Journal on Satisfiability, Boolean Modeling and Computation 1 (2004) 25-30

# March_eq: Implementing Efficiency and Additional Reasoning into a Lookahead SAT-Solver

**Marijn J.H. Heule*** m.j.h.heule@ewi.tudelft.nl
**Mark Dufour** m.dufour@student.tudelft.nl
**Hans van Maaren** h.vanmaaren@ewi.tudelft.nl
*Department of Software Technology,*
*Faculty of Electrical Engineering, Mathematics and Computer Sciences,*
*Delft University of Technology*

**Joris E. van Zwieten** zwieten@ch.tudelft.nl
*Department of Mediamatics,*
*Faculty of Electrical Engineering, Mathematics and Computer Sciences,*
*Delft University of Technology*

## Abstract

In this paper, several techniques are discussed that make the lookahead architecture for satisfiability (SAT) solvers more competing. Our contribution consists of reduction of the computational costs to perform lookahead and the cheap integration of both equivalence reasoning and local learning. Most proposed techniques are illustrated with experimental results of their implementation in our solver march_eq.

KEYWORDS: *SAT-solver, lookahead, equivalence reasoning, local learning*

*Submitted September 2004; revised October 2004; published November 2004*

## 1. Introduction

Lookahead SAT solvers usually consist of a simple DPLL algorithm [4] and a more sophisticated *lookahead procedure* to determine an effective branch variable. The lookahead procedure measures the effectiveness of variables by performing *lookahead* on a set of variables and evaluates the reduction of the formula. We refer to the lookahead on variable $x$ as the Iterative Unit Propagation (IUP) on a formula with additional unit clause $x$ (in short IUP($\mathcal{F} \cup \{x\}$)). The effectiveness of a variable $x_i$ is obtained using a lookahead evaluation function (in short DIFF), which evaluates the difference between $\mathcal{F}$ and and the reduced formula after IUP($\mathcal{F} \cup \{x_i\}$). A widely used DIFF counts the newly created binary clauses.

Besides the selection of a branch variable, the lookahead procedure may detect *failed literals*: If the lookahead on $\neg x$ results in a conflict, $x$ is forced to true. Detection of failed literals could result in a reasonable reduction of the DPLL-tree.

Last decade, several enhancements have been implemented in making the lookahead SAT solvers more powerful. In satz by Li [6] a heuristics PROP$_z$ is used, which restricts the number of variables that enter the lookahead procedure. Especially on random instances applying these heuristics results in a clear performance gain. However, the use of this heuristics is

* Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.306.

not clear from a general viewpoint. Experiments with our pre-selection heuristics show that different benchmark families require different numbers of variables entering the lookahead phase to perform optimally.

Since much reasoning is already performed during each node of the DPLL-tree, it is relatively cheap to extend the lookahead with (some) additional reasoning. For instance: Integration of equivalence reasoning in satz - implemented in eqsatz [7] - made it possible to solve various crafted and real-world problems which were beyond the reach of existing techniques. However, the performance may drop significantly on some problems, due to the integrated equivalence reasoning. Our alternative variant of equivalence reasoning extends the set of problems which favors from its integration and practically removes the disadvantages.

Another form of additional reasoning is implemented in the OKsolver[1.] [5] *local learning*. When performing lookahead on $x$, any unit clause $y_i$ found means that binary clause $\neg x \vee y_i$ is logically implied by the formula, and thus can be "learned", i.e. added to the current formula. Like equivalence reasoning, addition of these local learned resolvents could both increase and decrease the performance (depending on the formula). We propose a partial addition of these resolvents that will practically always result in a speed-up.

Generally, lookahead Sat solvers are effective on relatively small, hard formulas. Due to the high computational costs of the lookahead procedure, elaborate problems are often solved more efficiently by other techniques. Reduction of these costs is essential to make lookahead techniques more competing on a wider range of benchmarks problems. In this paper, we suggest (1) several techniques to reduce these costs and (2) a cheap integration of additional reasoning. This way, benchmarks that do not profit form the additional reasoning will not be significantly harder to solve.

Most topics discussed in this paper are illustrated with experimental results showing the performance gains by our proposed techniques. Benchmarks range from random 3-Sat near the threshold [1], to bounded model checking (`longmult` [3], `zarpas` [2]) factoring problems (`pylala braun` [9]), `quasigroup` [10] and crafted(`stanion hwb` [2]) problems. Only unsatisfiable instances were selected to provide a more stable overview.

All techniques are compared with a reference variant of march_eq which is slightly more optimised than march_eq_100; the solver that won two categories of the Sat 2004 competition [8]. Besides the recent optimasations this variant has uses exactly the same techniques as the winning variant: full (100%) look-ahead, addition of all constraint resolvents, tree-based look-ahead, equivalence reasoning, and removal of inactive clauses. All these techniques are discussed below.

## 2. Hierarchy Example

[Always write something at the beginning of a section, before starting new subsections.]

### 2.1 First Sub-section Level

#### 2.1.1 Second Sub-section Level

[section ommitted]

1. Version 1.2

## 3. Tree-based Lookahead

As march_eq performs an expensive, iterative look-ahead procedure in each node of the decision tree, we have put a substantial amount of effort into optimizing the respective part of the program. The structure of our look-ahead procedure is based on the observation that different literals that we plan to look-ahead on, often entail certain shared implications, and that we can form 'sharing' trees out of these relations, which may be used to reduce the amount of times these implications have to be propagated during look-ahead.

To give an example of how this works, suppose that two look-ahead literals share a certain implication. In this simple case, we could propagate the shared implication first, followed by a propagation of one of the look-ahead literals, backtracking the latter, then propagating the other look-ahead literal and only in the end backtracking to the initial state. This way, the shared implication has been propagated only once. In graphical form:

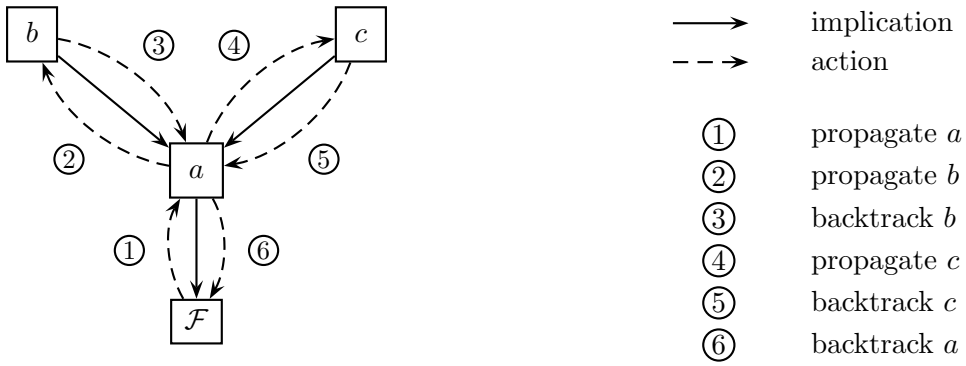


**Figure 1.** Graphical form of an implication tree with corresponding actions.

The implications among $a$, $b$ and $c$ form a small tree. Some thought reveals that the above process, when applied recursively, works for arbitrary trees. Based on this, our solver extracts, prior to look-ahead, trees from the implications known to exist among the literals selected for look-ahead, such that each literal occurs in exactly one tree. The look-ahead procedure is improved by recursively visiting these trees. Of course, the more dense the implication graph, the more possibilities there are for forming trees, so local learning will in many cases be an important catalyst for the effectiveness of this method.

Unfortunately, there are many ways of extracting trees from a graph, such that each vertex occurs in exactly one tree. Large trees are obviously desirable, as they imply more sharing, as does having literals with the most impact on the formula near the root of a tree. To this end, have developed a simple heuristic. More involved methods would probably give better results, although optimality in this area could easily mean we are solving NP-complete problems again. We consider this an interesting direction for future research.

Our heuristic requires a list of predictions to be available, of the relative amount of propagations that each look-ahead literal implies, to be able to construct trees that share as much of these as possible. In the case of march_eq, the pre-selection heuristic provides us with such a list.

The heuristic now travels this list once, in the order of decreasing prediction, while along the way constructing trees out of the corresponding literals. It does this by determining for

each literal, if available, one other look-ahead literal that will become its parent in some tree. When a literal is assigned a parent, this relationship remains fixed. At the outset, as much trees are created as there are look-ahead literals, each consisting of just the corresponding literal.

More specifically, for each literal that it encounters, the heuristic checks whether this literal is implied by any other look-ahead literals, that are the root of some tree. If any, these are made child nodes of the node corresponding to the implied literal. If not already encountered, these child nodes are now recursively checked in the same manner. At the same time, we remove the corresponding elements from the list, so that each literal will be checked exactly once, and will receive a position within exactly one tree.

As an example, we show the process for a small set of look-ahead literals, that share some implications. A gray box denotes the current position:

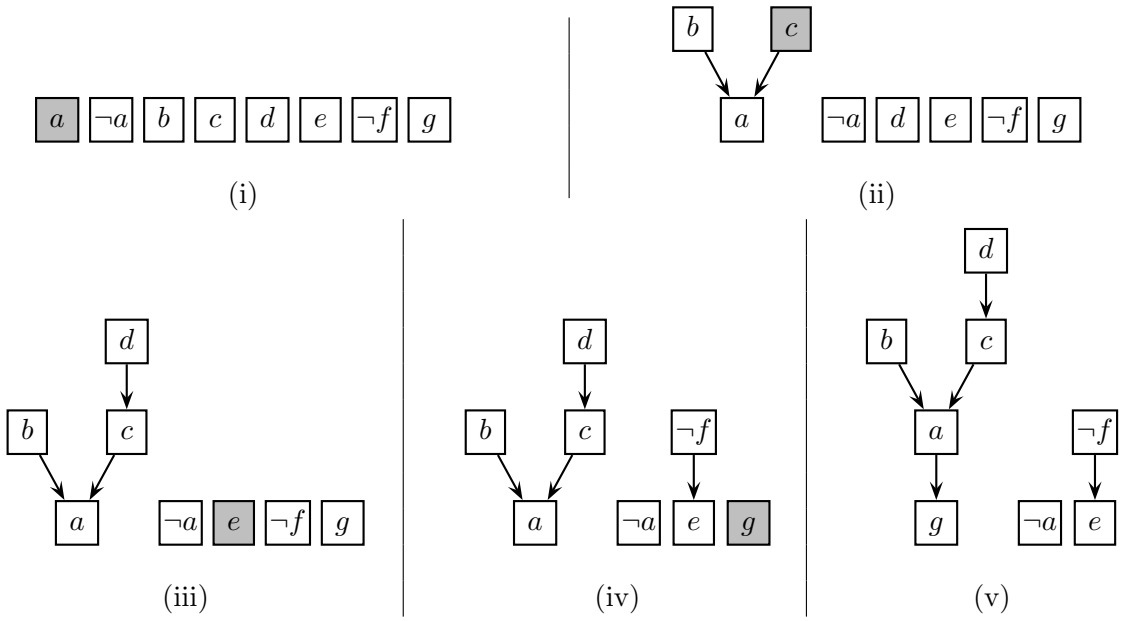


**Figure 2.** Five steps of building implication trees.

Because of the order in which the list is traveled, literals which have received higher predictions are made into parent nodes as early as possible. This is important, because of the fact that it is often possible to extract many different trees from an implication graph, and because every literal should occur in exactly one tree.

Having implication trees available opens up several possibilities of going beyond resolution. One such possibility is to detect implied literals. Whenever a node has descendants that are complementary, clearly the corresponding literal is implied. By approximation, we detect this for the most important literals, as these should have ended up near the roots of larger trees, by the above heuristic. For solvers unable to deduce such implications by themselves, a simple, linear-time algorithm that scans the trees may be used.

Some intriguing ideas for further research have occurred to us during the course of developing our tree-based lookahead procedure, but which, due to time constraints we

have not been able to pursue. One possible extension would be to add variables, that both positively and negatively imply some look-ahead literal, as full-fledged look-ahead variables. This way, we may discover important, but previously undetected variables to perform look-ahead, and possibly branch upon. Because of the inherent sharing, the overhead of doing this will be smaller than without a tree-based lookahead.

**Table 1.** Performance of march_eq on several benchmarks with and without the use of tree-based lookahead.

| | **normal lookahead** | | **tree-based lookahead** | | |
|---|---|---|---|---|---|
| **Benchmarks** | time($s$) | treesize | time($s$) | treesize | speed-up |
| unsat250 (100) | 1.24 | 3428.5 | 1.45 | 3391.7 | -16.94 % |
| unsat350 (100) | 40.57 | 74501.7 | 48.78 | 73357.2 | -20.24 % |
| hwb-n20-01 | 29.55 | 184363 | 23.65 | 183553 | 19.97 % |
| hwb-n20-02 | 40.93 | 227237 | 30.91 | 222251 | 24.48 % |
| hwb-n20-03 | 25.88 | 155702 | 21.70 | 163984 | 16.15 % |
| longmult8 | 332.64 | 7918 | 90.80 | 8149 | 72.70 % |
| longmult10 | 1014.09 | 10861 | 226.31 | 11597 | 77.68 % |
| longmult12 | 727.01 | 4654 | 176.85 | 5426 | 75.67 % |
| pb-unsat-35-4-03 | 1084.08 | 19093 | 662.93 | 19517 | 38.85 % |
| pb-unsat-35-4-04 | 1098.50 | 19493 | 659.04 | 19364 | 40.01 % |
| quasigroup3-9 | 8.85 | 1508 | 7.97 | 1495 | 9.94 % |
| quasigroup6-12 | 78.75 | 1339 | 58.05 | 1311 | 26.29 % |
| quasigroup7-12 | 13.03 | 268 | 10.03 | 256 | 23.02 % |
| rule14_1_15dat | 25.62 | 0 | 20.70 | 0 | 19.20 % |
| rule14_1_30dat | 192.30 | 0 | 186.27 | 0 | 3.14 % |

Also, once trees have been created, we could include non-lookahead literals in the sharing, as well as in the checking of implied literals. As for the first, suppose that literals $a$ and $b$ imply some literal $c$. In this case we could share not just the propagation of $c$, but also that of any other shared implications of $a$ and $b$. Sharing among tree roots could be exploited in the same manner, with the difference that in the case of many shared implications, we would have to determine which trees could best share implications with which other trees. More in general, it might be a better idea to look more in detail at possibilities of sharing, than the simple heuristic we have used.

## 4. Theory

**Theorem 1.** *Unless P equals NP, ...*

**Proof:** Obvious. □

[section ommitted]

## Acknowledgments

The authors wish to thank Arjen van Lith for designing the JSAT logo. We also thank our anonymous reviewers for their comments.

## Appendix A. March_eq Source Code

[section ommitted]

## References

[1] D. Mitchell, B. Selmon and H. Levesque, Hard and easy distributions of SAT problems. *Proceedings of AIII-1992* (1992), 459–465.

[2] D. Le Berre and L. Simon, The essentials of the SAT'03 Competition. Springer-Verlag, *Lecture Notes in Comput. Sci.* **2919** (2004), 452–467.

[3] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs. in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, *Lecture Notes in Comput. Sci.* **1579** (1999), 193–207.

[4] M. Davis, G. Logemann, and D. Loveland, A machine program for theorem proving. *Communications of the ACM* **5** (1962), 394–397.

[5] O. Kullmann, Investigating the behaviour of a SAT solver on random formulas. Submitted to *Annals of Mathematics and Artificial Intelligence* (2002).

[6] C.M. Li and Anbulagan, Look-Ahead versus Look-Back for Satisfiability Problems. Springer-Verlag, *Lecture Notes in Comput. Sci.* **1330** (1997), 342–356.

[7] C.M. Li, Equivalent literal propagation in the DLL procedure. *The Renesse issue on satisfiability* (2000). *Discrete Appl. Math.* **130** (2003), no. 2, 251–276.

[8] L. Simon, Sat'04 competition homepage. http://www.lri.fr/~simon/contest/results/.

[9] L. Simon, D. Le Berre, and E. Hirsch, The SAT 2002 competition. Accepted for publication in *Annals of Mathematics and Artificial Intelligence* (*AMAI*) **43** (2005), 343–378.

[10] H. Zhang and M.E. Stickel, Implementing the Davis-Putnam Method. *SAT2000* (2000), 309–326.