



# Lookahead in Partitioning SMT

Antti E. J. Hyvärinen   
 USI, Switzerland  
[antti.hyvaerinen@usi.ch](mailto:antti.hyvaerinen@usi.ch)

Matteo Marescotti  
 USI, Switzerland  
[matteseo@gmail.com](mailto:matteseo@gmail.com)

Natasha Sharygina   
 USI, Switzerland  
[natasha.sharygina@usi.ch](mailto:natasha.sharygina@usi.ch)

**Abstract**—Lookahead in propositional satisfiability has proven efficient as a heuristic in pre- and in-processing, for partitioning instances for parallel solving, and as the main driver of a stand-alone solver. While applying similar techniques in satisfiability modulo theories is potentially equally useful, adapting lookahead to learning theory clauses and to estimating search space sizes in the presence of first-order structures is not straightforward. This paper addresses both of these observations. We give a hybrid algorithm that integrates lookahead into the state-based representation of an SMT solver and show that in the vast majority of cases it is possible to compute full lookahead up to depth four on inexpensive theories. We also show the role of first-order structures in SMT search space: while in most of our benchmarks the partitions are easier to solve than the original instance, we identify cases where lookahead results in sequences of increasingly difficult instances for a computationally expensive theory.

## I. INTRODUCTION

Large scale parallel SMT solving that would result in linear speed-up reliably over any instance in a cloud environment is a lucrative prize that has been intensively studied over the recent years [26], [14], [13], [17]. A central sub-goal in this project is in understanding how to apply successfully the *cube-and-conquer* [24] approach in SMT solving. The lookahead heuristic in propositional logic [27], in addition to being efficient in solving certain types of structured problems [8], has recently proven to be a powerful tool in constructing partitions for divide-and-conquer-based parallel SAT solvers [10], [9]. The idea is to base the search-space traversal on the explicit principle of branching on literals that reduce maximally the remaining search space. In addition to SAT solvers, the heuristic has been implemented in SMT solvers such as Z3 [20], where it serves for in- and pre-processing, and by us in OpenSMT [11], [12] as an alternative implementation for the main SAT solver.

This paper studies how the literals chosen by lookahead algorithm for SMT affect the difficulty of the instance from the perspective of a standard CDCL-based SMT solver. This question is central to divide-and-conquer-style parallel SMT solving, where the lookahead heuristic is used to build a binary *lookahead tree* of depth  $d$ , with nodes labeled by the literals chosen with the lookahead heuristic, and root labeled with the true literal  $\top$ . Conjoining the literals in each rooted path to the leaves with the original instance produces  $2^{d-1}$  partitioned instances that do not share models. The resulting instances can be solved in parallel, and the original instance is satisfiable if and only if one of the partitioned instances is satisfiable.

Our main contributions are rigorously defining what we mean by lookahead heuristic for an SMT solver, and an experimental study on how the use of this heuristic affects the difficulty of the partitions. In defining the heuristic, we show that lookahead can be integrated tightly into a CDCL(T)-style algorithm that fully leverages learned clauses, including determining unsatisfiability while constructing partitions. We summarize our experimental results as follows. First, in many cases the heuristic runs in seconds when producing a non-trivial number of partitions (say, 16). This is already a non-trivial observation given that the full lookahead heuristic in SAT is known to be in most cases prohibitively expensive. Second, usually the approach results in partitions that are easier to solve than the original. While this result seems rather implicit and obvious, it is made interesting by the next observation: There are instances where the above described lookahead-based parallel algorithm's run time *increases* compared to the original instance even when no overhead from partitioning or communication is considered, and the number of partitions is in the thousands. We show some details on the latter cases that help to understand the underlying phenomena, and identify a possible reason arising from the way the theory solving algorithm for linear real arithmetics is implemented in most SMT solvers. These cases serve to illustrate the complexity of the ultimate goal of an efficient and general parallel solver.

Combining a lookahead algorithm with a CDCL-based SMT solver in a meaningful way is not straightforward. First, the lookahead heuristics assumes that the clauses of an instance are known at computing time. In contrast, an SMT solver produces a new clause whenever a propositional model is inconsistent in the theory. A potentially very large number of clauses remain invisible for the heuristic. Second, the explanation clauses guide the search through non-chronological backtracking. This means that the heuristic scores of variables change with each backtrack, and the algorithm may determine unsatisfiable entire sub-trees of the lookahead tree. The subtrees need to be re-computed to ensure that the approach produces  $2^d$  partitions. Finally, it is not clear how SMT solver's theory specific reasoning part interacts with the lookahead-heuristic that only measures the reduction in the propositional space.

To the best of our knowledge, this paper is the first to build lookahead partitioning into the SMT framework in a way that observes the search space reduction resulting from learned clauses, and guarantees the unit-propagation consistency of the resulting partitions in case instance satisfiability is not de-

terminated. We consider the theories of uninterpreted functions with equality [3] and linear real arithmetic [4]. These are the two central algorithms that constitute, together with a SAT solver, the core of most SMT solvers. Combinations of these two theories with pre-processing techniques are capable of handling the quantifier-free subset of the SMT-LIB benchmark library instances. The algorithm either produces exactly  $2^{d-1}$  instances none of which can be shown unsatisfiable through (theory-aware) unit-propagation in the current state of the SMT solver; or shows the original instance either satisfiable or unsatisfiable. The partitioning algorithm compromises in certain cases the exactness of the lookahead scores for decreased run time. We believe that the efficiency of our proof-of-concept implementation forms a solid basis for future research in this direction. Since the approach also sheds light to the observed slowdowns, we believe that the work will prove useful for designing more general parallelization algorithms for SMT.

The paper is organized as follows. After discussing related work, in Sec. III we define our SMT-related logical notation. In Sec. IV we adapt the rule-based description of SMT from [25] to the specific case of lookahead and introduce a running example. In Sec. V we present our lookahead partitioning algorithm, then provide experimental results in Sec. VI, and conclude in Sec. VII.

## II. RELATED WORK

The lookahead heuristic was first introduced in the context of DPLL-based SAT solving in [27]. The original idea uses the number of propagated literals as a measure of search space reduction [23], and is further extended to consider, e.g., equivalence reasoning [5], the clause-based Jeroslow-Wang heuristic [16], and approaches for choosing which variables to consider for lookahead [7].

Lookahead as a pre- and in-processor for clause-learning SAT solvers was formalized in [6]. However, it was not integrated into the CDCL algorithm in the sense that is done in this work. A similar pre- and in-processing approach was recently implemented for the SMT solver Z3 [20]. When used as a pre- and in-processor for an ordinary, CDCL-based solver, the lookahead implementation can be conceptually fairly straightforward. Lookahead is not directly involved in the CDCL search, and therefore the artifacts related to non-chronological backtracking need not be necessarily considered. In [12] we formalized an algorithm inspired by the lookahead heuristic for solving quantifier-free first-order formulas based on CDCL SMT solving. The approach is implemented in our SMT solver OpenSMT [11] and was shown experimentally to be efficient for solving linear integer arithmetic problems with Boolean structure. Compared to the publication, in the current work we give a more formal treatment of the implementation, define the lookahead algorithm for partitioning, and provide experimental data and analysis for parallel solving based on cube-and-conquer.

Our focus is in how SMT lookahead can implement partitioning in divide-and-conquer for parallel solving. The idea was introduced for parallel SAT solving in [10], and an

implementation for parallel SMT solving was used in [13], [17]. However, the details of this partitioning approach have not been discussed before. The lookahead-based partitioning implementation in [10] applies essentially lookahead-based binary partitioning recursively. The downside of this design is that it does not use the full information in the CDCL solver, and producing the partitions might miss an unsatisfiability high up in the tree. As a result it constructs partitions that are known to be unsatisfiable in an intermediate state of the partitioning algorithm.

The substantial amount of research in SAT heuristics, overviewed in [1] from the perspective of parallel solving, provides a promising foundation for partitioning in SMT. Recent relevant approaches include [15], where the authors recognize high-level information that can be used for better clause learning.

## III. PRELIMINARIES

The *Satisfiability Modulo Theories* (SMT) problem [22], [3] consists of determining whether a propositional formula is satisfiable, given that some of the atoms have an interpretation in first-order logic. A *conflict-driven clause learning* (CDCL) SMT solver searches first for propositional models, which are then checked for consistency with respect to the theory. If found inconsistent, the propositional structure is enriched with an *explanation*, that is, a clause containing in general theory atoms. If instead during the process the propositional part becomes unsatisfiable, the solver has shown the whole formula unsatisfiable. The formula is satisfiable if the solver finds a theory-consistent model.

1) *SMT solving*: This section fixes the notation for first-order logic and SMT. We define sets of function symbols, terms, constants, and predicate symbols as usual, the last containing the special symbols  $\top$ ,  $\perp$ , and  $=$  that represent, respectively **true**, **false**, and equality. We call applications of predicate symbols on terms *atoms*. Let  $U$  be a possibly infinite set of elements containing at least the truth values **true** and **false**. A *model*  $\mathcal{M}$  assigns to each constant a unique element from  $U$ , to each function symbol of arity  $n \geq 1$  a total function  $U^n \rightarrow U$ , to each predicate symbol of arity zero a truth value **true** or **false**, and to each predicate symbol of arity  $n \geq 1$  a total function  $U^n \rightarrow \{\mathbf{true}, \mathbf{false}\}$ . An *interpretation*  $\mathcal{A}$  is the extension of  $\mathcal{M}$  to general terms in the usual sense.

Given a finite set of atoms  $\text{At}$ , a *clause* is a set of *literals*, that is, positive and negative atoms  $x, \neg x$ ,  $x \in \text{At}$ . We extend the negation to clauses, and write  $\neg(l_1 \vee \dots \vee l_n)$  for  $\neg l_1 \wedge \dots \wedge \neg l_n$ . A *propositional formula in conjunctive normal form* (CNF) is a conjunction of clauses. Throughout the text we use both a set of literals and disjunction, and a set of clauses and a conjunction, interchangeably. We also treat conjunctions of unit clauses (*cubes*) as sets of literals when this cannot be confused with a disjunction. A sequence of literals is written  $l_1 \dots l_n$ , and when the order plays no role, we equate the sequence with the corresponding set  $\{l_1, \dots, l_n\}$ .

A set of literals  $X$  is *consistent* if for no  $x$  both  $x \in X$  and  $\neg x \in X$ . A consistent set  $\sigma$  is called an assignment. An

assignment is *total* if for all atoms  $x \in \text{At}$  either  $x \in \sigma$  or  $\neg x \in \sigma$ . An atom  $x$  is *assigned* if either  $x \in \sigma$  or  $\neg x \in \sigma$ . The assignment  $\sigma$  satisfies a clause  $c$  when  $\sigma \cap c \neq \emptyset$ , and a formula  $\phi$  if it satisfies all clauses of  $\phi$ . A *theory*  $T$  is a non-empty set of models. A CNF formula  $\phi$  is *T-satisfiable* if (i) there exists a satisfying total assignment  $\sigma$  for  $\phi$  and an interpretation  $\mathcal{A}$  that is an extension of a model  $\mathcal{M} \in T$ , and (ii) for each  $l \in \sigma$ ,  $l^{\mathcal{A}} \equiv \mathbf{true}$  if  $l$  is of the form  $x$ ; and  $l^{\mathcal{A}} \equiv \mathbf{false}$  if  $l$  is of the form  $\neg x$ , where  $x$  is an atom of  $\phi$ . In particular, given a formula  $\phi$  and an assignment  $\sigma$  that is total (with respect to  $\phi$ ), we write  $\sigma \models_T \phi$  if  $\sigma$  is such an assignment. In addition we write  $\phi' \models_p \phi$  if all assignments that satisfy  $\phi'$  also satisfy  $\phi$  propositionally, and  $\models_T c$  if  $c$  is entailed by the theory, that is, a *theory lemma* of a theory  $T$ . For a formula, clause, literal, or assignment  $\xi$  we denote by  $\text{Ats}(\xi)$  the set of atoms appearing in  $\xi$ .

In this work we study two theories: the theory of linear real arithmetic (LRA) and the theory of uninterpreted functions with equality (EUF). The universe of LRA consists of real numbers, function symbols  $*$  and  $+$  of arity two restricted to expressing linear terms, and the predicate symbol  $\leq$ ; all three have their usual interpretations. The EUF theory places no restrictions on the interpretations of constants, functions, or predicates (apart from the inherent ones for equality,  $\top$ , and  $\perp$ ).

2) *Parallel SMT solving*: Given an SMT instance  $\phi$ , *partitioning* produces instances  $\phi_1, \dots, \phi_k$  such that the satisfiability of  $\phi$  is equal to the satisfiability of the disjunction  $\phi_1 \vee \dots \vee \phi_k$ . In addition, we are interested in partitionings such that no two partitions  $\phi_i, \phi_j$ ,  $i \neq j$ , share a total satisfying assignment. The *partitioning approach*  $\text{Part}(k)$  consists of solving an SMT instance  $\phi$  by first constructing the partitions  $\phi_1, \dots, \phi_k$ , and then solving each resulting partition  $\phi_i$  in parallel until one of them is shown satisfiable, or all of them are shown unsatisfiable.

#### IV. CONFLICT-DRIVEN CLAUSE-LEARNING LOOKAHEAD IN SMT

The *CDCL lookahead algorithm* intuitively guides an SMT solver in a binary tree, using the solver's state to determine how to expand the tree. To more precisely describe the algorithm, we adapt here the rule-based presentation of CDCL( $T$ ) from [25], [21] to our needs. As usual, in the first phase an input SMT formula is converted into an equisatisfiable propositional formula  $\phi$  in CNF while preserving the atoms in the theories  $T$ . The *state*  $\langle \sigma \mid F \rangle$  of an SMT solver consists of  $\sigma$ , an initially empty assignment, and  $F$ , a set of clauses initially consisting of  $\phi$ . The execution of the solver proceeds according to a set of rules described below. In general, the algorithm alternates between *propagation*, choosing a *decision literal*, denoted by  $x^\delta$ , and analysing conflicts found in propagation. The labels  $L$  and  $E$  refer to *learned* and *explanation* clauses. When they appear on the left side of  $\rightarrow$ , the corresponding rule matches only to clauses that have the label.

- The *propagation* rule  $\langle \sigma \mid F \wedge (c \vee l) \rangle \xrightarrow{\text{Prop}} \langle \sigma l \mid F \wedge (c \vee l) \rangle$  where  $c$  is a clause, and  $\neg c \subseteq \sigma$ ,  $l \notin \sigma$  and

$\neg l \notin \sigma$ , expands the assignment with literals that are logical consequences in the current state.

- The *theory propagation* rule  $\langle \sigma \mid F \rangle \xrightarrow{\text{TProp}} \langle \sigma l \mid F \wedge (c \vee l)^L \rangle$  uses theory lemmas to lift information to the propositional level allowing new literals to propagate. It can be applied if  $\sigma \models_T l$ ,  $l$  or  $\neg l$  appears in  $F$ ,  $l \notin \sigma$  and  $\neg l \notin \sigma$ , and  $c$  is a clause such that  $\sigma \models_T \neg c$  and  $\models_T c \vee l$ .
- The *decision* rule  $\langle \sigma \mid F \rangle \xrightarrow{\text{Dec}} \langle \sigma l^\delta \mid F \rangle$  decides a literal  $l$ , where  $l$  or  $\neg l$  appears in  $F$ , and  $l \notin \sigma$  and  $\neg l \notin \sigma$ .
- The *theory explanation* rule  $\langle \sigma \mid F \rangle \xrightarrow{\text{TExp}} \langle \sigma \mid F \wedge c^E \rangle$  is used to lift theory to propositional level based on observed conflicts in the theory solver. It can be applied when each atom of  $c$  appears in  $\langle \sigma \mid F \rangle$ ,  $\sigma \models_T \neg c$ , and  $\models_T c$ .
- the *propositional explanation* rule  $\langle \sigma \mid F \rangle \xrightarrow{\text{PExp}} \langle \sigma \mid F \wedge (c_1 \vee c_2)^E \rangle$  is the standard resolution rule, which can be applied if  $c_1 \vee x \in F$  and  $c_2 \vee \neg x \in F$ . However, due to the invariants of the underlying SAT solver, we require in addition that  $\neg c_1 \subseteq \sigma$  and  $\neg c_2 \subseteq \sigma$ .
- the *backjump* rule  $\langle \sigma l^\delta \sigma' \mid F \wedge c^E \rangle \xrightarrow{\text{BJ}} \langle \sigma l' \mid F \wedge (c' \vee l')^L \rangle$  learns clauses that steer the search. It is applicable if  $\neg c \subseteq \sigma l^\delta \sigma'$ , there is a clause  $c' \vee l'$  such that (1)  $F, c \models_p c' \vee l'$  and  $\neg c' \subseteq \sigma$ ; (2)  $l' \notin \text{Ats}(\sigma)$  and  $\neg l' \notin \text{Ats}(\sigma)$ ; and (3)  $l'$  or  $\neg l'$  occurs in  $\sigma l^\delta \sigma'$  or  $F \wedge c$ .
- The *fail* rule  $\langle \sigma \mid F \wedge c \rangle \xrightarrow{\text{Fail}} \perp$  corresponds to determining unsatisfiability. It is applicable if  $\neg c \subseteq \sigma$ , and  $\sigma$  contains no decision literals.
- The *reset* rule  $\langle \sigma \mid F \rangle \xrightarrow{\text{Reset}} \langle \emptyset \mid F \rangle$  can be applied at any time.
- the *forget* rule  $\langle \sigma \mid F \wedge c^L \rangle \xrightarrow{\text{Forget}} \langle \sigma \mid F \rangle$  is used for forgetting learned clauses, essentially to keep memory usage in control
- The *undo* rule  $\langle \sigma l^\delta \sigma' \mid F \rangle \xrightarrow{\text{Undo}} \langle \sigma \mid F \rangle$  is finally required to implement the backtracking while computing lookahead.

A CDCL( $T$ )-based SMT solver works by applying the above rules with two restrictions. (i) The solver always computes the *unit propagation closure* before deciding a new literal, i.e. the rule *Dec* is never applied if the rule *Prop* is applicable; and (ii) to notice any theory inconsistencies when a propositional assignment is found, if the rule *Dec* cannot be applied (i.e., all atoms are assigned) the solver applies the rule *TProp*. The solver always terminates if both the rules *Reset* and *Forget* are applied with an increasing interval [2].

Since the unit-propagation closure has a central role in computing lookahead, we give here two useful, related definitions in the above notation. Given a solver state  $\langle \sigma \mid \phi \rangle$ , the *unit propagation closure*  $UP(\sigma, \phi)$  is the set of literals  $\sigma' \supseteq \sigma$ , where  $\langle \sigma' \mid \phi \rangle$  is the state obtained by applying the rules *Prop* and *TProp* until neither one applies. A solver state  $\langle \sigma \mid \phi \rangle$  is called *unit propagation consistent* or *consistent* if the set  $UP(\sigma, \phi)$  is consistent.

The following running example illustrates the use of the

rules. The notation  $Prop^*$  indicates a sequence of propagations.

*Example 1:* Consider the conjunction  $F = (\neg x \vee (b \leq c))^{(1)} \wedge (\neg x \vee (a \leq b))^{(2)} \wedge (\neg(a \leq d) \vee \neg(a \leq b) \vee \neg(a \leq c))^{(3)} \wedge ((c \leq d) \vee \neg(b \leq c) \vee (a \leq d))^{(4)} \wedge ((c \leq d) \vee \neg(a \leq d) \vee (a \leq c))^{(5)}$  where the numbers in parentheses label the clauses. The following is a possible computation of the CDCL( $T$ ) system.

$$\begin{aligned} \langle \emptyset \mid F \rangle &\xrightarrow{Dec} \langle x^\delta \mid F \rangle \xrightarrow{Prop^*} \langle x^\delta(b \leq c)(a \leq b) \mid F \rangle \xrightarrow{Dec} \\ &\langle x^\delta(b \leq c)(a \leq b)\neg(c \leq d)^\delta \mid F \rangle \xrightarrow{Prop^*} \\ &\langle x^\delta(b \leq c)(a \leq b)\neg(c \leq d)^\delta(a \leq d)\neg(a \leq c) \mid F \rangle \xrightarrow{PExp} \\ &\langle x^\delta(b \leq c)(a \leq b)\neg(c \leq d)^\delta(a \leq d)\neg(a \leq c) \mid \\ &\quad F \wedge ((c \leq d) \vee \neg(b \leq c) \vee (a \leq d)) \rangle \xrightarrow{BJ} \\ &\langle x^\delta(b \leq c)(a \leq b) \mid F \wedge C_1^L \rangle \end{aligned}$$

where the learned clause, obtained by resolution, is  $C_1^L := (c \leq d \vee \neg b \leq c \vee \neg a \leq b)^L$ . Continuing the example, we get

$$\xrightarrow{TProp} \langle x^\delta(b \leq c)(a \leq b)(c \leq d)(a \leq c) \mid F' \rangle$$

where  $F' := F \wedge C_1^L \wedge (\neg(a \leq b) \vee \neg(b \leq c) \vee (a \leq c))^L$ , the last being a valid clause in the theory, and

$$\begin{aligned} &\xrightarrow{Prop^*} \langle x^\delta(b \leq c)(a \leq b)(c \leq d)(a \leq c)\neg(a \leq d) \mid F' \rangle \\ &\xrightarrow{TExp} \langle x^\delta(b \leq c)(a \leq b)(c \leq d)(a \leq c)\neg(a \leq d) \mid \\ &\quad F' \wedge (\neg(a \leq c) \vee \neg(c \leq d) \vee (a \leq d)) \rangle \xrightarrow{BJ} \\ &\langle \neg x \mid F' \wedge \neg x^L \rangle \end{aligned}$$

where  $\neg x^L$  is obtained through a resolution derivation on clauses in  $F'$  and the explanation.

## V. LOOKAHEAD-BASED PARTITIONING FOR SMT

This section describes the lookahead-based algorithm for partitioning an SMT instance into  $2^d$  partitions or determining whether the instance is satisfiable.

### A. The Lookahead Score

Lookahead in a backtracking search consists in general of repeated trial and backtracking on all available branches at a certain point of the search, and committing to the one that seems most promising. We define the relation between SMT solver states before and after the trial branch, and the lookahead score as the difference between the two. The approach is oblivious to the details on how the lookahead score between two states  $s$  and  $s'$  is defined. Our implementation supports two scoring functions, one based on the number of free atoms in the instance globally [23], and the other on unassigned atoms in the clauses of the instance [8]. Our examples and experiments in this paper use the former, and we give a comparison to the latter in Appendix K.

Lookahead aims to assign with the rule  $Dec$  the literal that minimizes the upper bound for the remaining search space. Given a state  $s$  where neither  $Prop$  nor  $TProp$  applies, we define the *lookahead step* on a literal  $l$  as the sequence of rules starting from  $s$ , having  $Dec$  on  $l$  as the first rule, followed by

unit propagation closure computation resulting in the state  $s'$ , and finally an  $Undo$  on  $l$  ending in state  $s$ . This sequence is not always possible, and we describe in Sec. V how we handle the failed cases. For a consistent state  $\langle \sigma \mid \phi \rangle$ , the set  $UP(\sigma, \phi)$  is unique. Therefore we can define the lookahead score of a literal  $l$  based on a difference between  $\langle UP(\sigma, \phi) \mid \phi \rangle$  and  $\langle UP(\sigma l, \phi) \mid \phi \rangle$ . We denote the *lookahead score* of literal  $l$  by  $score(l) = |UP(\sigma \cup \{l\}, \phi) \setminus UP(\sigma, \phi)|$ , that is, the number of propagated literals after deciding  $l$ , and extend the definition to atoms  $x$  as

$$score(x) = \min(score(x), score(\neg x)), \quad (1)$$

which minimizes the sum of the upper bounds for the remaining search spaces [23].<sup>1</sup>

### B. Lookahead-Based Partitioning

---

**Algorithm 1:** The lookahead partitioning algorithm.

---

**Input** : An SMT instance  $\phi$  in CNF; Tree depth  $d$   
**Output**: Sat, Unsat, or a balanced binary tree of depth  $d$   
**Data** : Solver  $s$ , DFS stack  $stack$

```

1 restart ← true
2 while restart do
3   restart ← false;
4   r ← empty node;
5   stack.push(r);
6   while stack.size ≠ 0 do
7     n ← stack.pop();
8     res ← setSolverToNode(s, n);
9     if res = Unsat then return Unsat;
10    if res = BackJump then
11      restart ← true;
12      break;
13    if Depth of n is d then continue;
14    c, c', res ← expandTree(s);
15    if res = Unsat then return Unsat;
16    if res = Sat then return Sat;
17    if res = BackJump then
18      restart ← true;
19      break;
20    stack.push(c);
21    stack.push(c');
22  end
23 end
24 return the tree rooted at r;
```

---

The approach is presented in Alg. 1. The algorithm constructs a tree with nodes labelled with literals. The tree is constructed depth-first using the  $stack$ , with the help of a CDCL( $T$ ) SMT solver  $s$ . The intuition is that the tree is being built by guiding the SMT solver along the rooted paths and lookahead heuristic is used to expand a leaf node. The algorithm limits the search depth to the input value  $d$ , and is also a sound but incomplete (if  $|Ats\phi| > d$ ) SMT solver.

Let  $n^i$  denote a node  $n$  at depth  $i$  in the tree. Then each path in the tree from the root  $n^0$  to a leaf  $n^i$  corresponds to a partition as follows. We label the nodes  $n$  with a literal

<sup>1</sup>There are other definitions for lookahead score, but they all favor atoms that minimize the remaining search space on both polarities [8].

$Lab(n)$ , and  $n^0$  is labelled  $Lab(n^0) = \top$ . A path  $n^0 \dots n^i$  is interpreted as a cube, and  $n^0 \dots n^d$  in the tree corresponds to the partition  $\phi \wedge Lab(n^0) \wedge \dots \wedge Lab(n^d)$ .

The main work, done in the loop between lines 6 – 22, consists of two phases: *setting the solver  $s$  to a given node* on Line 8, and *expanding the lookahead tree* on Line 14. We describe both phases, referring to the rules in Sec. III.

1) *Expanding the lookahead tree*: The lookahead tree is expanded with new nodes  $c, c'$  by the function *expandTree* on Line 14. Using the solver  $s$  the function computes the lookahead step for each literal  $x, \neg x$  not assigned in  $\sigma$  as described in Sec. V-A. The process may be interrupted by three special conditions:

- The rule *Fail* becomes applicable. In this case the function returns *Unsat*.
- A total assignment is found: the function returns *Sat*.
- The rule *BJ* becomes applicable. In this case:
  - If *BJ* becomes applicable with  $l^\delta = x$  or  $l^\delta = \neg x$ , the function does a *local restart*: it forgets the computed lookahead scores and restarts the lookahead computation.
  - If *BJ* is applicable with  $l^\delta = y$  or  $l^\delta = \neg y$  for some earlier decision literal  $y \neq x$ , the function does a *complete restart* by returning *BackJump*.

If *expandTree* determines satisfiability, the algorithm terminates and reports the result immediately. The distinction between local and complete restarts is motivated by efficiency and has deep implications to the algorithm. We discuss this point in Sec. V-B3.

2) *Setting the solver to a given node*: A lookahead path obtained from the stack is used to set the solver  $s$  to the correct state where the lookahead scores of literals can be computed. This is done in Line 8 by the call to the function *setSolverToNode* that takes as arguments the solver  $s = \langle \sigma \mid F \rangle$ , and the current node  $n = n^k$ . The function initially applies the rule *Reset* on the solver, and computes the unit propagation closure at the root by  $\sigma = UP(\emptyset, F)$ . Then, for each  $n^0 \dots n^k$  the function applies *Dec* with  $l = Lab(n^i)$ , and sets  $\sigma = UP(\sigma l, F)$ . The process may be interrupted in two cases:

- *Fail* becomes applicable. This corresponds to the derivation of unsatisfiability, and the process returns *Unsat*.
- *BJ* becomes applicable. The node is locally unsatisfiable and our implementation restarts the construction of the lookahead tree to avoid unbalancedness.

Otherwise, setting solver to the node succeeds and the algorithm proceeds with expanding the tree.

To clarify the behavior of the algorithm, we show its execution on the running example (Example 1).

*Example 2*: Let  $\phi = F$  from Ex. 1 and  $d = 2$  for Alg. 1. The algorithm advances to line 14 to compute the lookahead scores of the variables using solver  $s$ . No conflicts are detected by  $s$ , literal  $x$  propagates  $\{b \leq c, a \leq b\}$ , and literals  $\neg b \leq c$  and  $\neg a \leq b$  propagate  $\{\neg x\}$ . No other branch results in

propagations. Hence the score from Eq. (1) is zero for all atoms.

Say the algorithm expands the tree, that up to now consisted only of the empty root, with nodes labeled  $\neg x, x$ , and pushes both nodes to the DFS stack. Assume that the algorithm first branches on  $\neg x$ . None of the free literals propagate, and tree is expanded for example with  $\neg a \leq d$  and  $a \leq d$ . Once these are popped from the stack, the tree would consist so far of branches  $(\neg x(a \leq d))$ ,  $(\neg x \neg(a \leq d))$ , and  $(x)$ .

The algorithm will now pop  $x$  on line 7. On line 14, during the execution of the lookahead heuristic, the algorithm will do the lookahead step on  $b \leq c$ . This triggers the conflict-handling sequence shown in Ex. 1 resulting in the solver state  $\langle \neg x \mid F \wedge ((c \leq d) \vee \neg(b \leq c) \vee \neg(a \leq b))^L \wedge (\neg(a \leq b) \vee \neg(b \leq c) \vee (a \leq c))^L \rangle$ . Backjump is on the earlier decision literal  $a \leq c$ , not on the most recent decision literal  $b \leq c$  (see the description above for *expandTree*), and therefore *expandTree* will return *BackJump*, restarting the tree construction.

The algorithm builds now the tree similar to the first time, but when computing lookahead in state  $\langle x(b \leq c)(a \leq b)(c \leq d)(a \leq c)\neg(a \leq d) \mid F' \rangle$  there are no free variables, and the algorithm reports satisfiability.

3) *Observations on the backjumps*: The backjump during the above execution is critical for the partition quality. It is relatively easy to see that applying recursively a lookahead algorithm on the original problem, as in [10], produces partitions that in a later state of the solver would not be unit-propagation consistent.

First, one could imagine a version of the algorithm that backtracks to the level indicated by the backjump, similar to the underlying SMT solver. This choice would intuitively result in less repeated work as the previously built lookahead tree would be preserved, and therefore conceivably in a more efficient algorithm. However, there are two reasons why the restart is necessary. First, a clause  $c$  learned in a backjump at *expandTree* on node  $n^i$  alters the lookahead scores in an unpredictable way in the solver states closer to the root. The current lookahead tree becomes in general invalid from the heuristic perspective. Without the restart, the clause should be considered in all previous invocations of *expandTree* at least in the nodes  $n^0 \dots n^{i-1}$ , and tracking such propagations would be expensive. Second, allowing backjumps in the lookahead tree means that when setting the solver to a new node (Line 8), a learned clause can cause a conflict not present when the node was pushed (lines 20 and 21). In this case it is unclear how the algorithm should proceed to construct the balanced binary tree with consistent partitions.

The distinction between local and complete restarts stems from the above two observations. Complete restarts are too expensive to be performed on every conflict, a relatively common event during the lookahead computation. Instead, they are done only on the long backjumps that are rare in lookahead-based branching. The consequence of having the local restarts is that *setSolverToNode* may result in a conflict. While this introduces a performance overhead, it turns out to

be very rare and therefore insignificant in practice.<sup>2</sup>

We still recompute the lookahead scores in a local restart, since the error caused by omitting this may grow very large, as shown by this example where not recomputing the lookahead after a conflict would mis-calculate a literal’s score with a maximum possible error.

*Example 3:* Consider the following derivation, where a lookahead at  $\langle \sigma \mid G \rangle$  on  $x^d$  fails with the learned clause  $(c \vee \neg x)^L$ :

$$\begin{aligned} \langle \sigma x^d \mid G \rangle &\xrightarrow{PExp} \langle \sigma x^d \mid G \wedge c^E \rangle \xrightarrow{BJ} \langle \sigma \neg x \mid G \wedge (c \vee \neg x)^L \rangle \\ &\xrightarrow{Prop} \langle \sigma \neg x \sigma' \mid G \wedge (c \vee \neg x)^L \rangle. \end{aligned}$$

Assume now that  $G$  has as a subformula  $(x \vee v \vee p_1) \wedge \dots \wedge (x \vee v \vee p_n) \wedge (x \vee \neg v \vee q_1) \wedge \dots \wedge (x \vee \neg v \vee q_n)$ , where  $p_i, q_i$  and  $v$  do not appear in  $\text{Ats}(\sigma')$ . Then the lookahead score of  $v$  at  $\langle \sigma \mid G \rangle$  is 0 but in the state  $\langle \sigma \neg x \sigma' \mid G \wedge (c \vee \neg x)^L \rangle$  the score is  $n$ . Note that  $n$  is upper bounded by  $|\text{Ats}\phi|$  which in our scoring is also the highest heuristic value.

4) *Correctness and termination:* We finish the discussion with proofs on correctness and termination for Alg. 1

*Theorem 1:* The algorithm either determines the satisfiability of the instance or constructs a balanced binary tree with each rooted path leading to the leaves corresponding to a unit-propagation consistent SMT instance.

*Proof.* The correctness of the Sat and Unsat results reported by the algorithm follow immediately from the observation that the result is obtained by modifying the solver state with the rules outlined in Sec. IV. Each rooted path of the tree corresponds to a unit propagation consistent instance. This follows from two observations. First, if *setSolverToNode* succeeds on a node  $n$ , the instance corresponding to the node is unit propagation consistent. Second, if *expandTree* succeeds, similarly by construction the instances corresponding to the nodes  $c$  and  $c'$  are consistent. The resulting tree is balanced, since unless the execution terminates in lines 9, 15, or 16, the algorithm performs a DFS with a cutoff at depth  $d$ .  $\square$

*Theorem 2:* The algorithm terminates.

*Proof.* The procedure *setSolverToNode* terminates since it performs a sequence that is bounded by the depth of the node and consists of rules *Dec* and unit propagation closure computations that both terminate. The procedure *expandTree* terminates in quadratic number of applications of *Dec*, *Undo* and unit propagation closure computations: the computation consists of lookahead steps each bounded by the number of atoms  $|\text{Ats}(\phi)|$ . The local restart at a node  $n$  can be done at most  $|\text{Ats}(\phi)|$  times, since each related backjump will assign at least one atom in the truth assignment of the solver state at node  $n$ .

The restarts in tree construction on lines 11 and 18 will not cause non-termination since the solver state is persistent (modulo possible applications of *Reset*) over such restarts. Following [18], the assignments of the solver together with the literals can be seen as a finite ordered sequence that is

<sup>2</sup>We observed three conflicts while partitioning over 9000 instances in different ways.

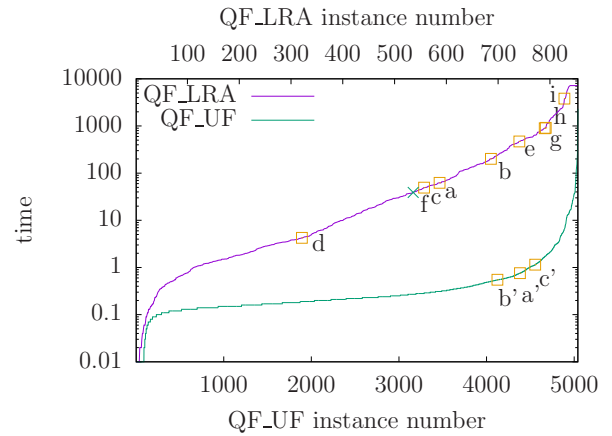


Fig. 1. Runtime for lookahead partitioning to 16 for QF\_LRA and QF\_UF. Labeled boxes and crosses refer to specific instances discussed below. Unsatisfiable instances are denoted with boxes ( $\square$ ), and satisfiable with crosses ( $\times$ ).

increased by every backjump and has a maximum element where every atom is assigned with no decision literals.  $\square$

## VI. EXPERIMENTS

We report experiments on our implementation on the non-incremental benchmark divisions QF\_UF and QF\_LRA of SMT-LIB.<sup>3</sup> The two divisions are chosen since they constitute the foundation of most other SMT logics and allow us to directly observe the behaviour of the congruence closure (egraph) and the Simplex algorithms under lookahead. All the experiments were run using the SMT solver OpenSMT [13]. The partitions are constructed with the implementation of Alg. 1, and, when applicable, solved with OpenSMT’s default CDCL( $T$ ) engine running the VSIDS heuristic [19], a setup similar to most CDCL( $T$ ) solvers. The CPU time consumed by the experiments is slightly under 338 CPU days. We used a Linux cluster, equipped with two Intel Xeon E5-2650 v3 @ 2.30GHz CPUs, yielding  $(2 \times 10)$  cores per node. Each node has 64GB of DDR4@2133MHz memory. We ran at most ten solvers on each node simultaneously, limiting the memory available for a solver to 4GB. The time out was 7200 s for both the partitioning and solving, except in Fig. 2 where the timeout was 1200 s. We first report on the efficiency of the partitioning implementation, and then show that the partitioning in general works well. Finally we study instances showing a slowdown anomaly. All times are given in seconds and refer to wall-clock times.

1) *Lookahead partitioning efficiency:* The plots in Fig. 1 illustrate the run times of Alg. 1 on the QF\_LRA and QF\_UF instances when partitioning into 16. The instances are ordered based on the run time. We only report the instances not solved during partitioning. The implementation is efficient in particular for QF\_UF, where the maximum stays in the majority of cases within a few seconds. The lookahead on QF\_LRA is

<sup>3</sup>The benchmarks are available at <https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks> under commit hash 33961bc4.



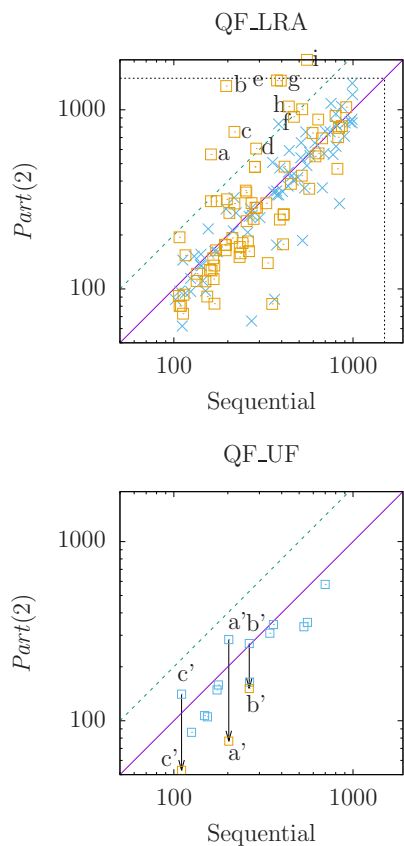


Fig. 2. Comparing sequential and  $Part(2)$  run times for QF\_LRA (top) and QF\_UF (bottom). On the top figure the boxes pointed to by the arrows are from  $Part(64)$  and show the approach efficient. The efficiency for QF\_LRA is studied separately.

much more involved, perhaps due to the more expensive theory solving. Our implementation partitions 98% of the benchmarks within two hours, showing that the approach is realistic.

2) *Effect of partitioning on instance difficulty*: To measure how partitioning affects the instance difficulty, we study instances that `OpenSMT` can solve between 100 and 1000 seconds sequentially, a range where parallelization is useful but the baseline can still be computed within a reasonable time. This resulted in 13 instances for QF\_UF and 144 instances for QF\_LRA. The reported times do not include partitioning.

Figure 2 compares  $Part(2)$  to sequential solving for QF\_LRA (top) and QF\_UF (bottom). We plot the line  $y = x$  corresponding to no speed-up, and the dashed line  $y = 2x$  corresponding to two-fold slowdown. The dashed horizontal and vertical lines in the top figure show the timeout of 1200 seconds. Crosses ( $\times$ ) and boxes ( $\square$ ) indicate satisfiable and unsatisfiable instances, respectively.

Except for three cases,  $Part(2)$  provides a consistent speed-up in QF\_UF. We ran these instances in  $Part(64)$  and each became easier to solve than the original instance (as shown by the downwards arrows that point to the corresponding  $Part(64)$  measurement). As a conclusion, it seems that looka-

head is efficient when combined with the congruence closure algorithm. This is somewhat expected since lookahead is efficient in purely propositional solving, and the congruence closure algorithm is scalable.

It is interesting to compare these results to QF\_LRA, where lookahead is efficient in 60% of the instances, but we also observe significant slowdowns, corresponding to up to 6-fold increase in run time. Repeating the experiment of partitioning with  $Part(64)$  did not result in a positive result similar to QF\_UF (see figures 3 – 4), suggesting that this phenomenon has a different origin.

The partitioning run times for the anomalies are shown with the labels in Fig. 1. Typically their run times are above the average.

3) *Slowdown analysis for partitioning*: Despite  $Part$  resulting in most cases in a consistent speed-up, the significant slowdowns in QF\_LRA warrant a separate study, as it poses a threat for lookahead partitioning in SMT. We label with (a) – (i) in Fig. 2 (top) nine instances where the run time more than doubles. We removed the randomness common in heuristic search by solving each partition several times with the `OpenSMT` VSIDS engine while changing the branching heuristic’s random seed. We refer to this approach as the *simulated parallel solver*, described in more detail in Appendix I.

We ran as a pre-processing phase  $Part(k)$  for  $k = 2, 4, 8, \dots, 2048$  for the instances (a) – (i) and stored the resulting partitions if the instance was not solved by  $Part$ . As a result of time outs and one of the instances being solved during partitioning, we could run the full experiment set only for the instances (a), (d), and (f). We concentrate on these three instances since they seem representative for the others as well, and report the rest in Appendix J.

Figure 3 (top) shows run times for the simulated parallel solver on the only satisfiable instance (f). While the slowdown is consistent for  $Part(2)$ , we observe speedup for  $Part(k), k \geq 4$ . Figure 3 (bottom) shows the simulated parallel median run times on instance (d). The partitions are easy only once a big number, 1024, is reached. We show in addition run time ranges (green bars) and medians (blue stars) for the individual partitions. The instance (i) behaves similarly to this. Figure 4 shows the results for the instance (a), where the minimum, median, and maximum run times consistently increase. We show also the individual  $Part$  runs as yellow boxes. Instances (b), (c), (e), (g), and (h) behave similarly to (a). While the lookahead clearly identifies easier partitions, the hardest partitions seem to get more difficult. In particular Figs. 3 (bottom) and 4 show a significant amount of partitions having the median time higher than the sequential median. The slowdown can be argued to result in part directly from these partitions.

The slowdown, affecting not uniformly all instances, seems to be the result of an intricate interaction between lookahead and the incremental Simplex implementation typically used in SMT solvers [4]. The implementation maintains an internal model for its real valued variables that satisfies all currently asserted inequalities. If a new inequality is not satisfied in the

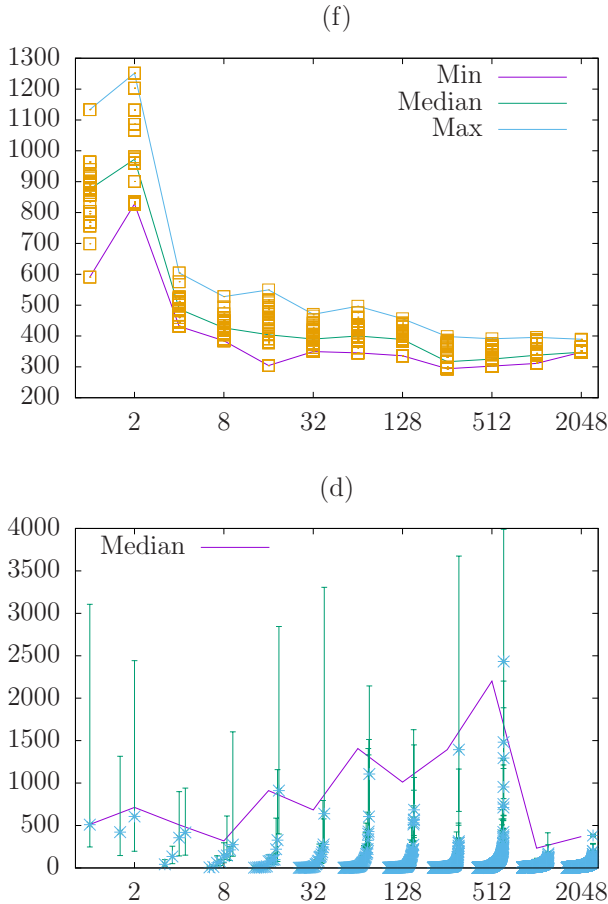


Fig. 3. Scalability for a satisfiable instance (*top*) and partition difficulty for an unsatisfiable instance (*bottom*). The horizontal axis refers to number of partitions produced, and the vertical axis to run time in seconds.

model, this triggers the pivoting sequence of Simplex that is in the worst-case exponential. SMT solvers try to avoid this behavior by branching as much as possible on inequalities that are consistent with the model. Because of lookahead, Simplex is sometimes forced to follow such a sequence, causing the increasing run times for some of the partitions. It is a natural further question how to generalize lookahead to mitigate or avoid these cases.

To conclude, we note that the lookahead partitioning produces in the vast majority of cases very balanced partitions and good speed-up. Nevertheless, the instance run times increase in a significant portion of the benchmarks. In the studied SMT-LIB benchmark divisions, we observed slowdown only for QF\_LRA. We believe that it is possible to obtain speed-up also for these instances by developing a version of the lookahead heuristic that considers also the configuration of the theory solvers run inside the SMT solver.

## VII. CONCLUSIONS

We present an algorithm for partitioning SMT with lookahead based on CDCL( $T$ ) calculus and show experimentally

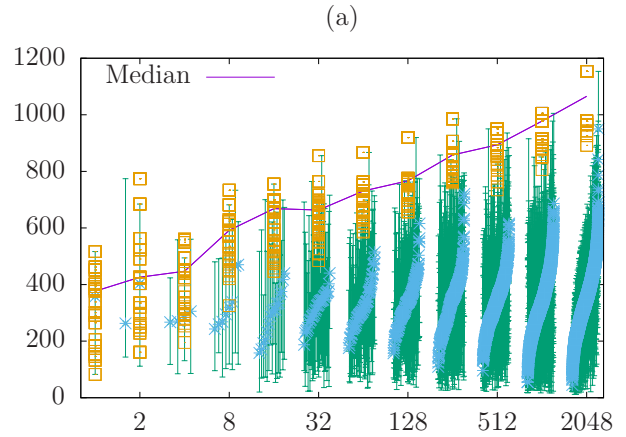


Fig. 4. Scalability and partition difficulty for an unsatisfiable instance. The horizontal axis refers to number of partitions produced, and the vertical axis to run time in seconds.

that the approach is highly promising. We also demonstrate that the classical propositional lookahead is not in general sufficient in SMT, where the theory reasoning engines may unexpectedly interfere with lookahead heuristic's view of the search space. In particular we found that in combination with Simplex as implemented in many SMT solvers, lookahead partitioning sometimes creates instances that are increasingly difficult to solve.

In future we plan to extend the lookahead heuristic to better consider the theories. In parallel, we will also study lookahead partitioning in a more applied setting, including theory combinations and non-convex theories, when new atoms are introduced.

*Acknowledgements.* This research was supported by the Swiss National Science Foundation grant number 200021\_185031.

## REFERENCES

- [1] Balyo, T., Sinz, C.: Parallel satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer (2018)
- [2] Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
- [3] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *Journal of the ACM* **52**(3), 365–473 (2005)
- [4] Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Proc. CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer (2006)
- [5] Heule, M., Dufour, M., van Zwieten, J., van Maaren, H.: March\_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: Hoos, H.H., Mitchell, D.G. (eds.) Proc. SAT2004. LNCS, vol. 3542, pp. 345–359. Springer (2005)
- [6] Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Proc. HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer (2012)
- [7] Heule, M., van Maaren, H.: March\_dl: Adding adaptive heuristics and a new branching strategy. *JSAT* **2**(1-4), 47–59 (2006)
- [8] Heule, M., van Maaren, H.: Look-ahead based SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 155–184. IOS Press (2009)



- [9] Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the Boolean Pythagorean triples problem via Cube-and-Conquer. In: Proc. SAT 2016. LNCS, vol. 9710, pp. 228–245. Springer (2016)
- [10] Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning SAT instances for distributed solving. In: Proc. LPAR 2010. LNCS, vol. 6397, pp. 372–386. Springer (2010)
- [11] Hyvärinen, A.E.J., Maescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Proc. SAT 2016. pp. 547 – 553. No. 9710 in LNCS, Springer (2016)
- [12] Hyvärinen, A.E.J., Maescotti, M., Sadigova, P., Chockler, H., Sharygina, N.: Lookahead-based SMT solving. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) Proc. LPAR-22. EPiC Series in Computing, vol. 57, pp. 418–434. EasyChair (2018)
- [13] Hyvärinen, A.E.J., Maescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Proc. SAT 2015. LNCS, vol. 9340, pp. 369–386. Springer (2015)
- [14] Hyvärinen, A.E.J., Wintersteiger, C.M.: Parallel satisfiability modulo theories. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 141 – 178. Springer (2018)
- [15] Iser, M., Kutzner, F., Sinz, C.: Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In: Proc. ICTAI 2017. pp. 1029–1036. IEEE Press (2017)
- [16] Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1**, 167–187 (1990)
- [17] Maescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Proc. ATVA 2016. pp. 428–443 (2016)
- [18] Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999)
- [19] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001. pp. 530–535. ACM (2001)
- [20] de Moura, L.M., Björner, N.: Z3: an efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337 – 340. Springer (2008)
- [21] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937 – 977 (2006)
- [22] Sebastiani, R.: Lazy satisfiability modulo theories. JSAT **3**(3-4), 141–224 (2007)
- [23] Simons, P.: Extending and Implementing the Stable Model Semantics. Ph.D. thesis, Helsinki University of Technology (2000)
- [24] van der Tak, P., Heule, M., Biere, A.: Concurrent cube-and-conquer - (poster presentation). In: Proc. SAT 2012. LNCS, vol. 7317, pp. 475–476. Springer (2012)
- [25] Tinelli, C.: A DPLL-based calculus for ground satisfiability modulo theories. In: Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings. pp. 308–319 (2002)
- [26] Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: A concurrent portfolio approach to SMT solving. In: Proc. CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer (2009)
- [27] Zabih, R., McAllester, D.: A rearrangement search strategy for determining propositional satisfiability. In: Proc. AAAI-88. pp. 155–160. ACM (1988)

## H. OPTIMIZATIONS

Our system for lookahead partitioning implements two high-level optimizations at the moment.

First, the lookahead computation traverses repeatedly the set of all atoms, occasionally interrupted by the restarts. The underlying implementation details give a natural order for the atoms: we currently store the atoms in a vector. It is useful not to start the lookahead computation from the beginning of the vector after a restart, but instead from the atom immediately following the most recently checked atom, as the former often results in repeating a computation that has just been performed without learning new information.

Second, given a solver state  $s$ , the literals  $b$  in a unit propagation closure computed as a side effect of computing the lookahead score for a literal  $a$  cannot propagate more than  $a$ . This follows from noting that  $a \wedge \sigma \wedge F \rightarrow b$  if  $b \in UP(\sigma a, F)$ , but it is not in general true that if  $b \in UP(\sigma a, F)$  then  $b \wedge \sigma \wedge F \rightarrow a$ . To implement this optimization we store for each atom  $c$  the scores  $score(c)$  and  $score(\neg c)$  as a pair  $(p_c, n_c)$ . Let the solver state be  $s = \langle \sigma \mid F \rangle$ , the highest known lookahead score in this state be  $(p_c, n_c)$ ,  $a$  and  $a'$  be literals such that  $b \in UP(\sigma a, F)$  and  $\neg b \in UP(\sigma a', F)$ ,  $score(a) = p_a$ , and  $score(a') = p_{a'}$ . We may apply the optimization in two cases: (1) if  $p_a \leq \min(p_c, n_c)$ , then there is no need to compute  $score(b)$ ; and (2) if  $p_{a'} \leq \min(p_c, n_c)$ , then there is no need to compute  $score(\neg b)$ . We use this observation by maintaining upper bounds for the literals appearing in unit propagation closures during the lookahead computation, and skip computing the scores for the literals whenever possible using (1) and (2).

## I. THE SIMULATED PARALLEL SOLVER

Run times of an SMT solver are unpredictable, and reliable measurement requires a large number of repeated experiments. Since the amount of experiments needed to reliably measure runs of a parallel solver gets quickly unreasonable, we instead adopt the following process, which we refer to as the *simulated parallel solver*.

The exact method of computing the run times of the simulated parallel solver uses the following procedure:

- 1) Construct  $n$  partitions.
- 2) Run each  $n$  partition 20 times, and store the results as

$$\begin{bmatrix} t_1^1 & \dots & t_{20}^1 \\ t_1^2 & \dots & t_{20}^2 \\ \vdots & \vdots & \vdots \\ t_1^n & \dots & t_{20}^n \end{bmatrix}$$

- 3) Repeat 100 times:
  - a) Pick a run time  $T^i$  randomly from  $t_1^i, \dots, t_{20}^i$  for each  $1 \leq i \leq n$
  - b) If the instance is unsatisfiable, report  $\max(T^1, \dots, T^n)$ . If the instance is satisfiable, report  $\min(T^{k_1}, \dots, T^{k_m})$  where  $\{T^{k_1}, \dots, T^{k_m}\} \subseteq \{T^1, \dots, T^n\}$  is the set of run times corresponding to satisfiable instances.

In the experimental results of this paper where we use the simulated parallel solver we exclude the time required to run Alg. 1, since in these cases we are not interested in the optimization details of the partitioner, but instead on the speed-up obtainable by a lookahead-based partitioning algorithm, given a “good” implementation of the partitioner.

## J. FURTHER EXPERIMENTS

We provide in this appendix a more complete picture than what is possible with the space limitations of the manuscript for the QF\_LRA instances (a) – (i) where we observe anomalies in the run times. The full names of the instances in SMT-LIB are show in Table I.

The results are shown in Figs. 5 – 13. Throughout, the figures show the number  $n$  of  $Part(n)$  on the horizontal axis and the time on the vertical axis.

The figures in the *top* show the median run time of the simulated parallel solver, and an illustration for the difficulty for each partition, grouped around  $n$ , as a green bar. The median run time of the partition, computed over several runs (the exact amount of the runs depends on  $n$  for practical reasons), is shown as a blue star plotted on the bar. For example, in Fig. 5 (*top*), we see two green bars grouped around  $n = 2$ . Each corresponds to a partition. Both partitions have the minimum run time around 100 seconds. The first partition (as ordered by the median) has a maximum run time of approximately 1300 seconds, and the second partition of approximately 2500 seconds. Both partitions seem to be easier than the original instance, shown as the leftmost bar in the figure, in the sense that the minimum and maximum of the partitions are lower. However, the median run time of the second partition is slightly higher than the original median.

As can be seen from the figure, the median run times of the partitions, the blue stars, are not the same as the median run time of the simulated parallel solver, shown as a solid line (except for the original instance). This is because in general the run times of the partitions overlap, as is expected for a good partitioning approach.

The figures on the *bottom* show run times of the simulated parallel solver over repeated experiments. We show with the blue line the maximum observed run times, with the green the median run time (also shown in the figures on the *top*), and with the purple line the minimum run time. In addition we show each individual experiment as a yellow box. Taking as example again Fig. 5, at  $Part(2)$  the yellow boxes correspond to the 100 samples reported by the above procedure. We see that there is a cluster of results around 750 s run time, where the median run time is placed according to the green line.

## K. AN ALTERNATIVE LOOKAHEAD HEURISTIC

To study the effect of the lookahead approach for a scoring system different from that of Eq. (1), and to verify that the results are not an artifact of the particular scoring, we implemented inside OpenSMT the Jeroslow-Wang (JW) based measure of search space size. Our implementation is motivated

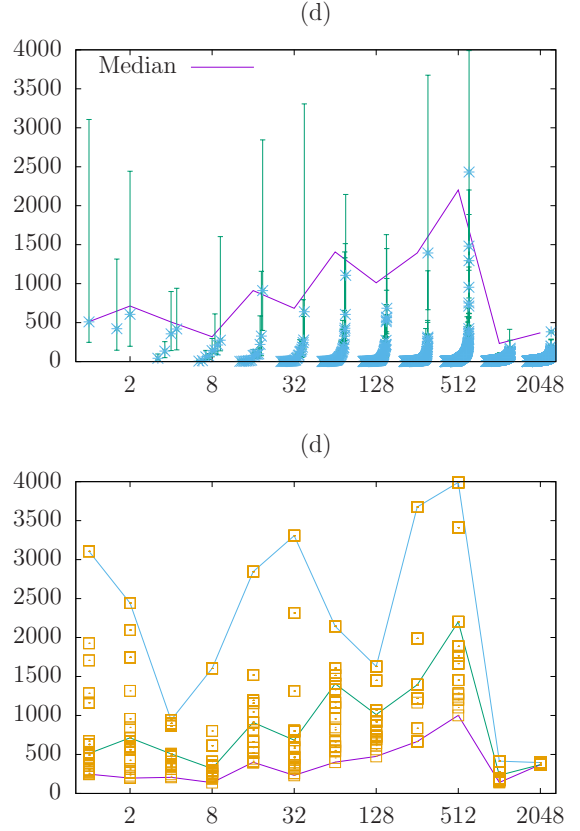


Fig. 5. (d) partition difficulty (*top*) and scalability (*bottom*)

by that of Z3 [20], [8].<sup>4</sup> Given a solver state  $s = \langle \sigma \mid F \rangle$ , we first compute the unit propagation closure  $u = UP(\sigma, F)$ , and then assign a score to each clause  $C \in F$  under  $u$  as

$$score_{JW}^s(C) := \begin{cases} 0 & \text{if } u \cap C \neq \emptyset \\ 2^{-\nu} & \text{otherwise} \end{cases},$$

where  $\nu$  is the number of free literals in  $C$  (i.e., literals  $l \in C$  such that both  $l \notin u$  and  $\neg l \notin u$ ). The score for a literal  $l$  in state  $\langle \sigma \mid F \rangle$  is then the sum of the reductions in the clause scores when  $l$  is assigned:

$$score_{JW}^{\langle \sigma \mid F \rangle}(l) := \sum_{C \in F} \left( score_{JW}^{\langle \sigma \mid F \rangle}(C) - score_{JW}^{\langle \sigma l \mid F \rangle}(C) \right)$$

Finally, the score of a variable  $v$  is defined as

$$score_{JW}^s(v) := p_0 + p_1 + 1024 * p_0 * p_1 \quad (2)$$

where  $p_0 = score_{JW}^s(v)$  and  $p_1 = score_{JW}^s(\neg v)$ .

In Fig. 14 we show the results of an experiment similar to that in Fig. 2 on solving our benchmark set with a virtual partitioning approach, this time partitioning with the lookahead algorithm using the score in Eq. (2). We see that the results are comparable, with one more 2-fold slowdown in QF\_LRA

<sup>4</sup>Note that this is not the same implementation, since Z3 does not implement Alg. 1.

TABLE I  
 INSTANCES OF QF\_LRA THAT EXPERIENCE AT LEAST TWO-FOLD SLOWDOWN IN PLAIN PARTITIONING IN TWO.

Label	Path in SMT-LIB
(a)	LassoRanker/SV-COMP/ NoriSharma-2013FSE-Fig8_true-termination.c_iteration1_Loop_4-pieceTemplate
(b)	LassoRanker/SV-COMP/ NoriSharma-2013FSE-Fig8_true-termination.c_iteration1_Lasso_4-pieceTemplate
(c)	LassoRanker/SV-COMP/ Masse_true-termination.c_iteration1_Loop_4-pieceTemplate
(d)	LassoRanker/Ultimate/ Braverman-2006CAV-Ex1-int.bpl_iteration1_Lasso_6-phaseTemplate
(e)	LassoRanker/Ultimate/ Gulwani.bpl_iteration1_Lasso_4-pieceTemplate
(f)	LassoRanker/CooperatingT2/ sas2.t2.c_iteration1_Lasso_5-phaseTemplate
(g)	LassoRanker/Ultimate/ yPositive-SIscaled75.bpl_iteration1_Loop_4-pieceTemplate
(h)	LassoRanker/SV-COMP/ KroeningSharyginaTsitovichWintersteiger-2010CAV-Fig1_true-termination.c_iteration1_Lasso_3-pieceTemplate
(i)	LassoRanker/SV-COMP/ GulwaniJainKoskinen-2009PLDI-Fig1_true-termination.c_iteration1_Lasso_3-pieceTemplate

in JW, while maximum slowdown being less extreme. For QF\_UF there are no slowdowns, suggesting that the Jeroslow-Wang heuristic works well in the propositional space, possibly since it sees “further” in the problem instance. Clearly the slowdowns in QF\_LRA cannot be addressed by the JW score, since the state information on the Simplex model is not available for this score either.

Finally, in Fig. 15 we show as a cactus plot the run times of our implementation of Alg. 1 in QF\_UF and QF\_LRA using both scores (1) and (2). The computational cost of the JW score is, not surprisingly, much higher, partly because we cannot apply all our optimizations in this case, and partly because the score needs to be evaluated on every clause. To make this implementation more practical we should apply pre-selection heuristics to limit the number of variables considered for lookahead. However, we chose not to do this optimization since it would have made the comparison between the two scores less exact.

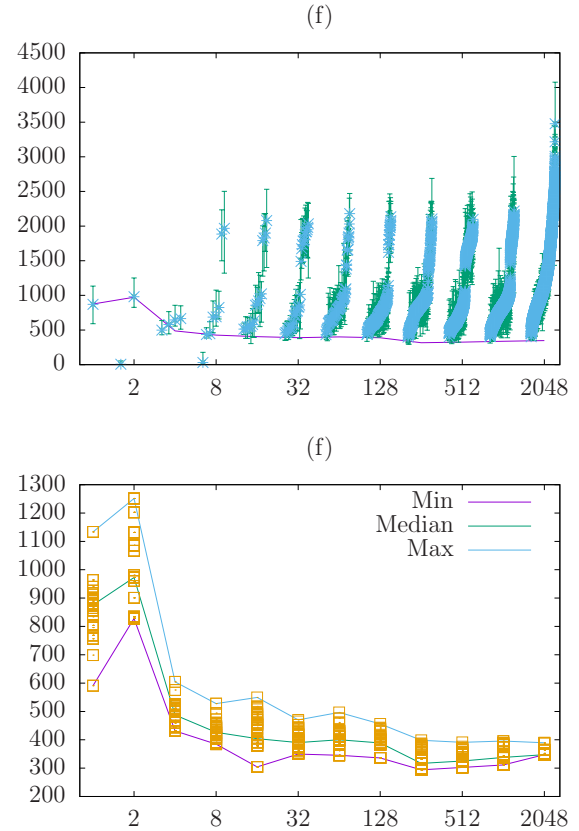


Fig. 6. (f) partition difficulty (*top*) and scalability (*bottom*)

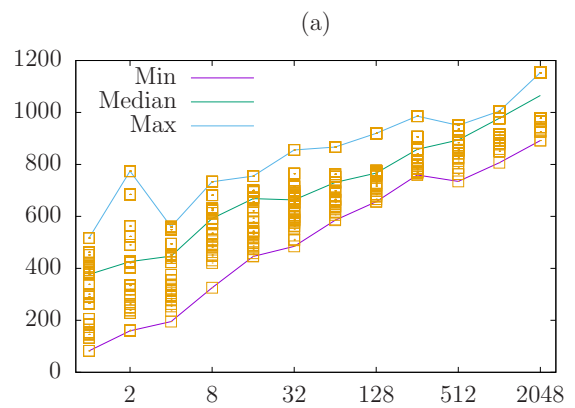
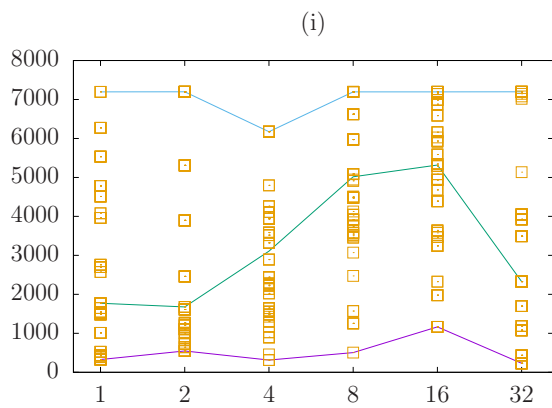
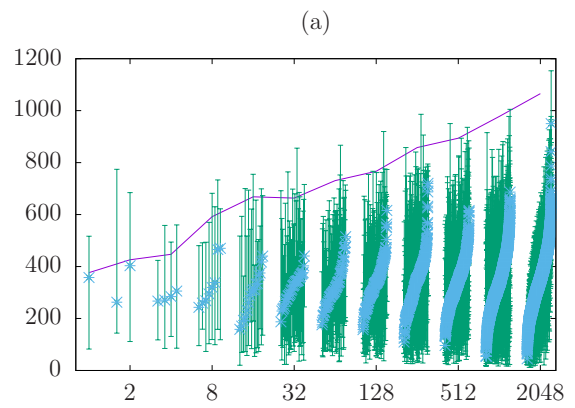
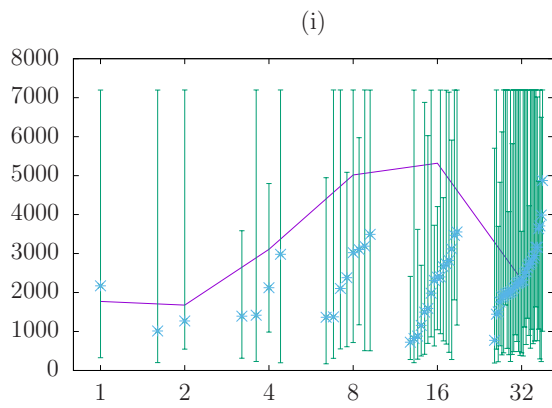


Fig. 7. (i) partition difficulty (*top*) and scalability (*bottom*)

Fig. 8. (a) partition difficulty (*top*) and scalability (*bottom*)

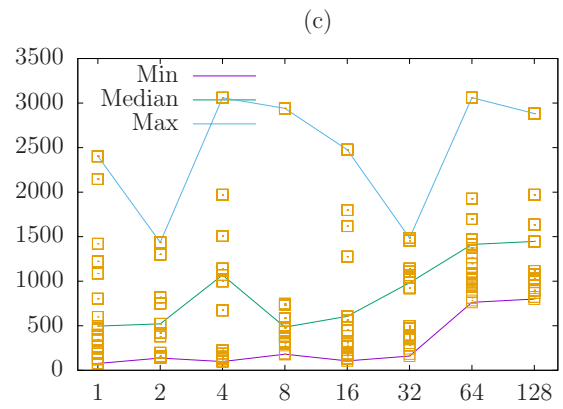
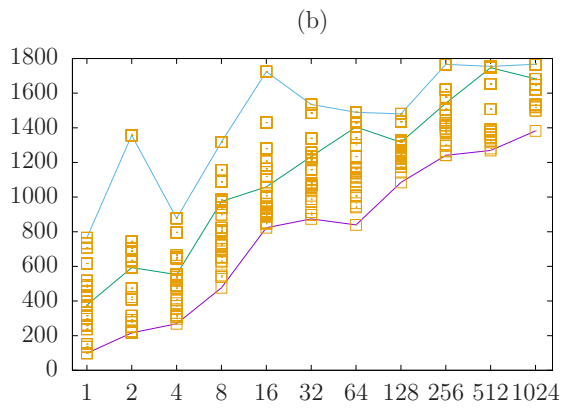
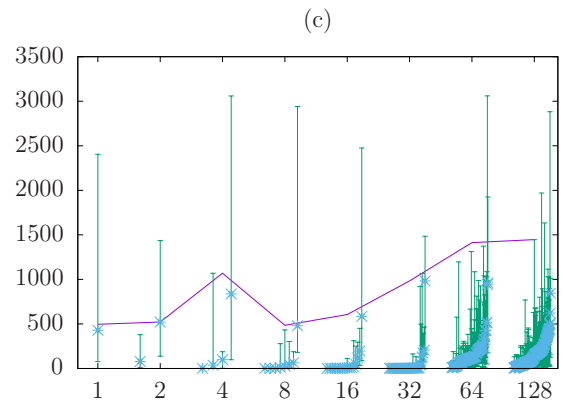
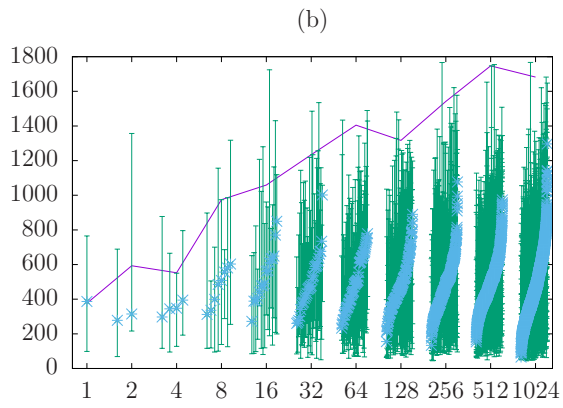


Fig. 9. (b) partition difficulty (*top*) and scalability (*bottom*)

Fig. 10. (c) partition difficulty (*top*) and scalability (*bottom*)

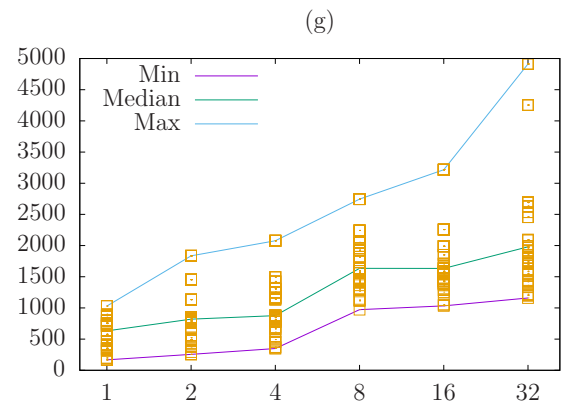
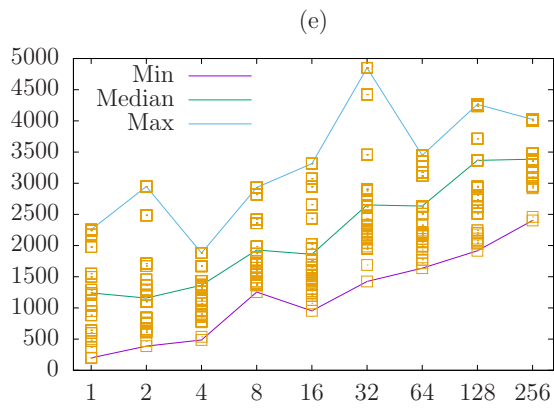
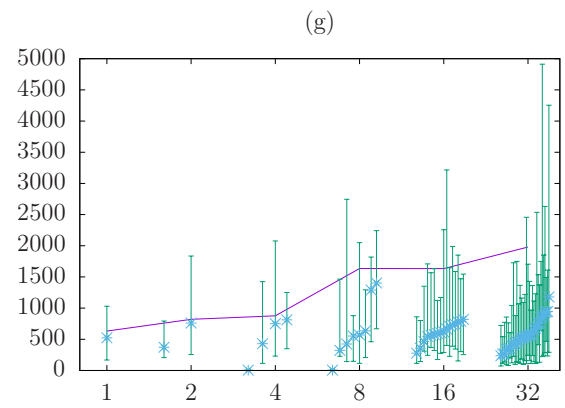
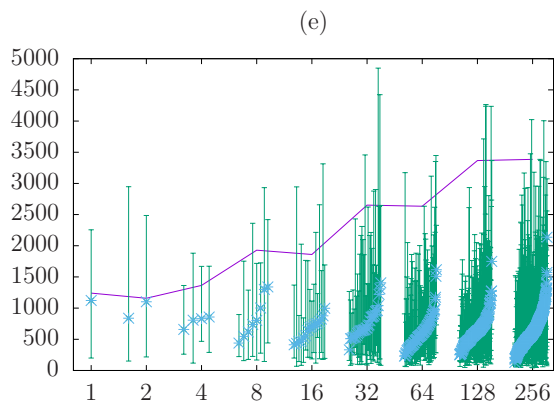


Fig. 11. (e) partition difficulty (*top*) and scalability (*bottom*)

Fig. 12. (g) partition difficulty (*top*) and scalability (*bottom*)



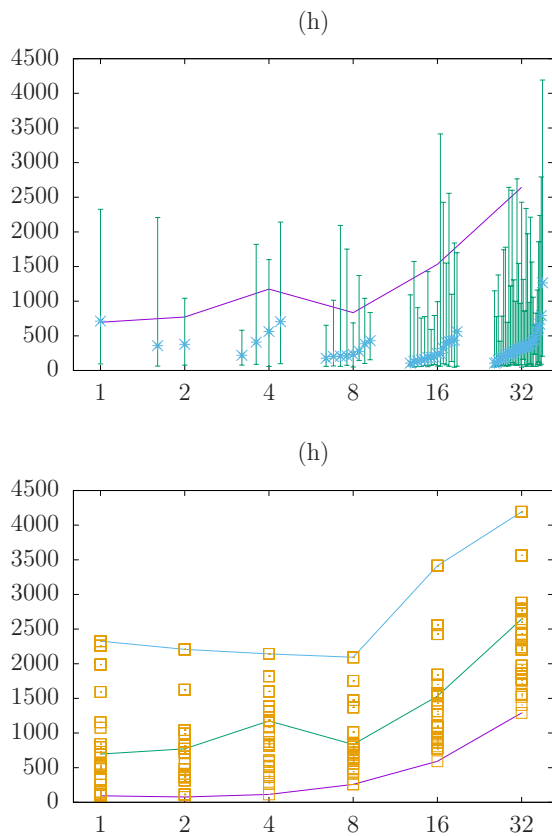


Fig. 13. (h) partition difficulty (*top*) and scalability (*bottom*)

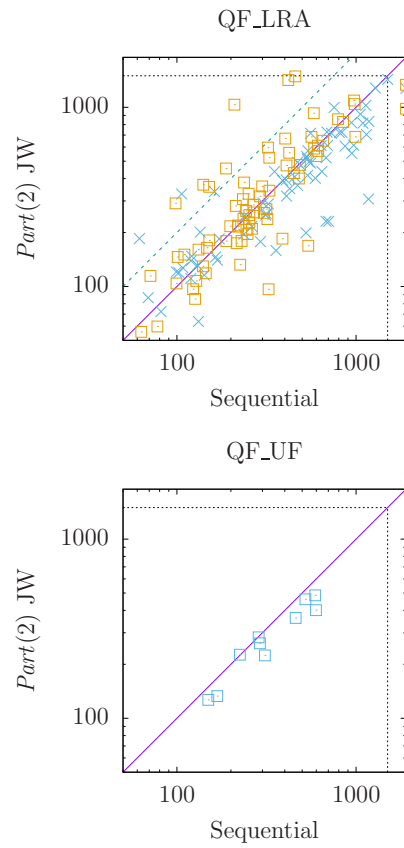


Fig. 14. Partitioning QF\_LRA (*top*) and QF\_UF (*bottom*) in two with the Jeroslow-Wang lookahead heuristic

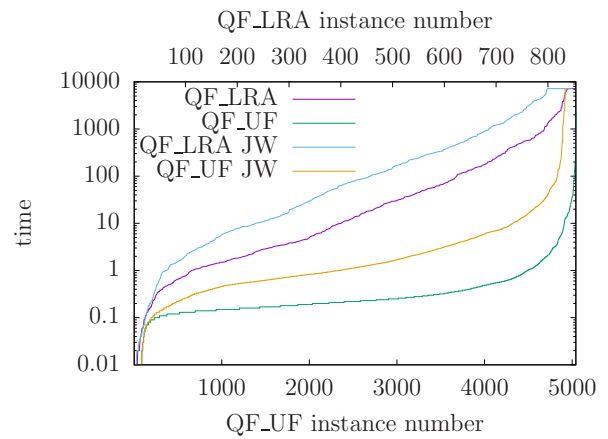


Fig. 15. Run time comparison of the partitioning heuristics