvärinen, M. Marescotti, and N. Sharygina, "Search-space partitioning for parallelizing smt solvers," in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer, 2015, pp. 369–386.

[6] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, "Lookahead in partitioning SMT," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 271–279.

[7] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.

[8] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, "Machsmt: A machine learning-based algorithm selector for smt solvers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 303–325.

[9] L. Xu, H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010, pp. 210–216.

[10] R. Singh, J. P. Near, V. Ganesh, and M. Rinard, "Avatarsat: An auto-tuning boolean sat solver," Technical Report MIT-CSAIL-TR-2009-039. Massachusetts Institute of Technology, Tech. Rep., 2009.

[11] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140. [Online]. Available: https://doi.org/10.1007/978-3-319-40970-2_9

[12] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh, "Machine learning-based restart policy for cdcl sat solvers," in *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*. Springer, 2018, pp. 94–110.

[13] C. M. Li, W. Wei, and H. Zhang, "Combining adaptive noise and look-ahead in local search for sat," in *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*. Springer, 2007, pp. 121–133.

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

[15] M. S. Cherif, D. Habet, and C. Terrioux, "Combining vsids and chb using restarts in sat," in *27th International Conference on Principles and Practice of Constraint Programming*, 2021.

[16] A. Biere, "Adaptive restart strategies for conflict driven sat solvers," in *Theory and Applications of Satisfiability Testing–SAT 2008: 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings 11*. Springer, 2008, pp. 28–33.

[17] N. Pimpalkhare, F. Mora, E. Polgreen, and S. A. Seshia, "Medleysolver: online smt algorithm selection," in *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*. Springer, 2021, pp. 453–470.

[18] H. Wu, C. Hahn, F. Lonsing, M. Mann, R. Ramanujan, and C. Barrett, "Lightweight online learning for sets of related problems in automated reasoning," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2023, pp. 1–11.

[19] H. Wu, C. Barrett, and N. Narodytska, "Cubing for tuning," 2025. [Online]. Available: https://arxiv.org/abs/2504.19039

[20] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, "Mariposa: Measuring smt instability in automated program verification," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, 2023, pp. 178–188.

[21] C. Barrett, P.-W. Chen, B. Cook, B. Dutertre, R. B. Jones, N. Le, A. Reynolds, K. Sheth, C. Stephens, and M. W. Whalen, "Smt-d: New strategies for portfolio-based smt solving," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 1–10.

[22] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for aws access policies using smt," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–9.