Effective Incorporation of Double Look-Ahead Procedures

Marijn Heule* and Hans van Maaren

Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Sciences
Delft University of Technology
marijn@heule.nl, h.vanmaaren@tudelft.nl

Abstract. We introduce an adaptive algorithm to control the use of the double look-ahead procedure. This procedure sometimes enhances the performance of look-ahead based satisfiability solvers. Current use of this procedure is driven by static heuristics. Experiments show that over a wide variety of instances, different parameter settings result in optimal performance. Moreover, a strategy that yields fast performance on one particular class of instances may cause a significant slowdown on other families. Using a single adaptive strategy, we accomplish performances close to the optimal performances reached by the various static settings. On some families, we clearly outperform even the fastest performance based on static heuristics. This paper provides a description of the algorithm and a comparison with the static strategies. This method is incorporated in march_dl, satz, and kcnfs. Also, the dynamic behavior of the algorithm is illustrated by adaptation plots on various benchmarks.

1 Introduction

Nowadays state-of-the-art satisfiability (SAT) solving shows two main solving architectures: conflict-driven and look-ahead driven. As tuned by the SAT competitions over the last years these two architectures seem to perform in an almost complementary way. The conflict-driven solvers dominate the so called industrial flavored problems (industrial category) while the look-ahead architecture dominates on random problems and problems with an intrinsic combinatorial hardness (part of crafted category). This paper deals with an engineering type of solver optimization with respect to one of the ingredients of look-ahead SAT solving.

The look-ahead architecture of (SAT) solvers has two important features: (1) It selects branching variables that result in a balanced search-tree; and (2) it detects failed literals to reduce the size of the search-tree. Many enhancements have been proposed for this architecture in recent years. One of the enhancements for look-ahead SAT solvers is the DOUBLELOOK procedure, which was

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

introduced by Li [7]. The usefulness of this procedure is straight forward: By also performing look-ahead on a second level of propagation, more failed literals could be detected, resulting in an even smaller search-tree.

By always performing additional look-aheads on the reduced formula, the computational costs rise drastically. One would like to restrict this enhancement in such a way that the overall computational time will decrease. Early implementations rely on restrictions based on static heuristics. Although these implementations significantly reduce the time to solve `random 3-`SAT formulas, they yield a clear performance slowdown on many structured instances.

We designed an algorithm for the DOUBLELOOK procedure that adapts towards the (reduced) CNF formula. Our algorithm has some key advantages: 1) Existing DOUBLELOOK implementations require only minor changes; 2) only one magic constant is used, which makes it easy to optimize the algorithm for a specific solver; and 3) this algorithm appears to outperform existing approaches.

In this paper, section 2 provides a general overview of the look-ahead architecture and zooms in on the DOUBLELOOK procedure. Section 3 deals with static heuristics for this procedure and their effect on the performance. Our algorithm is introduced in section 4 together with an alternative by Li. It offers detailed descriptions and motivates the decisions made regarding its design. Section 5 illustrates the usefulness and the behavior of the algorithm by experimental results and adaptation plots. Finally, we draw some conclusions in section 6.

2 Preliminaries

The look-ahead SAT architecture (introduced in [5]) consists of a DPLL search-tree [3] using a LOOKAHEAD procedure to reduce the formula and to determine a branch variable x_{branch} (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 $\mathcal{F}(l = 1)$.

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

```
1: if F = ∅ then
2:    return satisfiable
3: else if empty clause ∈ F then
4:    return unsatisfiable
5: end if
6: < F; x_branch > := LookAhead( F )
7: if empty clause ∈ F then
8:    return unsatisfiable
9: else if DPLL( F(x_branch = 1) ) = satisfiable then
10:   return satisfiable
11: end if
12: return DPLL( F(x_branch = 0) )
```

The effectiveness of the LookAhead procedure (see algorithm 2) depends heavily on the LookAheadEvaluation function which should favor variables that yield a small and balanced search-tree. Detection of failed literals could further reduce the size of the search-tree. Additionally, several enhancements are developed to boost the performance of Sat solvers based on this architecture.

One of these enhancements is the Preselect procedure, which preselects a subset of the variables (denoted by $\mathcal{P}$) to enter the look-ahead phase. By performing look-ahead only on variables in $\mathcal{P}$ the computational costs of the LookAhead procedure are reduced. However, this may result in less effective branching variables and less detected failed literals. All three solvers discussed in this paper, march_dl, satz, and kcnfs, use a Preselect procedure. Yet, their implementation of this procedure is different.

Another enhancement is the DoubleLook procedure (see algorithm 3), which was introduced by Li [7]. This procedure checks whether a formula resulting from a look-ahead on l is unsatisfiable - it detects l as a failed literal by performing additional look-aheads on the reduced formula. Since the computational costs of these extra unit-propagations are high, this procedure should not be performed on each reduced formula. In the ideal case, one would want to apply it only when the reduced formula could be detected to be unsatisfiable. This requires an indicator expressing the likelihood to observe a conflict.

Let $\mathcal{F}_2$ denote the set of binary clauses of formula $\mathcal{F}$. Li [7] suggests that the number of newly created binary clauses (denoted by $|\mathcal{F}_2 \setminus \mathcal{F}_2^*|$, with $\mathcal{F}_2^*$ referring to the set of binary clauses *before* the reduction) in the reduced formula is an effective indicator whether or not to perform additional look-aheads: If *many* new binary clauses are created during the look-ahead on a literal, the resulting formula is often unsatisfiable. In algorithm 3 the additional look-aheads are triggered when the number of newly created binary clauses exceeds the value of Δ_{trigger}. The optimal value of this parameter is the main topic of this paper.

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

```
1: P := Preselect( F )
2: for all variables x_i ∈ P do
3:    F' := DoubleLook( F(x_i = 0), F )
4:    F'' := DoubleLook( F(x_i = 1), F )
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:   else
12:      H(x_i) = LookAheadEvaluation( F, F', F'' )
13:   end if
14: end for
15: return < F; x_i with greatest H(x_i) >
```

Algorithm 3 DoubleLook($\mathcal{F}$, $\mathcal{F}^*$)

if empty clause $\in \mathcal{F}$ **then**
  return $\mathcal{F}$
end if
if $|\mathcal{F}_2 \setminus \mathcal{F}_2^*| > \Delta_{\text{trigger}}$ **then**
  for all variables $x_i \in \mathcal{P}$ **do**
    $\mathcal{F}' := \mathcal{F}(x_i = 0)$
    $\mathcal{F}'' := \mathcal{F}(x_i = 1)$
    if empty clause $\in \mathcal{F}'$ **and** empty clause $\in \mathcal{F}''$ **then**
      return $\mathcal{F}'$
    else if empty clause $\in \mathcal{F}'$ **then**
      $\mathcal{F} := \mathcal{F}''$
    else if empty clause $\in \mathcal{F}''$ **then**
      $\mathcal{F} := \mathcal{F}'$
    end if
  end for
end if
return $\mathcal{F}$

3 Static Heuristics

The DoubleLook procedure has been implemented in two look-ahead Sat solvers. Initially, Li proposed a static value for Δ_{trigger} [7]: In the first implementation in satz the DoubleLook procedure was triggered using $\Delta_{\text{trigger}} := 65$. (The latest version of satz uses a dynamic algorithm which will be discussed in the next section.) Dubois and Dequen use a variation in their solver kcnfs [4]: In their implementation, the DoubleLook procedure is triggered depending on the original number of variables (denoted by $\#vars$): $\Delta_{\text{trigger}} := 0.18\#vars$.

Both settings of Δ_{trigger} result from optimizing this parameter towards the performance on `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 clauses to trigger additional look-aheads. Even worse, on many other instances both Δ_{trigger} settings result in a pandemonium of additional look-aheads, which come down hard on the computational costs.

We selected a set of benchmarks from a wide range of families to illustrate these effects. We generated 20 `random 3-Sat` formulas with 350 variables with 1491 clauses (10 satisfiable and 10 unsatisfiable formulas) and used 10 `random 3color` instances from the SAT02 competition [9]. Additionally, we added some crafted and structured instances from various families:

- the `connamacher` family (generic uniquely extendible CSPs) contributed by Connamacher to Sat 2004 [2]. We selected those with $n = 600$ and $d = 0.04$;
- the `ezfact` family (factoring problems) contributed by Pehoushek. We selected the first three benchmarks of 48 bits from Sat 2002 [9];
- the `lksat` family, subfamily `l5k3` (random l-clustered k-Sat instances) contributed by Anton. Sat 2004 [10]. We selected all unsatisfiable instances;

- the `longmult` family (bounded model checking) contributed by Biere [1]. We used the instances of size 8, 10 and 12;
- the `philips` family (multiplier circuit) contributed by Heule to SAT 2004 [10];
- a `pigeon hole` problem (`phole10`) from `www.satlib.org`;
- the `pyhala braun` family (factoring problems) contributed by Pyhala Braun to SAT 2002 [9]. We selected the `unsat-35-4-03` and `unsat-35-4-04`, the two smallest instances from this family not solved during SAT 2002;
- the `stanion/hwb` family (equivalence checking problems) contributed by Stanion. We selected all three benchmarks of size 24 from SAT 2003 [6];
- SAT-encodings of `quasigroup` instances contributed by Zhang [11] We selected the harder unsatisfiable instances - `qg3-9`, `qg5-13`, `qg6-12`, and `qg7-12`.

Besides the random instances, all selected benchmarks are unsatisfiable to realize relatively stable performances. On most these families, the performance of look-ahead SAT solvers is strong[1] (compared to conflict-driven SAT solvers). We performed two tests: One that used constant numbers for Δ_{trigger} - analogue to early satz - and another used values depending on the original number of variables - analogue to kcnfs. For both tests we used the march_dl SAT solver[2]. 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 results of the first test are shown in table 1 and 2, for the low and high values of Δ_{trigger}, respectively.

Recall that satz uses $\Delta_{\text{trigger}} := 65$ - as a result of experiments on `random` 3-SAT instances. As expected, setting $\Delta_{\text{trigger}} := 65$ boosts performances on this family. However, instances from the `pyhala-braun` and `quasigroup` are hard to solve with this parameter setting: On these families the computational time can be reduced by 80% by changing the setting to $\Delta_{\text{trigger}} := 1500$. In general, we observe that a parameter setting which results in optimal performance for a specific family, yields far-from-optimal performances on other families.

Table 3 offers the results of the second test. On `random` 3-SAT optimal performance is realized by $\Delta_{\text{trigger}} := .20\#vars$: Indeed close to the setting used in kcnfs. However, none of the parameter settings result in close-to-optimal performances on all families. Moreover, the optimal performances on the families `3color`, `connamacher`, and `quasigroup` measured during the first test are about twice as fast as the optimal performances of the second test. So, all parameter settings used in the second test are far from optimal - at least for these families.

4 Adaptive DoubleLook

We developed an adaptive algorithm to control the DOUBLELOOK procedure. This algorithm updates Δ_{trigger} after each look-ahead in such fashion, that it adapts towards the characteristics of the (reduced) formula. This section deals with the decisions made regarding the algorithm. First and foremost - for reasons of elegance and practical testing - we focused on using only one magic constant.

[1] based on the results of the SAT competitions, see `http://www.satcompetition.org`
[2] available from `http://www.st.ewi.tudelft.nl/sat/`

Table 1. Performance of march_dl using various static (low) values for Δ_{trigger}.

family	0	10	30	65	100	150
3color (10)	118.69	39.91	**31.50**	62.87	67.96	70.42
anton (5)	276.74	269.00	184.73	119.99	80.31	**62.39**
connamacher (3)	5352.55	5407.50	4426.95	4373.89	4559.49	4852.63
ezfact48 (3)	650.01	451.55	287.67	321.70	264.79	187.93
longmult (3)	886.51	578.08	452.34	278.93	**219.35**	255.99
philips (1)	595.43	547.54	391.43	323.97	**273.99**	306.71
pigeon(1)	246.62	**140.05**	141.65	141.45	140.53	140.37
pyhala-braun(2)	4000.0	3024.49	2415.46	2019.37	1481.09	1224.92
quasigroup (4)	2351.98	2102.62	1649.78	1437.39	1362.80	1327.79
stanion (3)	2102.21	1661.80	**941.59**	971.29	964.34	972.18
random-sat (10)	157.12	136.95	96.04	**71.01**	75.44	86.09
random-uns (10)	322.68	285.80	199.03	**143.04**	156.70	178.00

Table 2. Performance of march_dl using various static (high) values for Δ_{trigger}.

family	250	400	600	850	1150	1500
3color (10)	67.26	70.26	70.24	70.49	72.21	73.52
anton (5)	64.09	73.28	75.02	75.07	77.22	78.98
connamacher (3)	4353.03	**2633.67**	2642.37	2861.83	4258.05	4099.12
ezfact48 (3)	69.87	**47.54**	55.78	57.16	54.56	51.91
longmult (3)	272.15	291.85	249.99	243.81	278.86	303.99
philips (1)	313.98	317.23	320.84	325.41	328.31	336.90
pigeon(1)	140.61	141.01	140.86	141.38	142.36	142.73
pyhala-braun(2)	1145.64	941.32	607.76	577.75	449.59	**428.26**
quasigroup (4)	1225.14	1011.26	849.64	507.18	455.84	**358.97**
stanion (3)	968.60	963.49	985.46	983.51	988.12	997.59
random-sat (10)	92.53	92.24	93.55	93.20	92.33	91.71
random-uns (10)	186.74	187.64	187.72	189.34	190.04	190.43

The algorithm has three components: (*i*) The Δ_{trigger} initial value, (*ii*) an increment strategy TriggerIncrease and (*iii*) a decrement strategy TriggerDecrease to update Δ_{trigger}. Both strategies consist of two parts: The location within the DoubleLook procedure and the size of the update value.

Regarding the first component: An effective initial value for Δ_{trigger} is probably as hard to determine as an effective global value for this parameter. Therefore, the algorithm should work on many initial values - even on zero, the most costly value at the root node. Hence our decision to initialize $\Delta_{\text{trigger}} := 0$.

The first aspect of the increment strategy is rather straight-forward: Assuming a strong correlation between the value of Δ_{trigger} and the detection of a conflict by the DoubleLook procedure, Δ_{trigger} should always be increased when the procedure fails to meet this objective. Algorithm 4 shows an adaptive variant of the DoubleLook procedure with the increment strategy located at line 17, the first position following a failure.

Table 3. Performance of march_dl using various static values for Δ_{trigger}. These static values are based on the original number of variables (denoted by $\#vars$).

family	.05 $\#vars$	.10 $\#vars$	.15 $\#vars$	.20 $\#vars$	.25 $\#vars$	.30 $\#vars$
3color (10)	**59.08**	67.98	70.19	67.08	68.06	65.87
anton (5)	146.13	83.24	62.67	**59.40**	64.19	67.15
connamacher (3)	4627.01	**4387.70**	4392.15	5078.09	4841.21	4807.81
ezfact48 (3)	324.14	202.17	61.19	50.75	**43.85**	47.64
longmult (3)	**205.46**	247.89	308.71	285.71	265.31	267.09
philips (1)	288.72	**285.43**	311.09	312.46	323.28	311.15
pigeon(1)	158.59	147.60	**142.02**	142.99	143.96	142.15
pyhala-braun(2)	1173.64	1095.74	753.08	590.00	546.79	**484.66**
quasigroup (4)	1473.65	1201.45	1035.91	1069.18	951.36	**837.54**
stanion (3)	1885.25	1110.04	**938.94**	949.83	956.62	973.57
random-sat (10)	118.50	88.57	72.86	**70.61**	71.55	75.97
random-uns (10)	254.60	185.96	155.18	**142.56**	150.69	165.46

The largest reasonable increment of Δ_{trigger} appears to make this parameter equal to the number of newly created binary clauses: Since no conflict was observed, Δ_{trigger} should be at least the number of new binary clauses ($|\mathcal{F}_2 \setminus \mathcal{F}_2^*|$) - which would have prevented the additional computational costs. The smallest value of the increment is a value close to zero and would result in a slow adaptation. The optimal value will probably be somewhere in between. We prefer a radical adaptation. For this reason we use the largest reasonable value:

$$\text{TriggerIncrease}() \ : \ \Delta_{\text{trigger}} := |\mathcal{F}_2 \setminus \mathcal{F}_2^*| \tag{1}$$

Algorithm 4 AdaptiveDoubleLook($\mathcal{F}$, $\mathcal{F}^*$)

```
1: if empty clause ∈ F then
2:    return F
3: end if
4: if |F2 \ F2*| > Δtrigger then
5:    for all variables xi ∈ P do
6:       F' := F(xi = 0)
7:       F'' := F(xi = 1)
8:       if empty clause ∈ F' and empty clause ∈ F'' then
9:          TriggerSuccess( )
10:         return F'
11:      else if empty clause ∈ F' then
12:         F := F''
13:      else if empty clause ∈ F'' then
14:         F := F'
15:      end if
16:   end for
17:   TriggerIncrease( )
18: else
19:   TriggerDecrease( )
20: end if
21: return F
```

Within the DoubleLook procedure, two events could suggest that Δ_{trigger} should be decreased[3] : (1) The detection of a conflict and (2) the number of newly created binary clauses is less than Δ_{trigger}. The first event seems the most logical: If the DoubleLook procedure detects a conflict, this is a strong indication that a slightly decreased Δ_{trigger} could increase the number of detected failed literals by this procedure. However, this may result in a deadlock situation: The increment strategy could update Δ_{trigger} such that no additional look-ahead will be executed, thereby making it impossible to decrease this parameter.

Placing the decrement strategy after the second event would guarantee that additional look-aheads will be executed every once in a while. Assuming that the computational time could diminish on all benchmarks by the DoubleLook procedure, then this location (algorithm 4 line 19) seems a more appealing choice.

How much should Δ_{trigger} be decreased if after a look-ahead the number of newly created binary clauses is less than this parameter? It seems hard to provide a motivated answer for this question. Therefore, we decided to obtain an effective value for the decrement using experiments.

These experiments were based on two considerations: First, the tests on static heuristics (see section 3) showed that effective parameter settings for Δ_{trigger} ranged from 10 to 1500. Therefore, the decrement should not be absolute but relative. So, it should be of the form $\Delta_{\text{trigger}} := c \times \Delta_{\text{trigger}}$ for some $c \in [0, 1]$.

Second, the size of preselected set $\mathcal{P}$ could vary significantly over different nodes. Therefore, the maximum decrement of Δ_{trigger} in each node depends on the size of $\mathcal{P}$. We believe this dependency is not favorable, so we decided to "neutralize" it. Notice that at most $2|\mathcal{P}|$ times in each node Δ_{trigger} could be decreased. Now, let parameter $\text{DL}_{\text{decrease}}$ denote the maximum relative decrement of Δ_{trigger} in a certain node. Then, combining these considerations, the decrement strategy could be formulated as follows:

$$\text{TriggerDecrease}() \ : \ \Delta_{\text{trigger}} := \sqrt[2|\mathcal{P}|]{\text{DL}_{\text{decrease}}} \times \Delta_{\text{trigger}} \tag{2}$$

The "optimal" value for parameter $\text{DL}_{\text{decrease}}$ is discussed in section 5.1.

The latest version of satz (2.15.2) also uses an adaptive algorithm: (*i*) It initializes $\Delta_{\text{trigger}} := .167\#vars$; (*ii*) it increases the Δ_{trigger} using the same TriggerIncrease() placed at the same location. The important difference lies in the location and size of (*iii*) the decreasing strategy: The algorithm realized the decrement at line 9 instead of line 19 of algorithm 4 - so Δ_{trigger} is only reduced after a successful DoubleLook call instead of slowly decrease after each look-ahead.

$$\text{TriggerSuccess}() \ : \ \Delta_{\text{trigger}} := .167\#vars \tag{3}$$

A drawback of this approach is that Δ_{trigger} could never be reduced to a value smaller than $.167\#vars$ - although we noticed from the experiments on static heuristics that significant smaller values are optimal in some cases (see table 3).

[3] Δ_{trigger} could also be decreased after lines 12 and 14 of algorithm 4: Each new forced literal on a second level of propagation increases the chance of hitting a conflict.

When a high value of Δ_{trigger} is optimal this approach might frequently alter between a relative low value ($\Delta_{\text{trigger}} := .167\#vars$) and a relative high value ($\Delta_{\text{trigger}} := |\mathcal{F}_2 \setminus \mathcal{F}_2^*|$) or result in the deadlock situation mentioned above.

5 Results

The adaptive algorithm as described above has been implemented in all look-ahead SAT solvers that contain a DOUBLELOOK procedure: march_dl, satz, and kcnfs. First, we show the effect of parameter $\text{DL}_{\text{decrease}}$ on the computational time. For this purpose, we use the modified march_dl. Second, the performance is compared between the original versions and the modified variants of satz and kcnfs. Third, the behavior of the algorithm is illustrated by adaptation plots. During the experiments we used the benchmarks as described in section 3.

5.1 The magic constant

The only undetermined parameter of the adaptive algorithm is $\text{DL}_{\text{decrease}}$. The computational times resulting from various settings for this parameter are shown in table 4. The data shows the effectiveness of the adaptive algorithm:

- Different settings for $\text{DL}_{\text{decrease}}$ result in comparable performances - generally close to the optimal values from the experiments using static heuristics.
- We observe that, for $\text{DL}_{\text{decrease}} := 0.85$, performances are realized for the `anton` and `philips` family that are nearly optimal, while on all the other families this setting outperforms all results using static heuristics.
- The optimal performances achieved by the adaptive heuristics are, on average, about 20% faster than those that are the result of static heuristics.

Table 4. Influence of parameter $\text{DL}_{\text{decrease}}$ on the computational time.

family	.75	.80	.85	.90	.95	.99
`3color` (10)	25.77	**25.39**	25.60	28.98	32.79	44.36
`anton` (5)	69.22	67.66	64.99	**63.26**	63.60	66.41
`connamacher` (3)	2258.59	2723.14	**1742.62**	3038.68	2872.84	4431.91
`ezfact48` (3)	39.00	**35.18**	37.87	38.66	38.68	46.08
`longmult` (3)	**197.29**	197.70	203.03	210.12	241.75	258.90
`philips` (1)	307.22	288.10	286.31	**267.17**	280.81	299.71
`pigeon`(1)	**99.31**	99.77	103.47	110.91	113.81	115.28
`pyhala-braun`(2)	369.49	**365.51**	372.98	366.89	376.89	405.05
`quasigroup` (4)	162.38	161.95	157.24	154.59	**150.63**	162.01
`stanion` (3)	**941.94**	946.38	950.44	965.30	984.20	1010.71
`random-sat` (10)	70.04	70.71	**69.21**	69.95	69.74	74.32
`random-uns` (10)	147.40	147.19	**145.95**	148.30	149.17	159.90

Table 5 shows the average values of Δ_{trigger} for various settings of $\text{DL}_{\text{decrease}}$. The average for each family is the mean of the averages of its instances, while

for each instance the average is the mean of the averages over all nodes. Because these values are not very accurate, we present only rounded integers.

Parameter $\mathrm{DL}_{\mathrm{decrease}}$ seems to have little impact on these average values. Note that - except for `pyhala-braun` and `quasigroup` instances - the average values of $\Delta_{\mathrm{trigger}}$ are very close to the optimal values shown in tables 1 and 2. In section 5.3 we provide a possible explanation for the two exceptions.

Table 5. Influence of parameter $\mathrm{DL}_{\mathrm{decrease}}$ on the average value of $\Delta_{\mathrm{trigger}}$.

family	.75	.80	.85	.90	.95	.99
`3color` (10)	23	24	25	28	33	42
`anton` (5)	129	134	141	162	176	220
`connamacher` (3)	538	575	589	527	462	292
`ezfact48` (3)	324	332	357	370	420	538
`longmult` (3)	76	78	80	90	100	127
`philips` (1)	99	102	107	110	117	142
`pigeon`	7	7	8	8	9	9
`pyhala-braun`(2)	105	108	112	117	127	148
`quasigroup` (4)	537	530	516	489	529	664
`stanion` (3)	21	22	23	25	29	36
`random-sat` (10)	57	58	62	67	77	98
`random-uns` (10)	57	59	62	67	78	98

5.2 Comparison

To test the general application of the adaptive algorithm, we also implemented it in both other SAT solvers that use a DOUBLELOOK procedure: satz and kcnfs. We modified the latest version of the source codes[4]. All three components were made according to the proposed adaptive algorithm: First, initialization is changed to $\Delta_{\mathrm{trigger}} := 0$. Second - only for kcnfs - a line is added to increase $\Delta_{\mathrm{trigger}}$ when no conflict is detected. Analogue to the march_dl and satz, $\Delta_{\mathrm{trigger}} := |\mathcal{F}_2 \setminus \mathcal{F}_2^*|$.

The third modification is implemented slightly differently, because in satz and kcnfs the size of the pre-selected set $\mathcal{P}$ is computed "on the fly". Therefore, $\sqrt[2^{|\mathcal{P}|}]{\mathrm{DL}_{\mathrm{decrease}}}$ would not be a constant value in each LOOKAHEAD procedure. As a workaround, we decided to use the average value of march_dl for $\sqrt[2^{|\mathcal{P}|}]{\mathrm{DL}_{\mathrm{decrease}}}$ instead. Additionally, from satz the decrement strategy TRIGGERSUCCESS is removed. While using $\mathrm{DL}_{\mathrm{decrease}} := 0.85$, this average appeared approximately 0.9985, which was used for an alternative decrement strategy:

$$\text{TRIGGERDECREASE}() \ : \ \Delta_{\mathrm{trigger}} := 0.9985 \times \Delta_{\mathrm{trigger}} \tag{4}$$

Notice that using value 1.0 instead of 0.9985 would drastically reduce the number of additional look-aheads, because $\Delta_{\mathrm{trigger}}$ would never be decreased.

[4] For satz we used version 215.2 (with the adaptive algorithm) which is available at http://www.laria.u-picardie.fr/~cli/satz215.2.c and for kcnfs we used the version available at http://www.laria.u-picardie.fr/~dequen/sat/kcnfs.zip

Table 6. Comparison between performances of the original and the modified versions of satz, kcnfs and march_dl.

	satz		kcnfs		march_dl	
family	original	modified	original	modified	prelim	final
`3color` (10)	52.71	**36.91**	37.89	**27.88**	72.51	**25.60**
`anton` (5)	183.97	**123.16**	3433.39	**2382.96**	80.75	**64.99**
`connamacher` (3)	> 6000	> 6000	4707.51	**4705.23**	4134.85	**1742.62**
`ezfact48` (3)	39.96	**32.98**	> 6000	> 6000	54.22	**37.87**
`longmult` (3)	2411.36	**1582.85**	440.34	**413.19**	265.88	**203.03**
`philips` (1)	1126.38	**710.75**	750.75	**443.27**	428.52	**286.31**
`pigeon`(1)	**23.72**	24.12	43.39	**40.25**	145.38	**103.47**
`pyhala-braun`(2)	1247.46	**881.91**	644.84	**466.92**	380.57	**372.98**
`quasigroup` (4)	172.40	**171.54**	230.59	**173.86**	351.85	**157.24**
`stanion` (3)	**3657.49**	3810.53	**3834.31**	3863.13	993.89	**950.44**
`random-sat` (10)	93.82	**92.56**	**79.63**	80.33	91.63	**69.21**
`random-uns` (10)	**260.13**	266.81	139.67	**138.22**	189.75	**145.95**

The performances of the original and the modified versions of satz, kcnfs, and march_dl are shown in table 6. The proposed adaptive algorithm generally outperforms the one in satz: On most instances from our test, the performance was improved up to 30%, while on the others only small losses were measured. Significant performance boosts are also observed in kcnfs, although the `stanion/hwb` instances are solved slightly slower. Since we did not optimize the magic constant, additional progress could probably be made.

The double look-ahead is the latest feature of march resulting in version march_dl. The preliminary version used has all features except the DoubleLookAhead procedure. The addition of this feature - using the proposed adaptive algorithm - boost the performance on the complete test set.

5.3 Adaptation plots

We selected four benchmarks (due to space limitations) to illustrate the behavior of the adaptive algorithm. For each benchmark, the first 10.000 (non-leaf) nodes of the DPLL-tree - using march_dl with $\Delta_{\text{trigger}} := .85$ - are plotted with a colored dot. Nodes are numbered in the (depth-first) order they are visited - so for the first few nodes their number equals their depth. The color is based on the depth of the node in the DPLL-tree. The horizontal axis shows the number of a certain node and the vertical axis shows the average value of parameter Δ_{trigger} in this node. These so-called *adaptation plots* are shown in figures 1, 2, 3 and 4.

In general, we observed that each family has its own kind of adaptation plot, while strong similarities between instances from different families were rare. For none of the tested instances Δ_{trigger} converged to a certain value, which is probably due to the design of the algorithm.

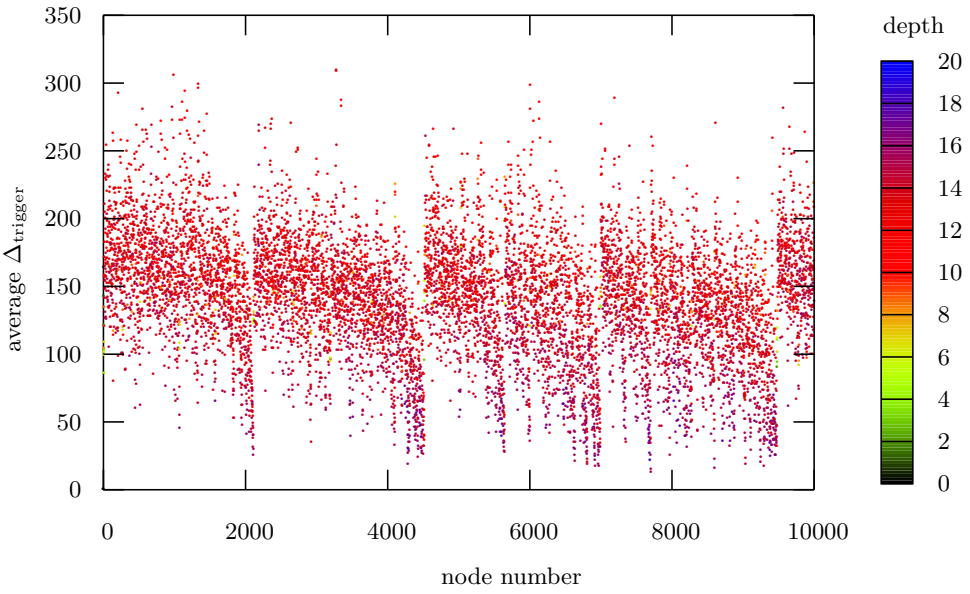

Fig. 1. Adaptation plot of `philips.cnf`

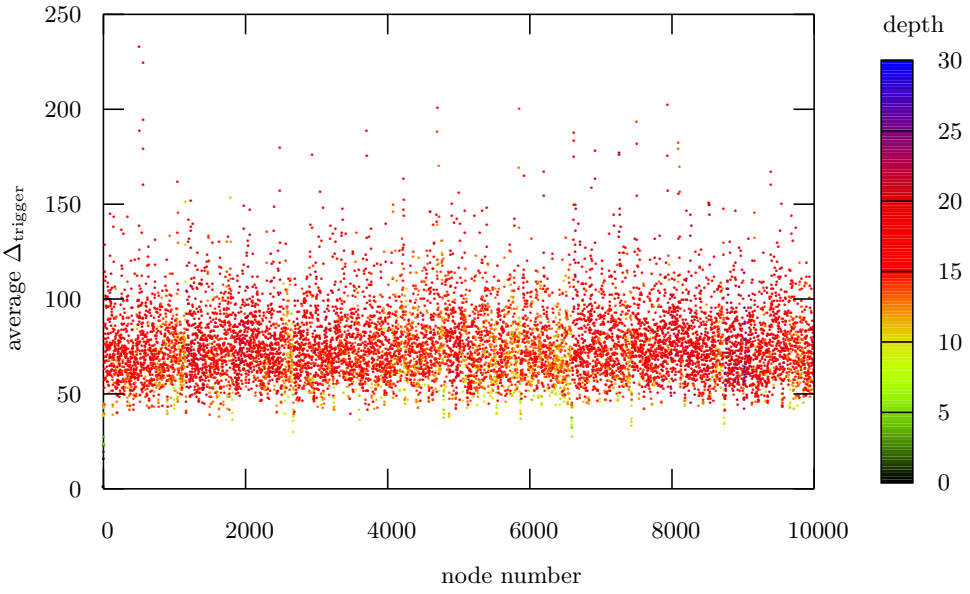

Fig. 2. Adaptation plot of a `random` 3-SAT instance with 350 variables and 1491 clauses.

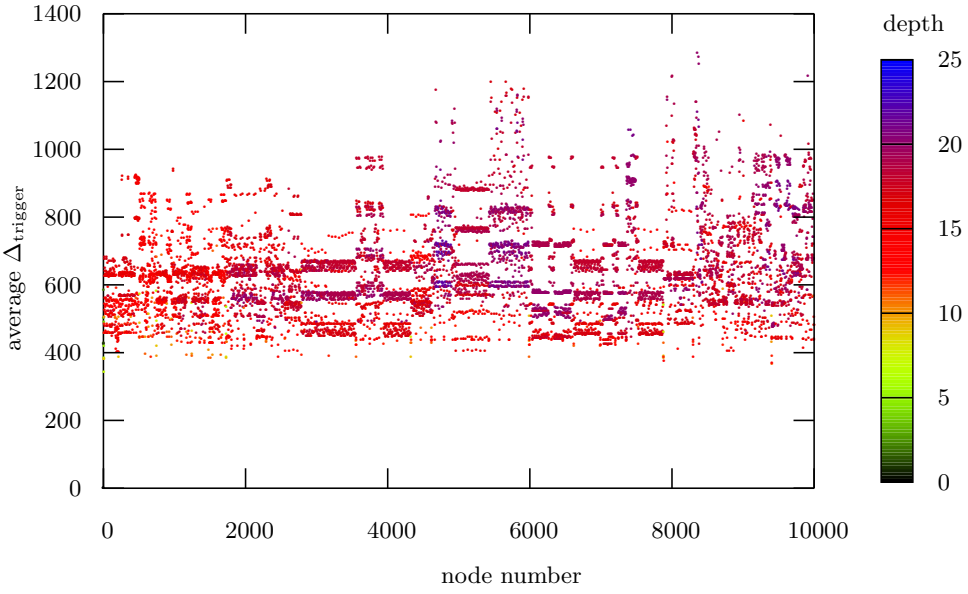

Fig. 3. Adaptation plot of `connm-ue-csp-sat-n600-d0.04-s1211252026.cnf`

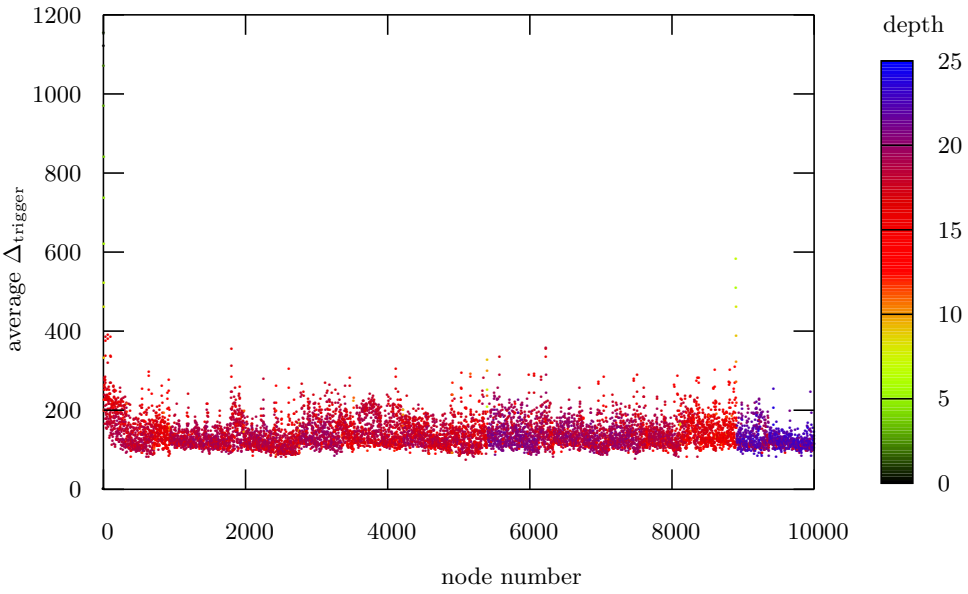

Fig. 4. Adaptation plot of `pyhala-braun-unsat-35-4-03.cnf`

For half of the families, the value of Δ_{trigger} tends to be above average at nodes near the root of the search-tree and / or tends to be below average at nodes near the leafs (see figure 1 and 4). For the other half of the families the opposite trend was noticed (see figure 2 and 3).

Recall that for `pyhala-braun` and `quasigroup` instances the average value for Δ_{trigger} was much lower than the optimum based on static heuristics. Figure 4 offers a possible explanation: Notice that nodes near the root use $\Delta_{\text{trigger}} \approx 1100$ while on average nodes use $\Delta_{\text{trigger}} \approx 100$. Adaptation plots for `quasigroup` instances showed a similar gap. A low static value for Δ_{trigger} will probably result in many additional look-aheads at the nodes near the root which could ruin the overall performance.

6 Conclusions

We presented an adaptive algorithm to control the DoubleLook procedure, which uses - like the static heuristic - only one magic constant. The algorithm has been implemented in all look-ahead Sat solvers that use a DoubleLook procedure. As a result of this modification, all three solvers showed a performance improvement on a wide selection of benchmarks. On macro level we observed that for most instances this algorithm approximates the family specific "optimal" static strategy, while on micro level the algorithm adapts to the (reduced) formula in each node of the search-tree.

References

1. A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, *Symbolic model checking without BDDs.* in Proc. Int. Conf. TACAS, Springer Verlag, LNCS **1579** (1999), 193–207.
2. H. Connamacher, *A random constraint satisfaction problem that seems hard for DPLL.* In the Proceedings of SAT 2004.
3. M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem proving.* Communications of the ACM **5** (1962), 394–397.
4. O. Dubois and G. Dequen, *source code of the kcnfs solver.* Available at `http://www.laria.u-picardie.fr/~dequen/sat/`.
5. J.W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms.* Ph.D. thesis, University of Pennsylvania, Philadelphia (1995).
6. D. Le Berre and L. Simon, *The essentials of the SAT'03 Competition.* Springer-Verlag, LNCS **2919** (2004), 452–467.
7. C.M. Li, *A constraint-based approach to narrow search trees for satisfiability.* Information processing letters **71** (1999), 75–80.
8. C.M. Li and Anbulagan. *Heuristics Based on Unit Propagation for Satisfiability Problems.* In Proc. of Fifteenth IJCAI (1997), 366–371.
9. L. Simon, D. Le Berre, and E. Hirsch, *The SAT 2002 competition.* Annals of Mathematics and Artificial Intelligence (AMAI) **43** (2005), 343–378.
10. L. Simon, *Sat'04 competition homepage.* `http://www.satcompetition.org/2004`
11. H. Zhang and M.E. Stickel, *Implementing the Davis-Putnam Method.* SAT2000 (2000), 309–326.