Lookahead-Based SMT Solving

Antti E. J. Hyvärinen[1], Matteo Marescotti[1], Parvin Sadigova[2], Hana Chockler[2], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Switzerland {first.last@usi.ch}
[2] King's College London, UK {first.last@kcl.ac.uk}

Abstract

The lookahead approach for binary-tree-based search in constraint solving favors branching that provide the lowest upper bound for the remaining search space. The approach has recently been applied in instance partitioning in divide-and-conquer-based parallelization, but in general its connection to modern, clause-learning solvers is poorly understood. We show two ways of combining lookahead approach with a modern DPLL(T)-based SMT solver fully profiting from theory propagation, clause learning, and restarts. Our thoroughly tested prototype implementation is surprisingly efficient as an independent SMT solver on certain instances, in particular when applied to a non-convex theory, where the lookahead-based implementation solves 40% more unsatisfiable instances compared to the standard implementation.

1 Introduction

Many practical questions can be reduced to determining whether a set of Boolean constraints has a solution. These types of problems are generally known as constraint satisfaction problems, and the most widely known example is the propositional satisfiability problem (SAT) of determining whether a propositional formula, represented in conjunctive normal form, is satisfiable. SAT forms the basis of another important class of problems called *satisfiability modulo theories* (SMT), where instead of Boolean atoms the formulas are built on ground atoms in first-order logic, often in practice expressible as equalities over arithmetics or uninterpreted functions. Algorithms for both SAT and SMT have been extensively studied. A well-established architecture for SMT is to run a SAT solver that regularly queries the consistency of the first-order atoms from the arithmetics and uninterpreted functions solvers. The information carried by the theories is communicated to the SAT solver on-the-fly during the search based on counterexamples that violate the axioms of the theories.

Lookahead is an approach used in algorithms for solving constraint satisfaction problems based on branching [21, 20, 7]. The approach selects a branch that results in the lowest upper bound in some measure for the remaining search space. The basic process involves trying every possible atom for a given branch and deducing an upper bound for the resulting search space through a process called Boolean constraint propagation. The number of unassigned atoms after propagation gives an upper bound for the remaining search space under the respective atom. A solver using lookahead picks the atom with the approximated smallest remaining search space. Lookahead is often considered expensive because of the big number of propagations needed for computing the upper bounds. Many modern solvers for propositional satisfiability and SMT therefore revert to more efficiently implementable heuristics that are based on different metrics that can be computed during the search. Most notably these include the *variable-state independent decaying sum* (VSIDS) heuristic [16] and its variations, that are tightly integrated to the respective decision procedures.

In this work we study the question of using lookahead in modern SMT solvers and show experimentally that the lookahead heuristic can solve almost 40% more unsatisfiable instances in some types of SMT problem collections, therefore being superior to the standard heuristic. We first define two approaches of integrating lookahead into an SMT solver. Both use elements of the DPLL(T) [19] algorithm for solving SMT, and are defined for the quantifier-free fragments of SMT theories. We further study two types of theories that are known as *convex* and *non-convex* [18]. From the perspective of a DPLL(T) solver, solving for convex theories requires adding information to the SAT formula that only mention the first-order atoms that already exist in the formula. For non-convex theories instead it is in general necessary to introduce new first-order atoms. We show that in particular non-convex theories raise further interesting theoretical and practical questions for lookahead.

The lookahead heuristic is defined in [21, 8] for SAT solvers using the DPLL algorithm [4, 3]. However, modern SMT solvers are based on the conflict-drive clause-learning (CDCL) algorithm [15] that behaves very differently from DPLL. While the DPLL algorithm is based on a binary search tree, the CDCL algorithm maintains a stack of assumptions and propagations. During its search the CDCL algorithm derives with resolution new clauses that are then used for backtracking and expanding the stack. Due to this difference it is not obvious how to generalize the lookahead algorithm to a CDCL-based solver, and even less so for an SMT solver. In the following we will use the somewhat non-standard term CDCL(T) instead of the more established DPLL(T) to emphasize the difference between the DPLL-based lookahead procedure and the DPLL(T)-based SMT-solving procedure which in most modern SMT solvers apply CDCL-based techniques for driving the search. We identify four ways in which CDCL(T) differs from DPLL with respect to lookahead, and provide solutions for addressing the challenges arising from each:

- CDCL has no search tree, while lookahead assumes that each branch is associated with an atom and its negation;
- CDCL and in particular CDCL(T) add clauses during the search, which has implications on propagations and as a result on what is the remaining search space of a branch;
- CDCL(T) can *theory propagate* atoms, which reduce the remaining search space of a branch but are invisible for the Boolean structure; and
- The search space of non-convex theories can increase on a branch, since theory solvers may need to add atoms to the instance.

The first algorithm we present, Plain, assumes the structure of the CDCL(T) search but instead of applying the VSIDS heuristic, chooses the atoms based on lookahead. However, the algorithm Plain is not a straightforward extension of the CDCL(T) algorithm, since computing the lookahead heuristic can result in resolution steps and therefore the CDCL(T) branching needs to be modified accordingly. The second algorithm, Tree, maintains instead an explicit DPLL tree, and extends and prunes the tree by simulating the branches of the DPLL tree on a CDCL(T) solver's stack. This results in a dramatically different algorithm from the standard CDCL(T).

Based on a comparative study we identify the more promising approach for lookahead. The approach is implemented directly on the SMT solver OpenSMT2 to facilitate the evaluation of the experimental results. In particular the theory solvers are implemented in an efficient but simple form, thus avoiding complicated interaction between lookahead and certain optimizations such as solver layering [6] used in many other SMT solvers. We evaluate the implementation on the algorithm Tree on all the instances of the widely used SMTLIB benchmark set's quantifier-free categories of equality and uninterpreted functions (QF_UF), linear real arithmetics (QF_LRA), and linear integer arithmetics (QF_LIA). We show that the approach is particularly efficient on QF_LIA, suggesting that lookahead might be an interesting alternative to provide efficient search methodology for non-convex theories. In general the lookahead-based solver is slower than the corresponding VSIDS-based implementation. This however depends on the benchmark instance, and we could identify several instances that are much more efficiently solved on the lookahead implementation also in the category QF_LRA.

Related work. The lookahead heuristic is originally introduced in [21] in the context of SAT solving. It found one of the biggest early successes in applications to computing stable models in [20], and later again in sequential SAT solving in [7]. A CDCL solver for SAT that uses lookahead is presented in [2]. Unlike the two algorithms we present in this work where CDCL and lookahead are tightly integrated, lookahead in [2] is used as a pre- and in-processing technique.

An algorithm similar to Tree presented in this work was first used in parallel SAT solving approaches based on divide-and-conquer in [10], and later for SMT solving in [12, 14]. However, the lookahead-based partitioning algorithm was never published for lack of space. Unlike in the previous works, in this work the algorithm Tree is extended to non-convex theories and used as a stand-alone solver instead of as an aide for parallelization. Recently, a lookahead-based parallelization technique was also successfully applied in constructing a large-scale proof in [9].

The experiments in this paper done on QF_LIA show that the lookahead heuristic can be used for guiding the search in non-convex SMT theories to accomplish in some cases termination of a straightforward, incomplete implementation. An orthogonal approach involves altering the *cutting plane* algorithm, as proposed in, e.g., [1, 13].

Finally, a recent result in [17] shows that the non-chronological backtracking of a CDCL solver might be harmful in some SAT instances. We believe that some of the performance gains we observe in the lookahead implementation are due to a similar phenomenon in SMT where the solver backtracks too much during its search.

Structure of the paper. We first give the necessary background for lookahead, CDCL solvers, and SMT solvers in Sec. 2. We then describe our two approaches Plain and Tree to lookahead-based SMT solving in Sec. 3. In Sec. 4 we discuss some of the most central implementation details including observations on the use of Tree for parallel SAT and SMT solving. We report our experiments in Sec. 5, before concluding in Sec. 6.

2 Preliminaries

An *instance* of the *propositional satisfiability problem* (SAT) connects a finite set of literals of form either x or $\neg x$, where x is a *Boolean atom*, with *Boolean connectives* conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). We equate the literal $\neg\neg x$ with x. All SAT instances can be represented in an equisatisfiable formula in *conjunctive normal form* (CNF), that is, a conjunction of clauses, which are disjunctions of literals, with a polynomial increase in the number of atoms. We assume that a CNF instance does not have multiple copies of the same clause and that a clause does not have multiple copies of the same literal, and therefore represent clauses as sets of literals and the CNF instance as a set of clauses. In addition we assume that tautological clauses, containing both the literal x and $\neg x$, are removed. We represent the atoms appearing in a CNF formula ϕ by $At(\phi)$, and use the same notation $At(l)$ for the atom of a literal l. The truth values of the atoms are represented by a *partial truth assignment* $\sigma \subseteq \{x, \neg x \mid x \in Var(\phi)\}$ such that for no atom x both $x \in \sigma$ and $\neg x \in \sigma$. If $x \in \sigma$, then x

is *true*, if $\neg x \in \sigma$, then x is *false*, and if neither is in σ, then x is *unassigned.* Similarly, for a literal l, if $l \in \sigma$ we say that l is *true*. We define two constant literals, $\top$ and $\bot$, that are respectively *true* and *false* under any truth assignment. A set of literals σ such that for some x both $x \in s$ and $\neg x \in s$ is called an *inconsistent truth assignment.*

Unit propagation is a polynomial-time process that, given a formula ϕ and a partial truth assignment σ, determines how σ must be extended with new literals in order for it to satisfy ϕ. Given a truth assignment σ and a CNF formula ϕ, the *unit propagation operator* $up_\phi(\sigma)$ returns either (1) a new truth assignment $\sigma' \supseteq \sigma$ of the literals of ϕ such that for all clauses $c \in \phi$, if $c \cap \sigma = \{l\}$ for some literal l, then $l \in \sigma'$; or, if such a σ' in case (1) would be inconsistent, (2) the constant literal $\bot$. The *unit propagation closure* $UP(\phi, \sigma)$ returns the smallest, unique set of literals $U \supseteq \sigma$ that is closed under up_ϕ, if such a set is not inconsistent.

DPLL-based SAT solvers. A SAT solver is a tool that determines whether the atoms of a CNF instance ϕ can be assigned truth values such that all clauses $c \in \phi$ contain a true literal. DPLL-based SAT solvers build a labelled binary tree called the *DPLL search tree*, where each node corresponds to a partial truth assignment for the formula ϕ. The labelling L is a bijection between the nodes of the tree and $\{x, \neg x \mid x \in At(\phi)\} \cup \{\top\}$. Let n_0 be the root of the tree, and n_k an arbitrary node in the tree. We denote by $n_0, \ldots, n_k$ the unique path from the root to n_k. Semantically node n_k corresponds to the truth assignment $\sigma_{n_k} = \{L(n_0), \ldots, L(n_k)\}$. In addition to a label, the nodes of the tree are marked either *closed*, indicating that the algorithm has shown that the corresponding truth assignment is not a subset of any satisfying truth assignment for ϕ, or *open*, indicating that it is not known whether the truth assignment can be extended to a satisfying assignment. The solver determines the labels heuristically so that sibling nodes have the label x and $\neg x$ for some atom $x \in At(\phi)$ that the heuristic considers promising. The atom labelling the children of n_k is chosen from the *unassigned atoms* at node n_k, that is, from the set $At(\phi) \setminus \{At(l) \mid l \in UP(\phi, \sigma_{n_k})\}$. If the closure is consistent, the search will continue in this branch. If the closure is inconsistent, the problem corresponding to $\phi \wedge \sigma$ is unsatisfiable. Therefore the node is marked closed, and the solver backtracks to the nearest ancestor of n_k that has a non-closed child. If the both children of a node are marked closed, also the node is marked closed. The search terminates if (1) the root of the tree is marked closed indicating that the instance is unsatisfiable, or (2) when in some node n_k, after computing the unit propagation closure, all atoms in $At(\phi)$ are assigned, indicating that $UP(\phi, \sigma_{n_k})$ is a satisfying truth assignment for ϕ.

Lookahead. Given a literal l and an assignment σ, the *propagation score* of l assuming σ is $PS_{\phi\wedge\sigma}(l) = |UP(\phi, \{l\} \cup \sigma) \setminus UP(\phi, \sigma)|$, if $UP(\phi, \{l\} \cup \sigma) \neq \bot$, and undefined otherwise. We extend the propagation score to atoms x by defining $\textit{la-score}_{\phi\wedge\sigma}(x) = \min(PS_{\phi\wedge\sigma}(x), PS_{\phi\wedge\sigma}(\neg x))$.

The lookahead approach for DPLL-based SAT solvers labels nodes with the literals that have the highest propagation score. In the *lookahead phase*, given a node n_k and the corresponding truth assignment σ_k, the propagation scores are computed for all unassigned atoms at n_k as $\textit{la-score}_{\phi\wedge\sigma_k}(x)$. The highest scored atom x_{best} is chosen as the labels x_{best}, $\neg x_{\text{best}}$ of the two children. However, lookahead is not simply a heuristic but can also give insight into the satisfiability of the problem. If during the lookahead phase $PS_{\phi\wedge\sigma_k}(l)$ results in an inconsistent truth assignment, then any satisfying truth assignment for $\phi \wedge \sigma_k$ must include $\neg l$. If also $PS_{\phi\wedge\sigma_k}(\neg l)$ results in an inconsistent truth assignment, then σ_k cannot be extended to a satisfying truth assignment, and the node n_k can be marked closed. Similarly, the lookahead heuristic can be used, for instance, for computing equalities of atoms under truth assignments [8].

Clause-learning, conflict driven SAT solvers. Unlike a DPLL-based SAT solver, a clause-learning, conflict driven (CDCL) SAT solver does not maintain an explicit search tree but guarantees the completeness of a search by computing new clauses that satisfy certain criteria. The CDCL solver alternates between heuristically selecting a *decision literal* and computing unit propagation closure. During the search, a CDCL solver maintains a *decision stack* $s = (E_0, \ldots, E_n)$, where n is defined as the *decision level* of s, and $E_0 = UP(\phi, \emptyset)$. For $i > 0$, $E_i = \{d_i, l_i^1, \ldots, l_i^k\}$ where d_i is the decision literal and $l_i^1, \ldots, l_i^k$ are the new literals propagated by d_i, that is, $\{l_i^1, \ldots, l_i^k\} = UP(\phi, \bigcup_{j=0}^{i-1} E_j \cup \{d_i\}) \setminus \bigcup_{j=0}^{i-1} E_j)$. We use the infix notation $\cdot$ to denote pushing an element to a stack. More formally, we write $(E_0, \ldots, E_n) \cdot E_{n+1} = (E_0, \ldots, E_{n+1})$. Each literal in E_i is associated with the decision level i. If a unit propagation closure at some decision level n is inconsistent, the CDCL solver learns a conflict clause c where exactly one literal $l \in c$ is in the decision level n and all other literals appear negated in some E_i, where $i < n$. The clause c is learned essentially through simulating resolution steps on the clauses that implied literals during previous unit propagations. The instance ϕ is updated to consist of $\phi \cup \{c\}$, and the decision stack is backtracked until the second highest decision level i' appearing in the clause c. The CDCL solver then recomputes the stack element $E_{i'}$ as $UP(\phi \cup \{c\}, \bigcup_{j=0}^{i'} E_j) \setminus \bigcup_{j=0}^{i'-1} E_j$, repeating the process until either a consistent unit propagation closure is found at some stack element $E_{i'}$ or $UP(\phi, E_0)$ is inconsistent. In the former case the CDCL solver chooses a new literal $d_{i'+1}$ heuristically and extends the decision stack by the corresponding $E_{i'+1}$, while in the latter case the instance ϕ is shown unsatisfiable.

CDCL solvers are sound, as can be seen from the following reasoning. First, assume that ϕ is unsatisfiable. The algorithm learns new clauses through resolution on the previous clauses, and therefore the new clauses are logical consequences of the old clauses. Therefore an inconsistent unit propagation closure on ϕ implies that also the original instance is unsatisfiable. Second, assume that ϕ is satisfiable. If there is a complete truth assignment σ such that unit propagation closure $UP(\phi, \sigma)$ is consistent, then σ satisfies ϕ and all its learned clauses. CDCL solvers are guaranteed to terminate on satisfiable instances since they will eventually find a complete truth assignment if one exists. CDCL solvers are also guaranteed to terminate on unsatisfiable instances, since an execution of the solver forms an ordered sequence of decision stacks, the sequence having a maximal element: each new learned clause propagates at least one new literal on a lower decision level, and the maximum element of this sequence is the one with all literals assigned on the decision level 0 [15].

SMT solvers based on CDCL(T). Most SMT solvers are built as an extension of a CDCL solver where periodically the CDCL solver sends to the theory solver T a subset $\sigma_T = \{l_1, \ldots, l_k\} \subseteq UP(\phi, \bigcup_{j=0}^{n} E_j)$ of literals that have an interpretation in T. The theory solver then attempts to guide the search further depending on the satisfiability of the set σ_T and the state of the solver.

An SMT theory is called *convex* if for every finite set of literals σ_T and for every non-empty disjunction of atoms $(x_1 = y_1), \ldots, (x_n = y_n)$ over $At(\sigma_T)$ it holds that $\sigma_T \models x_i = y_i$ for some $1 \leq i \leq n$ if and only if $\sigma_T \models \bigvee_{1 \leq i \leq n} x_i = y_i$. In particular, assume that T is a convex theory, σ_T is an assignment over literals of T such that $\sigma_T \models \neg((x_1 = y_1) \wedge \ldots \wedge (x_n = y_n))$. Then the clause $\neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n)$ can be used for guiding further the search as a learned clause. Dually, if T is not convex, there is in general no such clause even if σ_T is inconsistent with the axioms of the theory T, and the search needs to be guided in a different manner.

A theory solver may reply to a query in five different ways that we name in the following list:

inco-e: If σ_T is inconsistent in T and the inconsistency can be expressed in terms of $At(\phi)$, the solver returns as an explanation a clause that is a subset of $\neg l_1, \ldots, \neg l_k$ corresponding to a tautology in T that σ_T does not satisfy.

inco-n: If σ_T is inconsistent in T but the inconsistency cannot be expressed in terms of $At(\phi)$, the theory solver introduces new atoms not in $At(\phi)$ to express the conflict, and returns a clause that consists of the new literals that the solver needs to satisfy in future;

inco-o: If σ_T is inconsistent in T, and the inconsistency can be expressed by a clause that the theory solver has already communicated to the CDCL solver, the theory solver asks the CDCL solver to continue the search, that is, to try to extend σ_T.

con-p: If σ_T is consistent in T, the theory solver might still be able to return a clause consisting of theory literals in $At(\phi)$ that are tautological in theory T and imply a literal in $UP(\phi, \sigma_T)$, through a process called *theory propagation*.

con-n: If σ_T is consistent in T but the theory solver cannot find theory implications, the theory solver simply states that σ_T is consistent with the theory and asks the CDCL solver to continue the search similar to case *inco-o*.

Depending on the theory, not all the cases listed above are relevant. In particular the cases *inco-n* and *inco-o* only occur in theories that are not convex, either inherently or because of being a combination of two or more convex theories.

The CDCL(T) solvers are designed to be sound, but depending on the theory they might be incomplete either due to the undecidability of the theory, or since complete decision procedures for the theory are slower than incomplete implementations.

3 Algorithms for Lookahead-Based SMT Solving

The core idea of the lookahead implementation for SMT is to combine a tree-based lookahead search with a CDCL(T) SMT solver. However, there are several interesting design choices that need to be evaluated to build an algorithm for lookahead-based SMT solving. Perhaps most importantly, the combination can be done in two ways:

- Selecting decision atoms in the CDCL(T) framework using lookahead, or
- Building a DPLL tree by computing the lookahead heuristic using a CDCL(T) solver.

In the following we will cover the two approaches in more detail. The first approach, that we call Plain, is relatively straightforward to implement as an extension to the CDCL(T) algorithm. The second approach, called Tree, is somewhat more involved. We generalize the lookahead procedure described in Sec. 2 in a number of ways. First, we always include the theory propagations from *con-p* in the unit propagation closure computations, which will then be taken into account when computing the lookahead scores. Second, every time a new clause is learned either through *inco-e* or through standard CDCL learning, we update the propagation scores accordingly to reflect the new propagations resulting from the clause. Third, the lookahead phase provides information on whether setting some literal true would result in the cases *inco-n* or *inco-o* before committing to choosing the literal as a decision literal. We integrate this information into the score of the literals, giving low score to atoms that result in *inco-n* or *inco-o*.

```
   Input  : An SMT instance in CNF $\phi$.
   Output: $\{SAT, UNSAT\}$
   Data   : Decision stack $s$, literal $x_{\text{best}}$, map scores : $At(\phi) \to (\mathbb{N} \cup \{-\infty\})$
 1 while True do
 2     while Propagate($s$) results in conflict clause $c$ do
 3         if the decision level of $s$ is 0 then return UNSAT ;
 4         $\phi \leftarrow \phi \cup \{c\}$
 5         Backtrack $s$ until $c$ becomes implying
 6     end
 7     if $s$ contains all $x \in \phi$ then return SAT ;
 8     $x_{\text{best}} \leftarrow \top$
 9     scores $\leftarrow \{x_{\text{best}} \mapsto -\infty\}$
10     for all $x \in At(\phi)$ not in $s$ do
11         scores($x$) $\leftarrow$ $\textit{la-score}_{\phi \wedge \bigwedge_{i=0}^{n} E_i}(x)$
12         if conflict during a propagation then
13             goto line 2
14         else if scores($x$) > scores($x_{best}$) then
15             $x_{\text{best}} \leftarrow x$
16         end
17     end
18     $s \leftarrow s \cdot (\{x_{\text{best}}\})$
19 end
```

Algorithm 1: The Lookahead-Based CDCL(T) algorithm Plain.

3.1 Lookahead Heuristic in the CDCL(T) Framework

Algorithm 1 describes the CDCL(T) algorithm Plain that uses lookahead heuristic for selecting the decision literals. The algorithm gets as input the SMT instance ϕ in CNF and returns either *SAT* or *UNSAT* depending on the satisfiability of ϕ. The algorithm maintains the decision stack $s = (E_0, \ldots, E_n)$ where n is the decision level of s. Lines 2 – 6 correspond to computing the unit propagation closure $UP(\phi, s)$, and conflict resolution. The procedure *Propagate* computes both Boolean and theory unit propagation closure. If $UP(\phi, \bigcup_{i=0}^{n} E_i)$ is inconsistent, the *Propagate*(s) call results in a conflict clause c. Otherwise, s is updated to include the propagation closure. The simplified learned clause is inserted to ϕ on Line 4 and guides backtracking on Line 5. On Line 7, if the stack is consistent and all atoms are assigned, the algorithm returns SAT. The lookahead phase is implemented on Lines 10 – 17, where the solver computes the propagation scores of all unassigned atoms of $At(\phi)$. The propagation score is computed on Line 11 by calling *Propagate* on each unassigned atom. In case of a conflict, the algorithm returns to the conflict resolution process (Lines 2 – 6) by the *goto* statement on Line 13. In case of no conflicts, the algorithm extends the decision stack by the new frame containing the heuristically best literal x_{best} on Line 18 and returns to the main propagation phase.

It is interesting to note that implementing lookahead to the CDCL(T) solver results in an algorithm that is genuinely different from CDCL(T), since the clause learning takes place in two different phases, first on Line 2 as in the standard CDCL(T), and then inside the computation of the score on Line 11.

3.2 DPLL-based CDCL(T) Solver with Lookahead

Instead of integrating the lookahead as a heuristic to CDCL(T) the integration can be done in the opposite way, so that CDCL(T) is called as a sub-procedure in the DPLL-search. This approach is natural since DPLL is the basis of the original propositional lookahead algorithm. The resulting algorithm Tree is shown in Alg. 2.

In the Tree algorithm we "simulate" the DPLL tree with the CDCL(T) reasoning engine while still allowing the resulting system to learn clauses and even perform non-chronological backtracking. Let n_k be a node in the tree and $\sigma = \bigwedge_{i=0}^{k} L(n_i)$ the conjunction of the literals labelling the nodes in the path from the root n_0 to n_k. We call this simulation *forcing the decision stack*. To determine the labels of the children n_{k+1}, n'_{k+1} on Lines 35 – 37, first we force the decision stack correctly for the CDCL(T) algorithm on Lines 7 – 21, and then run lookahead on Lines 25 – 34.

Forcing the decision stack. To determine the labels for the children of n_k we backtrack the CDCL(T) solver to decision level zero. The decision stack of the solver is then set to $(E_0, \ldots, E_k)$ using the literals $L(n_i)$ for the decision literals d_i (Lines 7 – 21). The DPLL tree is built in depth-first order using a DFS stack during the stack forcing phase. The decisions are maintained in the stack of the CDCL(T) solver and apart from forcing the stack, the CDCL algorithm is run in a normal way. Forcing the stack has two possible outcomes:

- the CDCL(T) solver successfully reaches the decision level k (Lines 9 and 19), or
- the CDCL(T) solver encounters a conflict at some unit propagation on a frame E_j, $j \leq k$ (Lines 13 and 17).

In the former case the CDCL(T) solver enters the lookahead phase. In the latter case the solver is backtracked to the decision level j indicated by the conflict. The decision stack is then re-forced from the node n_j onwards.

On lines 7 – 21 the CDCL(T) solver stack is set according to the current node n_k. The loop handle situations where the decision forced by the DPLL tree have already been assigned in previous propagations. The node n_k can be closed if either propagating the decision $L(n_k)$ results in conflict on Line 12, or the negation of the decision literal is already in the CDCL(T) stack on Line 16. The CDCL(T) stack is forced normally if propagating $L(n_k)$ does not result in conflict on Line 9, and an empty decision stack element is inserted on Line 19 if $L(n_k)$ was already asserted previously. If the node n_i was closed while forcing the CDCL(T) stack, the search on n_i is abandoned and a new node is taken from the stack on Line 13. If after forcing the stack until n_k there are no unassigned atoms, the algorithm can deduce that the problem is satisfiable on Line 22.

Observations on forcing the stack. In general the CDCL(T) solver must explicitly be set to decision level k by backtracking it to a level j based on conflict analysis and forcing the decisions $L(n_j), \ldots, L(n_k)$ because of possible conflicts arising from the newly learned clauses, as well as to keep the sets of literals obtained with both Boolean and theory unit propagation in the different decision levels consistent. A conflict obtained at the stack $(E_0, \ldots, E_k)$ that results in backtracking to the stack $(E_0, \ldots, E_j)$, $j < k$, implies that $UP(\phi, d_1 \wedge \ldots \wedge d_k)$ is inconsistent, but not necessarily that $UP(\phi, d_1 \wedge \ldots \wedge d_{j+1})$ is unsatisfiable. As a result, a conflict at level k means that the corresponding node n_k must be closed but the algorithm needs to still study the remaining open nodes between n_j and n_k in the DPLL tree.

```
   Input  : An SMT instance in CNF $\phi$.
   Output: $\{SAT, UNSAT\}$
   Data   : Decision stack $s$, literal $x_{\text{best}}$, DFS stack $q$, DPLL root $n_0$, map
            scores : $At(\phi) \rightarrow \mathbb{N} \cup \{-\infty\}$
 1 $L(n_0) \leftarrow \top$
 2 $q.push(n_0)$
 3 while $True$ do
 4     $s \leftarrow ()$
 5     $n_k \leftarrow q.pop()$
 6     if $n_k$ is closed then continue;
 7     for $n_i$ in $n_0, \ldots, n_k$ do
 8         if $At(L(n_i))$ not assigned in $E_0 \cup \ldots \cup E_{i-1}$ then
 9             $s \leftarrow s \cdot (\{L(n_i)\})$
10             if $Propagate(s)$ results in conflict then
11                 if the decision level of $s$ is 0 then return $UNSAT$ ;
12                 close $n_i$
13                 goto line 3
14             end
15         else if $\neg L(n_i) \in E_0 \cup \ldots \cup E_{i-1}$ then
16             close $n_i$
17             goto line 3
18         else
19             $s \leftarrow s \cdot (\emptyset)$
20         end
21     end
22     if $s$ contains all $x \in At(\phi)$ then return $SAT$ ;
23     $x_{\text{best}} \leftarrow \top$
24     scores $\leftarrow \{x_{\text{best}} \mapsto -\infty\}$
25     for all $x \in At(\phi)$ not in $s = (E_0, \ldots, E_k)$ do
26         scores$(x) \leftarrow$ $la\text{-}score_{\phi \wedge \bigwedge_{i=0}^{k} E_i}(x)$
27         if conflict during a propagation then
28             if the decision level of $s$ is 0 then return $UNSAT$ ;
29             $q.push(n_k)$
30             goto line 3
31         else if scores$(x) >$ scores$(x_{best})$ then
32             $x_{\text{best}} \leftarrow x$
33         end
34     end
35     Create child nodes $n_{k+1}, n'_{k+1}$ for $n_k$
36     $L(n_{k+1}) \leftarrow x_{\text{best}}$
37     $L(n'_{k+1}) \leftarrow \neg x_{\text{best}}$
38     $q.push(n_{k+1})$
39     $q.push(n'_{k+1})$
40 end
```

Algorithm 2: The DPLL-based lookahead with CDCL(T) Tree.

Lookahead. Once the stack has been forced on the node n_k, the labels of the children of n_k are determined in the lookahead phase, which attempts to compute the heuristic score of all unassigned atoms of $\phi \wedge UP(\phi, \bigwedge_{i=0}^{k} E_i)$ (Lines 23 – 34). There are again two possible outcomes of the lookahead phase the algorithm needs to consider:

- the lookahead phase runs to completion, in which case the atom x_{best} with the highest propagation score is identified and the children of n_k will be labelled with x_{best} and $\neg x_{\text{best}}$ (Lines 35 – 37); or
- the lookahead phase finds a conflict in which case the CDCL(T) solver returns to forcing the decision stack according to the DPLL tree (Line 29).

The actual lookahead phase is performed on Line 26. If the phase detects a conflict on decision level zero, the algorithm returns *UNSAT* on Line 28, and for conflicts on higher decision levels the algorithm tries to force the stack again, now with the newly learned clauses in ϕ. Finally, after a successful lookahead phase, the algorithm extends the DPLL tree with the nodes n_{k+1}, n'_{k+1} labelled with the atom with the highest propagation score.

3.3 Algorithm Comparison

The two algorithms Plain and Tree are dramatically different in their pseudocode implementation. We show schematically the two algorithms in Fig. 1 highlighting their shared components and their differences. We emphasize that this representation is by nature informal and intended to clarify the relationship of the two approaches visually rather than to serve as a reference implementation. The boxes labelled *Lookahead* and *Conflict Handling* correspond to computing the lookahead and analysing and learning the conflict clauses, and are shared between both algorithms. In contrast, the CDCL(T) solver stack is built differently in the two approaches. Plain chooses a literal to propagate in the box *Propagate chosen literal*, while Tree does this in the box *Propagate forced stack*. In addition Tree maintains the search tree, denoted in the figure by the box DPLL *Tree*.

For Alg. 1, the box *Propagate chosen literal* corresponds to Line 2, the box *Conflict Handling* corresponds to Lines 2 – 6, and the box *Lookahead* corresponds to the Lines 10 – 17. For Alg 2, the box DPLL *Tree* does not correspond to specific lines but represents the concept of the tree that the algorithm maintains. The box *Propagate forced stack* corresponds to the Lines 7 – 21, the box *Lookahead* corresponds to the Lines 25 – 34, while the box *Conflict Handling* corresponds to the Lines 10 – 17.

4 Implementation

We have only implemented the algorithm Tree and not Plain. Tree more closely follows the original structure of lookahead implementations, and differs sufficiently from a standard CDCL implementation to offer a complementary view to how SMT solving can be done. The comparison of Tree and Plain is left for future work. The implementation of Tree (Alg. 2) is based on our SMT solver OpenSMT2 [11]. The solver supports the quantifier-free fragments of the logics of uninterpreted functions with equality, linear real arithmetics, and linear integer arithmetics, allowing us to experiment with both convex and non-convex theories.

The implementation of the lookahead phase includes two important optimizations. First, the heuristic needs to often backtrack due to inconsistent unit propagation, and if the atoms have a fixed order $x_1 \prec \ldots \prec x_n$ in which their propagation score is computed, this results in the lookahead score of the atoms in the beginning of the order being computed much more often than that of the atoms at the end of the order. To allow a more fair propagation schedule, we order the atoms as a ring so that $x_n \prec x_1$, and, after computing an inconsistent unit propagation closure at atom x_i, we continue the computation at atom $x_{(i+1) \bmod n}$.

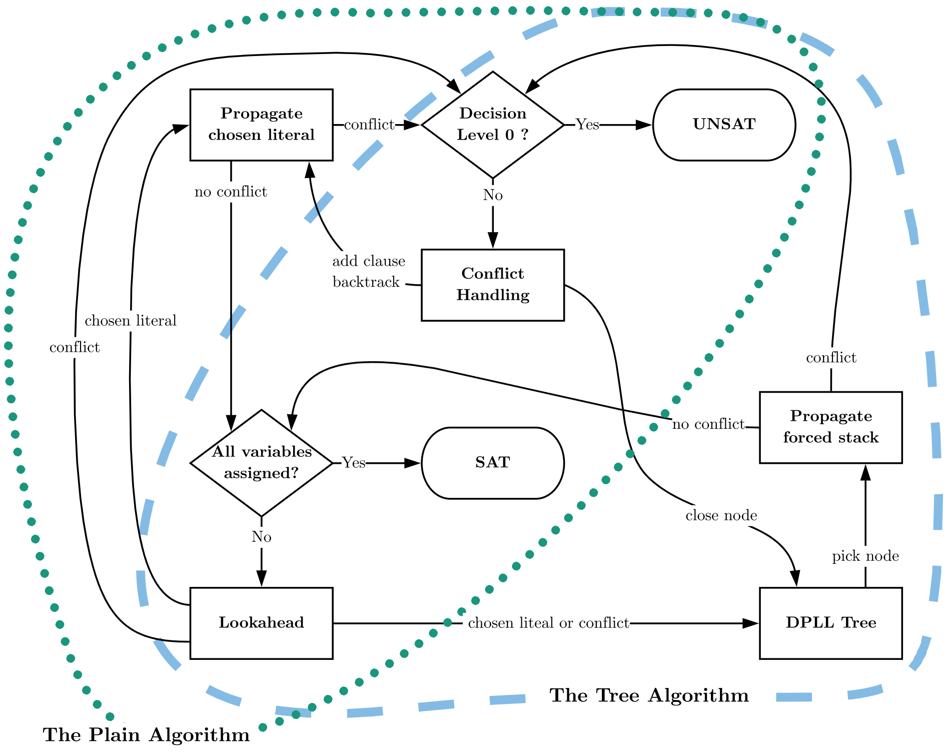

Figure 1: The algorithms Plain and Tree. The executions of the algorithms Plain and Tree start, respectively, in the box labelled *Propagate chosen literal*, and DPLL *Tree*.

Second, since the propagation heuristic we implemented looks for the atom $x \in At(\phi)$ that maximizes $\min_{p \in \{x, \neg x\}} |UP(\phi, p)|$, we can often avoid computing the propagation scores for certain atoms for which an upper bound of the propagation is already known. In particular, let l be a literal and $Ub(l)$ be an upper bound for $|UP(\phi, l)|$ that is initially set to zero. During the lookahead phase the algorithm maintains the current best atom x_{best}. Whenever $l \in |UP(\phi, q)|$ for some literal q for which the lookahead score is being computed, we update $Ub(l) \leftarrow \max(|UP(\phi, x)|, Ub(l))$. When computing the score for an atom x_k, if $Ub(x_k) < \mathit{la\text{-}score}(x_{\text{best}})$ or $Ub(\neg x_k) < \mathit{la\text{-}score}(x_{\text{best}})$, we know that $\mathit{la\text{-}score}(x_k) < \mathit{la\text{-}score}(x_{\text{best}})$, and therefore there is no need to compute the exact score of x_k.

We also implemented a simple restart scheme for building the DPLL tree of Alg. 2. In case the algorithm reaches a geometrically increasing restart limit on number of conflicts on the calls to *Propagate*, the algorithm is re-started while preserving the learned clauses added to the instance ϕ.

4.1 Applications of the Lookahead Heuristic to Parallelization

Often a randomization helps in obtaining better results in SMT solving through algorithm portfolios. However, the lookahead heuristic is by nature very deterministic. We introduce a simple form of randomization to the lookahead heuristic. Instead of maintaining a single best scored atom x_{best}, we maintain the k best scored atoms $x^1_{\text{best}}, \ldots, x^k_{\text{best}}$, and choose the decision literal randomly among them.

Finally, we adapted Alg. 2 for the divide-and-conquer approach in [12, 14]. The idea is to construct the DPLL tree until depth k to produce 2^k partitions, corresponding to the instances $\phi_i = \phi \wedge L(n^i_0) \wedge \ldots \wedge L(n^i_k)$ where $n^i_0 = n_0$ is the root of the tree, and n^i_k are the 2^k nodes at depth k. The instances ϕ_i can, for instance, be solved in parallel. This is done by adding a check immediately after the Line 22 of Alg. 2 on whether the decision level is k, and in that case jumping to the beginning of the while-loop. Note that due to the completeness of the algorithm Tree it is possible that not all 2^k partitions are created. In the extreme case, for example, the instance is shown unsatisfiable before reaching the level k for the first time. Results of this implementation for equality and uninterpreted functions and linear real arithmetics have been reported in [12, 14].

4.2 The Linear Integer Arithmetics Solver in OpenSMT2

The core components of a solver for linear integer arithmetics in modern SMT solvers are (1) a simplex solver that finds a solution to the query where free first-order constants are interpreted to range over the real numbers; and (2) a system that adds *cuts* that forbid non-integer values in future queries. A careful selection of cuts is critical for efficient implementation of the solver. In our current implementation in OpenSMT2 we limit the cuts to be of the form $x \leq c$ and $x \geq c + 1$, where x is a free first-order constant of the original query and $c \in \mathbb{Z}$ is obtained from a rational solution given by the simplex solver by rounding upwards or downwards to the next or previous integer. This implementation could be made more efficient by using more sophisticated methods for choosing the cuts. Fore more details, see, for example [13, 1, 6]. While we plan to study the relationship of more sophisticated cuts in the future in conjunction with the lookahead approach, we now prefer to keep the solver implementation simple to better understand the effect of lookahead in isolation.

5 Experiments

The goal of our experiments is to understand how the lookahead algorithm and its modifications work on different types of practical SMT instances in comparison to a standard SMT implementation which we refer to as CDCL(T) in the results.

The experiments were executed on a cluster where each node has 16 Intel(R) Xeon(R) E5-2630 v3 CPUs running at 2.40GHz. We executed 20 solvers simultaneously on each node and set the memory limit to 4GB for each solver. The evaluation was done on the SMTLIB benchmark set[1] using the set's quantifier-free formulas over equalities and uninterpreted functions (QF_UF), linear real arithmetics (QF_LRA) and linear integer arithmetics (QF_LIA). The set consists of 14531 instances in total. Each job, consisting of a solver and an instance, has a run time limit of 1000 seconds[2], and performing the full set of experiments reported in this paper takes roughly two days of computing in our setup, where we usually obtain 16 nodes simultaneously. To maximize our confidence on the correctness of the approach, we cross-checked all results against the results reported by the Z3 SMT solver [5]. The source code for our tool OpenSMT is available at http://verify.inf.usi.ch/content/lookahead. We use *scatter-plots* for reporting the results. The idea is to plot results on a plane, taking the horizontal

[1] http://smtlib.cs.uiowa.edu/

[2] We set the limit on the user time as defined by the Linux kernel.

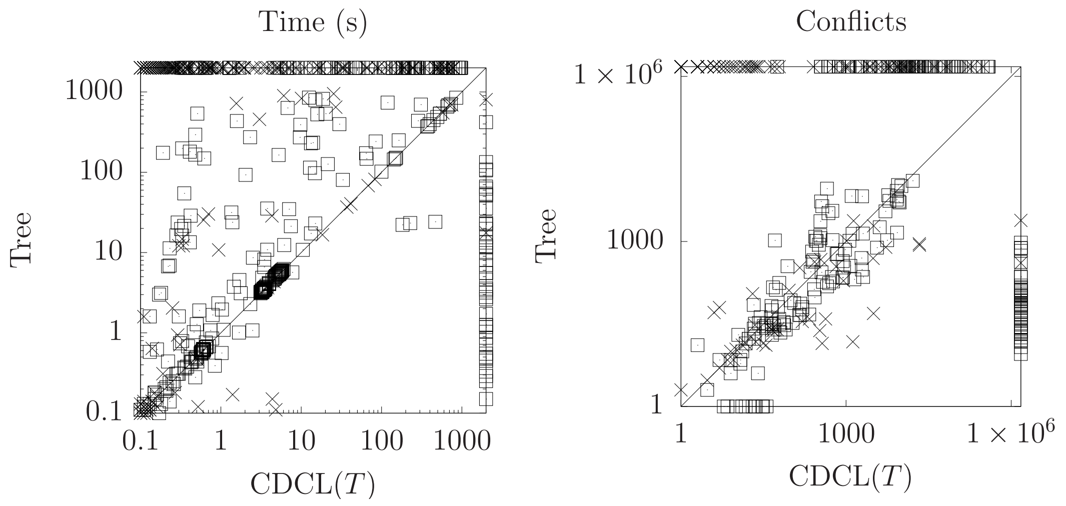

Figure 2: The Tree algorithm (*vertical axis*) compared to CDCL(T) (*horizontal axis*) on linear integer arithmetics (QF_LIA) formulas.

and vertical coordinates from the two approaches that we compare. Time-outs are drawn on the edges of the plots, and the diagonal line represents the points where the results of the two approaches would be on a par. In the figures we use the convention of boxes ($\boxdot$) denoting unsatisfiable and crosses ($\times$) denoting satisfiable instances.

Figure 2 (*left*) compares the run times of the Tree algorithm (Alg. 2) and the CDCL(T) algorithm on SMT instances with first-order atoms consisting of equalities and inequalities in linear integer arithmetics. On instances solved by both CDCL(T) and Tree, the CDCL(T) is usually faster. However, a closer analysis of the results shows that Tree solves 2% more instances than CDCL(T), and almost 40% more unsatisfiable instances than CDCL(T). In total in the QF_LIA benchmark set using OpenSMT2, the CDCL(T) solves 390 satisfiable and 421 unsatisfiable instances within the time limit, while Tree solves 252 satisfiable and 576 unsatisfiable instances. Surprisingly, two of these instances that are only solved by Tree are also satisfiable. Some of the instances are on the diagonal, and the search in most of these cases consists of a single, relatively expensive query to the theory solver. In general the results suggest that lookahead works extremely well in combination with a simple non-convex theory solver, and we assume that this is at least partly due to the tendency of the lookahead approach to search for promising literals instead of exploring branches containing decision literals that have a high VSIDS score but in the end result in the algorithm getting into a loop. In Fig. 2 (*right*) we show the number of conflicts obtained on the instances that both approaches solved. Interestingly, there is no big consistent bias in the unsatisfiable instances either for CDCL(T) or for Tree, while the conflicts for the satisfiable instances are lower for Tree. These observations support the conclusion that the efficiency of Tree in QF_LIA is due to its ability to avoid the branches which might cause our implementation to enter a loop.

In Fig. 3 we compare the Tree algorithm against the standard CDCL(T) algorithm on the linear real arithmetics instances. The results show that, while in most cases the CDCL(T) algorithm is an order of magnitude faster, there are certain unsatisfiable instances where the lookahead-based implementation is clearly faster, and a handful of instances where CDCL(T) times out while Tree determines the solution. We believe that the reason for the efficiency

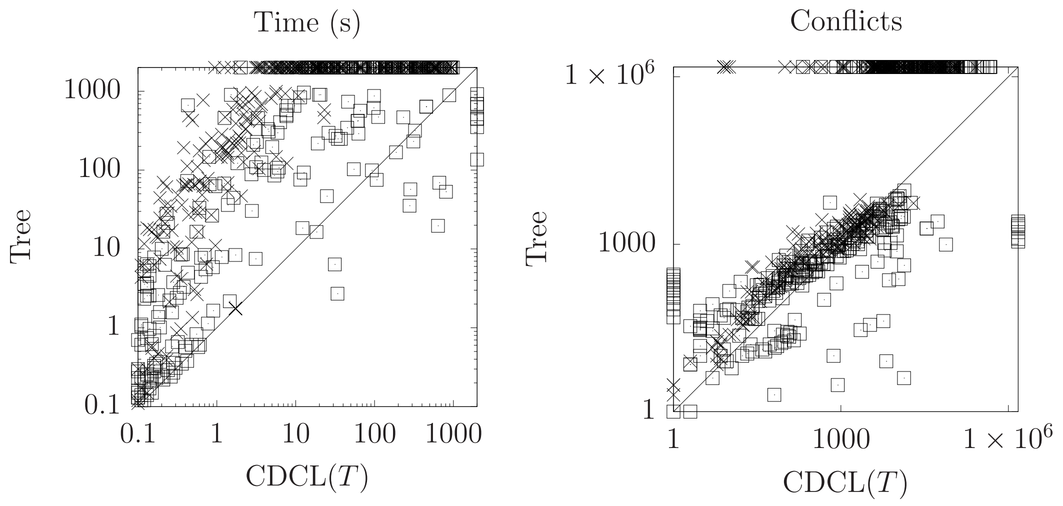

Figure 3: The Tree algorithm (*vertical axis*) compared to standard CDCL(T) (*horizontal axis*) on linear real arithmetics (QF_LRA) formulas.

of Tree is that the lookahead phase can quickly find many theory conflicts that would appear shallow in the search, but which go unnoticed by VSIDS for a longer time. In addition to the run time in Fig. 3 (*left*), we report also the number of conflicts reached by the time of determining the satisfiability of the instance in Fig. 3 (*right*). The graph shows that, at least on the instances that were solved by Tree, the number of decisions is often lower on the lookahead-based implementation than on the CDCL(T)-based implementation.

In Fig. 4 we make a similar comparison on the category QF_UF. Apart from a small number of exceptions, similar behavior where our implementation of Tree would outperform the standard CDCL(T) implementation is absent. This is true both for run times (*left*) and, although less dramatically, for the number of conflicts (*right*). We note that if the CDCL(T) can determine the satisfiability of the instance in few conflicts, the Tree algorithm almost never determines the satisfiability as fast. However, if the CDCL(T) algorithm does not determine the satisfiability in a few conflicts, there are some cases where Tree can determine the satisfiability in significantly less conflicts.

In Fig. 5 we report the results on implementing the restarts on Alg. 2. The results suggest that in the QF_LRA category there are many unsatisfiable instances where restarts give a small but consistent speed-up.

In general the efficiency of the algorithm Tree varies significantly depending on the benchmark, and it is safe to say that most benchmarks in the SMTLIB2 benchmark set we experimented with strongly favor the standard CDCL(T) algorithm. This is not a very surprising result given that a similar behavior is observed in SAT solving. However, we find it very interesting that there are some instances in the standard benchmark library for which lookahead consistently performs better, especially on QF_LIA, but also on QF_LRA. We believe that it is possible to implement Tree more efficiently in comparison to our prototype, and this will further increase the number of instances that are solved faster with lookahead.

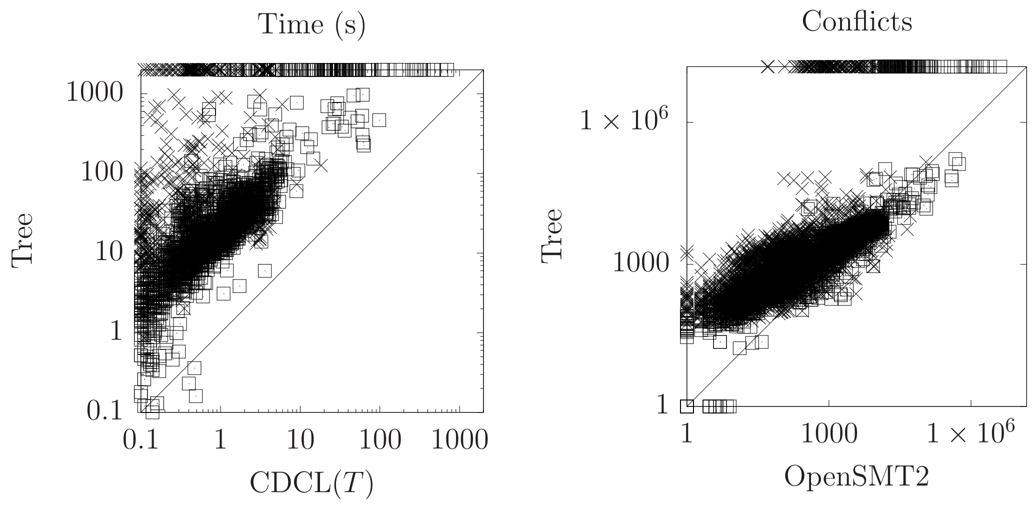

Figure 4: The Tree algorithm (*vertical axis*) compared to CDCL(T) on formulas over equality and uninterpreted functions (QF_UF) (*horizontal axis*)

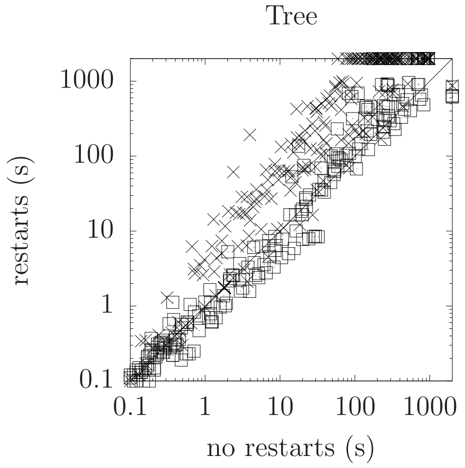

Figure 5: Effects of restarts on the Tree algorithm on linear real arithmetics (QF_LRA) formulas.

6 Conclusions

Lookahead is an expensive but powerful heuristic based on greedily branching on atoms that result in a lowest upper-bound on remaining search space. In this work we study the possibility of integrating lookahead with conflict-driven clause learning SAT solvers and in particular with DPLL(T) based SMT solvers. We believe that our results contribute to the understanding of the interaction between lookahead and SMT both in theory and practice.

On the theoretical level, we discuss how the lookahead concept of upper bound for search space changes in SMT and in particular in the non-convex theory of linear integer arithmetics.

We propose two ways of integrating lookahead with SMT, the algorithms Plain and Tree, and provide a pseudocode implementation for both. We describe certain optimizations to make the lookahead computation more efficient and discuss applying the algorithm Tree to parallelization of SMT solvers.

On the practical level, we report experimental evaluation of the algorithm Tree on the full set of instances from the widely used SMT benchmark categories QF_UF, QF_LRA, and QF_LIA. Our evaluation shows that lookahead in QF_LIA allows us to solve significantly more instances compared to the baseline implementation, and is efficient on some benchmarks from QF_LRA, suggesting a non-trivial interaction between lookahead and SMT.

In future we are interested in applying lookahead in combinations of SMT theories. In addition we are interested in understanding whether the proofs produced by solvers applying lookahead-based ideas are suitable for constructing inductive invariants in model checking. Finally, we plan to study better implementation techniques for lookahead, such as generalizing the two-watched-literal scheme [16] to a three-watched-literal scheme that would detect clauses that can be made implying by assigning a single atom.

Acknowledgements. This work was supported by the SNF grant number 200020_166288.

References

[1] Martin Bromberger. A reduction from unbounded linear mixed arithmetic problems into bounded problems. In *Proc. IJCAR*, volume 10900 of *LNCS*, pages 329–345. Springer, 2018.

[2] Jingchao Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoing techniques. In *Proc. SAT 2009*, volume 5584 of *LNCS*, pages 298–311. Springer, 2009.

[3] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. TACAS 2008*, volume 4963 of *LNCS*, pages 337 – 340. Springer, 2008.

[6] Alberto Griggio. A practical approach to satisability modulo linear integer arithmetic. *JSAT*, 8(1/2):1–27, 2012.

[7] Marijn Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *JSAT*, 2(1-4):47–59, 2006.

[8] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 155–184. IOS Press, 2009.

[9] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Proc. SAT 2016*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.

[10] Antti E. J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Proc. LPAR 2010*, volume 6397 of *LNCS*, pages 372–386. Springer, 2010.

[11] Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In *Proc. SAT 2016*, number 9710 in LNCS, pages 547 – 553. Springer, 2016.

[12] Antti E. J. Hyvärinen, Matteo Marescotti, and Natasha Sharygina. Search-space partitioning for parallelizing SMT solvers. In *Proc. SAT 2015*, volume 9340 of *LNCS*, pages 369–386. Springer, 2015.

[13] Daniel Kroening, Jérôme Leroux, and Philipp Rümmer. Interpolating quantifier-free presburger arithmetic. In *Proc. LPAR 2010*, volume 6397 of *LNCS*, pages 489–503. Springer, 2010.

[14] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. Clause sharing and partitioning for cloud-based SMT solving. In *Proc. ATVA 2016*, pages 428–443, 2016.

[15] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC 2001*, pages 530–535. ACM, 2001.

[17] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *Proc. SAT 2018*, volume 10929 of *LNCS*, pages 111–121. Springer, 2018.

[18] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[19] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937 – 977, 2006.

[20] Patrick Simons. *Extending and Implementing the Stabel Model Semantics.* PhD thesis, Helsinki University of Technology, 2000.

[21] Ramin Zabih and David McAllester. A rearrangement search strategy for determinig propositional satisfiability. In *Proc. AAAI-88*, pages 155–160. ACM, 1988.