Partitioning SAT Instances for Distributed Solving

Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä

Aalto University School of Science and Technology
Department of Information and Computer Science
PO Box 15400, FI-00076 AALTO, Finland
{Antti.Hyvarinen,Tommi.Junttila,Ilkka.Niemela}@tkk.fi

Abstract. In this paper we study the problem of solving hard propositional satisfiability problem (SAT) instances in a computing grid or cloud, where run times and communication between parallel running computations are limited. We study analytically an approach where the instance is partitioned iteratively into a tree of subproblems and each node in the tree is solved in parallel. We present new methods which combine clause learning and look-ahead to construct partitions, evaluate their efficiency experimentally, and finally demonstrate the power of the approach in a real grid environment by solving several instances that were not solved in a SAT solver competition.

1 Introduction

This paper develops novel techniques for solving hard propositional satisfiability (SAT) problems in widely distributed computing environments such as computing grids or clouds. An example of this kind of an environment is NorduGrid (`http://www.nordugrid.org/`) that we use in some of the experiments of this paper.

NorduGrid provides a high number of readily available, high-end, heterogeneous computing facilities cost-efficiently, suggesting that this type of computing is an interesting target for applications such as SAT solving. The distributed environments in this work differ in some significant aspects from some other common distributed computing environments, such as multi-core workstations, and even local parallel environments, such as computing clusters: once a computation or a job is submitted to a grid or a cloud, the execution of the job is not immediate, as it is delayed by an amount of time depending on the availability of the computing resources; the executing jobs have limited communication capabilities restricted by the site security policies; and the jobs are only allowed to use a predetermined amount of CPU time and memory during their execution.

The goal of this work is to develop distributed SAT solving for such environments using the best available SAT solvers as black-box subroutines with minimal modifications. The goal can be straightforwardly achieved by exploiting the randomized nature of current state-of-the-art SAT solvers with the *simple distribution* (SD) approach, where one just runs a randomized solver a number of times independently. This leads to surprisingly good speed-ups even in a grid environment with substantial communication and other delays [1]. The approach could be extended by applying particular restart strategies [2, 3] or employing an algorithm portfolio scheme [4, 5]. Another key feature

in modern SAT solvers is the use of conflict driven clause learning techniques. Extending the simple distribution approach with this feature leads to a powerful SAT solving technique [6].

While the SD approach has led to surprisingly good performance, it provides no mechanism for splitting the search into more manageable portions which can be treated in parallel. A *partition function* maps a SAT instance ϕ to a set of *derived* instances $\phi_1, \ldots, \phi_n$ by including additional constraints to ϕ so that the original instance is satisfiable if and only if at least one of the derived instances is. We call *plain partitioning* an approach where a partition function is used once and each resulting instance is solved in parallel. A typical approach to splitting the search space is to use guiding path or semantic decomposition based techniques [7–11]. However, these techniques impose constraints on the underlying solver technology and are not ideal for grids and clouds since they require relatively frequent communication between jobs. Furthermore, an improper partition function can produce derived instances as difficult to solve as the original instance. In such cases plain partitioning leads to a detrimental speed-up anomaly, where an increase in parallelism results in an increase in expected run time even if the delays in the environment are ignored [12]. In an environment with run time limitations this results in decrease of solving probability.

This work studies an approach for distributed SAT solving with *partition trees* [13]. The approach has modest communication requirements and can use any SAT solver as a black-box subroutine. The basic idea in the partition tree approach is straightforward: jobs consisting of a solver and a SAT instance run in the distributed environment. A partition function is used to construct the SAT instances, which are organized as a tree. The first job, consisting of the original instance, will be the root of the tree; the subsequent child jobs are constructed by applying the partition function to the original instance, and later recursively to the derived instances. The resulting tree is expanded until a solution can be determined or all parallel resources are in use. In the latter case, the resource limits guarantee that jobs will eventually terminate and more constrained, hopefully easier to solve, jobs can be submitted to the environment.

The main contributions of this work are the following. (i) We prove that when using the partition tree approach the expected run time will never increase as more resources are introduced. (ii) We develop novel partition functions, which use conflict driven clause learning techniques, and two use in addition *unit propagation lookahead* [14] to form the derived instances. (iii) We develop a novel method for speeding up the computation of lookahead, and (iv) present results for a learning lookahead solver. (v) We show that the partition tree approach is in many cases superior to the simple distribution approach, and (vi) finally show the efficiency of the approach by solving several instances not solved in the SAT 2009 solver competition (`http://www.satcompetition.org/`).

2 Preliminaries

A literal l is a propositional variable x or its negation $\neg x$; as usual, we define that $\neg\neg x = x$. A clause is a disjunction of literals and a (CNF) formula is a conjunction of clauses. A clause is unit if it only contains one literal. Whenever convenient, we may

interpret a formula as a set of clauses and a clause as a set of literals. For instance, the formula $\phi = (x) \wedge (\neg x \vee \neg y) \wedge (y \vee \neg x \vee z) \wedge (x \vee y) \wedge (y \vee v \vee \neg w)$ can be represented as a set $\{\{x\}, \{\neg x, \neg y\}, \{y, \neg x, z\}, \{x, y\}, \{y, v, \neg w\}\}$. Let $\mathrm{vars}(\phi)$ denote the set of variables occurring in ϕ and $\mathrm{lits}(\phi) = \{x, \neg x \mid x \in \mathrm{vars}(\phi)\}$. A truth assignment τ for a formula ϕ is a subset of $\mathrm{lits}(\phi)$; τ is (i) inconsistent if both $x \in \tau$ and $\neg x \in \tau$ for some variable x, (ii) consistent if it is not inconsistent, and (iii) complete for ϕ if either $x \in \tau$ or $\neg x \in \tau$ for each $x \in \mathrm{vars}(\phi)$. A literal l is assigned to true by τ if $l \in \tau$ and to false if $\neg l \in \tau$; if it is assigned to either, it is called assigned by τ. For any truth assignment or other set of literals τ, we write $\phi \wedge \tau$ to denote the formula $\phi \wedge \bigwedge_{l \in \tau}(l)$. A complete and consistent truth assignment satisfies the formula ϕ if it includes at least one literal of each clause in ϕ; if such a satisfying assignment exists, ϕ is satisfiable and unsatisfiable otherwise.

Given a formula ϕ and a truth assignment τ for it, we use $\mathrm{up}(\phi, \tau)$ to denote the set of literals implied by *unit propagation* on ϕ under τ; formally, $\mathrm{up}(\phi, \tau)$ is the smallest set U of literals satisfying (i) $\tau \subseteq U$, and (ii) if a clause $(l_1 \vee ... \vee l_n)$ is in ϕ and $\{\neg l_1, ..., \neg l_{i-1}, \neg l_{i+1}, ..., \neg l_n\} \subseteq U$, then $l_i \in U$. Note that if ϕ contains a unit clause (l), then $l \in \mathrm{up}(\phi, \tau)$ for all truth assignments τ. Obviously, if $\mathrm{up}(\phi, \tau)$ is inconsistent, then the formula $\phi \wedge \tau$ is unsatisfiable. As an abbreviation, we may write $\mathrm{up}(\phi)$ to denote $\mathrm{up}(\phi, \emptyset)$.

3 Approaches for Distributed Solving

This work studies distributed solving of difficult SAT instances by partitioning an instance into independently solvable subproblems, solving the subproblems, and determining the satisfiability of the original instance by combining the results. The presented solving methods are designed for environments such as computing grids and clouds. These differ from parallel or multi-core environments by providing large amounts of easily available computing power on the expense of placing limitations on the run times and communication of the executions. Furthermore, the framework developed here assumes nothing on the implementation of the SAT solver, but instead treats the solver as a black-box which takes as input a SAT instance and returns either satisfiable or unsatisfiable. In the following analysis we will first ignore the run time limitations, and then later mention how they affect the results.

Simple Distribution. Grids and clouds encourage certain approaches for SAT solving. An efficient approach is the simple distribution (SD), where *randomized SAT solvers*, taking as input the instance and optionally a seed used to initialize the random number generator of the solver, are run in parallel and the solution is obtained from the first solver that finishes and finds the instance either satisfiable or unsatisfiable. The approach is especially suitable for SAT solving, since most modern sequential solvers already employ some form of randomization. This approach has an obvious drawback, since minimum run time of the instance limits the obtainable speed-up.

Plain Partitioning. The simple distribution approach can be strengthened by forcing constraints on the overlap of the work of independent SAT solvers. The constraints

result in smaller search spaces for the solvers and, hopefully, executions which can be finished in shorter time than the original execution. In this work, we use *partition functions* to constrain the overlap. A *partition function* $\mathcal{P}(\phi, n)$ maps a SAT instance ϕ and an integer $n \geq 2$ to a set $\{\phi_1, \ldots, \phi_n\}$ of *derived instances* such that (i) $\phi_1 \vee \ldots \vee \phi_n \equiv \phi$, and (ii) $\phi_i \wedge \phi_j$ is unsatisfiable for $i \neq j$. From (i) it follows that the instance ϕ is satisfiable if and only if at least one derived instance is satisfiable.

A simple approach to solving SAT instances in a distributed environment called *plain partitioning* works by applying $\mathcal{P}(\phi, n)$ to an instance ϕ and solving the derived instances $\phi_1, \ldots, \phi_n$ in parallel. The plain partitioning approach bears close resemblance to the guiding path approach [9, 15], and, provided that the run time reduction caused by the partition function depends naturally on n, the plain partitioning approach can solve arbitrarily difficult problems given a sufficiently large n [12]. However, plain partitioning has some fundamental flaws, which prevent it from being a practical solving method in distributed environments as such. Firstly, it is of course problematic to determine the value for n. Secondly, a more subtle problem is that if no guarantees can be given on the run times of the derived instances and the instance to be solved is unsatisfiable, increasing n arbitrarily might increase the expected run time. We call a badly working partition function, which results in derived instances with run times equal to that of the original instance, a *void partition function*[1]. Void partition functions are especially harmful for proving unsatisfiability, since it is not possible to obtain any speedup with a void partition function in plain partitioning [12]:

Proposition 1. *Let ϕ be an unsatisfiable instance and $\mathcal{P}(\phi, \cdot)$ a void partition function. Then the expected run time of the plain partitioning approach is at least as large when using the partition function $\mathcal{P}(\phi, n)$ as when using $\mathcal{P}(\phi, n-1)$.*

The Partition Tree Approach. To overcome these difficulties, we use instead $\mathcal{P}(\phi, n)$ to construct a *partition tree* and attempt the solving of the nodes in the tree in parallel. A partition tree of a formula ϕ is a tree rooted at ϕ, where nodes are propositional formulas, all except the leaves having n children constructed with a partition function. A SAT instance can be shown satisfiable by showing any node of the partition tree satisfiable, and unsatisfiable by solving at least one instance on every path from the root to the leaves. See Fig. 1 (right) for an example of a partition tree and such a set of instances. This approach has the advantage over plain partitioning that the expected run time of the approach cannot be higher than that of solving ϕ with a sequential SAT solver: since ϕ is in the root of the partition tree, showing it unsatisfiable suffices to prove unsatisfiability. We prove the following stronger claim for the partition tree approach, which intuitively states that if more parallel resources are used, the expected time required to solve the instance ϕ decreases. We formalize this assuming that all leaves of the tree have the same amount k of ancestors (i.e., the *height* of the tree is k), and all instances in the tree are executed simultaneously with no delay.

Proposition 2. *Let ϕ be an unsatisfiable instance, T_k and T_m be two partition trees of height k and m, respectively, constructed with a void partition function, and $k < m$.*

[1] Given a SAT solver that efficiently concentrates on solving unsatisfiable instances, a partition function can be void, for example, when the SAT instance consists of an unsatisfiable and a satisfiable part sharing no variables and the partitioning constrains only the satisfiable part.

Then the expected run time of the partition tree approach is at most as large when using T_m as when using T_k.

Proof. We show by induction on the height of the partition tree that the probability that ϕ is solved within time t cannot decrease, from which the claim follows. Let $q(t)$ be the probability that ϕ is solved sequentially within time t, $q'(t)$ its derivative at t, and $q_i(t)$ denote the probability that ϕ is solved within time t using a partition tree of height i. Then the probability $q_0(t) = q(t)$. The probability that the instance is solved within time t with the partition tree approach and tree of height one is $q_1(t) = \int_0^t (q'(\tau) + (1 - q(\tau))nq'(\tau)q(\tau)^{n-1}\mathrm{d}\tau$, that is, the sum of probability $q'(\tau)\mathrm{d}\tau$ that the instance is solved in the root of the tree at time τ, and the probability that the instance has not been solved in the root, has been solved by all children but one by time τ, and is solved at time τ in the last child. A direct calculation shows that $q_1(t) \geq q_0(t)$. Assume now that $q_k(t) \geq q_{k-1}(t)$ for all $t \geq 0$. As previously, $q_{k+1}(t) = \int_0^t (q'(\tau) + (1 - q(\tau))nq'_k(\tau)q_k(\tau)^{n-1})\mathrm{d}\tau = q(t) + q_k(t)^n - \int_0^t q(t)nq'_k(\tau)q_k(\tau)^{n-1}\mathrm{d}\tau$. Integration by parts on the negative term results in $q_{k+1}(t) = q(t) + q_k(t)^n - q_k(t)^n q(t) + \int_0^t q_k(\tau)^n q'(\tau)\mathrm{d}\tau = q(t) + (1 - q(t))q_k(t)^n + \int_0^t q_k(\tau)^n q'(\tau)\mathrm{d}\tau$. By the induction hypothesis $q_{k+1}(t) \geq q(t) + (1 - q(t))q_{k-1}(t)^n + \int_0^t q_{k-1}(\tau)^n q'(\tau)\mathrm{d}\tau = q_k(t)$ □

In practice the computing environment places some limitations on the approach. The number of parallel computing resources is limited, and therefore only a subset of the instances of an arbitrary partition tree can be executed simultaneously. Since the run times of these instances are bounded, the resources will always become available again. Therefore the partition tree is constructed on-the-fly. In our experiments, we use an approach where an instance is first submitted for solving and, while it is being solved, the partition function is applied to the instance locally to obtain the children of the instance, later submitted and repartitioned. The partition order is breadth first. Once a subtree is shown unsatisfiable, it is no longer expanded, and therefore it is not necessary to determine the height of the tree in advance.

The comparison between plain partitioning and the partition tree approach is similar also when the execution times are limited, only this time it is also possible that the instance cannot be solved. Figure 1 shows an example where an instance ϕ is solved using a randomized SAT solver in a distributed environment and plain partitioning (*left*); and the partition tree approach (*right*) using a void partition function. As the partition function is void, all derived instances are as difficult as the original instance. The plain partitioning approach is "unlucky" and fails to find a solution since the instance ϕ_1 is terminated after its execution time bound is exceeded. The partition tree approach is even more unlucky, and fails to find solutions for the two instances ϕ_1 and ϕ'_2. In the partition tree approach, however, the instance ϕ'_1 is solved and therefore the subtree rooted at ϕ'_1 can be determined unsatisfiable even though ϕ_1 times out. On the other hand, the subtree rooted at ϕ'_2 can be determined unsatisfiable even though ϕ'_2 times out since ϕ_3 and ϕ_4 are solved. Note that the same scenario is possible even if the partition function is not void.

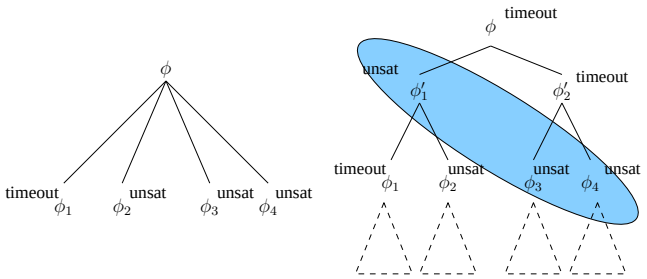


Fig. 1. An example of solving an unsatisfiable SAT instance ϕ with plain partitioning (*left*), where the instance is not solved due to a time out in ϕ_1, and the partition tree approach (*right*), where solving succeeds, since ϕ'_1 is shown unsatisfiable. All paths from the root to the leaves pass through the unsat instances in the shaded set.

4 DPLL-Based Partitioning with Lookahead

Our first new partition function utilizes and extends the ideas applied in traditional, chronologically backtracking and non-learning, DPLL-based SAT solvers employing unit propagation lookahead (see e.g. [14]) such as SATZ and MARCH. Given a formula ϕ, such solvers basically try to iteratively build a satisfying truth assignment τ by heuristically selecting a currently unassigned literal l and then considering two branches: one for the truth assignment $\mathrm{up}(\phi, \tau \cup \{\neg l\})$ and one for $\mathrm{up}(\phi, \tau \cup \{l\})$. If the considered truth assignment is inconsistent, the branch is closed and the solver backtracks chronologically. In order to prune the resulting *search tree*, these solvers also apply the so-called *one-step unit propagation lookahead* (or simply lookahead) to extend truth assignments in a satisfiability preserving way and, thus, also to detect inconsistencies and close unsuccessful branches earlier. As the truth assignments in the search tree nodes k steps below the root are mutually exclusive and cover all satisfying truth assignments, our core idea here is to use these as partitioning constraints when we want to partition a formula into 2^k derived formulas.

We next review the lookahead procedure (Sect. 4.1) and formally describe the partition function (Sect. 4.2). In addition, we give a novel technique for speeding up the computation of lookahead (Sect. 4.3) and, for the sake of analyzing the results in forthcoming sections, evaluate the efficiency of the lookahead procedure when applied as a formula simplifying technique (Sect. 4.4).

4.1 One-Step Unit Propagation Lookahead

The lookahead is based on the so-called failed literal rule. Given a formula ϕ and a truth assignment τ for it, a literal $l \in \mathrm{lits}(\phi)$ is a *failed literal* under $\phi \wedge \tau$ if $\mathrm{up}(\phi, \tau \cup \{l\})$ is inconsistent. As a consequence, if l is a failed literal under $\phi \wedge \tau$, then $\phi \wedge (\tau \cup \{l\})$ is unsatisfiable, implying that $\phi \wedge \tau$ is satisfiable iff $\phi \wedge (\tau \cup \{\neg l\})$ is. The failed literal rule states that if l is a failed literal under $\phi \wedge \tau$, then one can extend τ with $\neg l$ when searching for the satisfying truth assignments for $\phi \wedge \tau$. Note the monotonicity of failed

literals: if l is a failed literal under $\phi \wedge \tau$, then it is also under $\phi \wedge \tau'$ for any truth assignment $\tau' \supseteq \tau$.

Example 1. Assume a formula $\phi = (\neg x_4 \vee \neg x_7 \vee x_{15}) \wedge (\neg x_{15} \vee \neg x_3 \vee x_{21}) \wedge (\neg x_{21} \vee x_5 \vee x_{17}) \wedge (\neg x_{17} \vee \neg x_{60} \vee x_{89}) \wedge (\neg x_{17} \vee \neg x_{89}) \wedge ...$ and a truth assignment $\tau = \{x_7, x_3, \neg x_5, x_{60}\}$. Now x_4 is a failed literal under $\phi \wedge \tau$ as $\mathrm{up}(\phi, \tau \cup \{x_4\})$ is conflicting; thus $\phi \wedge \tau$ is satisfiable iff $\phi \wedge (\tau \cup \{\neg x_4\})$ is.

The lookahead procedure then applies the failed literal rule until there are no more failed literals unassigned by the truth assignment τ. The result of applying lookahead on a formula ϕ and truth assignment τ, denoted by $\mathrm{lookahead}(\phi, \tau)$, is, thus, the smallest set U of literals including τ and closed under the failed literal rule:

1. $\tau \subseteq U$, and
2. if a literal $l \in \mathrm{lits}(\phi)$ is a failed literal under $\phi \wedge U$, then $\neg l \in U$.

Observe that (i) if $\mathrm{lookahead}(\phi, \tau)$ is inconsistent, then $\phi \wedge \tau$ is unsatisfiable, and (ii) $\phi \wedge \tau$ is satisfiable iff $\phi \wedge \mathrm{lookahead}(\phi, \tau)$ is.

Note that computation of the lookahead can very time consuming: the best currently known techniques require in the worst case at least cubic time in the number of variables in the formula. Thus, solvers such as SATZ and MARCH do not usually compute the full lookahead but only apply the failed literal rule only to a subset of unassigned variables. Naturally, many heuristics have been developed to speedup lookahead computation (see e.g. [14] for existing ones and Sect. 4.3 below for a new one). In addition to the failed literal rule, there are also other search tree pruning rules such as the "necessary assignments" and "double lookahead" rules (see e.g. [14, 16]); evaluation of the efficiency of these rules in the context of partitioning is left to future work.

4.2 The Partition Function

As mentioned above, the proposed partition function uses the (up to) 2^k nodes at depth k in the search tree of a non-learning lookahead SAT solver to partition the formula ϕ. The pseudo-code for the function is shown in Fig. 4(a); to obtain a partitioning, it is invoked with *latree-partition*$(\phi, \emptyset, 0, k)$.

In the pseudo-code, the function *lookahead*(ϕ, τ') computes both the lookahead set $\tau'' = \mathrm{lookahead}(\phi, \tau')$ and a *variable selection heuristic* function h associating a value to each variable x not assigned by τ''. In our experiments, we use the following *lookahead balancing heuristics* function (adopted from the SMODELS stable models solver [17]) that tries to estimate the worst case search tree size after selecting x. As x is not assigned by τ'', we define the "remaining search tree size estimates" $h^-_{\phi,\tau}(x) = 2^{|\mathrm{vars}(\phi)| - |\mathrm{up}(\phi, \tau \cup \{\neg x\})|}$ and $h^+_{\phi,\tau}(x) = 2^{|\mathrm{vars}(\phi)| - |\mathrm{up}(\phi, \tau \cup \{x\})|}$, i.e. the numbers of unassigned variables after branching on $\neg x$ and x, respectively, and performing unit propagation. To estimate the remaining search tree size on both branches, we define the heuristic function h to be

$$h_{\phi,\tau}(x) = \max\left\{h^-_{\phi,\tau}(x), h^+_{\phi,\tau}(x)\right\}$$

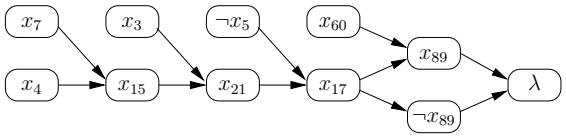


Fig. 2. A conflict graph

and select a variable that minimizes the function (ties are broken, e.g., randomly). Observe that this heuristic function is obtained practically as a side product when computing the lookahead. The computation of the lookahead and the heuristic function h is very important for obtaining small search trees (and, thus, good partitionings); therefore, whenever a time limit is imposed on the partition function, we try to divide the available time evenly between the lookahead functions in the search tree nodes.

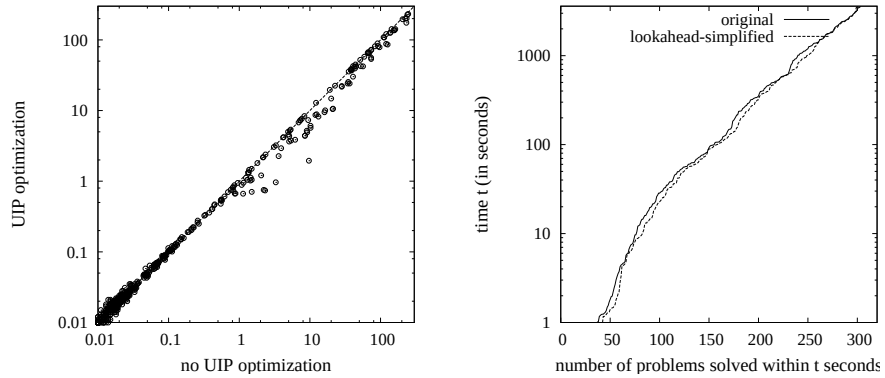
4.3 Exploiting Unique Implication Points

We now show how to apply the conflict analysis technique [18, 19] used in modern clause learning SAT solvers to enhance the failed literal rule in a way that allows it to sometimes detect multiple failed literals at the same time.

Assume a formula ϕ, a consistent truth assignment τ for ϕ that is closed under unit propagation (meaning that $\mathrm{up}(\phi, \tau) = \tau$), and a literal $l \in \mathrm{lits}(\phi)$ that is not assigned by τ. We now want to check whether l is a failed literal under $\phi \wedge \tau$, i.e. whether $\mathrm{up}(\phi, \tau \cup \{l\})$ is inconsistent. To do this, we start from $\tau \cup \{l\}$ and apply the unit propagation rule until no new literals can be deduced or an inconsistency is found. We now assume that $\mathrm{up}(\phi, \tau \cup \{l\})$ is inconsistent. Assuming that the computation of $\mathrm{up}(\phi, \tau \cup \{l\})$ is terminated as soon as an inconsistency is detected, it can be characterized by the corresponding *conflict graph*, which is a directed acyclic graph $\mathcal{G}_{\phi,\tau,l} = \langle \mathcal{V}, \mathcal{E} \rangle$ with a vertex set $\mathcal{V} \subseteq \mathrm{lits}(\phi) \cup \{\lambda\}$ for a special symbol $\lambda \notin \mathrm{vars}(\phi)$ and fulfilling the following conditions:

1. If $l' \in \mathcal{V}$, $l' \neq \lambda$, and $l' \in \tau \cup \{l\}$, then l' does not have any incoming edges.
2. If $l' \in \mathcal{V}$, $l' \neq \lambda$, and $l' \notin \tau \cup \{l\}$, then l' has a non-empty set of incoming edges originating from the vertices l_1,...,l_k and the clause $(\neg l_1 \vee ... \vee \neg l_k \vee l')$ is in ϕ.
3. There is exactly one variable $x \in \mathrm{vars}(\phi)$ such that both $x, \neg x \in \mathcal{V}$. The special symbol λ is a vertex that has incoming edges from these vertices x and $\neg x$ only.

A vertex $l' \neq \lambda$ such that $\neg l' \notin \tau$ is a *unique implication point* in the conflict graph if all the paths from the vertex l to the vertex λ go through l' (observe that the vertex l is a unique implication point). As the edges incoming to a vertex describe one application of the unit propagation rule, we have the following result. Take any unique implication point l' and the sub-graph of $\mathcal{G}_{\phi,\tau,l}$ induced by vertex set consisting the vertex l' and the vertices having a path to λ not visiting l'. Now this sub-graph is the conflict graph corresponding to a computation of $\mathrm{up}(\phi, \tau \cup \{l'\})$ and, thus, $\mathrm{up}(\phi, \tau \cup \{l'\})$ is inconsistent and l' is a failed literal under $\phi \wedge \tau$.



(a) comparing lookahead computation times (in seconds)

(b) the effect of applying lookahead as a preprocessing technique

Fig. 3. Some experimental results on computing and applying lookahead

Example 2. Assume a formula $\phi = (\neg x_4 \vee \neg x_7 \vee x_{15}) \wedge (\neg x_{15} \vee \neg x_3 \vee x_{21}) \wedge (\neg x_{21} \vee x_5 \vee x_{17}) \wedge (\neg x_{17} \vee \neg x_{60} \vee x_{89}) \wedge (\neg x_{17} \vee \neg x_{89}) \wedge \ldots$ and a truth assignment $\tau = \{x_7, x_3, \neg x_5, x_{60}\}$ closed under unit propagation. Now x_4 is a failed literal under $\phi \wedge \tau$ as $\mathrm{up}(\phi, \tau \cup \{x_4\})$ is conflicting; the conflict graph corresponding to a sequence of unit propagation rule applications is shown in Fig. 2. Now the nodes x_4, x_{15}, x_{21} and x_{17} are all unique implication points and thus the literals x_4, x_{15}, x_{21} and x_{17} are all failed literals under $\phi \wedge \tau$. Observe that $\neg x_{15}$, $\neg x_{21}$ and $\neg x_{17}$ do not necessarily belong to the set $\mathrm{up}(\phi, \tau \cup \{\neg x_4\})$ and thus this unique implication point exploitation method is able to derive truly additional failed literals.

We have experimentally evaluated the efficiency of this technique by implementing a lookahead procedure including it on top of the MINISAT solver [20] (version 1.14) and computing the lookahead set $\mathrm{lookahead}(\phi, \emptyset)$ for all the 887 formulas ϕ in the "crafted", "industrial", and "random" categories of the SAT-COMP 2007 solver competition benchmark set (see `http://www.satcompetition.org/`). The lookahead computation time was limited to 300 seconds: of the 887 formulas, the lookahead computation finished within 300 seconds on 853/855 formulas (without/with unique implication point exploitation) while for the others possibly only a subset of $\mathrm{lookahead}(\phi, \emptyset)$ was computed. The results in Fig. 3(a) show that on a number of problems it is definitely worthwhile to find and use the unique implication points as a 2–4 times speedup can be obtained. Furthermore, the results also show that the computation of unique implication points is not computationally too expensive when compared to unit propagation performed during the lookahead computation; thus when the unique implication point technique does not help, it does not slow down the lookahead computation, either.

4.4 Lookahead as a Preprocessing Technique

One may now think that applying lookahead as a preprocessing step might make the formula significantly easier to solve. That is, given a formula ϕ, if we compute the

lookahead $\mathrm{lookahead}(\phi, \emptyset)$ and add the unit clauses found to ϕ, then the resulting formula $\phi \wedge \mathrm{lookahead}(\phi, \emptyset)$ would be much easier to solve than ϕ. Unfortunately, this is not the case, at least when considering the 573 formulas in the "crafted" and "application" categories of the SAT-COMP 2009 solver competition (see `http://www.satcompetition.org/`). Figure 3(b) shows a digest of the run times of MINISAT [20] (version 2 without the SatElite preprocessor) on the original 573 formulas as well as on the corresponding lookahead-simplified formulas. The lookahead computation time, which was limited to 300 seconds, is not included in the "lookahead-simplified" plot; of the 573 formulas, the lookahead computation finished within 300 seconds on 516 formulas. The results clearly show that performing lookahead-based formula simplification does not give substantial run time benefits on these instances. And if the lookahead computation time was included in the "lookahead-simplified" plot, the result would slightly worse than the "original" plot.

We have included this negative result as it shows that the positive results we obtain later in this paper, when applying lookahead for partitioning SAT formulas, are not caused by the fact that simply applying lookahead to the original formula would have produced as good results.

5 Partitioning with Scattering

The partition function presented in Sect. 4 can be contrasted to the *scattering* approach presented in [13]. The approach is a generalization of DPLL-based partitioning, in a sense that not only literals but also longer clauses are conjoined with the original instance. More formally, given an input formula ϕ, the derived instances are of the form

$$\phi_i = \begin{cases} \phi \wedge c_1 & \text{when } i = 1, \\ \phi \wedge \neg c_1 \wedge \ldots \wedge \neg c_{i-1} \wedge c_i & \text{when } 1 < i < n, \text{ and} \\ \phi \wedge \neg c_1 \wedge \ldots \wedge \neg c_{n-1} & \text{when } i = n, \end{cases}$$

where $c_i = x_1 \wedge \ldots \wedge x_{d_i}$ conjoins literals and $\neg c_i = \neg x_1 \vee \ldots \vee \neg x_{d_i}$ is a clause.

Fig. 4(b) presents the *cdcl-partition* algorithm for scattering. The algorithm takes as input an instance ϕ and a sequence $d_1, \ldots, d_n$ which determines the number of literals in the clauses $c_1, \ldots, c_{n-1}$. The algorithm is based on the conflict-driven clause learning (CDCL) solver search, altered so that once a sufficient amount of decision literals are chosen, the literals are used to produced a partition, the solver backtracks to the topmost decision level, inserts a clause consisting of the negation of the decision literals, and continues the partitioning on the altered instance on lines 10–14.

We extend the algorithm described in [13] by a lookahead-type call similar to the one described in Sect. 4 by computing not only the unit propagation $\mathrm{up}(\phi, \tau)$ but also $\mathrm{lookahead}(\phi, \tau)$, effectively implementing a CDCL solver with lookahead. In contrast to the *lascatter-partition*() algorithm in Fig. 4(a), we use the first unique implication point (1-UIP), as in most CDCL solvers, to guide the backtracking.

There are several possibilities for determining the values for d_i. In this work we use the approach presented in [13], where d_i minimizes the "error function" $Err(d_i) = |\ 2^{-d_i} - \frac{1}{n-i+1}|$. The error function can be motivated as follows. Given an instance ϕ with search space size $||\phi||$ and a number n, if the search space of ϕ is to be divided

latree-partition(ϕ,τ,i,m):
1 let $\tau' := \mathrm{up}(\phi,\tau)$
2 let $\langle \tau'', h\rangle$:= *lookahead*(ϕ,τ')
3 if τ'' is inconsistent
4 return
5 if τ'' satisfies ϕ or $i = m$
6 Output the derived formula $\phi \wedge \tau''$
7 return
8 let x be h-best and unassigned by τ''
9 call *latree-partition*$(\phi,\tau'' \cup \{x\},i+1,m)$
10 call *latree-partition*$(\phi,\tau'' \cup \{\neg x\},i+1,m)$

(a) DPLL-based partitioning with lookahead

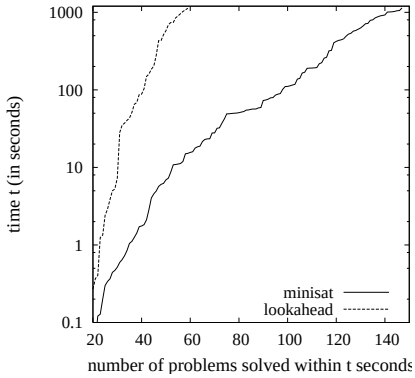
cdcl-partition$(\phi, d_1, \ldots, d_n)$:
1 let $\tau := \emptyset$, $dl := 0$, and $i := 1$
2 while true
3 let $\tau' := \mathrm{up}(\phi,\tau)$
4 let $\langle \tau'', h\rangle$:= *lookahead*(ϕ,τ')
5 if τ'' is inconsistent
6 let dl := *analyze*$()$
7 if $dl = -1$ return done
8 else *backtrack*(dl)
9 else if $dl = d_i$
10 Output the derived formula $\phi \wedge \tau''$
11 if $i = n$ return done
12 let $\phi := \phi \wedge (\neg x_1 \vee \ldots \vee \neg x_{d_i})$
13 let $i := i + 1$
14 *backtrack*(0)
15 else
16 let $dl := dl + 1$
17 let x_{dl} be h-best and unassigned by τ''
18 if $x_{dl} =$ None return sat
19 let $\tau := \tau \cup \tau'' \cup \{x_{dl}\}$

(b) Scattering-based lookahead partitioning

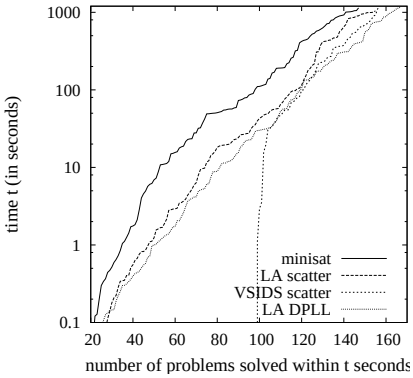
Fig. 4. Algorithms for partitioning

iteratively into n equally sized, non-overlapping partitions $\phi_1, \ldots, \phi_n$ of size $||\phi||/n$, then a direct calculation shows $\frac{||\phi||}{n} = \left(||\phi|| - (i-1)\frac{||\phi||}{n}\right) r_i$, and $r_i = 1/(n - (i - 1))$. We approximate the size of the search space with the worst case behavior, that is, $||\phi|| = 2^{\mathrm{vars}(\phi)}$, and that $||\phi \wedge x_1 \wedge \ldots \wedge x_{d_i}|| = 2^{\mathrm{vars}(\phi)-d_i}$, where $x_1, \ldots, x_{d_i}$ are literals. Hence, the closest approximation of r_i is obtained by minimizing the error function $Err(d_i) = |2^{-d_i} - r_i|$.

The algorithm proves an instance unsatisfiable on line 7, if it has not altered the instance on line 12. Hence, the algorithm can be used for sequential SAT solving by disabling the partition construction on lines 10–14. We implemented the algorithm on top of MINISAT version 1.14, and compare its performance against the unaltered MINISAT v1.14 implementation on a randomly selected set of instances from the crafted and applications categories of the SAT-COMP 2009 solver competition in Fig. 5 (a). The timeout limit was 1200 seconds, and the implementation uses the lookahead balancing heuristic in branching. The results show that the original version of MINISAT solves more than twice as many instances as the lookahead implementation, which is in accordance with the results in [21]. Perhaps surprisingly, in the following experiments we will see that lookahead is useful in producing partitions.



(a) Comparing lookahead based CDCL solver and standard CDCL solver



(b) Comparing plain MINISAT, vsids scattering, lookahead scattering and DPLL-based lookahead partitioning

Fig. 5. Some experimental results on solving and partitioning

6 Experiments

This section reports the results of the experiments conducted for comparing the partition functions discussed in this work. We use the SAT-COMP 2009 benchmark instances for the evaluation. Our first experiment compares the partition functions when using the plain partitioning approach, where the partition function is applied once and the resulting derived instances are then solved in parallel. We then continue the experiments on the partition functions using the partition tree approach, first applying it to instances that were not solved in SAT-COMP 2009, and then comparing the solving times of the partition tree approach to run times in SAT-COMP 2009. Finally, we study examples where MINISAT performs significantly worse than some sequential SAT solver in SAT-COMP 2009, and show that these instances are also difficult for the partition tree approach.

6.1 The Plain Partitioning Approach

This experiment compares the three partition functions by partitioning an instance once and then solving the resulting instances. The 324 instances were randomly selected from the 573 instances in the *crafted* and *application* categories of the SAT-COMP 2009. The two lookahead-based partition functions described in Sections 4 (DPLL-based partitioning with lookahead) and 5 (scattering with lookahead) are compared against a VSIDS-based partition function described in [13], which was completely reimplemented using MINISAT v1.14 as the basis. As described in [13], the VSIDS-scatter function is obtained from the *cdcl-partition* algorithm in Fig. 4(b) by replacing the call to lookahead by unit propagation. Furthermore, in the re-implemented version the solver runs a normal CDCL search both initially and after deriving each instance.

The results are reported in Fig. 5 (b). Each instance was divided into eight partitions using the scattering-based lookahead (*LA scatter*) and VSIDS heuristic with

cdcl-partition (*VSIDS scatter*), and the DPLL-based lookahead partition function with *latree-partition* (*LA DPLL*). The time-out for each partition function was 300 seconds. The resulting partitions were solved using a standard SAT solver (MINISAT v1.14) and 1200 s time-out for each partition. The figure shows the minimum run time of a satisfiable partition, and the maximum run time of the unsatisfiable partitions of an unsatisfiable instance. For comparison, the figure also provides the run time of the standard SAT solver on the original instances with no partitioning (*minisat*).

The approach using DPLL-based lookahead partitioning solves most instances from the benchmark set. This is a surprising result, since in principle, the greater freedom of the scattering approach in choosing the literals should increase the solving performance. The relatively straightforward scattering-based partition function with VSIDS heuristic also outperforms the lookahead-based scattering, although the difference is small. The figure shows that the VSIDS implementation solves instances in zero time, as some instances were solved with this method already in the partition phase.

6.2 The Partition Tree Approach

The experiments in this section study the partition tree approach and efficiency of the partition functions using NorduGrid as the distributed computing environment. The partition tree is formed breadth first. The time-out for the whole approach is 6 hours and it uses 64 CPUs in parallel. The partition functions have a time-out of 300 seconds, and the time-outs for the executions in the grid vary from 60 to 90 minutes[2].

We attempted the solving of 38 instances of the *applications* category that were not solved in the SAT-COMP 2009 competition. In total there are approximately 60 such instances. Table 1 lists the run times for the instances we could solve. The time-out in the competition was 10000 seconds, which is approximately half of the partition tree approach time-out (6h) in our experiment, but almost twice the maximum execution time limit of 90 minutes. The reported values are wall clock times in seconds, and time-outs (when no solution was found in 6 hours) are marked with a dash. The *Type* column reports the instance satisfiable (SAT) or unsatisfiable (UNSAT) unless no approach could solve the instance due to a time-out. The lowest solving time is typeset in boldface.

Using the partition tree approach we solved 11 new instances that were not solved in SAT-COMP 2009, although in some cases the run time was higher than the competition time-out. The results show that DPLL-based partitioning with lookahead heuristic performs usually best, having the lowest solving time in approximately half of the cases. This supports the conclusion in the previous experiment on partition functions. However, we see that the VSIDS partitioning solved two more instances compared to the 8 instances solved by the DPLL-based lookahead approach. The column *SD 64* reports the run time of the simple distribution approach when using 64 CPUs, that is, the minimum run time of 64 independent MINISAT v1.14 solvers if it was less than 6 hours. Based on these results the partition approaches perform usually much better than SD.

To study the performance of the partition tree approach, we also attempted to solve all 15 instances from the *crafted* and *applications* categories of SAT-COMP 2009 that

[2] The time limit is not constant to avoid a decrease in performance caused by simultaneous finishing of a large number of jobs.

Table 1. Wall-clock solving times in seconds for instances from the *applications* category not solved in SAT-COMP 2009

Name	Type	LA DPLL	LA scatter	VSIDS scatter	SD 64
AProVE07-25	UNSAT	**8992.60**	9176.91	11347.42	—
dated-5-19-u	UNSAT	16557.82	20155.96	**4124.62**	—
eq.atree.braun.12.unsat	UNSAT	3157.19	**2357.55**	3006.19	20797.60
eq.atree.braun.13.unsat	UNSAT	**7117.39**	8504.50	8158.85	—
gss-24-s100	SAT	1977.19	3449.55	2271.24	**968.23**
gss-26-s100	SAT	10844.22	—	**6057.80**	—
gss-32-s100	SAT	—	**16412.40**	—	—
gus-md5-14	UNSAT	**14779.03**	16264.37	16098.04	—
ndhf_xits_09_UNSAT	UNSAT	—	—	**14793.78**	—
rpoc_xits_09_UNSAT	UNSAT	—	—	**12388.32**	—
total-10-17-u	UNSAT	**4431.21**	7198.23	5099.73	—

were solved by at least one solver but the best run time was over 1 hour. The results are reported in Table 2. For comparison the table also reports the results for running 64 independent MINISAT solvers with 6 hour timeout and taking the minimum over the solving times (the *SD 64* column), and the competition results (*SAT-COMP*).

The results show again that the DPLL-based partition function with lookahead is superior to the other partition functions, forming the fastest approach in almost half of the instances. The approach is in all cases better than the approach using independent MINISAT solvers. In this experiment, all three approaches are able to solve instances not solved by the other approaches. In two cases the SD 64 approach is better than either the VSIDS or lookahead-based scatter approach. The average run time of MINISAT seems to be for both instances much higher than the minimum reported in the *SD 64* column. The table also reports a number of instances where the result of at least one solver in SAT-COMP 2009 was better than any of the partition tree results. In these cases also the MINISAT solver could not find a solution before the timeout. We could experimentally find some more instances which turned out to be relatively easy for some SAT solver in the competition but extremely challenging for our approach which is based on MINISAT. These instances are presented in the bottom part of Table 2. The partition tree approach treats SAT solvers as black boxes and it would be interesting to see how the partition tree approach performs if the solver is changed.

7 Conclusions

This work studies central concepts in distributed solving of propositional satisfiability problem (SAT) instances. We identify challenges of the commonly used strategies, simple distribution and plain partitioning [12], and show that the *partition tree approach* combines the two strategies by avoiding some of their inherent problems.

We develop three *partition functions* and use the partition tree approach to study their efficiency. The experimental results show that the partition tree approach performs well with these functions, being much more powerful in practice than the initial results

Table 2. Wall-clock times in seconds for the partition tree approaches, simple distribution and the SAT-COMP results

Name	Type	LA DPLL	LA scatter	VSIDS scatter	SD 64	SAT-COMP
9dlx_vliw_at_b_iq7	UNSAT	—	—	—	—	**6836.20**
AProVE07-01	UNSAT	1465.22	**1322.04**	2451.36	20230.30	6816.94
dated-5-13-u	UNSAT	**3881.60**	4745.52	4563.15	—	8005.27
gss-22-s100	SAT	**830.77**	1151.13	4246.25	2280.82	4326.83
gss-27-s100	SAT	—	—	9156.71	—	**7132.69**
gus-md5-11	UNSAT	**1190.28**	2077.99	2092.54	5057.39	4518.06
maxor128	UNSAT	—	—	—	—	**7131.52**
maxxor064	UNSAT	—	—	—	—	**5162.75**
minandmaxor128	UNSAT	—	—	—	—	**5143.44**
mod4block_3vars_7gates	UNSAT	**1740.17**	1755.47	2326.02	—	4109.89
new-difficult-26-243-24-70	SAT	**3260.86**	8887.61	5087.98	3311.62	4440.72
rbcl_xits_08_UNSAT	UNSAT	4557.86	**2390.50**	3695.97	—	3892.92
sgen1-unsat-109-100	UNSAT	**1363.14**	3000.48	4196.36	14675.60	4045.49
UTI-20-10p1	SAT	—	7097.74	—	—	**6289.06**
UR-20-10p1	SAT	**4463.24**	—	—	—	8766.23
Challenge instances for MINISAT						
countbitsarray02_32	UNSAT	1746.29	3003.50	997.84	2504.93	**834.519**
vange-col-abb313GPIA-9-c	SAT	—	—	—	—	**445.09**
velev-pipe-uns-1.0-8	UNSAT	—	—	—	—	**307.48**
vmpc_34	SAT	12452.59	1350.17	1479.62	2796.19	**35.347**
simon-s02b-k2f-gr-rcs-w8	UNSAT	3816.20	3106.70	14756.10	—	**6.40**

in [13] suggest. The partition tree approach solves several instances which were not solved in the SAT-COMP 2009 solver competition. While we could find other instances that were solved in the competition and we failed to solve, our results suggest that also the simple distribution approach fails to solve these instances. We conclude that the partition tree approach genuinely improves the efficiency of the solver, and similar results cannot be achieved simply by running randomized SAT solvers in parallel. The results raise an interesting future question on how other solvers would benefit from the approach.

References

1. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Strategies for solving SAT in Grids by randomized search. In: Proc. AISC'08. Volume 5144 of LNAI., Springer (2008) 125–140
2. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4) (1993) 173–180
3. Luby, M., Ertel, W.: Optimal parallelization of Las Vegas algorithms. In: Proc. STACS'94. Volume 775 of LNCS., Springer (1994) 463–474
4. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(5296) (1997) 51–54

5. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1–2) (2001) 43–62
6. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Incorporating clause learning in grid-based randomized SAT solving. Journal on Satisfiability, Boolean Modeling and Computation **6** (2009) 223–244
7. Speckenmeyer, E., Monien, B., Vornberger, O.: Superlinear speedup for parallel backtracking. In: Proc. Supercomputing '87. Volume 297 of LNCS., Springer (1988) 985–993
8. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver: Efficient workload balancing. Annals of Mathematics and Artificial Intelligence **17**(4–3) (1996) 381–400
9. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21**(4) (1996) 543–560
10. Jurkowiak, B., Li, C., Utard, G.: A parallelization scheme based on work stealing for a class of SAT solvers. Journal of Automated Reasoning **34**(1) (2005) 73–101
11. Michel, L., See, A., van Hentenryck, P.: Parallelizing constraint programs transparently. In: Proc. CP'07. Volume 4741 of LNCS., Springer (2007) 514–528
12. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning the search space of a randomized search. In: Proc. AI*IA'09. Volume 5883 of LNAI., Springer (2009) 243–252
13. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: Proc. SAT'06. Volume 4121 of LNCS., Springer (2006) 430–435
14. Heule, M., van Maaren, H.: Look-ahead based SAT solvers. In: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009) 155–184
15. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing **29**(7) (2003) 969–994
16. Le Berre, D.: Exploiting the real power of unit propagation lookahead. In: Proc. SAT 2001. Volume 9 of Electronic Notes in Discrete Mathematics., Elsevier (2001) 59–80
17. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
18. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
19. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: Proc. ICCAD'01, ACM (2001) 279–285
20. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT'03. Volume 2919 of LNCS., Springer (2004) 502–518
21. Giunchiglia, E., Maratea, M., Tacchella, A.: (In)Effectiveness of look-ahead techniques in a modern SAT solver. In: Proc. CP'03. Volume 2833 of LNCS., Springer (2003) 842–846