Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 47-59

March_dl: Adding Adaptive Heuristics and a New Branching Strategy

Marijn J.H. Heule* m.j.h.heule@ewi.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

Abstract

We introduce the march_dl satisfiability (SAT) solver, a successor of march_eq. The latter was awarded state-of-the-art in two categories during the SAT 2004 competition. The focus lies on presenting those features that are new in march_dl. Besides a description, each of these features is illustrated with some experimental results. By extending the preprocessor, using adaptive heuristics, and by using a new branching strategy, march_dl is able to solve nearly all benchmarks faster than its predecessor. Moreover, various instances which were beyond the reach of march_eq, can now be solved - relatively easily - due to these new features.

KEYWORDS: *Adaptive heuristics, Look-ahead, SAT solving, SAT competition*

Submitted October 2005; revised December 2005; published March 2006

1. Introduction

The satisfiability (SAT) competitions of the last years have boosted the development of SAT solvers: Each year, several unsolvable benchmarks (within the given time limit), were easily solved the year after. Modern SAT solver architectures can be split into three divisions: Conflict-driven (minisat, vallst, zChaff), look-ahead (kcnfs, march, OKsolver) and local search (AdaptNovelty+, R+ AdaptNovelty+, unitwalk). All solvers mentioned above won a category in the past competitions [9, 10, 11, 15]. Each architecture outperforms the other two on parts of the spectrum of available CNF instances. For instance, conflict-driven solvers dominate on industrial formulas, look-ahead solvers are very strong on unsatisfiable random formulas, while local-search techniques are unbeatable on large satisfiable random formulas.

As a look-ahead SAT solver, early development of march was focused on fast performance on unsatisfiable uniform random 3-SAT formulas. Frustrated by the poor performance on structured instances, we attempted to increase the speed on this latter kind of benchmarks by additional reasoning and eager data-structures. The resulted solver, march_eq, is described in detail in [8]. Since equivalence reasoning - an important part of march_eq and thus march_dl - is not further developed, we will ignore this aspect of the solver in this paper.

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

The march_eq solver was quite successful: It won two (crafted) categories during the Sat 2004 competition [10]. However, various benchmarks - relatively easy to solve by conflict-driven solvers - were still unsolvable by march_eq. We developed some enhancements in order to solve several of these instances. These enhancements are the primary focus of this paper.

The usefulness of each enhancement is illustrated by some experiments. We selected a small set of benchmarks for this purpose, since extensive experiments are beyond the scope of this paper. Because look-ahead Sat solvers perform relatively well on unsatisfiable uniform random 3-Sat formulas, we generated[1.] 200 of them (100 of 250 variables with 1075 clauses, and 100 of 350 variables with 1500 clauses) as a reference. We added some crafted and structured instances from five families:

- the `connamacher` family contributed by Connamacher to Sat 2004. This family consists of encodings of the generic uniquely extendible constraint satisfaction problem [3].
- the `ezfact` family contributed by Pehoushek to Sat 2002 [15]. These benchmarks are encodings of factorization problems.
- the `lksat` family contributed by Anton to Sat 2004 [10]. These are random l-clustered k-Sat instances.
- the `longmult` family contributed by Biere. Instances from this family arise from bounded model checking [1].
- the `philips` family contributed by Heule to Sat 2004 [10]. Encoding of a multiplier circuit provided by Philips.

All experiments were performed on a system with an Intel 3.0 GHz CPU and 1 Gb of memory running on Fedora Core 4.

The remaining part of this paper is structured as follows: Section 2 provides a short introduction on the look-ahead architecture together with some references to the origin of the techniques. Section 3 deals with two small enhancements to the march_dl pre-processor. Two new adaptive heuristics are introduced in section 4, and a new branching strategy is presented in section 5. Finally, section 6 concludes with some results on the overall performance.

2. Look-ahead architecture

Since march_dl is a look-ahead Sat solver, we will first provide a brief introduction on its general architecture. This architecture (introduced in [6]) consists of a DPLL search-tree [4] using a LookAhead procedure to determine a branch literal l_{branch} [14] (see algorithm 1). We refer to a look-ahead on literal l as assigning l to true and performing iterative unit propagation. If a conflict occurs during this unit propagation (the empty clause is generated), then l is called a *failed literal* - forcing l to be fixed on false. The resulting formula after a look-ahead on l is denoted by $\mathcal{F}(l = 1)$.

1. using mkcnf available from www.satlib.org.

Algorithm 1 DPLL($\mathcal{F}$)

if $\mathcal{F} = \emptyset$ **then**
  return `satisfiable`
else if empty clause $\in \mathcal{F}$ **then**
  return `unsatisfiable`
end if
l_{branch} := LookAhead($\mathcal{F}$)
if DPLL($\mathcal{F}(l_{\text{branch}} = 1)$) = `satisfiable` **then**
  return `satisfiable`
else
  return DPLL($\mathcal{F}(l_{\text{branch}} = 0)$)
end if

Algorithm 2 LookAhead($\mathcal{F}$)

$\mathcal{P}$:= PreSelect($\mathcal{F}$)
for each variable x_i in $\mathcal{P}$ **do**
  $\mathcal{F}' := \mathcal{F}(x_i = 0)$
  if empty clause $\notin \mathcal{F}'$ **and** $\mathcal{F}'_2 \gg \mathcal{F}_2$ **then**
    $\mathcal{F}'$:= DoubleLook($\mathcal{F}'$)
  end if
  $\mathcal{F}'' := \mathcal{F}(x_i = 1)$
  if empty clause $\notin \mathcal{F}''$ **and** $\mathcal{F}''_2 \gg \mathcal{F}_2$ **then**
    $\mathcal{F}''$:= DoubleLook($\mathcal{F}''$)
  end if
  if empty clause $\in \mathcal{F}'$ **and** empty clause $\in \mathcal{F}''$ **then**
    return `unsatisfiable`
  else if empty clause $\in \mathcal{F}'$ **then**
    $\mathcal{F} := \mathcal{F}''$
  else if empty clause $\in \mathcal{F}''$ **then**
    $\mathcal{F} := \mathcal{F}'$
  else
    $H(x_i)$ = MixDiff(Diff($\mathcal{F}$, $\mathcal{F}'$), Diff($\mathcal{F}$, $\mathcal{F}''$))
  end if
end for
x_{branch} := x_i with greatest $H(x_i)$
return GetDirection(x_{branch})

Five subprocedures of the LookAhead procedure (see algorithm 2) are common in most modern look-ahead Sat solvers:

- Preselect - In general, performing a look-ahead on all unfixed variables is very costly. Therefore, most look-ahead Sat solvers pre-select a subset (denoted by $\mathcal{P}$) of the unfixed variables in each node of the search-tree to enter the LookAhead procedure. This enhancement was introduced by Li *et al.* [14]. In each node, variables are ranked based on their occurrences in binary and ternary clauses. Variables with the highest ranking are pre-selected. Some modifications to these pre-selection heuristics are discussed in section 4.1.

- DoubleLook - If during the look-ahead on a literal, many new binary clauses are created (denoted by $\mathcal{F}_2' \gg \mathcal{F}_2$), the resulting formula is frequently unsatisfiable. The DoubleLook procedure attempts to find a conflict in the resulted formula by performing additional look-aheads. This subprocedure was first implemented in satz to reduce the search tree while solving uniform random 3-Sat formulas [12]. Details on this subprocedure are presented in section 4.2.

- Diff - The look-ahead evaluation function (Diff) used in march_dl is identical to the one used in march_eq: Newly created binary clauses and reduced equivalence clauses are counted in a weighted fashion. The resulting sum is used as an indicator of the size of the search-tree of the reduced formula. A higher sum suggests a smaller search-tree. For a full description we refer to [8].

- MixDiff - Combines the two Diff numbers. Let $L := \text{Diff}(\mathcal{F}, \mathcal{F}(x = 0))$ and $R := \text{Diff}(\mathcal{F}, \mathcal{F}(x = 1))$ then $\text{MixDiff}(L, R) := 1024 \times LR + L + R$. Motivation for this formula is that an ideal branching variable splits the formula into two small search-trees (realized by LR). The $L + R$ addition is used for tie-breaking purposes. The formula is used in most look-ahead Sat solvers (kcnfs [5], march_eq [8], posit [6], and satz [14]) and originates from [6].

- GetDirection - Given a branch variable x_{branch}, it pays off (only on satisfiable instances) to choose wisely whether to first enter branch $\mathcal{F}(x_{\text{branch}} = 1)$ or branch $\mathcal{F}(x_{\text{branch}} = 0)$. If $\text{Diff}(\mathcal{F}, \mathcal{F}(x_{\text{branch}} = 1)) < \text{Diff}(\mathcal{F}, \mathcal{F}(x_{\text{branch}} = 0))$, march_dl enters the first, otherwise the latter.

3. Pre-processor enhancements

In practice, the performance of look-ahead Sat solvers is highly related to the size of the formula: Large CNF's require generally much more solving time regardless the complexity of the underlying problem. Both other architectures do not appear very sensitive to this. Therefore, pre-processing (reducing the size of) the formula is essential for fast performance of a look-ahead Sat solver. march_dl simplifies the formula - like march_eq - by binary equivalence propagation, detection of failed literals and subsumption of clauses [8].

3.1 Root look-ahead

Unlike the other solvers participating in the SAT competitions, all march versions use a 3-SAT translator in the pre-processor. march_dl uses the same translator as the one used in march_eq [8]. In the pre-processor, most march versions perform an *iterative full look-ahead* procedure to reduce the formula. This procedure checks for all literals (full) whether unit propagation will result in a conflict. This is iteratively performed until no new failed literals are detected.

In march_eq, this procedure was executed *after* applying the 3-SAT translator. The resulted formula - after the iterative full look-ahead procedure - frequently contained many dummy (introduced by the translation) variables. By executing the procedure *before* the 3-SAT translator, generally, less dummy variables have to be used.

Table 1. Performance of march_dl and the number of used dummy variables by applying the root look-ahead before or after the 3-SAT translator.

	before translation		**after translation**	
Benchmarks	time(s)	#dummies	time(s)	#dummies
`random-unsat-250` (100)	0.52	0	0.52	0
`random-unsat-350` (100)	15.04	0	15.04	0
`connm-n600-d0.04-sat04-975`	406.52	148	406.52	148
`connm-n600-d0.04-sat04-978`	535.05	142	535.05	142
`connm-n600-d0.04-sat04-981`	220.78	141	220.78	141
`ezfact48-1`	**8.61**	1803	13.34	1850
`ezfact48-2`	**8.84**	1792	14.76	1846
`ezfact48-3`	**19.93**	1817	28.86	1857
`lksat-n1000-k3-l5-sat04-930`	26.47	0	26.47	0
`lksat-n1000-k3-l5-sat04-931`	25.41	0	25.41	0
`lksat-n1000-k3-l5-sat04-932`	6.97	0	6.97	0
`longmult8`	**35.62**	195	51.14	377
`longmult10`	**117.50**	246	140.72	471
`longmult12`	**218.92**	293	352.17	565
`philips`	292.81	0	292.81	0

Table 1 shows some experimental results of the effects - on solving time and number of generated dummies - on performing a root look-ahead before or after the 3-SAT translator. This table gives quite an accurate illustration of the effect of this enhancement: If swapping the execution order of the root look-ahead and the translator results in less generated dummy variables, then less computational time is required. Otherwise, no difference is noticed in solving times too.

3.2 Ternary resolvents

While pre-processing a formula, many resolvents could be added. Addition of all possible resolvents will - in general - significantly increase the size of the problem. Even adding only all resolvents of length two in the preprocessing phase, will increase solving time in most cases. Therefore, adding resolvents in march_eq is restricted to all binary constraint resolvents [8], both in the pre-processor and in the actual solving phase.

In march_eq, we already implemented a prototype procedure adding some ternary resolvents. This procedure is now efficiently implemented in march_dl and adds - just after the 3-Sat translation - only ternary resolvents of a certain type: All ternary resolvents are added to the formula that could be created by resolving two ternary clauses:

$$(x_i \vee x_j \vee x_r) \otimes_{x_j} (x_i \vee \neg x_j \vee x_s) = (x_i \vee x_r \vee x_s) \tag{1}$$

In this equation, $\otimes_{x_j}$ refers to the resolution operator on variable x_j. Notice that added ternary resolvents could be used to create other ternary resolvents using the same equation.

The motivation to add these resolvents is first observed in [2]. On uniform random 3-Sat formulas, their addition in the pre-processor reduces on average the computational costs by about 10% [13]. We experimented with the addition of these resolvents on various structured benchmarks. Within our experimental domain, this addition appeared to have either a favorable influence or no influence at all regarding the required computation time.

Table 2. Performance of march_dl on several benchmarks with and without adding ternary resolvents during the pre-processing phase.

Benchmarks	$\#T_{trans}$	$\#T_{resolve}$	**with**	**without**
`random-unsat-250` (100)	1075	92.7	**0.52**	0.55
`random-unsat-350` (100)	1500	89.3	**15.04**	16.13
`connm-n600-d0.04-sat04-975`	7640	16840	**406.52**	> 2000
`connm-n600-d0.04-sat04-978`	7292	17908	**535.05**	> 2000
`connm-n600-d0.04-sat04-981`	7242	16888	**220.78**	> 2000
`ezfact48-1`	8086	4292	**8.61**	> 2000
`ezfact48-2`	8063	3969	**8.84**	> 2000
`ezfact48-3`	8104	4596	**19.93**	> 2000
`lksat-n1000-k3-l5-sat04-930`	3629	1080	**26.47**	392.91
`lksat-n1000-k3-l5-sat04-931`	3602	715	**25.41**	875.20
`lksat-n1000-k3-l5-sat04-932`	3634	926	**6.97**	233.51
`longmult8`	1638	48	**35.62**	37.48
`longmult10`	2142	57	117.50	**111.85**
`longmult12`	2670	62	**218.92**	233.85
`philips`	896	0	292.81	292.81

Table 2 shows the number of ternary clauses after the 3-Sat translator ($\#T_{trans}$) and the number of ternary resolvents that - using (1) - could be added ($\#T_{resolve}$). The last two columns show the computational cost of march_dl with and without this addition. The table convincingly shows the usefulness of adding these ternary resolvents on a wide scale of benchmarks. In only one case the performance is slightly decreased. Since, on structured benchmarks, the far majority of the clauses has length two, this performance boost can be realized by the addition of relatively few clauses.

4. Adaptive heuristics

Most heuristics used in look-ahead Sat solvers are heavily optimized towards fast performance on uniform random formulas. These heuristics are partly the cause of mediocre performance on structured instances. By developing heuristics that adapt towards each specific instance, we tried to perform well 'across the board'.

4.1 Pre-selection heuristics

The main differences between the four march versions submitted to the Sat 2004 competition (march_001, march_007, march_eq_010, and march_eq_100) is the number of variables pre-selected to enter the LookAhead procedure. Each version pre-selects a fixed number of variables determined as a percentage of the original number of variables. The last suffix (_xxx) denotes this percentage. For instance, while solving a CNF with 1234 initial variables, march_eq_010 will pre-select 123 variables in each node of the DPLL search-tree to enter the LookAhead procedure. If, in a certain node, there are less than 123 variables unfixed, all remaining variables will be pre-selected. Hence, deeper in the search-tree relative more unfixed variables are pre-selected.

The motivation to use a different percentage in each of the submitted versions originates from the observation that the optimal percentage is benchmark dependent [8]. Therefore, we decided to use more dynamic pre-selection heuristics in march_dl. We also observed that - within our experimental domain - the optimal percentage was closely related to the frequency of detected failed literals: When relatively many failed literals were detected, higher percentages appeared optimal. Let $\#\text{failed}_i$ be the number of detected failed literals in node i. We tried to exploit the correlation mentioned above by using the average number of detected failed literals as an indicator for the maximum size of the pre-selected set in node n (denoted by $\mathcal{P}^n_{\max}$):

$$\mathcal{P}^n_{\max} := \mu + \frac{\gamma}{n} \sum_{i=1}^{n} \#\text{failed}_i \tag{2}$$

In the above, parameter μ refers to the lowerbound of $\mathcal{P}_{\max}$ in each node (namely when the average tends to zero) and γ is a parameter modelling the importance of failed literals. During small scale experiments on various structured and random instances, the values $\mu := 5$ and $\gamma := 7$ resulted in favorable performance on most instances. Notice that the above adaptive pre-selection heuristics are heavily influenced by the branching strategy - which is also affected by these heuristics.

In most nodes $|\mathcal{P}| = \mathcal{P}_{\max}$. Only when the number of unfixed variables in the formula is smaller than $\mathcal{P}_{\max}$, then all variables are pre-selected - resulting in $|\mathcal{P}| < \mathcal{P}_{\max}$. It could happen that, during the LookAhead procedure, all variables in $\mathcal{P}$ are forced - due to the detection of failed literals. In these cases the procedure is restarted with the reduced formula.

Table 3. Performance of march_dl and three modified versions march_dl$^*_{xxx}$ with a constant number of pre-selected variables. Subscript xxx denotes the percentage of the original number of variables used for this constant.

Benchmarks	march_dl	march_dl$^*_{001}$	march_dl$^*_{010}$	march_dl$^*_{100}$
`random-unsat-250` (100)	**0.52**	0.94	0.54	0.71
`random-unsat-350` (100)	**15.04**	33.62	15.80	25.31
`connm-n600-d0.04-sat04-975`	406.52	1150.86	**389.65**	584.96
`connm-n600-d0.04-sat04-978`	535.05	**517.55**	661.29	707.19
`connm-n600-d0.04-sat04-981`	220.78	814.78	**210.31**	291.60
`ezfact48-1`	8.61	**7.72**	8.62	8.67
`ezfact48-2`	8.84	**8.52**	9.02	9.05
`ezfact48-3`	19.93	**17.64**	19.78	19.89
`lksat-n1000-k3-l5-sat04-930`	26.47	39.18	**23.83**	52.85
`lksat-n1000-k3-l5-sat04-931`	25.41	40.77	**23.94**	47.50
`lksat-n1000-k3-l5-sat04-932`	6.97	9.87	**6.72**	11.65
`longmult8`	**35.62**	36.01	43.68	44.21
`longmult10`	117.50	**102.43**	107.65	111.87
`longmult12`	218.92	**130.54**	153.24	165.43
`philips`	292.81	**282.16**	432.84	441.83

The effect of using these adaptive pre-selection heuristics on the performance is shown in table 3. As a reference, three columns are added with the computational costs of modified versions of march_dl with static percentages. Although the adaptive variant rarely results in the fastest performance; in general, its performance is relatively - compared to the references - close to optimal. Since these adaptive heuristics are still in an experimental phase, we expect to achieve even better results by further optimizing the settings.

4.2 Double look-ahead

The DoubleLook procedure (see algorithm 3) checks whether a formula resulting from a look-ahead is unsatisfiable. It does so by performing additional unit-propagations. Since the computational costs of the DoubleLook procedure are high, it should not be called after every look-ahead. In the ideal case, one would only call it when the procedure would detect that the input formula is unsatisfiable. This could be done by an indicator expressing the usefullness of a DoubleLook call.

Li [12] suggests that the number of newly created binary clauses found during a look-ahead is an effective indicator whether or not to call the DoubleLook procedure: Many

newly created binary clauses during a look-ahead increases the chance that DoubleLook will detect a conflicting formula. Li's solver satz calls DoubleLook if the number of new binary clauses in the reduced formula (after a look-ahead) is larger than a certain constant. We refer to this constant as $\mathrm{DL}_{\mathrm{trigger}}$. In satz, $\mathrm{DL}_{\mathrm{trigger}} := 65$ is used.

Algorithm 3 DoubleLook($\mathcal{F}$)

```
1: P := PreSelect( F )
2: for each variable x_i in P do
3:    F' := F(x_i = 0)
4:    F'' := F(x_i = 1)
5:    if empty clause ∈ F' and empty clause ∈ F'' then
6:       return F'
7:    else if empty clause ∈ F' then
8:       F := F''
9:    else if empty clause ∈ F'' then
10:      F := F'
11:   end if
12: end for
13: return F
```

Dubois and Dequen use a slight variation of the above setting in their solver kcnfs [5]. Here, the DoubleLook procedure is triggered when the number of new binary clauses is larger than $\mathrm{DL}_{\mathrm{trigger}} := 0.175$ #vars + 5 (#vars refers to the initial number of variables). Both settings of $\mathrm{DL}_{\mathrm{trigger}}$ result from optimizing this parameter towards the performance on uniform random 3-Sat formulas. On these instances they appear quite effective. However, on structured formulas - industrial and crafted - these settings are far from optimal: On some families, practically none of the look-aheads generate enough new binary to trigger DoubleLook. Even worse, on many other structured instances both $\mathrm{DL}_{\mathrm{trigger}}$ settings result in a pandemonium of calls of the DoubleLook procedure, which will come down hard on the computational costs to solve these instances.

To counter these unfavorable effects, march_dl uses a more abstract parameter $\mathrm{DL}_{\mathrm{success}}$. This parameter refers to the aimed ratio of successful calls on the DoubleLook procedure. A DoubleLook call is successful if it detects that the input formula is unsatisfiable. For instance, $\mathrm{DL}_{\mathrm{success}} := \frac{3}{4}$ means that the solver tries to call the DoubleLook procedure in such a way that three out of four calls are successful.

We achieve this success ratio by using a dynamic $\mathrm{DL}_{\mathrm{trigger}}$ parameter: Depending on the success of a certain DoubleLook call, $\mathrm{DL}_{\mathrm{trigger}}$ is updated using $\mathrm{DL}_{\mathrm{update}}$ (see equation (3)). Under the assumption that the number of newly created binary clauses is an effective indicator for the success probability of a DoubleLook call, it is expected that $\mathrm{DL}_{\mathrm{trigger}}$ will converge to a certain value. In practice either it stabilizes or $\mathrm{DL}_{\mathrm{trigger}}$ reaches early in the solving phase a high value such that DoubleLook is never triggered in a later stadium.

$$\mathrm{DL}_{\mathrm{update}} := \begin{cases} -\dfrac{1-\mathrm{DL}_{\mathrm{success}}}{\mathrm{DL}_{\mathrm{success}}} & \text{if DoubleLook}(\mathcal{F}) \text{ is successful} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

Experiments show that a wide range of settings of $DL_{success}$ (0.7 to 0.95) result in a similar (fast) performance. In march_dl we choose $DL_{success} := \frac{9}{10}$. Additionally, we initialized $DL_{trigger} := 0.1\#vars$. A full description with large-scale experiments to analyze and explain the above parameters and the behavior of the evolving sequence of $DL_{trigger}$, will be the subject of a forthcoming paper.

Table 4. Four different settings of the parameter $DL_{trigger}$ implemented in march_dl: (a) adaptive, the default setting in march_dl; (b) $DL_{trigger} := 65$ as used in satz; (c) $DL_{trigger} := 0.175$ #vars + 5 as used in kcnfs; and (d) turning it OFF, so $DL_{trigger} := \infty$.

Benchmarks	adaptive	à la satz	à la kcnfs	OFF
random-unsat-250 (100)	0.52	**0.51**	0.52	0.59
random-unsat-350 (100)	15.04	**14.68**	14.88	17.94
connm-n600-d0.04-sat04-975	**406.52**	823.37	815.69	517.41
connm-n600-d0.04-sat04-978	**535.05**	1808.77	1816.78	1205.27
connm-n600-d0.04-sat04-981	**220.78**	1149.71	1112.08	729.69
ezfact48-1	**8.61**	111.04	9.30	10.25
ezfact48-2	**8.84**	117.33	10.86	10.96
ezfact48-3	**19.93**	223.86	22.21	21.29
lksat-n1000-k3-l5-sat04-930	**26.47**	50.49	26.81	32.63
lksat-n1000-k3-l5-sat04-931	**25.41**	47.61	25.48	31.00
lksat-n1000-k3-l5-sat04-932	**6.97**	14.97	7.04	8.41
longmult8	35.62	73.38	34.83	**34.68**
longmult10	117.50	231.99	120.42	**111.67**
longmult12	218.92	241.80	207.14	**198.12**
philips	218.92	239.47	218.31	**215.87**

Table 4 shows the performance (in seconds) of four different approaches: (1) our proposed adaptive heuristics; (2) the one used in satz; (3) the one used in kcnfs; and as reference (4) no DoubleLook at all (used in march_eq). The adaptive heuristics are by far the best option within our experimented domain. The down sides of the heuristics used in satz and kcnfs are clearly visible: On average the OFF setting performs better than both these random-instance-motivated methods.

5. Local branching

The look-ahead evaluation heuristic H (see algorithm 2) has an unfavorable effect: The selected branch variables only have a high MixDiff value in common. On structured instances this could result in branch variables that are scattered all over the structure. Because no conflict clauses are added in look-ahead Sat solvers, this increases the chance that local conflicts must be resolved multiple times.

By branching only on variables occurring in reduced clauses, we try to counter this effect. We refer to this branching strategy as *local branching*. Clearly, this is not applicable for the first node, because the initial formula has no reduced clauses. Recall that march_dl uses a 3-

Sat translator in the pre-processing, so all clauses are either binary or ternary. Therefore, local branching in this special case means that march_dl only branches on variables that occur in binary clauses that originated from ternary clauses in the initial formula.

This new branching strategy is realized by modifying the PreSelect procedure: Instead of performing the pre-selection heuristics on the whole formula in a certain node, we discard all clauses that also occur in the initial formula (denoted by $\mathcal{F}_{\text{initial}}$). So, only variables occurring in reduced clauses are pre-selected. The resulted procedure is called LocalPreSelect and is shown in algorithm 4. Notice that LocalPreSelect does not only pre-selects different (compared to PreSelect) variables to enter the look-ahead phase, it also could select less variables: The number of variables occurring in reduced clauses in the formula of node n could be smaller the $\mathcal{P}^n_{\text{max}}$.

Algorithm 4 LocalPreSelect($\mathcal{F}$)

1: $\mathcal{F}_{\text{reduced}} := \mathcal{F} \setminus \mathcal{F}_{\text{initial}}$
2: **if** $\mathcal{F}_{\text{reduced}} = \emptyset$ **then**
3: $\quad \mathcal{F}_{\text{initial}} := \mathcal{F}$
4: $\quad$ **restart**
5: **end if**
6: **return** PreSelect($\mathcal{F}_{\text{reduced}}$)

On some families - satisfiable and unsatisfiable - using local branching resulted in large speed-ups. Two examples of this kind are (1) the `ferry` family (all satisfiable) contributed by Maris to the SAT 2003 competition [9] and (2) the `homer` family (all unsatisfiable) contributed by Aloul to the SAT 2002 competition [15]. Table 5 shows the performance of the march versions submitted to the Sat 2004 and 2005 competition on small instances of these families. Clearly, march_dl is the only solver that - due to local branching - could solve these instances. On benchmarks where the new branching strategy did not realize such a speed-up, no significant performance gains or losses were noticed.

Table 5. Performance of different march versions on instances of the `ferry` and `homer` families.

Benchmarks	march_dl	march_001	march_007	march_eq_010	march_eq_100
`ferry8.sat03-384`	**2.18**	> 2000	> 2000	> 2000	> 2000
`ferry8u.sat03-385`	**2.54**	> 2000	> 2000	> 2000	> 2000
`ferry9.sat03-386`	**1.70**	> 2000	> 2000	> 2000	> 2000
`ferry9u.sat03-387`	**1.37**	> 2000	> 2000	> 2000	> 2000
`fpga10-11-uns-rcr`	**118.93**	> 2000	> 2000	> 2000	> 2000
`fpga10-12-uns-rcr`	**136.21**	> 2000	> 2000	> 2000	> 2000
`fpga10-13-uns-rcr`	**154.78**	> 2000	> 2000	> 2000	> 2000

The original motive to implement local branching was to increase the chance of finding *autarkies* [7] - partial assignments that satisfy all clauses that they "touch". The remaining formula, after removing all satisfied clauses by an autarky, is satisfiability equivalent to the original formula. A pure literal is an example of an autarky. Especially on unsatisfiable

benchmarks, detection of autarkies is useful: Unsatisfiability of the remaining formula yields unsatisfiability of the original formula, resulting in a smaller search-tree.

This aspect is also shown in algorithm 4: Whenever the formula $\mathcal{F}$ in a certain node does not contain reduced clauses - compared to $\mathcal{F}_{\text{initial}}$ - an autarky is detected. So, $\mathcal{F}$ is satisfiability equivalent to $\mathcal{F}_{\text{initial}}$. To reduce the computational cost to solve $\mathcal{F}_{\text{initial}}$, we restart the DPLL procedure with $\mathcal{F}$. Although many autark assignments were found in various families, none of these detections resulted in significant performance gains. This disappointing result could be explained by the fact that nearly all autarkies were found on satisfiable instances.

6. Results and conclusions

Five enhancements are presented which were developed to increase the overall performance of march. All were illustrated using some experimental results showing their contribution of reducing the computational costs. For comparisons with other solvers we refer to the Sat competition pages[2].

The resulted version - march_dl - participated in the Sat 2005 competition. It was awarded with three silver and two bronze medals [11]. Unlike previous competitions, march_dl performed relatively good on industrial benchmarks too: It ended midway in the final ranking in that category. However, much progress is still required to make look-ahead based solvers competitive on these kind of structured instances.

References

[1] 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.

[2] A. Billionnet and A. Sutter, An efficient algorithm for the 3-Satisfiability problem. *Operations Research Letters* **12** (1992), 29–36.

[3] H. Connamacher, A random constraint satisfaction problem that seems hard for DPLL. In *Proceedings of SAT 2004*.

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

[5] O. Dubois and G. Dequen, source code of the kcnfs solver. Available at http://www.laria.u-picardie.fr/~dequen/sat/.

[6] J.W. Freeman, Improvements to Propositional Satisfiability Search Algorithms. Ph.D. thesis, Department of computer and Information science, University of Pennsylvania, Philadelphia (1995).

[7] O. Kullmann, Investigations on autark assignments. *Discrete Applied Mathematics* **107**(1-3) (2000), 99–137.

2. www.satcompetition.org

[8] M.J.H. Heule, J.E. van Zwieten, M. Dufour and H. van Maaren, March_eq: Implementing Additional Reasoning into an Efficient Lookahead Sat Solver. Springer-Verlag, *Lecture Notes in Comput. Sci.* **3542** (2005), 345–359.

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

[10] D. Le Berre and L. Simon, Fifty-five solvers in Vancouver: The sat 2004 competition. Springer-Verlag, *Lecture Notes in Comput. Sci.* **3542** (2005), 321–344.

[11] D. Le Berre and L. Simon, *Sat'05 competition homepage*. http://www.satcompetition.org/2005/.

[12] C.M. Li, A constraint-based approach to narrow search trees for satisfiability. *Information processing letters* **71** (1999), 75–80.

[13] C.M. Li and Anbulagan, Look-Ahead Versus Look-Back for Satisfiability Problems. In *Proceedings of CP 1997*, Springer-Verlag, *Lecture Notes in Comput. Sci.* **1330** (1997) 342–356.

[14] C.M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. of Fifteenth International Joint Conference on Artificial Intelligence* (1997), 366–371.

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