March_eq
Implementing Additional Reasoning into an Efficient Look-Ahead SAT Solver

Marijn Heule*, Mark Dufour,
Joris van Zwieten and Hans van Maaren

Department of Information Systems and Algorithms,
Faculty of Electrical Engineering,
Mathematics and Computer Sciences,
Delft University of Technology
marijn@heule.nl, m.dufour@student.tudelft.nl,
zwieten@ch.tudelft.nl, h.vanmaaren@its.tudelft.nl

Abstract. This paper discusses several techniques to make the look-ahead architecture for satisfiability (SAT) solvers more competitive. Our contribution consists of reduction of the computational costs to perform look-ahead and a 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.

1 Introduction

Look-ahead SAT solvers usually consist of a simple DPLL algorithm [5] and a more sophisticated *look-ahead procedure* to determine an effective branch variable. The look-ahead procedure measures the effectiveness of variables by performing *look-ahead* on a set of variables and evaluating the reduction of the formula. We refer to the look-ahead on literal x as the Iterative Unit Propagation (IUP) on the union of a formula with the unit clause x (in short IUP($\mathcal{F} \cup \{x\}$)). The effectiveness of a variable x_i is obtained using a look-ahead evaluation function (in short DIFF), which evaluates the differences between $\mathcal{F}$ and the reduced formula after IUP($\mathcal{F} \cup \{x_i\}$) and IUP($\mathcal{F} \cup \{\neg x_i\}$). A widely used DIFF counts the newly created binary clauses.

Besides the selection of a branch variable, the look-ahead procedure may detect *failed literals*: if the look-ahead on $\neg x$ results in a conflict, x is forced to true. Detection of failed literals can result in a substantial reduction of the DPLL-tree.

During the last decade, several enhancements have been proposed to make look-ahead SAT solvers more powerful. In satz by Li [9] pre-selection heuristics PROP$_z$ are used, which restrict the number of variables that enter the look-ahead

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

H.H. Hoos and D.G. Mitchell (Eds.): SAT 2004, LNCS 3542, pp. 345–359, 2005.

procedure. Especially on random instances the application of these heuristics results in a clear performance gain. However, the use of these heuristics is not clear from a general viewpoint. Experiments with our pre-selection heuristics show that different benchmark families require different numbers of variables entering the look-ahead phase to perform optimally.

Since much reasoning is already performed at each node of the DPLL-tree, it is relatively cheap to extend the look-ahead with (some) additional reasoning. For instance: integration of equivalence reasoning in satz - implemented in eqsatz [10] - 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 variant of equivalence reasoning extends the set of problems which benefit from its integration and aims to remove the disadvantages.

Another form of additional reasoning is implemented in the OKsolver [1] [8]: *local learning*. When performing look-ahead on x, any unit clause y_i that is found means that the binary clause $\neg x \vee y_i$ is implied by the formula, and can be "learned", i.e. added to the current formula. As with 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 which results in a speed-up practically everywhere.

Generally, look-ahead SAT solvers are effective on relatively small, hard formulas. Le Berre proposes [2] a wide range of enhancements of the look-ahead procedure. Most of them are implemented in march_eq. Due to the high computational costs of the an enhanced look-ahead procedure, elaborate problems are often solved more efficiently by other techniques. Reducing these costs is essential for making look-ahead techniques more competitive 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. Due to the latter, benchmarks that do not profit from additional reasoning will not take significantly more time to solve.

Most topics discussed in this paper are illustrated with experimental results showing the performance gains by our proposed techniques. The benchmarks range from `uniform random` 3-SAT near the threshold [1], to bounded model checking (`longmult` [4], `zarpas` [3]), factoring problems (`pyhala braun` [12]) and crafted problems (`stanion/hwb` [3], `quasigroup` [14]). Only unsatisfiable instances were selected to provide a more stable overview. Comparison of the performance of march_eq with performances of state-of-the-art solvers is presented in [7], which appears in the same volume.

All techniques have been implemented into a reference variant of march_eq, which is essentially a slightly optimised version of march_eq_100, the solver that won two categories of the SAT 2004 competition [11]. This variant 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.

[1] Version 1.2 at http://cs-svr1.swan.ac.uk/∼csoliver/OKsolver.html

2 Translation to 3-SAT

The translation of the input formula to 3-SAT stems from an early version of march_eq, in which it was essential to allow fast computation of the pre-selection heuristics. Translation is not required for the current pre-selection heuristics, yet it is still used, because it enables significant optimisation of the internal data structures.

The formula is pre-processed to reduce the amount of redundancy introduced by a straightforward 3-SAT translation. Each pair of literals that occurs more than once together in a clause in the formula is substituted by a single *dummy* variable, starting with the most frequently occurring pair. Three clauses are added for each dummy variable to make it logically equivalent to the disjunction of the pair of literals it substitutes. In the following example $\neg x_2 \vee x_4$ is the most occurring literal pair and is therefore replaced with the dummy variable d_1.

$$\begin{array}{r} x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5 \\ x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee x_6 \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_6 \\ \neg x_1 \vee \neg x_2 \vee x_4 \vee x_5 \vee x_6 \end{array} \Leftrightarrow \begin{array}{r} x_1 \vee d_1 \vee \neg x_3 \vee \neg x_5 \\ x_1 \vee d_1 \vee \neg x_3 \vee x_6 \\ \neg x_1 \vee d_1 \vee \neg x_3 \vee \neg x_6 \\ \neg x_1 \vee d_1 \vee x_5 \vee x_6 \end{array} \wedge \begin{array}{r} d_1 \vee x_2 \\ d_1 \vee \neg x_4 \\ \neg d_1 \vee \neg x_2 \vee x_4 \end{array}$$

It appears that to achieve good performance, binary clauses obtained from the *original* ternary clauses should be given more weight than binary clauses obtained from ternary clauses which were *generated by translation*. This is accomplished by an appropriate look-ahead evaluation function, such as the variant of DIFF proposed by Dubois *et al.* [6], which weighs all newly created binary clauses.

3 Time Stamps

March_eq uses a time stamp data structure, *TimeAssignments* (TA), which reduces backtracking during the look-ahead phase to a single integer addition: increasing the *CurrentTimeStamp* (CTS).

All the variables that are assigned during look-ahead on a literal x are stamped: if a variable is assigned the value true, it is stamped with the CTS; if it is assigned the value false, it is stamped with $CTS + 1$. Therefore, simply adding 2 to the CTS unassigns all assigned variables.

The actual truth value that is assigned to a variable is not stored in the data structure, but can be derived from the time stamp of the variable:

$$\mathrm{TA}[x] = \begin{cases} stamp < CTS & \text{unfixed} \\ stamp \geq CTS \textbf{ and } stamp \equiv 0 \pmod 2 & \text{true} \\ stamp \geq CTS \textbf{ and } stamp \equiv 1 \pmod 2 & \text{false} \end{cases}$$

Variables that have already been assigned before the start of the look-ahead phase, i.e. during the solving phase, have been stamped with the *MaximumTimeStamp* (MTS) or with $MTS + 1$. These variables can be unassigned by

stamping them with the value *zero*, which happens while backtracking during the solving phase (i.e. *not* during the look-ahead phase). The MTS equals the maximal even value of an (32-bit) integer. One has to ensure that the CTS is always smaller than the MTS. This will usually be the case and it can easily be checked at the start of each look-ahead.

4 Constraint Resolvents

As mentioned in the introduction, a binary resolvent could be added for every unary clause that is created during the propagation of a look-ahead literal - provided that the binary clause does not already exist. A special type of resolvent is created from a unary clause that was a ternary clause prior to the look-ahead. In this case we speak of *constraint resolvents*.

Constraint resolvents have the property that they cannot be found by a look-ahead on the complement of the unary clause. Adding these constraint resolvents results in a more vigorous detection of failed literals. An example:

First, consider only the original clauses of an example formula (figure 1 (a)). A look-ahead on $\neg r$, IUP($\mathcal{F} \cup \{\neg r\}$), results in the unary clause x. Therefore, one could add the resolvent $r \vee x$ to the formula. Since the unary clause x was originally a ternary clause (before the look-ahead on $\neg r$), this is a constraint resolvent. The unique property of constraints resolvents is that when they are added to the formula, look-ahead on the complement of the unary clause results in the complement of the look-ahead-literal. Without this addition this would not be the case. Applying this to the example: after addition of $r \vee x$ to the formula, IUP($\mathcal{F} \cup \{\neg x\}$) will result in unary clause r, while without this addition it will not.

IUP($\mathcal{F} \cup \{\neg r\}$) also results in unary clause $\neg t$. Therefore, resolvent $r \vee \neg t$ could be added to the formula. Since unary clause $\neg t$ was originally a binary clause, $r \vee \neg t$ is not a constraint resolvent. IUP($\mathcal{F} \cup \{t\}$) would result in unary clause r.

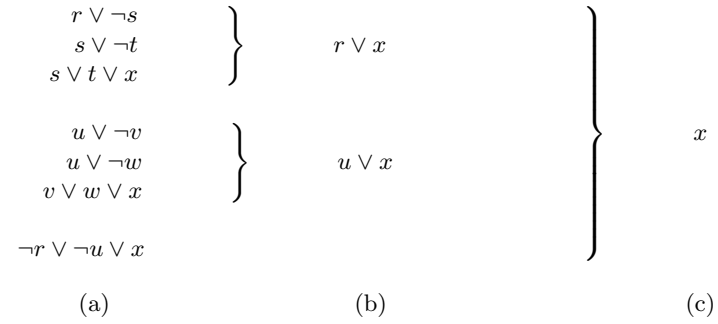


Fig. 1. Detection of a failed literal by adding constraint resolvents. (a) The original clauses, (b) constraint resolvents and (c) a forced literal

Table 1. Performance of march_eq on several benchmarks with three different settings of addition of resolvents during the look-ahead phase

	no resolvents		**all binary resolvents**		**all constraint resolvents**	
Benchmarks	time(s)	treesize	time(s)	treesize	time(s)	treesize
random_unsat_250 (100)	1.61	4059.1	1.51	3389.2	**1.45**	3391.7
random_unsat_350 (100)	55.41	89709.4	51.28	72721.1	**48.78**	73357.2
stanion/hwb-n20-01	31.52	282882	24.76	180408	**23.65**	183553
stanion/hwb-n20-02	41.32	345703	33.94	219915	**30.91**	222251
stanion/hwb-n20-03	30.54	280561	23.48	161687	**21.7**	163984
longmult8	139.13	15905	341.46	8054	**90.8**	8149
longmult10	504.92	330094	915.84	11877	**226.31**	11597
longmult12	836.78	41522	847.95	5273	**176.85**	5426
pyhala-unsat-35-4-03	781.19	29591	1379.33	19100	**662.93**	19517
pyhala-unsat-35-4-04	733.44	28312	1366.19	18901	**659.04**	19364
quasigroup3-9	11.67	2139	11.09	1543	**7.97**	1495
quasigroup6-12	117.49	3177	66.13	1362	**58.05**	1311
quasigroup7-12	14.47	346	11.06	248	**10.03**	256
zarpas/rule14_1_15dat	> 2000	-	46.59	0	**20.7**	0
zarpas/rule14_1_30dat	> 2000	-	> 2000	-	**186.27**	0

Constraint resolvent $u \vee x$ is detected during IUP($\mathcal{F} \cup \{\neg u\}$). After the addition of both constraint resolvents (figure 1 (b)), the look-ahead IUP($\mathcal{F} \cup \{\neg x\}$) results in a conflict, making $\neg x$ a failed literal and thus forces x. Obviously, IUP($\mathcal{F} \cup \{\neg x\}$) will not result in a conflict if the constraint resolvents $r \vee x$ and $u \vee x$ were not added both.

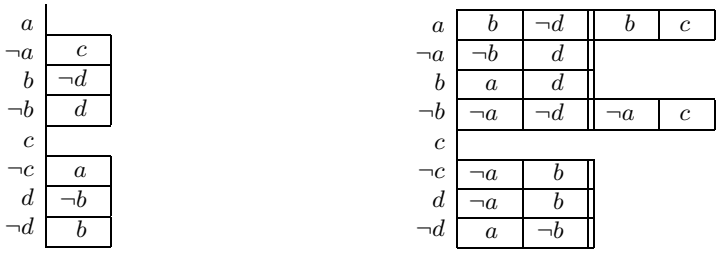
Table 1 shows the usefulness of the concept of constraint resolvents: in all our experiments, the addition of mere constraint resolvents outperformed a variant with full local learning (adding all binary resolvents). This could be explained by the above example: adding other resolvents than constraint resolvents will not increase the number of detected failed literals. These resolvents merely increase the computational costs. This explanation is supported by the data in the table: the tree-size of both variants is comparable.

When we look at zarpas/rule_14_1_30dat, it appears that only adding constraint resolvents is essential to solve this benchmark within 2000 seconds. The node-count of zero means that the instance is found unsatisfiable during the first execution of the look-ahead procedure.

5 Implication Arrays

Due to the 3-SAT translation the data structure of march_eq only needs to accommodate binary and ternary clauses. We will use the following formula as an example:

$$\mathcal{F}_{\text{example}} = (a \vee c) \wedge (\neg b \vee \neg d) \wedge (b \vee d) \wedge (a \vee \neg b \vee d) \wedge (\neg a \vee b \vee \neg d) \wedge (\neg a \vee b \vee c)$$



(i) (ii)

Fig. 2. The binary (i) and ternary (ii) implication arrays that represent the example formula $\mathcal{F}_{\text{example}}$

(i) (ii)

Fig. 3. A common clause database / variable index data structure. All clauses are stored in a clause database (i), and for each literal the variable index lists the clauses in which it occurs (ii)

Binary and ternary clauses are stored separately in two *implication arrays*. A binary clause $a \vee c$ is stored as two implications: c is stored in the binary implication array of $\neg a$ and a is stored in the binary implication array of $\neg c$. A ternary clause $(a \vee \neg b \vee d)$ is stored as three implications: $\neg b \vee d$ is stored in the ternary implication array of $\neg a$ and the similar is done for b and $\neg d$. Figure 2 shows the implication arrays that represent the example formula $\mathcal{F}_{\text{example}}$.

Storing binary clauses in implication arrays requires only half the memory that would be needed to store them in an ordinary clause database / variable index data structure. (See figure 3.) Since march_eq adds many binary resolvents during the solving phase, the binary clauses on average outnumber the ternary clauses. Therefore, storing these binary clauses in implication arrays significantly reduces the total amount of memory used by march_eq. Furthermore, the implication arrays improve data locality. This often leads to a speed-up due to better usage of the cache.

March_eq uses a variant of iterative unit propagation (IFIUP) that propagates binary implications before ternary implications. The first step of this procedure is to assign as many variables as possible using only the binary implication arrays. Then, if no conflict is found, the ternary implication array of each variable that was assigned in the first step is evaluated. We will illustrate this second step with an example.

Suppose look-ahead is performed on $\neg c$. The ternary implication array of $\neg c$ contains $(\neg a \vee b)$. Now there are five possibilities:

1. If the clause is already satisfied, i.e. a has already been assigned the value false or b has already been assigned the value true, then nothing needs to be done.
2. If a has already been assigned the value true, then b is implied and so b is assigned the value true. The first step of the procedure is called to assign as many variables implied by b as possible. Also, the constraint resolvent $(c \vee b)$ is added as two binary implications.
3. If b has already been assigned the value false, then $\neg a$ is implied and so a is assigned the value false. The first step of the procedure is called to assign as many variables implied by $\neg a$ as possible. Also, the constraint resolvent $(c \vee \neg a)$ is added as two binary implications.
4. If a and b are unassigned, then we have found a new binary clause.
5. If a has already been assigned the value true and b has already been assigned the value false, then $\neg c$ is a failed literal. Thus c is implied.

The variant of DIFF used in march_eq weighs new binary clauses that are produced during the look-ahead phase. A ternary clause that is reduced to a binary clause that gets satisfied in the same iteration of IFIUP, should *not* be included in this computation. However, in the current implementation these clauses are in fact included, which causes noise in the DIFF heuristics. The first step of the IFIUP procedure, combined with the addition of constraint resolvents, ensures that the highest possible amount of variables are assigned *before* the second step of the IFIUP procedure. This reduces the noise significantly.

An advantage of IFIUP over general IUP is that it will detect conflicts faster. Due to the addition of constraint resolvents, most conflicts will be detected in the first call of the first step of IFIUP. In such a case, the second step of IFIUP is never executed. Since the second step of IFIUP is considerably slower than the first, an overall speed-up is expected.

Storage of ternary clauses in implication arrays requires an equal amount of memory as the common alternative. However, ternary implication arrays allow optimisation of the second step of the IFIUP procedure. On the other hand, ternary clauses are no longer stored as such: it is not possible to efficiently verify if they have already been satisfied and early detection of a solution is neglected. One knows only that a solution exists if all variables have been assigned and no conflict has occurred.

6 Equivalence Reasoning

During the pre-processing phase, march_eq extracts the so-called equivalence clauses $(l_1 \leftrightarrow l_2 \leftrightarrow \cdots \leftrightarrow l_i)$ from the formula and places them into a separate data-structure called the *Conjunction of Equivalences* (CoE). After extraction, a solution for the CoE is computed as described in [7, 13].

In [7] - appearing in the same volume - we propose a new look-ahead evaluation function for benchmarks containing equivalence clauses: let eq_n be a weight for a reduced equivalence clause of new length n, $\mathcal{C}(x)$ the set of all reduced equivalence clauses ($\mathcal{Q}_i$) during a look-ahead on x, and $\mathcal{B}(x)$ the set of all newly created binary clauses during the look-ahead on x. Using both sets, the look-ahead evaluation can be calculated as in equation (2). Variable x_i with the highest $\text{DIFF}_{eq}(x_i) \times \text{DIFF}_{eq}(\neg x_i)$ is selected for branching.

$$eq_n = 5.5 \times 0.85^n \tag{1}$$

$$\text{DIFF}_{eq} = |\mathcal{B}| + \sum_{\mathcal{Q}_i \epsilon \mathcal{C}} eq_{|\mathcal{Q}_i|} \tag{2}$$

Besides the look-ahead evaluation and the pre-selection heuristics (discussed in section 7), the intensity of communication between the CoE- and CNF-part of the formula is kept rather low (see figure 4). Naturally, all unary clauses in all phases of the solver are exchanged between both parts. However, during the solving phase, all binary equivalences are removed from the CoE and transformed to the four equivalent binary implications which in turn are added to the implication arrays. The reason for this is twofold: (1) the binary implication structure is faster during the look-ahead phase than the CoE-structure, and (2) for all unary clauses y_i that are created in the CoE during $\text{IUP}(\mathcal{F} \cup \{x\})$, constraint resolvent $\neg x \vee y_i$ can be added to the formula without having to check the original length.

We examined other forms of communication, but only small gains were noticed on only some problems. Mostly, performance decreased due to higher com-

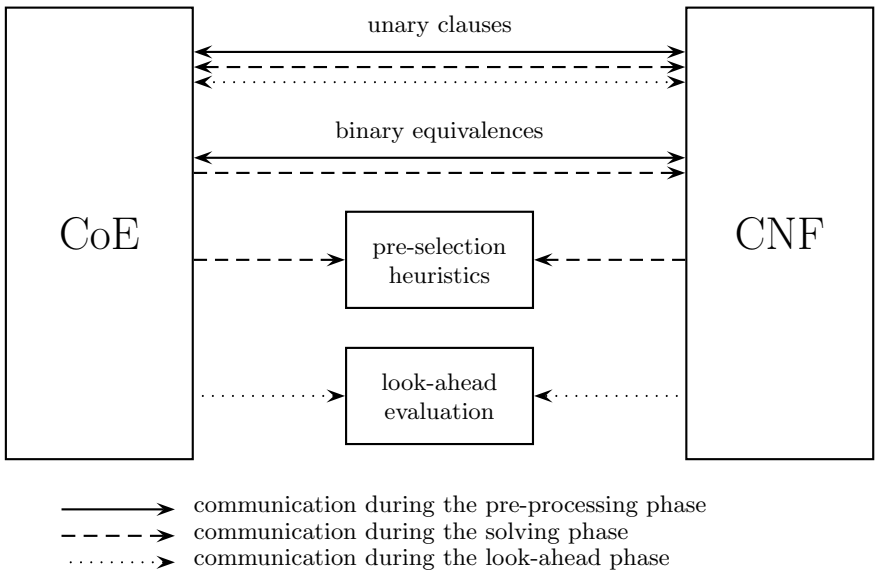


Fig. 4. Various forms of communication in march_eq

Table 2. Performance of march_eq on several benchmarks with and without equivalence reasoning

	without equivalence reasoning		**with equivalence reasoning**		
Benchmarks	time(s)	treesize	time(s)	treesize	speed-up
`random_unsat_250` (100)	1.45	3391.7	1.45	3391.7	-
`random_unsat_350` (100)	48.78	73357.2	48.78	73357.2	-
`stanion/hwb-n20-01`	42.88	182575	23.65	183553	44.85 %
`stanion/hwb-n20-02`	55.34	222487	30.91	222251	44.15 %
`stanion/hwb-n20-03`	42.08	164131	21.70	163984	48.43 %
`longmult8`	76.69	8091	90.80	8149	-18.40 %
`longmult10`	171.66	11597	226.31	11597	-31.84 %
`longmult12`	126.36	6038	176.85	5426	-39.96 %
`pyhala-unsat-35-4-03`	737.15	19513	662.93	19517	10.07 %
`pyhala-unsat-35-4-04`	691.04	19378	659.04	19364	4.63 %
`quasigroup3-9`	7.97	1495	7.97	1495	-
`quasigroup6-12`	58.05	1311	58.05	1311	-
`quasigroup7-12`	10.03	256	10.03	256	-
`zarpas/rule14_1_15dat`	21.68	0	20.70	0	4.52 %
`zarpas/rule14_1_30dat`	219.61	0	186.27	0	15.18 %

munication costs. For instance: communication of binary equivalences from the CNF- to the CoE-part makes it possible to substitute those binary equivalences in order to reduce the total length of the equivalence clauses. This rarely resulted in an overall speed-up.

We tried to integrate the equivalence reasoning in such a manner that it would only be applied when the performance would benefit from it. Therefore, march_eq does not perform any equivalence reasoning if no equivalence clauses are detected during the pre-processing phase (if no CoE exists), making march_eq equivalent to its older brother march.

Table 2 shows that the integration of equivalence reasoning in march rarely results in a loss of performance: on some benchmarks like the `random_unsat` and the `quasigroup` family no performance difference is noticed, since no equivalence clauses were detected. Most families containing equivalence clauses are solved faster due to the integration. However, there are some exceptions, like the `longmult` family in the table.

If we compare the integration of equivalence reasoning in march (which resulted in march_eq) with the integration in satz (which resulted in eqsatz), we note that eqsatz is much slower than satz on benchmarks that contain no equivalence clauses. While satz[2] solves 100 `random_unsat_350` benchmarks near the treshold on average in 22.14 seconds using 105798 nodes, eqsatz[3] requires on

[2] Version 2.15.2 at http://www.laria.u-picardie.fr/∼cli/EnglishPage.html
[3] Version 2.0 at http://www.laria.u-picardie.fr/∼cli/EnglishPage.html

average 795.85 seconds and 43308 nodes to solve the same set. Note that no slowdown occurs for march_eq.

7 Pre-selection Heuristics

Overall performance can be gained or lost by performing look-ahead on a subset of the free variables in a node: gains are achieved by the reduction of computational costs, while losses are the result of either the inability of the *pre-selection heuristics* (heuristics that determine the set of variables to enter the look-ahead phase) to select effective branching variables or the lack of detected failed literals. When look-ahead is performed on only a subset of the variables, only a subset of the failed literals is most likely detected. Depending on the formula, this could increase the size of the DPLL-tree.

During our experiments, we used pre-selection heuristics which are an approximation of our combined look-ahead evaluation function (Ace) [7]. These pre-selection heuristics are costly, but because they provide a clear discrimination between the variables, a small subset of variables could be selected. Experiments with a *fixed number* of variables entering the look-ahead procedure is shown in

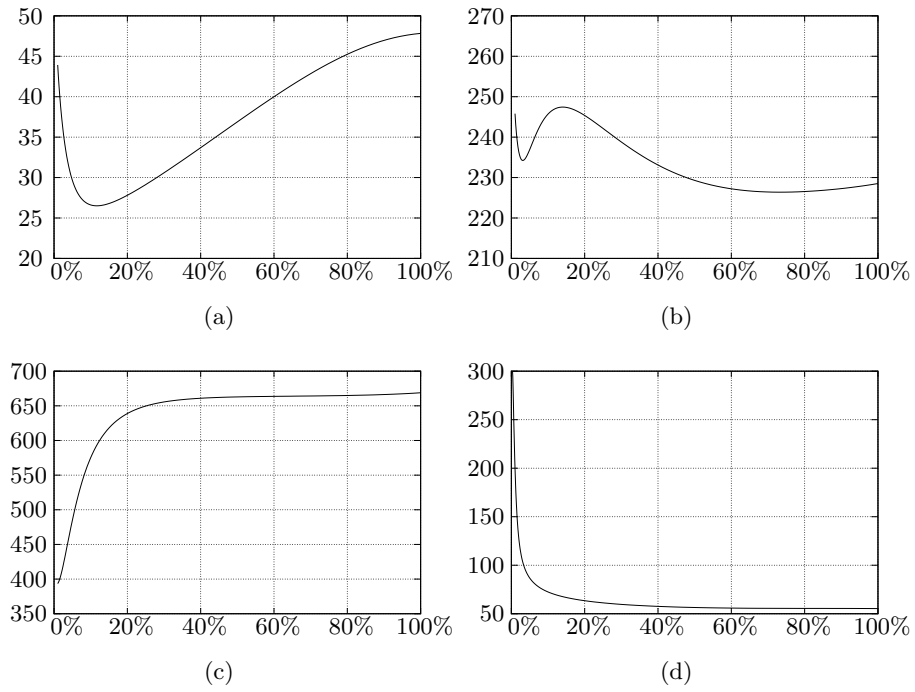


Fig. 5. Runtime(s) vs. percentage look-ahead variables on single instances: (a) `random_unsat_350`; (b) `longmult10`; (c) `pyhala-braun-unsat-35-4-04`; and (d) `quasigroup6-12`

figure 5. The fixed number is based on a percentage of the original number of variables and the "best" variables (with the highest pre-selection ranking) are selected.

The plots in this figure do not offer any indication of which percentage is required to achieve optimal general performance: while for some instances 100% look-ahead appears optimal, others are solved faster using a much smaller percentage.

Two variants of march_eq have been submitted to the SAT 2004 competition [11]: one which selects in every node the "best" 10 % variables (march_eq_010) and one with full (100%) look-ahead (march_eq_100). Although during our experiments the first variant solved the most benchmarks, at the competition both variants solved the same number of benchmarks, albeit different ones. Figure 5 illustrates the influence of the number of variables entering the look-ahead procedure on the overall performance.

8 Tree-Based Look-Ahead

The structure of our look-ahead procedure is based on the observation that different literals, often entail certain shared implications, and that we can form 'sharing' trees from these relations, which in turn may be used to reduce the number of times these implications have to be propagated during look-ahead.

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, backtrack the latter, then propagate the other look-ahead literal and only in the end backtrack to the initial state. This way, the shared implication has been propagated only once.

Figure 6 shows this example graphically. The implications among a, b and c form a small tree. Some thought reveals that this process, when applied recursively, could work for arbitrary trees. Based on this idea, our solver extracts - prior to look-ahead - trees from the implications among the literals selected for look-ahead, in such a way that each literal occurs in exactly one tree. The look-

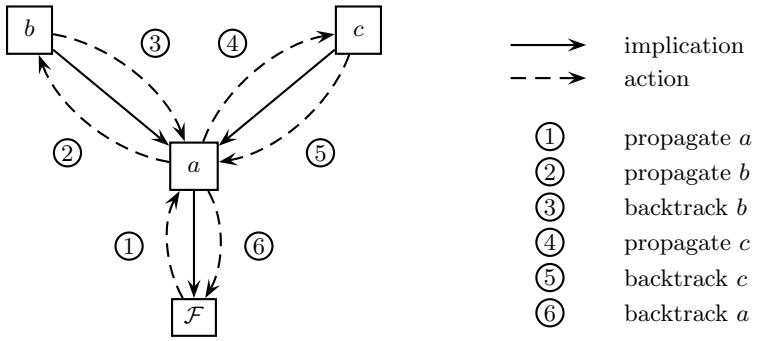


Fig. 6. Graphical form of an implication tree with corresponding actions.

Table 3. Performance of march_eq on several benchmarks with and without the use of tree-based look-ahead

	normal look-ahead		**tree-based look-ahead**		
Benchmarks	time(s)	treesize	time(s)	treesize	speed-up
random_unsat_250 (100)	1.24	3428.5	1.45	3391.7	-16.94 %
random_unsat_350 (100)	40.57	74501.7	48.78	73357.2	-20.24 %
stanion/hwb-n20-01	29.55	184363	23.65	183553	19.97 %
stanion/hwb-n20-02	40.93	227237	30.91	222251	24.48 %
stanion/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 %
pyhala-unsat-35-4-03	1084.08	19093	662.93	19517	38.85 %
pyhala-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 %
zarpas/rule14_1_15dat	25.62	0	20.70	0	19.20 %
zarpas/rule14_1_30dat	192.30	0	186.27	0	3.14 %

ahead procedure is improved by recursively visiting these trees. Of course, the more dense the implication graph, the more possibilities are available 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, so 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, we have developed a simple heuristic. More involved methods would probably produce better results, although optimality in this area could easily mean 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 order of decreasing prediction, while 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. On 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 so, these are labelled child nodes of the node corresponding to the implied literal. If not already encountered, these child nodes are now recursively

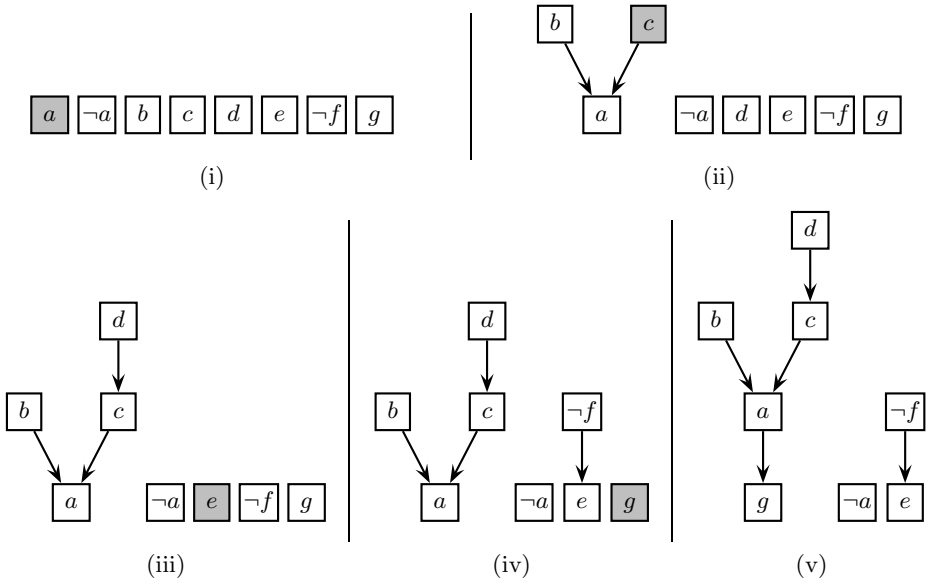


Fig. 7. Five steps of building implication trees

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. A gray box denotes the current position:

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

Availability of implication trees 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, we suggest a simple, linear-time algorithm that scans the trees.

Some intriguing ideas for further research have occurred to us during the development our tree-based look-ahead procedure, but which, we have not been able to pursue due to time constraints. 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 on and possibly branch upon. Because of the inherent sharing, the overhead will be smaller than without a tree-based look-ahead.

Also, once trees have been created, we could include non-look-ahead 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 each other. In general, it might be a good idea to focus in detail on possibilities of sharing.

9 Removal of Inactive Clauses

The presence of inactive clauses increases the computational costs of the procedures performed during the look-ahead phase. Two important causes can be appointed: first, the larger the number of clauses considered during the look-ahead, the poorer the performance of the cache. Second, if both active and inactive clauses occur in the active data-structure during the look-ahead, a check is necessary to determine the status of every clause. Removal of inactive clauses from the active data-structure prevents these unfavourable effects.

When a variable x is assigned to a certain truth value during the solving phase, all the ternary clauses in which it occurs become inactive in the ternary implication arrays: the clauses in which x occurs positively become satisfied, while those clauses in wich it occurs negatively are reduced to binary clauses. These binary clauses are moved to the binary implication arrays.

Table 4. Performance of march_eq on several benchmarks with and without the removal of inactive clauses on the chosen path

	without removal		**with removal**		
Benchmarks	time(s)	treesize	time(s)	treesize	speed-up
`random_unsat_250` (100)	1.70	3393.7	1.45	3391.7	14.71 %
`random_unsat_350` (100)	63.38	73371.9	48.78	73357.2	23.04 %
`stanion/hwb-n20-01`	24.92	182575	23.65	183553	5.10 %
`stanion/hwb-n20-02`	33.78	222487	30.91	222251	8.50 %
`stanion/hwb-n20-03`	23.68	164131	21.70	163984	8.36 %
`longmult8`	114.71	8091	90.80	8149	20.84 %
`longmult10`	287.37	11597	226.31	11597	21.25 %
`longmult12`	254.51	6038	176.85	5426	30.51 %
`pyhala-unsat-35-4-03`	783.52	19513	662.93	19517	15.39 %
`pyhala-unsat-35-4-04`	772.59	19378	659.04	19364	14.70 %
`quasigroup3-9`	11.73	1497	7.97	1495	32.05 %
`quasigroup6-12`	136.70	1335	58.05	1311	57.53 %
`quasigroup7-12`	22.53	256	10.03	256	55.48 %
`zarpas/rule14_1_15dat`	29.80	0	20.70	0	30.54 %
`zarpas/rule14_1_30dat`	254.81	0	186.27	0	26.90 %

Table 4 shows that the removal of inactive clauses during the solving phase is useful on all kinds of benchmarks. Although the speed-up is only small on uniform random benchmarks, larger gains are achieved on more structured instances.

10 Conclusion

Several techniques have been discussed to increase the solving capabilities of a look-ahead SAT solver. Some are essential for solving various specific benchmarks: a range of families can only be solved using equivalence reasoning, and as we have seen, `march_eq` is able to solve a large `zarpas` benchmark by adding only constraint resolvents.

Other proposed techniques generally result in a performance boost. However, the usefulness of our pre-selection heuristics is as yet undoubtedly subject to improvement and will be subject of future research.

References

1. D. Mitchel, B. Selmon and H. Levesque, *Hard and easy distributions of SAT problems*. Proceedings of AIII-1992 (1992), 459–465.
2. D. Le Berre, *Exploiting the Real Power of Unit Propagation Lookahead*. In LICS Workshop on Theory and Applications of Satisfiability Testing (2001).
3. D. Le Berre and L. Simon, *The essentials of the SAT'03 Competition*. Springer-Verlag, Lecture Notes in Comput. Sci. **2919** (2004), 452–467.
4. 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.
5. M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem proving*. Communications of the ACM **5** (1962), 394–397.
6. O. Dubois and G. Dequen, *A backbone-search heuristic for efficient solving of hard3-sat formulae*. International Joint Conference on Artificial Intelligence 2001 **1** (2001), 248–253.
7. M.J.H. Heule and H. van Maaren, *Aligning CNF- and Equivalence-Reasoning*. Appearing in the same volume.
8. O. Kullmann, *Investigating the behaviour of a SAT solver on random formulas*. Submitted to Annals of Mathematics and Artificial Intelligence (2002).
9. C.M. Li and Anbulagan, *Look-Ahead versus Look-Back for Satisfiability Problems*. Springer-Verlag, Lecture Notes in Comput. Sci. **1330** (1997), 342–356.
10. 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.
11. L. Simon, *Sat'04 competition homepage*. http://www.lri.fr/∼simon/contest/results/
12. 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.
13. J.P. Warners, H. van Maaren, *A two phase algorithm for solving a class of hard satisfiability problems*. Oper. Res. Lett. **23** (1998), no. 3-5, 81–88.
14. H. Zhang and M.E. Stickel, *Implementing the Davis-Putnam Method*. SAT2000 (2000), 309–326.