New Updating Criteria for Conflict-Based Branching Heuristics in DPLL Algorithms for Satisfiability

Renato Bruni, Andrea Santori
Università di Perugia - D.I.E.I.
Via G. Duranti, 93 - 06125 Perugia, Italy.
E-mail: `renato.bruni@diei.unipg.it, santori.andrea@libero.it`

Abstract

The paper is concerned with the computational evaluation and comparison of a new family of conflict-based branching heuristics for evolved DPLL Satisfiability solvers. Such a family of heuristics is based on the use of new scores updating criteria developed in order to overcome some of the typical unpleasant behaviors of DPLL search techniques. In particular, a score is associated with each literal. Whenever a conflict occurs, some scores are incremented with different values, depending on the character of the conflict. The branching variable is then selected by using the maximum among those scores. Several variants of this have been introduced into a state-of-the-art implementation of a DPLL SAT solver, obtaining several versions of the solver having quite different behavior. Experiments on many benchmark series, both satisfiable and unsatisfiable, demonstrate advantages of the proposed heuristics.

Keywords: Branching Rules, Conflict-Based Search Frameworks, Satisfiability

1 Introduction

A propositional formula $\mathcal{F}$ in *conjunctive normal form* (CNF) is a conjunction of clauses C_j, each clause being a disjunction of literals, each literal being either a positive (x_i) or a negative $(\neg x_i)$ propositional variable, with $j \in \{1, \dots, m\}$, $i \in \{1, \dots, n\}$. By denoting with I_j the set of variables of C_j, and with $[\neg]$ the possible presence of $\neg$, this is

$$\bigwedge_{j=1\dots m} \; (\bigvee_{i \in I_j} [\neg] \; x_i)$$

The *satisfiability* problem (SAT) consists in determining whether there exists a truth assignment in $\{0, 1\}$ (or equivalently in $\{$*False, True*$\}$) for the variables such that $\mathcal{F}$ evaluates to 1. Extensive references can be found in [6, 14, 23]. Many problems arising from different fields, such as artificial intelligence, logic circuit design and testing, cryptography, database systems, software verification, are usually encoded as SAT. Moreover, SAT carries considerable theoretical interest as the original NP-complete problem [7, 11]. From the practical point of view, this implies that many instances require an exponentially bounded computational time for their solution, but also that investing on the cleverness of the solution algorithm can result in very large savings in such computational times. The above has motivated a wide stream of research in practically efficient SAT solvers. As a consequence, many algorithms for solving the SAT problem have been proposed, based on different techniques (see for instance [8, 9, 12, 14, 17]). Computational improvements in this field are impressive, see e.g. [17, 22]. However, even if size and difficulty of the instances which can be solved are greatly increasing, also size and difficulty of the instances which are needed to be solved is greatly increasing (just to give an example, think about the case of microprocessor verification).

A solution method is said to be *complete* if it guarantees (given enough time) to find a solution if one exists, or prove lack of solution otherwise. Incomplete, or *stochastic*, methods, on the contrary, cannot guarantee finding the solution, although they may scale better than complete methods, mainly on large satisfiable problems. Most of the best complete solvers are based on so-called Davis-Putnam-Logemann-Loveland (DPLL) enumeration techniques. From the initial relatively simple DPLL backtracking algorithm described in [8], SAT solvers have evolved experimenting with several more sophisticated branching and backtracking frameworks, and eventually incorporating the best ones. Noteworthily examples of this have been non-chronological backtracking and conflict-driven clause learning [1, 19]. These techniques greatly improve the efficiency of DPLL algorithms, especially for structured SAT instances. Subsequently, a further generation of solvers paying special attention to implementation aspects appeared: SATO [25], Chaff [21], BerkMin [13] and several others, sometimes referred to as *chaff-like* solvers [17]. Such solvers nowadays appear to be the most competitive in solving real-world satisfiability problems.

As a matter of fact, a relevant influence on computational behavior is given by the *branching rule*, or *branching heuristic*, that is how to chose, at each branching, the next variable assignment. Different branching heuristics for the same basic algorithm may result in completely different computational results [20, 24]. Early branching heuristics (e.g. Böhm [4], MOM [14], Jeroslow-Wang [16]) have often been viewed as greedy trials of simplifying as much as possible the current subproblem, for instance by satisfying the most clauses. Such heuristics are based on *a priori* statistics on the instance, and have a certain effectiveness in the case of randomly generated problems. However, they usually cannot capture hidden problem structure, and real world problems typically are quite well structured. In order to tackle such problems, heuristics based on the history of the search, and in particular on the history of conflicts, have been proposed. Examples are VSIDS heuristic of Chaff [21], the adaptive branching rule of ACS [2], BerkMin decision making strategy [13], the dynamic selection of branching rules [15]. Conflict-based heuristics generally keep dynamically updated scores associated with variables. A central issue is then the policy for updating such scores. Recent studies on evolved scores updating techniques are reported also in [5] and in a preliminary version of present paper [3].

We report here a computational study of new scores updating criteria for conflict-based branching heuristics. Such criteria have been developed in order to overcome a part of the typical time-wasting behaviors of DPLL search techniques, as described in Section 2. In particular, a score is associated with each literal. Whenever a conflict occurs, some scores are incremented with different values, depending on the character of the conflict, as illustrated in detail in Section 3. The branching variable is then selected by using the maximum among those scores. Therefore, a new family of conflict-based branching heuristics for evolved DPLL Satisfiability solvers, called *reverse assignment sequence* (RAS), is obtained. Such heuristics have been introduced into a state-of-the-art implementation of a DPLL SAT solver, obtaining several versions of the solver having quite different behaviors, as described in Section 4. Experiments on many benchmark series, both satisfiable and unsatisfiable, show that the proposed branching heuristics are often able to improve solution times. Moreover, notwithstanding the fact that the introduced counters updating requires some computational overhead for its operations, total solution times on each series are always in favor of one of the new versions of the solver.

2 Motivations and Aims of New Updating Criteria

For DPLL-based algorithm, the search evolution is often represented as the exploration of a *search tree*, where each node subproblem is obtained by assigning a variable. The fact that SAT is an NP-complete problem implies that, for satisfiable instances, if one could choose at every node subproblem the correct truth assignment, that is the correct branch in the search tree, a satisfying solution would be obtained in a polynomial number of assignments [11]. Unfortunately, unless P=NP, it seems unlikely that some practical algorithm doing this in polynomial time may in general exist. Moreover, the problem of choosing at every node such an assignment for DPLL algorithms has been proven to be NP-hard as well as coNP-hard [18]. Therefore, the (heuristic) policy governing the choice of the variable assignments is generally called *branching heuristic*. Different branching heuristics may produce drastically different sized search trees for the same basic algorithm.

Conflict-based branching heuristics generally keep, for each variable x_i, a counter, or *score* s_i, or sometimes two counters, for the two possible truth assignments, or phases, of x_i. Score s_i is incremented when x_i is somehow involved in a *conflict*, i.e. an empty clause is derived by current truth assignments. Branching variables are selected according to the values of such scores. Counters are often periodically proportionally reduced, both for avoiding overflow problems, and for giving to earlier history of the search progressively less importance than recent history. For instance, zChaff [21] heuristic (called VSIDS, variable state independent decaying sum) uses for each variable two scores initialized to the number of occurrences of each literal in the instance. Whenever a new clause is *learned*, the counter of each of its literals is incremented by 1. The variable assignment corresponds to the literal having maximum score. Also the adaptive branching heuristic of ACS [2] uses a score for each clause, since it operates with a clause-based branching tree. The score of each clause is incremented by a penalty p_v each time an assignment aimed at satisfying that clause is made, and by another penalty p_f each time that that clause causes a conflict. The variable assignment is selected among literal contained in the unsatisfied clause having maximum score. BerkMin [13] heuristic uses one score for each variable. Whenever a conflict occurs, the scores of all variables contained in the clauses that are responsible for the conflict are increased by 1. The variable assignment corresponds to the literal whose variable has maximum score among those contained in the last learned clause that is unresolved.

Conflict-based branching heuristics have the advantages of requiring low computational overhead and of being often able to detect the hidden structure of a problem. They therefore generally produce good results on large real-world instances. The motivations of this can be explained by noticing that such heuristics try to avoid, or at least to postpone, the exploration of some regions of the search space which are likely to produce an unpleasant behavior of the DPLL search algorithm.

We therefore try to follow along this line and develop more evolved techniques for altogether avoiding other unpleasant phases of a DPLL search algorithm. There are in fact a number of situations that may denote that the search is passing through a non promising and time-wasting phase. Note that the simple occurrence of such situations cannot guarantee that the search is exploring a useless region of the search space. Therefore, such phases cannot be just forbidden, or the search would become incomplete. Our aim is to avoid them, or at least postpone them, in order to tackle them only when no better option is available. We propose, in particular, techniques for avoiding the unpleasant search phases denoted by the three situations described below, and also illustrated in Fig. 1. In the following description, let the h-th *level* of the search tree be the set of nodes the search tree having the same search tree depth h. We will speak intuitively of first levels, i.e. the nearest ones to the root, and of low levels, i.e. the most distant ones from the root.

i) A first situation denoting an unpleasant phase is having many backtracks at the low levels of the search tree (Fig. 1 part i). If indeed backtracks could be moved all at the very first levels of the search tree, either unsatisfiability would be detected much earlier, or a satisfiable solution would be reached within a very limited number of useless variable assignments.

ii) A second situation (Fig. 1 part ii) denoting an unpleasant phase is the repetition, in different branches of the search tree, of the same sequence of variable assignments leading to a conflict (e.g. $\ldots\ x_i = v_i,\ x_j = v_j,\ x_k =$

$v_k,\ x_l = v_l$). Conflict clause learning can only avoid, each time, the repetition of the last assignment of such a sequence, but, without some adaptive heuristic, it does not prevent the search to move again in the same direction (e.g. $\ldots\ x_i = v_i,\ x_j = v_j,\ x_k = v_k$). Although this search phase cannot be forbidden without making the search incomplete, it would be preferable to avoid it as far as it is possible.

iii) Finally, it may often happen that some of the variables of an instance are related in such a way that, for large portions of the branching tree, a conflict is obtained always at about the same decision level and due to a small set of variables (Fig. 1 part iii). Such phase is clearly time-wasting and should be avoided, even if, again, it cannot be forbidden.

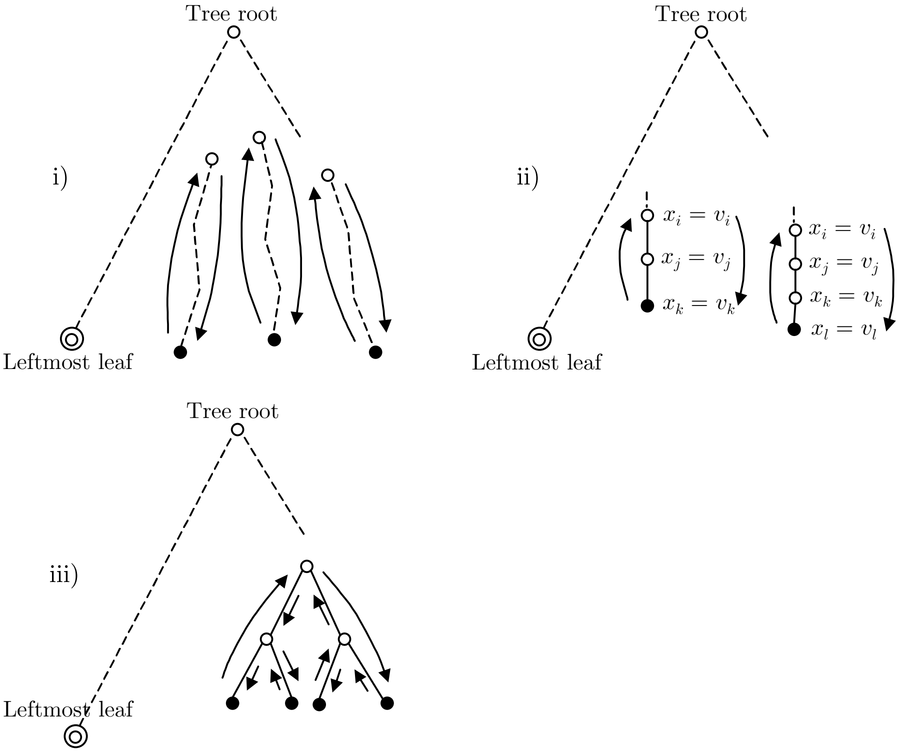

Figure 1: Representation of the described unpleasant search situations for a DPLL algorithm. Nodes corresponding to subproblems where an empty clause is derived, hence backtrack is performed, are represented in black. Search trees are represented in such a way that their exploration chronologically proceeds from right to left.

The above three aims can be pursued by using the scores updating mechanism. Since in fact the branching decision is taken on the basis of the maximum among such scores, by incrementing them in a suitable way we would be able to guide the search in order to avoid, but not forbid, the above phases. Note that such a list of situations denoting phases that should be avoided during the search, but cannot be forbidden without making the search incomplete, could also be enriched, still remaining in the proposed algorithmic framework.

3 The Proposed Updating Criteria

For each variable $x_i,\ i \in \{1, \ldots, n\}$, we use two counters, or scores, s_i^0 and s_i^1 for the two possible phases of x_i. Counters are therefore associated with the two possible literals $v_0(x_i) = \neg x_i$ and $v_1(x_i) = x_i$. When branching is needed, we assign, as usual, variable x_i at value $v \in \{0, 1\}$ by choosing the maximum score, as follows.

$$x_i = v \ \text{ such that } \ s_i^v = \max\{s_1^0, s_1^1, \ldots, s_n^0, s_n^1\}$$

Similarly to other conflict-based heuristics, scores are initialized to the number of occurrences of each literal in the instance, and periodically proportionally reduced. The main issue clearly is how scores $\{s_1^0, s_1^1, \ldots, s_n^0, s_n^1\}$ are incremented.

In order to pursue the above point i), we try to assign at first the more *difficult* variables, in the sense of the more constrained ones. This because, when assigning them in the upper levels of the search tree, either we should discover unsatisfiability earlier, or we should remain with only *easy* variables to assign in the lower levels of the search tree, and therefore little backtrack should be needed there. Whenever a new *learned clause* $C_l = \{v(x_{l1}), \ldots, v(x_{lh})\}$ is added to the clause set by effect of a conflict, what we have actually discovered is that variables $\{x_{l1}, \ldots, x_{lh}\}$ contained in C_l are a bit more constrained than other variables. In fact, C_l represents just an explicitation of such constraint, that is already implied by the original clauses. Therefore, we increment the scores of those literals by a *penalty* for learning p_l, as follows:

$$s_i^v \leftarrow s_i^v + p_l, \quad \forall v(x_i) \in C_l$$

(where $a \leftarrow a + b$ means that new value of a is obtained by adding b to its old value). The effect can also be viewed as trying to satisfy C_l. Note that, so far, this is also zChaff's policy.

Moreover, in order to pursue the above point ii), we try to reverse every sequence of assignments which leads to a conflict. Whenever a sequence of assignments produces an *empty clause*, this sequence is at risk of being repeated again in the search tree, leading again to the same conflict. The use of learned clauses, together with the increment of the scores of their literals, can only partially solve the problem. We therefore try to satisfy the *failed clause* $C_f = \{v(x_{f1}), \ldots, v(x_{fk})\}$ (the clause which has become empty) by incrementing the scores of its literals by a penalty for failure p_f, as follows:

$$s_i^v \leftarrow s_i^v + p_f, \quad \forall v(x_i) \in C_f$$

After doing so, the subsequent assignments would be different, thus preventing the repetition of the above conflicting sequences of assignments. However, since increasing scores has a cost, and moreover implies an even higher cost for reordering the scores in order to choose the higher value, we consider also the possibility of applying some simplifications to the above algorithm. In fact, adding p_f to only one of the counters corresponding to the literals of the failed clause C_f, and in particular to the last assigned literal except the conflicting literal, decreases computational overhead while maintaining most of the positive features. Several other alternatives were tested, but the above proposed one appears more stable, in the sense of producing good results on different types of problems.

Finally, in order to pursue the above point iii), we would like to avoid frequent backtracks due to the same conflicting literal $v(x_f)$ at the same decision level d. We therefore keep in memory the set of the last c conflict literals and their corresponding levels, obtaining the set of couples $M = \{(v(x_{f1}), d_{q1}), \ldots, (v(x_{fc}), d_{qc})\}$. Whenever a new conflict occurs due to literal $v(x_f)$ at decision level d_q, if the couple $(v(x_f), d_q)$ is already contained in M, we increment the score of the direct conflicting literal by a penalty p_d, and the score of the negation of the conflicting literal by a penalty p_n, as follows:

$$\left\{ \begin{array}{l} s_f^v \leftarrow s_f^v + p_d \\ s_f^{\neg v} \leftarrow s_f^{\neg v} + p_n \end{array} \right. \quad \text{if} \quad \left\{ \begin{array}{l} v(x_f) \text{ conflicts at level } d_q \\ \text{and already } (v(x_f), d_q) \in M \end{array} \right.$$

There are in fact reasons for increasing the score of the conflicting literal $v(x_f)$, and also reasons for increasing the score of the negation of the conflicting literal $\neg v(x_f)$. This is because, in the absence of further information, it should be convenient to try to assign such a variable at an upper decision level, and, moreover, both its values may reveal to be useful since they both were “needed”. Since, however, increasing the two counters has a relatively high computational cost, we also consider the possibility of increasing only the counter of the conflicting literal $v(x_f)$. We will briefly refer to the above operation as “frequent conflicting literals detection”.

The following example illustrates in detail the counters updating performed after a typical conflict.

Example 3.1. Consider an instance $\mathcal{F}$ containing, among others, the clauses:

$$C_a = (\neg x_1 \vee x_3 \vee x_5) \qquad C_b = (x_2 \vee \neg x_4 \vee \neg x_5)$$

Imagine that $\{x_1$ to 1, x_2 to 0, x_3 to 0 and x_4 to $1\}$ have already been assigned, and that a conflict due to x_5 at the same decision level d where the search currently is has already occurred within the last c conflicts, hence $(x_5, d) \in M$. We now have C_a reduced to a unit clause, which forces assigning $\{x_5$ to $1\}$. So far C_b becomes empty, and we learn $C_l = (\neg x_1 \vee x_2 \vee x_3 \vee \neg x_4)$, while C_f is in this case C_b and the conflict literal is x_5. Therefore, scores corresponding to all literals of the learned clause C_l are increased by p_l, scores corresponding to all literals of the failed clause C_b are increased by p_f, score corresponding to the conflict literal x_5 is increased by p_d and score corresponding to the negation of the conflict literal $\neg x_5$ is increased by p_n. Updating is as follows:

$$\begin{array}{ll} s_1^0 \leftarrow s_1^0 + p_l & s_2^1 \leftarrow s_2^1 + p_l + p_f \\ s_3^1 \leftarrow s_3^1 + p_l & s_4^0 \leftarrow s_4^0 + p_l + p_f \\ s_5^1 \leftarrow s_5^1 + p_d & s_5^0 \leftarrow s_5^0 + p_f + p_n \end{array}$$

4 Computational Analysis

The described heuristics were implemented in the state-of-the-art DPLL solver zChaff [21, 10], obtaining several solver versions. Parameters are chosen in order to cross combinations. In particular, for what concerns the following tables,

- 'zChaff' is the original version of zChaff 2004 [10];
- 'zCh1' is the version incrementing all literals of learned clauses using $p_l = 1$ and frequent conflicting literals (not their negations) using $c = 2$ and $p_d = 2$;
- 'zCh2' is the version incrementing all literals of learned clauses using $p_l = 1$ and frequent conflicting literals and their negations using $c = 2$, $p_d = 2$ and $p_n = 2$;
- 'brChaff' is the version incrementing all literals of learned clauses using $p_l = 1$ and the last literal of failed clauses except the conflicting literal using $p_f = 2$;
- 'brCh1' is the same as 'brChaff' but also incrementing frequent conflicting literals using $c = 2$ and $p_d = 2$;
- 'brCh2' is the same as 'brChaff' but also incrementing frequent conflicting literals and their negations using $c = 2$, $p_d = 2$ and $p_n = 2$;
- 'bChaff' is the version incrementing all literals of learned clauses using $p_l = 1$ and all literals of failed clauses using $p_f = 2$;
- 'bCh1' is the same as 'bChaff' but also incrementing frequent conflicting literals using $c = 2$ and $p_d = 2$;
- 'bCh2' is the same as 'bChaff' but also incrementing frequent conflicting literals and their negations using $c = 2$, $p_d = 2$ and $p_n = 2$.

Note that zChaff 2004 may also use, for a limited number of times, other branching heuristics in addition to the classical VSIDS one. Our branching heuristics substituted completely the VSIDS one and only that one. Experiments are conducted on a 2.5GHz Intel Celeron PC with 512MB RAM and using MS VC++ compiler. Note also that some libraries may be different using other compilers, therefore results may vary (we experienced it) but maintaining about the same average results on each series.

We report, in the first line of each box of the tables, running times in CPU seconds. Time limit was set at 3600 sec. (1 hours), when exceeded we report

"-". Total solution times are obtained by counting each time-out as 3600 sec, except for problems not solved by any solver (global time-outs), which are not counted in the totals. Since the total for solvers incurring in non-global time-outs is actually a lower bound, we denote this by writing a > before the value. We report in bold face the best total time. We also report, in the second line of each box of the tables, the number of decisions, that is how many times the solver needs to select a variable and to assign it. Assignments which are just forced consequences of such decisions (e.g. unit propagation) are not counted as decisions themselves. The total number of decisions are obtained by counting each time-out as the maximum among the numbers of decisions made by the other solvers which solved the time-outed problem, except for the problems which are not solved by any solver, which are not counted in the totals.

The considered benchmark series were provided by different authors to the SAT community and are now publicly available. The majority of them were used as benchmarks in recent SAT Solver Competitions (see [22], both for benchmark details and for past and probably future results of other solvers on them). Most of the considered series are real-world problems, therefore structured, but we also considered one randomly generated series. The series are either all satisfiable, or all unsatisfiable, or mixed.

As a general remark, notwithstanding the fact that the introduced counter updating techniques require a computational overhead for its operations compared to the original zChaff branching heuristic (especially for the detection of frequent conflicting literals), computational times often decrease, proving the algorithmic effectiveness of the proposed updating criteria. Moreover, our experiments fully confirm that the branching rule has a very relevant influence

Barrel	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
barrel2	U	0.01 3	0.00 3	0.00 3	0.00 5	0.00 5	0.01 5	0.00 5	0.00 5	0.01 5
barrel3	U	0.02 154	0.01 127	0.02 101	0.02 119	0.02 119	0.02 119	0.02 192	0.02 152	0.02 228
barrel4	U	0.08 197	0.07 197	0.09 197	0.07 182	0.07 182	0.07 182	0.07 368	0.07 368	0.09 368
barrel5	U	3.60 11248	3.05 10832	2.78 9882	2.43 9083	2.34 9826	2.12 10596	1.55 8172	1.55 7860	1.86 7354
barrel6	U	18.60 38816	14.91 34104	13.32 35280	11.74 31381	13.62 36311	12.43 32159	9.70 24727	11.39 29779	10.85 27057
barrel7	U	35.45 54429	51.89 84568	26.03 57035	27.15 49911	25.32 54894	32.63 57721	16.93 46357	14.67 47706	14.90 47433
barrel8	U	227.03 180853	189.79 199608	127.08 143429	118.54 149886	164.93 174569	122.45 124932	74.81 122492	66.15 136210	80.40 128030
barrel9	U	152.40 417906	167.04 438875	133.79 368223	134.11 347597	130.56 365419	126.79 342494	98.95 282883	88.87 255763	103.51 282827
Total		437.19 703606	426.76 768314	303.11 614150	294.08 588164	336.87 641325	296.52 568208	202.04 485196	**182.73** 477843	211.62 493302

Table 1: Comparison on bounded model checking problems.

Des-encryption	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
cnf-r3-b1-k1.1	S	7.40 28871	13.01 41410	4.68 21088	11.30 40471	8.48 28927	7.79 24138	11.32 35820	10.86 29596	6.61 23334
cnf-r3-b1-k1.2	S	4.34 10755	9.01 25464	4.05 9449	16.49 29682	6.06 14819	20.10 46061	4.26 10287	16.42 40170	13.89 36021
cnf-r3-b2-k1.1	S	0.92 1328	0.74 862	0.77 1299	0.50 638	0.76 1236	0.75 899	1.16 1465	0.84 1042	0.72 994
cnf-r3-b2-k1.2	S	2.05 1243	2.17 2147	2.67 2034	1.75 1317	1.04 656	2.14 1378	3.97 3223	1.77 1240	1.48 915
cnf-r3-b3-k1.1	S	1.17 1129	0.57 577	0.70 890	1.35 1196	1.25 1217	1.82 1531	1.00 741	1.35 1063	1.10 1016
cnf-r3-b3-k1.2	S	1.77 538	2.03 567	1.91 818	2.18 890	2.33 798	1.63 447	1.05 302	2.04 650	1.76 518
cnf-r3-b4-k1.1	S	0.91 497	1.04 607	1.12 513	1.04 706	0.95 415	1.75 1239	1.23 491	1.32 573	1.25 562
cnf-r3-b4-k1.2	S	2.66 640	2.04 421	1.86 557	1.66 242	2.05 477	3.03 631	1.85 341	2.03 326	2.66 675
cnf-r4-b1-k1.1/.2	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
cnf-r4-b2-k1.1	S	- -	1010.48 2625565	- -	1270.94 2901880	3348.77 5340918	- -	1610.48 3682226	- -	- -
cnf-r4-b2-k1.2	S	- -	- -	- -	1026.18 2221151	2174.86 4008092	- -	2095.12 3332414	- -	1656.77 2894610
cnf-r4-b3-k1.1	S	705.20 1768757	1314.44 2292924	- -	1482.11 2445872	2611.29 3967580	- -	667.92 1348190	1556.71 3401421	1150.48 2217073
cnf-r4-b3-k1.2	S	647.39 981196	- -	1551.82 2390076	558.12 1107932	- -	- -	2067.51 3169485	693.69 1053167	252.23 451757
cnf-r4-b4-k1.1	S	422.61 816444	1361.78 2179918	2624.32 3913826	767.18 1402128	1663.30 2083653	1060.80 1769687	1106.74 1428725	1348.00 1683870	1041.60 1596734
cnf-r4-b4-k1.2	S	647.29 657150	776.89 1003928	316.88 432733	577.05 716099	490.60 631319	977.68 1025376	342.22 465776	155.54 211683	494.75 655010
Total		> 9643.71 > 14950384	> 11694.20 > 18856226	> 15310.78 > 22796037	**5717.85** 2193369	> 13911.74 > 21431025	> 16477.49 > 24195059	7915.84 13479486	> 10990.57 > 17106637	> 8225.30 > 13319227

Table 2: Comparison on data encryption problems.

on computational behavior: small modifications in it may cause completely different computational results. Versions incrementing literals of failed clauses tend to be good compromises between speed and stability. On the other hand, versions incrementing frequent conflicting literals tend to be less stable: sometimes they are the fastest, but they are often the slowest on easy instances due to their heavier computational load.

FVP 2.0	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
3pipe	U	3.68 19628	3.80 18693	4.15 20080	5.52 21680	4.76 22070	4.04 21043	4.20 17454	3.98 18425	4.43 19436
3pipe_1_ooo	U	2.30 12192	3.11 14849	2.59 12635	2.67 12922	2.56 13235	3.14 16170	2.94 12271	2.93 14327	2.61 12827
3pipe_2_ooo	U	3.81 15274	5.57 17929	4.55 16551	5.16 17438	5.25 18359	4.91 18225	5.52 16421	5.03 17190	4.73 16412
3pipe_3_ooo	U	6.13 23080	6.01 22783	5.54 19648	5.73 22853	4.81 17715	7.06 21979	5.23 19429	5.20 20288	5.72 20753
4pipe	U	27.61 129609	21.14 111765	22.32 100785	23.30 96440	34.47 107298	24.97 110342	25.80 100481	21.61 104631	21.23 95646
4pipe_1_ooo	U	27.01 78558	28.80 89836	27.68 82552	36.22 113272	24.38 89511	29.80 95661	27.04 72697	29.10 90805	27.96 100942
4pipe_2_ooo	U	35.93 106541	27.65 93181	36.88 118288	28.26 98158	37.47 106880	38.54 108101	35.21 100979	45.17 114632	47.08 112418
4pipe_3_ooo	U	32.23 113404	31.39 124496	30.38 108679	24.27 99329	33.73 129273	35.99 130067	31.45 112883	33.10 120171	33.71 122940
4pipe_4_ooo	U	37.36 128679	36.68 126102	37.96 132114	37.12 115851	39.08 132240	38.09 135063	39.44 125913	40.44 129635	40.38 142974
5pipe	U	33.54 203587	31.92 200877	33.08 209056	34.79 214131	35.62 220432	32.09 199249	31.54 204618	33.13 210301	32.34 202836
5pipe_1_ooo	U	91.96 204155	99.08 243868	84.16 194285	91.09 223536	88.08 205210	95.00 226436	91.70 215116	73.23 195158	87.09 207876
5pipe_2_ooo	U	79.72 179907	90.37 226359	90.46 224927	81.42 198077	91.51 224617	81.75 211995	93.76 229906	90.22 212275	87.62 216447
5pipe_3_ooo	U	88.62 217160	77.94 211042	95.75 233444	85.51 218758	90.77 218790	101.31 237086	76.98 186882	98.72 263911	91.13 212417
5pipe_4_ooo	U	166.98 430352	171.78 439102	169.43 440374	176.46 468466	177.69 443954	176.30 434125	183.19 451661	164.79 410764	185.20 474555
5pipe_5_ooo	U	95.38 237306	102.01 270781	108.58 288828	102.35 250826	92.44 241146	89.26 229072	91.43 235347	96.88 247624	97.21 236855
6pipe	U	288.37 841057	324.90 881971	329.82 837488	268.67 790432	289.89 925736	228.32 796393	280.19 863981	290.11 827799	261.05 705900
6pipe_6_ooo	U	436.94 749135	464.71 801359	540.42 932423	533.04 889996	412.12 803188	520.30 931822	519.82 919249	541.46 935302	493.96 892079
7pipe	U	824.51 1541933	1029.76 1887586	669.83 1868722	719.03 1990022	674.44 1676084	671.30 2039544	697.07 1845979	710.61 2094339	920.16 1805369
7pipe_bug	S	21.96 143775	559.34 1320204	16.92 129570	390.81 1179127	510.53 1302818	5.19 45325	3.86 34490	3.90 34491	3.83 34487
Total		2301.74 5375332	3112.85 7102783	2307.91 5970449	2648.75 7021310	2647.04 6898556	**2184.21** 6007698	2243.63 5765757	2286.88 6062068	2444.83 5633169

Table 3: Comparison on hardware verification problems.

Miters	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
c1355-s	U	1.05 8692	1.16 9061	1.29 10345	1.24 8682	1.64 10022	1.43 9057	0.96 8180	1.37 9313	1.29 9531
c1355	U	1.19 8792	1.35 9930	0.96 7958	5.23 18452	1.26 7891	1.66 10144	1.11 8719	2.02 12156	1.06 8865
c1908-s	U	2.06 9634	1.92 9212	2.37 11125	2.04 9954	2.92 11970	5.03 17665	2.32 10122	3.51 12622	2.35 9714
c1908	U	2.63 9910	2.30 9732	2.38 9635	1.81 8153	2.77 11507	2.13 9625	2.19 9751	2.78 11750	2.93 10810
c1908_bug	S	1.71 8766	2.04 9400	2.83 12023	2.48 10464	2.11 9274	2.52 10335	1.63 8551	1.51 7890	2.73 11788
c2670-s	U	2.86 19780	2.83 20161	2.67 19116	2.61 18909	2.12 16464	2.79 18693	2.99 19531	3.25 22893	3.62 23365
c2670	U	1.50 15121	2.97 20801	2.22 18411	2.41 16634	2.33 17753	2.10 16839	1.97 16889	2.15 17672	2.46 19330
c2670_bug	S	0.04 1223	0.04 1223	0.05 1223	0.38 4814	0.30 4660	0.35 4058	0.19 4731	0.19 4718	0.25 5944
c3540-s	U	73.34 92500	45.08 60189	57.67 74185	68.86 83350	63.15 81780	76.64 91941	56.52 73214	44.62 63795	68.47 83003
c3540	U	63.56 79945	52.63 74435	67.17 76123	54.98 75548	63.59 74249	50.33 67219	83.54 91694	75.54 85670	49.83 63814
c3540_bug	S	0.01 50	0.01 50	0.01 50	0.01 50	0.01 50	0.01 50	0.01 50	0.01 50	0.01 50
c432-s	U	0.08 1352	0.08 1395	0.11 1417	0.12 1396	0.10 1370	0.10 1463	0.08 1412	0.08 1372	0.09 1409
c432	U	0.10 1446	0.09 1440	0.09 1378	0.08 1374	0.11 1495	0.08 1389	0.07 1113	0.08 1177	0.11 1602
c499-s	U	0.50 8111	1.59 13501	0.54 7213	0.67 8452	0.67 8810	1.63 14866	0.67 9356	1.21 10357	1.00 11526
c499	U	1.13 12213	0.63 9743	1.91 14880	1.28 12833	0.66 8768	1.22 11826	0.89 11198	0.95 10009	0.73 8831
c5315-s	U	20.21 96129	23.41 99753	22.17 97248	23.08 102289	25.19 104273	23.14 100821	22.84 95347	26.78 104671	22.91 100992
c5315	U	21.29 94999	23.39 103893	23.92 100395	21.44 92809	23.57 94978	23.95 98990	25.90 111641	21.35 97445	18.01 84440
c5315_bug	S	1.29 11723	1.04 11782	0.28 5513	0.56 6521	0.66 5892	2.12 24283	0.88 17809	1.86 23710	1.56 24451
c6288-s	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
c6288	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
c7552-s	U	52.32 198656	59.71 224167	55.76 210143	54.88 201799	51.16 200505	51.97 195630	59.33 213695	51.38 193681	64.78 227098
c7552	U	54.74 193567	54.69 200748	52.52 193449	56.89 209101	53.73 200137	45.91 183096	51.26 195410	53.43 205476	45.35 169670
c7552_bug	S	3.11 31040	2.07 19863	1.47 16474	1.62 19727	0.57 9843	0.48 8832	1.47 17264	0.64 8474	0.64 9537
c880-s	U	1.03 7299	1.19 7850	0.67 5943	1.10 7600	1.00 7309	1.14 7787	1.19 8444	1.38 9024	0.98 7815
c880	U	1.04 7299	1.15 7850	0.68 5943	1.14 7600	1.04 7309	1.18 7787	1.18 8444	1.39 9024	1.02 7815
Total		306.55 918247	**280.63** 926179	299.47 900190	302.84 926511	297.88 896309	296.66 912396	321.34 942565	290.12 922949	285.68 901400

Table 4: Comparison on combinational equivalence checking problems.

Effects are however quite different on the various benchmark series. In particular, on the Barrel series (bounded model checking problems) the versions incrementing all literals of learned clauses and all literals of failed clauses (bChaff, bCh1, bCh2) are the fastest, and advantages are quite uniform and stable. On the Des-encryption series (data encryption problems) the version incrementing all literals of learned clauses and the last literal of failed clauses except the conflicting literal (brChaff) is by far the fastest. However advantages of the proposed techniques are not uniform. On the contrary, on the FVP series (hardware verification problems) running times are quite similar, and the proposed techniques produce more uniform results. The fastest is in this case the version incrementing all literals of learned clauses, the last literal of failed clauses except the conflicting literal, and frequent conflicting literals and their negations (brCh2). On the Miters series (equivalence checking problems) the version incrementing all literals of learned clauses and frequent conflicting literals is the fastest (zCh1), but running times are relatively similar. On the Quasigroup series (latin squares logical problems) running times are again quite similar, although the version incrementing all literals of learned clauses and the last literal of failed clauses except the conflicting literal (brChaff) is again the fastest. On the Ferries series (industrial planning problems from the 2005 SAT Competition) the version incrementing all literals of learned clauses, all literals of failed clauses and both frequent conflicting literals and their negations (bCh2) is by far the fastest, even if results of the various versions are here quite different. On the VMPC inversion series (open cryptographic problems from the 2005 SAT Competition) the version incrementing all literals of learned clauses and frequent conflicting literals (zCh1) is the fastest, even if results of the various versions are here considerably heterogeneous. Note, in particular, that the version incrementing all literals of learned clauses, all literals of failed clauses and both frequent conflicting literals and their negations (bCh2) is incredibly fast on some difficult problems of the series, although has a poor behavior on others. Finally, on the Hardnm series (randomly generated problems from 2003 SAT Competition, where we omitted for brevity the central part of the names, e.g. hardnm-L19-02-S125896754.shuffled-as.sat03-916 → hrdnm-L19-02-03-916) results are again not uniform, but the version incrementing all literals of learned clauses and frequent conflicting literals (zCh1) is by far the fastest.

We mainly focus our attention on running times, which is the most important practical aspect. Clearly not on its absolute values, which will rapidly become outdated, but on the comparison among the different solver versions, since the proposed technique may be introduced in any generic DPLL SAT solver (and probably also in other branching-based algorithms used for solving different problems). Note, however, some interesting absolute results: problems vmpc_29 and vmpc_32, not solved by any complete solver in the most recent (at the time of writing) SAT Competition 2005 (within their time limit and on their machine) [22], are solved by some of the modified versions in quite short times.

We furthermore observe that the number of decisions, for a given problem, is only roughly proportional, and not exactly, to running times. This because the propagation performed after variable assignments may require different times for different variables, depending on their situation within the formula.

Quasigroup	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
qg1-07	S	0.09 140	0.08 140	0.07 140	0.09 158	0.11 158	0.09 195	0.08 137	0.07 137	0.08 143
qg2-07	S	0.03 42	0.03 43	0.04 42	0.03 42	0.03 42	0.03 42	0.03 43	0.03 43	0.03 43
qg2-08	S	60619 60619	47505 47505	49563 49563	48895 48895	57467 57467	61654 61654	28473 28473	51409 51409	61956 61956
qg3-08	S	0.05 157	0.05 157	0.05 157	0.09 354	0.08 336	0.06 257	0.06 279	0.07 249	0.11 418
qg3-09	U	78.27 49221	70.21 46020	92.94 55095	62.67 45095	110.16 65019	103.46 60786	96.90 56553	104.00 57712	119.14 63909
qg4-08	U	0.45 1416	0.26 852	0.29 879	0.36 1171	0.40 1333	0.44 1347	0.30 993	0.33 1026	0.30 1005
qg4-09	S	0.01 34	0.01 34	0.00 34	0.01 35	0.00 35	0.01 35	0.00 36	0.01 36	0.01 36
qg5-09	U	0.02 65	0.02 65	0.02 65	0.02 66	0.02 66	0.02 66	0.02 66	0.02 66	0.02 66
qg5-10	U	0.05 159	0.04 125	0.04 159	0.04 127	0.04 133	0.04 128	0.03 131	0.04 150	0.04 143
qg5-11	S	0.08 133	0.05 91	0.08 133	0.10 349	0.10 341	0.10 342	0.08 152	0.06 92	0.05 92
qg5-12	U	1.12 1508	1.25 1670	1.15 1584	1.09 1423	1.06 1288	1.31 1638	1.10 1389	1.00 1297	1.27 1610
qg5-13	U	85.97 58282	93.83 59252	102.28 63582	75.41 51393	82.49 55888	94.33 63119	90.91 60545	93.74 60119	81.49 54145
qg6-09	S	0.01 16	0.01 16	0.01 16	0.01 16	0.01 16	0.01 16	0.01 16	0.01 16	0.01 16
qg6-10	U	0.22 495	0.22 547	0.21 490	0.31 704	0.24 553	0.33 548	0.28 692	0.29 684	0.33 697
qg6-11	U	2.40 4116	2.14 3419	2.58 4462	2.73 4251	2.08 3711	2.17 3584	3.00 4396	2.16 3690	2.19 3399
qg6-12	U	44.47 37289	47.03 41871	59.82 44813	52.74 42167	51.62 42712	45.81 39929	43.31 35621	55.43 42334	57.32 45375
qg7-09	S	0.01 8	0.01 8	0.01 8	0.00 8	0.01 8	0.01 8	0.01 8	0.01 8	0.01 8
qg7-10	U	0.10 269	0.09 269	0.09 269	0.08 240	0.07 226	0.10 228	0.10 281	0.09 263	0.10 264
qg7-11	U	0.89 1671	1.09 1977	1.07 2101	0.76 1663	0.79 1624	1.11 2216	1.03 1802	1.17 2006	0.83 1487
qg7-12	U	9.37 11993	10.17 13152	8.44 11235	8.72 10806	6.93 9433	6.52 9133	12.74 14443	12.01 13957	10.05 11421
qg7-13	S	9.53 32794	4.74 18107	2.59 10801	2.20 4387	1.05 1566	1.36 2256	4.18 9897	2.52 6540	4.59 12153
Total		304.84 260427	272.19 235320	319.60 245628	**248.57** 213350	309.33 241955	313.74 247527	275.37 215953	318.38 241834	333.72 258386

Table 5: Comparison on latin squares logical problems.

On the contrary, when considering different problems, the ratios between number of decisions and running times are almost completely unrelated, since, for each decision, time spent in the propagation phase depends heavily on the size of the problem, and can therefore vary greatly.

Ferries	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
ferry_5_ks99i	S	0.09 1257	0.06 1275	0.06 1267	0.09 1093	0.06 1101	0.09 1083	0.09 1173	0.09 1179	0.09 1275
ferry_5_v01i	S	0.09 973	0.51 4856	0.06 913	0.09 997	0.09 1102	0.09 1130	0.26 2520	0.34 3288	0.20 2274
ferry_6_ks99a	S	0.09 704	0.20 1181	0.09 667	0.14 938	0.23 1208	0.23 1196	0.20 1129	0.12 686	0.18 1129
ferry_6_ks99i	S	0.74 7148	0.66 6874	1.21 10215	0.65 6762	0.17 3230	3.49 16572	1.65 11948	0.91 9262	0.12 2999
ferry_6_v01a	S	0.09 705	0.20 1168	0.20 1066	0.20 1132	0.20 1097	0.20 1119	0.23 1155	0.20 1155	0.17 1011
ferry_6_v01i	S	0.17 1788	0.14 1830	1.60 10406	0.86 7077	1.77 12557	1.54 10002	1.49 9813	0.20 2212	1.00 6904
ferry_7_ks99a	S	0.09 859	0.06 840	0.09 838	0.06 849	0.09 850	0.03 872	0.06 946	0.06 947	0.06 939
ferry_7_ks99i	S	7.15 30372	5.95 27619	3.43 18846	5.63 25760	4.74 24800	0.17 4282	0.03 3560	0.06 3560	0.06 3560
ferry_7_v01a	S	0.03 427	0.01 427	0.03 427	0.03 425	0.03 424	0.03 424	0.03 442	0.01 442	0.03 445
ferry_7_v01i	S	0.51 8610	25.31 56654	4.06 26116	11.32 38862	5.57 29490	9.43 34315	0.86 11073	1.54 13382	1.23 13164
ferry_8_ks99a	S	0.03 1091	0.06 1091	0.03 1091	0.06 1031	0.06 992	0.06 1084	0.06 958	0.06 958	0.12 1219
ferry_8_ks99i	S	8.78 42296	6.46 40972	7.06 39414	8.75 45256	8.43 41061	13.27 51134	32.88 74253	12.72 51431	11.92 54193
ferry_8_v01a	S	0.06 1181	0.06 1188	0.09 1429	0.09 1014	0.06 969	0.09 999	0.09 1248	0.17 2262	0.20 2582
ferry_8_v01i	S	24.36 68748	19.45 61474	157.65 245520	140.34 164733	1.83 23907	56.93 102114	24.65 72620	139.71 145180	1.77 21720
ferry_9_ks99a	S	0.03 2457	0.06 3323	0.03 2457	0.06 3589	0.06 3578	0.06 3578	0.06 2975	0.06 2969	0.06 2959
ferry_9_v01a	S	0.06 1870	0.06 1874	0.03 1869	0.03 1731	0.06 1731	0.03 1731	0.03 1355	0.03 1349	0.03 1349
ferry_10_ks99a	S	0.60 4775	0.79 8503	1.83 9502	1.29 7711	0.74 6434	0.43 4915	5.31 20225	6.63 4652	1.54 9326
ferry_10_v01a	S	3.55 11452	1.66 8044	3.95 11896	0.12 2366	6.23 11726	0.20 2920	0.27 3065	0.31 3623	0.11 2215
Total		46.52 186713	61.70 229193	181.50 383939	169.81 311326	30.42 166257	86.37 239470	68.25 220458	163.22 248537	**18.89** 129263

Table 6: Comparison on industrial planning problems.

VMPC	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
vmpc_21	S	40.17 51447	35.69 49383	87.07 71934	10.92 37382	73.54 64253	16.98 32587	113.26 76797	4.32 26858	84.12 64690
vmpc_22	S	17.27 41243	58.96 66741	167.33 97775	13.75 29278	51.30 51426	71.17 68315	81.26 71840	26.05 47160	20.27 31513
vmpc_23	S	23.50 32087	14.70 40043	20.84 30101	80.38 77444	459.02 165147	8.38 36982	139.37 82271	7.46 19977	12.87 38440
vmpc_24	S	487.83 164331	1466.91 344462	- -	62.56 72555	15.78 27549	295.43 124715	2781.95 453645	191.09 98820	1.00 6663
vmpc_25	S	45.75 52606	12.04 26166	1212.98 274411	1898.02 363854	- -	471.25 164424	1703.79 342933	2677.69 429575	- -
vmpc_26	S	683.22 203309	109.23 78038	245.40 114239	- -	3078.87 491855	3178.86 486241	- -	1037.55 243033	2.53 16153
vmpc_27	S	456.90 172813	55.27 62336	391.76 178963	176.97 116529	415.67 175928	591.35 191731	480.43 163112	859.53 238902	162.13 99316
vmpc_28	S	1167.88 285294	34.05 52479	2729.96 457885	- -	- -	- -	2404.73 387145	892.95 242817	- -
vmpc_29	S	- -	- -	- -	- -	- -	- -	- -	- -	239.96 93700
vmpc_30	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
vmpc_31	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
vmpc_32	S	- -	- -	198.72 110112	837.14 281415	- -	865.82 249933	- -	- -	- -
vmpc_33	S	- -	- -	- -	- -	- -	- -	- -	- -	1493.40 322177
vmpc_34	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
Total		>13722.52 > 1700422	> **12586.85** > 1416940	> 15854.06 > 2204942	> 19454.74 > 2666253	> 22094.18 > 2560910	> 16299.24 > 2230690	> 22104.79 > 2766890	> 16496.64 > 2044434	> 12816.28 > 1841527

Table 7: Comparison on cryptographic problems.

Hardnm shuffled	Sol	zChaff	zCh1	zCh2	brChaff	brCh1	brCh2	bChaff	bCh1	bCh2
hrdnm-L19-01-03-915	S	9.24 25236	10.65 27521	123.56 85254	41.61 62255	23.29 42143	8.73 30594	17.11 38479	13.47 32548	15.34 33604
hrdnm-L19-02-03-916	S	8.63 25860	12.48 31683	2.65 11253	8.61 25337	5.75 20925	3.68 16423	39.10 56102	12.42 32242	30.01 58647
hrdnm-L19-03-03-917	S	82.55 83046	13.65 34632	11.35 30803	25.29 49694	27.20 48626	8.40 26517	5.36 19278	12.34 29194	7.71 26262
hrdnm-L22-01-03-920	S	2.52 12932	13.20 33368	5.15 22519	7.25 26418	5.25 24026	7.31 23146	8.45 30238	4.36 29668	22.02 20652
hrdnm-L22-02-03-921	S	5.27 22451	2.87 16762	7.96 28590	6.82 26141	8.77 29480	13.48 38988	7.59 29398	38.98 65562	3.82 20067
hrdnm-L22-03-03-922	S	8.59 30934	8.02 28817	5.69 24651	5.49 24323	8.99 30140	10.10 33155	10.13 31967	12.29 33514	4.19 19584
hrdnm-L23-01-03-925	S	16.01 48217	69.29 98247	29.05 66796	30.06 66299	66.75 96265	380.35 173629	23.63 63596	65.94 98711	29.31 68294
hrdnm-L23-02-03-926	S	28.25 59509	18.01 49629	51.02 77861	14.71 44176	29.01 62696	22.20 51443	46.11 80912	16.74 46323	22.95 56691
hrdnm-L23-03-03-927	S	21.28 64106	36.35 70584	12.52 44783	22.91 55877	18.15 48843	20.77 55804	25.54 68574	14.67 45455	39.23 70973
hrdnm-L25-01-03-930	S	75.86 108616	13.82 42366	313.96 209249	8.02 34393	12.86 43058	110.88 117917	47.48 84246	101.13 128466	115.36 137370
hrdnm-L25-02-03-931	S	40.59 82238	102.39 113462	47.66 89920	157.36 159150	183.31 164969	103.57 138525	276.68 192270	681.51 330759	100.76 135463
hrdnm-L25-03-03-932	S	60.23 86583	17.68 55290	13.26 42828	137.41 140286	112.19 121455	293.26 214873	56.41 96379	51.83 84411	24.02 62075
hrdnm-L29-01-03-935	S	535.23 317705	28.93 95837	117.23 179297	359.71 290982	255.53 262211	664.79 371819	405.31 348645	184.02 245036	241.84 205747
hrdnm-L29-02-03-936	S	346.96 287345	66.03 141558	210.92 215490	239.92 308060	361.16 289431	633.12 361345	347.30 281170	131.63 171277	552.09 376329
hrdnm-L29-03-03-937	S	317.73 295238	118.39 175396	17.80 71344	176.63 185118	230.73 242715	317.96 244672	679.81 393134	387.76 346807	312.98 277702
hrdnm-L32-01-03-940	S	42.91 135702	31.97 113662	30.62 121878	60.06 152236	24.38 107706	30.54 116380	105.60 207157	38.69 141886	31.76 119094
hrdnm-L32-02-03-941	S	41.55 142873	58.82 161608	150.33 248032	36.60 133982	44.97 144234	319.41 348274	26.39 100668	66.88 152691	39.84 137348
hrdnm-L32-03-03-942	S	53.62 160376	25.26 115426	31.62 119618	138.09 261505	96.51 179978	50.66 147518	49.16 141343	38.39 143495	235.42 366033
Total		1707.02 1988967	**647.81** 1405848	1182.34 1690166	1476.55 2046232	1515.67 1958901	2997.15 2511022	2176.02 2263556	1877.14 2158045	1810.98 2191935

Table 8: Comparison on randomly generated problems.

5 Conclusions

The branching heuristic has a relevant influence on computational behavior of DPLL SAT solvers. Conflict-based branching heuristics have the advantages of requiring low computational overhead and of being often able to detect the hidden structure of a problem. We report here a computational study of new scores updating criteria for conflict-based branching heuristics. Such criteria have been developed in order to overcome some of the typical time-wasting behaviors of DPLL search techniques. In particular, the proposed family of conflict-based heuristics has three main aims: i) to assign at first the more constrained variables; ii) to reverse every sequence of assignments which have led to a conflict, by satisfying at first clauses which have become empty; and iii) to assign at first variables that, due to their relations with the others, cause frequent backtracks at the same decision level of the search tree. For the above reasons, this family of branching heuristics has been called *reverse assignment sequence* (RAS). Such heuristics have been implemented into the state-of-the-art DPLL SAT solver zChaff 2004, obtaining several solver versions having quite different behaviors. Experiments on many benchmark series, both satisfiable and unsatisfiable, show that the proposed branching heuristics are often able to improve solution times. Moreover, notwithstanding the fact that the introduced counters updating requires some computational overhead for its operations, total solution times on each series are always in favor of one of the new versions of the solver.

As a final remark, the authors suppose that similar score based branching heuristics for guiding the search performed by a generic complete branching algorithm can be adapted also to the case of problems different from the propositional Satisfiability one.

References

[1] R. Bayardo, R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of 14th National Conference on Artificial Intelligence* (AAAI), 1997.

[2] R. Bruni, A. Sassano. Restoring Satisfiability or Maintaining Unsatisfiability by finding small Unsatisfiable Subformulae. In *Proceedings of Theory and Applications of Satisfiability Testing* (SAT2001), 2001.

[3] R. Bruni, A. Santori. Adding a New Conflict-Based Branching Heuristic in two Evolved DPLL SAT Solvers. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing* (SAT2004), 2004.

[4] M. Buro, H. Kleine Büning. Report on a SAT Competition. *Bulletin of the European Association for Theoretical Computer Science*, 49, 143–151, 1993.

[5] E. Cervalho, J.P. Marques-Silva. Using Rewarding Mechanisms for Improving Branching Heuristics. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing* (SAT2004), 2004.

[6] V. Chandru and J.N. Hooker. *Optimization Methods for Logical Inference.* Wiley, New York, 1999.

[7] S.A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of Third Annual ACM Symposium on Theory of Computing*, 1971.

[8] M. Davis, G. Logemann, D. Loveland. A machine program for theorem proving. *Communications of the ACM* 5, 394-397, 1962.

[9] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the ACM* **7**, 201–215, 1960.

[10] Z. Fu, Y. Mahajan, S. Malik. New Features of the SAT'04 version of zChaff. In *SAT 2004 Competition: Solver Descriptions*, 2004.

[11] M.R. Garey, D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and co., San Francisco, 1979.

[12] I.P. Gent, H. van Maaren, T. Walsh editors. *SAT 2000*, IOS Press, Amsterdam, 2000.

[13] E. Goldberg, Y. Novikov. BerkMin: a Fast and Robust SAT-Solver. In Proceedings of Design Automation & Test in Europe (DATE 2002), 2002.

[14] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. *DIMACS Series in Discrete Mathematics* American Mathematical Society, 1999.

[15] M. Herbstritt, B. Becker. Conflict-based Selection of Branching Rules in SAT-Algorithms. In E. Giunchiglia, A. Tacchella eds., *Sixth International Conference on Theory and Applications of Satisfiability Testing -Selected Papers*, LNAI 2919, Springer, 2003.

[16] R.E. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and AI* **1**, 167–187, 1990.

[17] D. Le Berre, L. Simon. The Essentials of the SAT 2003 Competition. In E. Giunchiglia, A. Tacchella eds., *Sixth International Conference on Theory and Applications of Satisfiability Testing -Selected Papers*, LNAI 2919, Springer, 2003.

[18] P. Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1-2):315-326, 2000.

[19] J.P. Marques-Silva, K.A. Sakallah. Conflict Analysis in Search Algorithms for Propositional Satisfiability. In *Proceedings of IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[20] J.P. Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), 1999.

[21] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 39th Design Automation Conference*, 2001.

[22] SAT competitions web site, organizers D. Le Berre and L. Simon. `http://www.satcompetition.org/`, and also `http://www.satlive.org/`.

[23] K. Truemper. *Effective Logic Computation.* Wiley, New York, 1998.

[24] L. Zhang, S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Proceedings of CADE 2002 and CAV 2002*, 2002.

[25] H. Zhang, M.E. Stickel. Implementing the Davis-Putnam Method. In I.P. Gent, H. van Maaren, and T. Walsh eds. *SAT 2000*, IOS Press, Amsterdam, 2000.