Conflict-based Selection of Branching Rules in SAT-Algorithms

Marc Herbstritt *Bernd Becker*

Institute of Computer Science, Albert–Ludwigs–University,
D-79110 Freiburg im Breisgau, Germany
{herbstri,becker}@informatik.uni-freiburg.de

ABSTRACT

The problem of proving that a propositional boolean formula is satisfiable (SAT) is one of the fundamental problems in computer science. The application of SAT solvers in VLSI CAD has become of major interest. The most popular SAT algorithms are based on the well known Davis-Putnam procedure. There, to guide the search, a branching rule is applied for selecting and assigning unassigned variables. Additionally, conflict analysis methods are available that result in non-chronological backtracking that prevents the SAT algorithm from searching nonrelevant parts of the search space.

In this paper we focus on the impact of different branching rules and present an approach which (1) allows the use of several branching rules to be applied (not limited to one static rule) and (2) uses information from non-chronological backtracking to dynamically adapt the probabilities of the branching rules to be selected.

Our approach results in a faster and more robust behaviour of the SAT algorithm.

1 Introduction

The problem of proving that a propositional boolean formula is satisfiable (SAT) is one of the fundamental problems in computer science [6]. Many problems can be transformed into a SAT problem such that a solution of the SAT problem can be used to get a solution to the original problem. In the last years important developments were made resulting in powerful SAT algorithms [24, 20, 30, 1, 15, 29] which have been successfully applied to real-world problems [21, 3]. Particularly, the impact of SAT based techniques in problems related to VLSI CAD (e.g. equivalence checking, test pattern generation, timing analysis, and model checking) is continuously increasing [21, 18, 2, 26, 14].

In modern SAT algorithms two features are very important: the application of a branching rule and the use of non-chronological backtracking. In this paper we propose a technique that is based on a set of branching rules instead of only one. Each branching rule is attached a preference value which reflects the usefulness of the branching rule. Depending on these values we are able to perform a dynamic selection. Moreover, data from non-chronological backtracking is used to customize the dynamic selection of branching rules. Thus, two important parts of modern SAT algorithms are combined which traditionally do not depend on each other. In principal our approach may be integrated into any Davis-Putnam-like SAT algorithm, e.g. GRASP [20], SATO [30], or Chaff [24].

In this work we use GRASP as the basis for implementation since it already provides many branching rules and non-chronological backtracking. Experimental results show the feasibility of our approach.

The paper is structured as follows. After giving preliminaries about the SAT problem and corresponding algorithms, we give an overview of popular branching rules. Then a detailed description of our approach follows. Experimental results are presented and discussed. Section 9 then concludes the paper.

2 Preliminaries

In this section we give notations used throughout the paper. *Boolean variables* are denoted by $x_1, x_2, \ldots, x_n$. A boolean variable x_i can be assigned a value 0 or 1 and this value is denoted by $v(x_i)$. A *literal* is a variable x_i or its negation $\neg x_i$. A *clause* c_j is a disjunction $(l_{j,1} \vee \ldots \vee l_{j,k_j})$ of literals where all literals are pairwise disjoint. Clauses can be considered as sets of literals, i.e. $c_j = \{l_{j,1}, \ldots, l_{j,k_j}\}$.

The size of a clause c_j is given by the number of literals in c_j and is denoted by $s(c_j)$. A clause c_j is a *unit clause* if c_j consists of only one literal, i.e. $s(c_j) = 1$. The *empty* clause is denoted by ϵ. A *conjunctive normal form* (CNF) φ is a conjunction $(c_1 \wedge \ldots \wedge c_m)$ of clauses. According to clauses and literals, a CNF φ can be considered as a set of clauses, i.e. $\varphi = \{c_1, \ldots, c_m\}$. Finally, φ can be written as

$$\varphi \;=\; \{\{l_{1,1}, \ldots, l_{1,k_1}\}, \ldots, \{l_{m,1}, \ldots, l_{m,k_m}\}\} .$$

A literal l is *satisfied* iff $l = x_i$ and $v(x_i) = 1$, or $l = \neg x_i$ and $v(x_i) = 0$. A literal l is *unsatisfied* iff $l = x_i$ and $v(x_i) = 0$, or $l = \neg x_i$ and $v(x_i) = 1$. A clause c_j is *satisfied* iff at least one of its literals is satisfied. A clause c_j is *unsatisfied* iff all literals are unsatisfied. A clause c_j is *unresolved* if it is neither satisfied nor unsatisfied. Note that the empty clause ϵ is never satisfied and is indicating a contradiction of variable assignments wrt φ. Finally, a CNF φ is *satisfied* iff all of its clauses are satisfied.
Using this notation we formulate the propositional satisfiability problem as follows.

```
Davis-Putnam( $\varphi$ ) {
  **if** $\varphi$ is empty **return** *satisfiable*
  **if** $\epsilon \in \varphi$ **return** *not satisfiable*
  /* unit propagation */
  **if** $\exists c_j \in \varphi$ with $s(c_j) = 1$ ($c_j = \{l\}$)
    **then** satisfy $l$ and simplify $\varphi$ to $\varphi'$
        **return** Davis-Putnam( $\varphi'$ )
    **else**
        /* branching rule */
        select unassigned variable $x_i$ and an
        assignment $v(x_i) = a$;
        simplify $\varphi$ to $\varphi'$;
        **if** Davis-Putnam( $\varphi'$ ) == satisfiable
          **return** *satisfiable*
        **else**
          change assignment of $x_i$ to $v(x_i) = \neg a$;
          simplify $\varphi$ to $\varphi''$;
          **return** Davis-Putnam( $\varphi''$ )
}
```

Figure 1: Davis-Putnam Procedure

Problem: SAT
Given: A CNF φ over boolean variables $x_1, \ldots, x_n$.
Question: Does there exist an assignment $(v(x_1), \ldots, v(x_n))$ of variables $x_1, \ldots, x_n$ such that φ is satisfied?

Consequently a formula φ is *unsatisfiable* iff no assignment to the variables exists such that φ is satisfied. There exist conceptually different approaches to solving SAT. We restrict ourselves to Davis-Putnam based algorithms.

3 Davis-Putnam Procedure

Davis-Putnam based algorithms basically traverse the search space of all possible variable assignments using depth-first search and backtracking [9, 8]. The basic procedure is sketched in Figure 1.
Modern SAT algorithms extend the basic Davis-Putnam procedure with the addition of a more intelligent implication analysis (e.g. fixing monotone variables in φ and clause subsumption), and powerful conflict analysis mechanims [19]. In particular the latter aspect leads to the application of *non-chronological backtracking* (i.e. backtracking more than one level). This allows the search procedure to ignore parts of the search space which indeed will not lead to satisfiable solutions. Additionally, different algorithms differ in the branching rule they apply.
In this work, we focus on the implementation of the branching rule which is applied during the search to decide which part of the search space should be traversed next.

4 Branching Rules

It is widely accepted that the application of a *"good"* branching rule is essential for solving a SAT problem. "Good" means that in case of a satisfiable SAT instance the branching rule selects variables and corresponding assignments in a sequence such that the amount of traversed search space is small compared to the size of the entire search space. On the other hand, in case of an unsatisfiable SAT instance the branching rule should select variables and corresponding assignments in a sequence that quickly leads to contradictions and thus minimizes search costs.
It should be noted that the problem of choosing the optimal variable and an assignment to this variable in Davis-Putnam based SAT algorithms has been proven to be NP-hard as well as coNP-hard [16].
In the following sections we review the most common used branching rules which are also available in GRASP [20].

Böhm: The Böhm branching rule [4] selects the variable x with maximal vector $H(x) = (H_1(x), H_2(x), \ldots, H_n(x))$ (in lexicographic order), where $H_i(x)$ is given by

$$H_i(x) = \alpha \cdot \max(h_i(x), h_i(\neg x)) + \beta \cdot \min(h_i(x), h_i(\neg x)),$$

and $h_i(x)$ denotes the number of unresolved clauses with i literals. This rule gives preference to variables that are satisfying small clauses ($v(x) = 1$) or that are reducing the size of already small clauses ($v(x) = 0$).

MOM: MOM is shorthand for *Maximum Occurences on Clauses of Minimum Size* [10]. The MOM branching rule selects a variable x that maximizes

$$(f^*(x) + f^*(\neg x)) \cdot 2^k + f^*(x) \cdot f^*(\neg x),$$

where $f^*(l)$ is the number of occurences of literal l in the smallest unresolved clauses. This rule gives preference to variables that occur frequently as positive as well as negative literal in many clauses. Note that the value of k can vary. E.g. MOM is used in Satz [15] with $k = 10$.

Jeroslaw-Wang Rules: There are two branching rules proposed by Jeroslow and Wang in [12]. They are based on the function $\mathrm{JW}(l)$ for a literal l whereby $\mathrm{JW}(l) = \sum_{l \in c \wedge c \in \varphi} 2^{-s(c)}$. Both branching rules give preference to variables that occur often in small unresolved clauses.

OS-JW: The *One-Sided Jeroslow-Wang* (OS-JW) branching rule selects the literal l with the largest value $\mathrm{JW}(l)$ and makes an assignment that satisfies l.

TS-JW: The *Two-Sided Jeroslow-Wang* (TS-JW) branching rule selects variable x with the largest combined value $\mathrm{JW}(x) + \mathrm{JW}(\neg x)$ and assigns $v(x) = 1$ if $\mathrm{JW}(x) \geq \mathrm{JW}(\neg x)$ and $v(x) = 0$ otherwise.

Literal count heuristics: The next three branching rules are so-called literal count heuristics, since only the numbers of occurences of literals in unresolved clauses are considered [17]. They make use of the following functions for a variable x:

$$\begin{aligned} \mathrm{UC_p}(x) &= |\{c \mid x \in c \text{ and } c \text{ is unresolved}\}| \\ \mathrm{UC_n}(x) &= |\{c \mid \neg x \in c \text{ and } c \text{ is unresolved}\}| \end{aligned}$$

DLCS: *Dynamic Largest Combined Sum* (DLCS) selects the variable x with largest value $\mathrm{UC_p}(x) + \mathrm{UC_n}(x)$ and assigns $v(x) = 1$ if $\mathrm{UC_p}(x) \geq \mathrm{UC_n}(x)$ and $v(x) = 0$ otherwise.

DLIS: *Dynamic Largest Individual Sum* (DLIS) selects the variable x where

$$x = \arg\max_y(\mathrm{UC_p}(y), \mathrm{UC_n}(y)),$$

and assigns $v(x) = 1$ if $\mathrm{UC_p}(x) \geq \mathrm{UC_n}(x)$ and $v(x) = 0$ otherwise. DLIS is the default branching rule used in GRASP.

Class	#Inst.	#SAT	#UNSAT
aim-200	24	16	8
bf	4	0	4
dubois	13	0	13
ii16	10	10	0
ii32	17	17	0
jnh	50	16	34
pret	8	0	8
ssa	8	4	4
$\sum$	134	63	71

Table 1: DIMACS benchmarks subset.

Branching rule	Time [sec]	Aborts
Böhm	1817.45	8
MOM	1428.04	7
OS-JW	807.82	4
TS-JW	911.28	4
DLCS	746.30	3
DLIS	409.14	1
RDLIS	439.16	1.1
RAND	1431.85	5.7

Table 2: Application of different branching rules. Results for RDLIS and RAND are average values of 30 experiments.

RDLIS: To balance the greedy behaviour of DLIS, *Random DLIS* selects the same variable as DLIS, but makes a random assignment $v(x)$.

RAND: *RAND* selects randomly an unassigned variable x and assigns randomly 0 or 1 to x.

5 Comparison of branching rules

In [17], a detailed comparison between the branching rules mentioned above is presented. To validate these results we performed experiments for the same set of DIMACS benchmarks (see Table 1) on a AMD Athlon(TM) XP1700+, restricted to 512MB main memory and 180sec of CPU runtime for each instance. Each class of the benchmark set consists of several instances[1] which can be either *satisfiable* (see column #SAT) or *unsatisfiable* (see column #UNSAT). For each branching rule we applied GRASP to the set of benchmarks. Note that DLIS is the default branching rule of GRASP. Results are given in Table 2. We have counted aborted instances with 180sec wrt time. Note that the results for RDLIS and RAND are average results from 30 experiments. The standard deviation for time (number of aborts) is 179.04sec (0.94 aborts) for RDLIS, and 198.35sec (1.10 aborts) for RAND, respectively. The application of DLIS produces the best results, in particular only one instance is aborted.

6 Dynamic selection

The idea of our approach is to not only apply one branching rule during the entire search process, but to give each branching rule the possibility to make a decision from time to time. Therefore we consider a set $B = \{\varrho_1, \ldots, \varrho_t\}$ where each $\varrho \in B$ denotes a branching rule. When a decision assignment must be made, we have to select a branching rule $\varrho \in B$ which selects an unassigned variable and its assignment. What are the criteria for selecting a ”‘good”’ branching rule from B?

Preference values

For this, we assign a *preference value* $\mathrm{Pref}(\varrho)$ to each branching rule $\varrho \in B$ which models the probability of ϱ to be selected. To be consistent with probability theory, we require the following constraints:

$$\forall \varrho \in B : \quad 0 < \mathrm{Pref}(\varrho) < 1 \tag{1}$$

$$\sum_{\varrho \in B} \mathrm{Pref}(\varrho) = 1 \tag{2}$$

[1]An *instance* is nothing else than a concrete SAT problem.

Selecting a branching rule

Assuming a valid probability distribution of all $\mathrm{Pref}(\varrho)$ we apply well known selection methods from genetic algorithms [23] to the set B at each decision level. From the huge set of possible selection methods we consider only the following:

Roulette-Wheel Selection: Each $\varrho \in B$ is selected with probability $\mathrm{Pref}(\varrho)$.

Linear Ranking Selection: Consider all $\varrho \in B$ sorted in increasing order by their preference values $\mathrm{Pref}(\varrho)$. Then, if ϱ has rank r, ϱ is selected with probability

$$r/(\sum_{i=1}^{t} i),$$

where t denotes the number of branching rules, i.e. $t = |B|$.

k-Tournament Selection: Select randomly k branching rules $B' = \{\varrho_1, \ldots, \varrho_k\}$ from B, where each $\varrho \in B$ has equal probability to be selected. Now choose from B' the branching rule ϱ_m with maximum preference value $\mathrm{Pref}(\varrho_m)$, i.e.

$$\arg\max_{\varrho \in B'} \mathrm{Pref}(\varrho).$$

The selected branching rule never has the minimum preference value among all branching rules. We set $k = 2$ in our experiments.

Conflict-triggering branching rule

As previously mentioned modern SAT solvers make use of powerful conflict analysis techniques. When a conflict occurs on level d (i.e. the empty clause ϵ is deduced), these methods analyze the conflict by identifying the causes of the conflict. Based on this causes often a backtrack level d' can be derived which is located several levels above the conflict level ($d' = d - \delta$ and often $\delta > 1$). Therefore the search space wrt the variables between level d and level d' can be ignored. This issue is called *non-chronological backtracking*.

The use of non-chronological backtracking is mandatory when solving unsatisfiable SAT instances in practice. On the other hand, in the case of a satisfiable SAT instance, it is clear that there exists a search path leading to a solution of the instance, and that there doesn't occur any conflict when traversing this path. This is why triggering a conflict can be positive or negative dependent on the solvability of the underlying instance. Hence we make the following definition.

Definition 6.1 (Conflict-triggering branching rule)
A branching rule $\varrho \in B$ triggers a conflict *iff*

1. *A conflict occurs at decision level d.*
2. *Non-chronological backtracking results in backtracking to decision level d'.*
3. *ϱ was applied at decision level d', $(d' \leq d)$.*

Rewarding or punishing?

As conflicts are needed for solving unsatisfiable instances, we should give a reward to conflict-triggering branching rules. Also in case of satisfiable instances, we should punish conflict-triggering branching rules. But being faced with an unknown instance, we need some estimation on the solvability of the instance to decide if we should reward or punish! In our approach we use the clauses/variables ratio to estimate the solvability of an instance. The clauses/variables ratio is known from the theory of phase transitions [5, 22]. E.g. for random 3-SAT, it is known that instances with a clauses/variables ratio of 4.3 are hard to solve. Instances with smaller values belong to the less constrained instances, and higher values indicate a more constrained instance. But these values hold for a *class of instances* only. Therefore we introduce some kind of instance specific phase transition approximation.

Definition 6.2 (Individual averaged #C/#V ratio)
The individual averaged clauses/variables ratio of instance I is set at the beginning of the search according to:

$$\mathrm{AR}(I) = no\_of\_clauses(I)/no\_of\_variables(I).$$

During the search, after each time after a conflict occurs, we update $\mathrm{AR}(I)$ *by setting*

$$\mathrm{AR}_{new}(I) = \frac{1}{2} \cdot (\mathrm{AR}_{old}(I) + \frac{no\_unresolved\_clauses(I)}{no\_free\_variables(I)}).$$

Now we can decide wether to punish or to reward:

1. If
$$\frac{no\_unresolved\_clauses(I)}{no\_free\_variables(I)} < \mathrm{AR}_{old}(I),$$
we reside in a *relatively less constrained* region being more probably satisfiable. Thus conflict-triggering branching rules should be punished. We will denote this by *punishing mode*.
2. If the opposite holds we are in a *relatively more constrained* region being more probably unsatisfiable. Hence conflict-triggering branching rules should be given a reward. This will be denoted by *reward mode*.

In the following let variable *mode* be defined as[2]

$$mode := \begin{cases} 1 & : \text{in punishing mode} \\ 0 & : \text{in reward mode} \end{cases} \tag{3}$$

To achieve an estimation of the importance of a branching rule, two counters are maintained for each branching rule ϱ:

$$\begin{aligned} \mathrm{Used}(\varrho) &= \text{the number of applications of } \varrho \text{ at some decision level} \\ \mathrm{Trigger}(\varrho) &= \text{the number of how often } \varrho \text{ triggered a conflict according to Definition 6.1} \end{aligned}$$

These counters are updated every time a branching rule is applied at a decision level and every time a branching rule has triggered a conflict. Based on these counters we can compute an *update factor* with respect to the preference value of branching rule ϱ:

$$\mathrm{Update}(\varrho) = 1 + (-1)^{mode} \cdot \frac{\mathrm{Trigger}(\varrho)}{\mathrm{Used}(\varrho)} \tag{4}$$

Now, if ϱ triggers a conflict we compute a new preference value of ϱ by setting

$$\mathrm{Pref}_{\mathrm{new}}(\varrho) = \mathrm{Update}(\varrho) \cdot \mathrm{Pref}_{\mathrm{old}}(\varrho) \tag{5}$$

Thus, the probability of ϱ to be selected as branching rule at a decision level is decreased in punishing mode and increased in reward mode.

[2]Please note that in our implementation we use a 10% deviation for switching *mode* to compensate small ”irritations”.

Difference distribution

To ensure property (2) we have to distribute the difference $\mathrm{Pref}_{\mathrm{diff}}(\varrho)$ between the old and the new preference value of ϱ to the other branching rules $B' = B \setminus \{\varrho\}$, where

$$\mathrm{Pref}_{\mathrm{diff}}(\varrho) = (-1)^{mode} \cdot (\mathrm{Pref}_{\mathrm{new}}(\varrho) - \mathrm{Pref}_{\mathrm{old}}(\varrho)) \tag{6}$$

This distribution can be done in different ways. We examined the two following distribution mechanisms (t denotes the number of branching rules):

Uniform Distribution: Each branching rule $\psi \in B'$ gets the same portion:

$$\mathrm{Pref}_{\mathrm{new}}(\psi) = \mathrm{Pref}_{\mathrm{old}}(\psi) + (-1)^{(1-mode)} \cdot \mathrm{Pref}_{\mathrm{diff}}(\varrho) \cdot \frac{1}{(t-1)}. \tag{7}$$

Weighted Distribution: Each branching rule $\psi \in B'$ gets a portion proportional to its own preference value. Using

$$\mathrm{Pref}_{B'} = \sum_{\psi \in B'} \mathrm{Pref}(\psi),$$

we compute the new preference values by setting

$$\mathrm{Pref}_{\mathrm{new}}(\psi) = \mathrm{Pref}_{\mathrm{old}}(\psi) + (-1)^{(1-mode)} \cdot \mathrm{Pref}_{\mathrm{diff}}(\varrho) \cdot \frac{\mathrm{Pref}_{\mathrm{old}}(\psi)}{\mathrm{Pref}_{B'}} \tag{8}$$

Afterwards we have to normalize the preference values in order to fulfill properties (1) and (2).

Initialization of preference values

To complete the framework of our approach suitable initialization values for the preference values should be given. We use the results from single branching rule application given in Table 2. We analyzed the results with respect to runtime and number of aborted instances. Applying a linear ranking restricted to these attributes results in initialization values which we will denote by Time-Rank, Abort-Rank, and Time-Abort-Rank, respectively.
It is clear that the initialization of the preference values takes place before starting the search.

7 Integration

Instead of giving a detailed description of our approach as a closed SAT algorithm we explain how our approach can be integrated into Davis-Putnam based SAT algorithms.

1. Initialize the set B and the preference values of available branching rules with some reasonable values fulfilling properties (1) and (2).
2. At each decision level remember the applied branching rule ϱ.
3. Instead of applying only one branching rule at a decision level, proceed in the following way:

   (a) Let *selection-method* be one of the proposed selection methods Roulette-Wheel, Linear Ranking or 2-Tournament.
   (b) Apply *selection-method* to the set B of available branching rules and let ϱ denote the selected branching rule.
   (c) Increment $\mathrm{Used}(\varrho)$.
   (d) Apply ϱ and return the computed variable and its assignment.

4. When a conflict occurs at decision level d and non-chronological backtracking results in backtracking to level d', do the following:

Select	Dist	Time-Rank				Abort-Rank				Time-Abort-Rank			
		$\oslash$-T	$\sigma(T)$	$\oslash$-A	$\sigma(A)$	$\oslash$-T	$\sigma(T)$	$\oslash$-A	$\sigma(A)$	$\oslash$-T	$\sigma(T)$	$\oslash$-A	$\sigma(A)$
RW	uni	374.48	149.15	1.07	0.89	363.43	181.16	1.00	0.86	337.59	120.87	0.90	0.70
	weight	303.35	147.28	0.83	0.73	394.52	179.05	0.93	0.81	320.60	122.91	0.93	0.77
LR	uni	290.41	168.15	0.87	0.76	207.23	74.93	0.57	0.50	234.18	137.78	0.63	0.66
	weight	329.11	135.78	1.13	0.76	201.07	116.99	0.60	0.49	269.06	118.14	0.83	0.58
2T	uni	387.01	162.21	1.13	0.85	255.16	136.10	0.67	0.75	337.97	148.49	1.07	0.85
	weight	411.52	126.50	1.40	0.76	292.56	150.00	0.83	0.93	316.30	155.51	0.93	0.85

Table 3: Experimental results by conflict-based selection of branching rules.

(a) Let ϱ denote the branching rule applied at decision level d'.
(b) Compute the update factor $\mathrm{Update}(\varrho)$ and re-compute the preference value $\mathrm{Pref}(\varrho)$ according to formula (5).
(c) Distribute the difference $\mathrm{Pref}_{\mathrm{diff}}(\varrho)$ using one of the proposed distribution mechanisms Uniform Distribution or Weighted Distribution.
(d) Increment $\mathrm{Trigger}(\varrho)$.
(e) After backtracking to decision level d', update the averaged clauses/variables ratio AR according to definition 6.2. Update *mode* according to the discussion following definition 6.2 in the text.

8 Experimental results

We have integrated our approach into GRASP. Since there are several possibilities for choosing a preference value initialization (3x), branching rule selection method (3x) and difference distribution mechanism (2x) we have conducted experiments with $(3 \cdot 3 \cdot 2) = 18$ configurations. Since our approach is a *randomized* method (namely the application of the proposed selection methods), we handled each instance of the benchmark set (see Table 1) 30 times for each of the 18 parameter settings. All experiments were performed on a AMD Athlon(TM) XP1700+, restricted to 512MB main memory and 180sec of CPU runtime for each instance.
Table 3 gives the results. Column *Select* gives the used branching rule selection method, where RW denotes Roulette-Wheel, LR denotes Linear Ranking, and 2T denotes 2-Tournament. In column *Dist* the applied distribution mechanism is named (*uni* denotes Uniform Distribution and *weight* denotes Weighted Distribution). For each preference value initilization (Time-Rank, Abort-Rank, Time-Abort-Rank) the corresponding column is splitted into a twofold-column wrt time and aborts, respectively. Each column is splitted into two columns for average CPU time in seconds $\oslash(T)$ (average aborts $\oslash(A)$) and standard deviation of time $\sigma(T)$ (standard deviation of aborts $\sigma(A)$). As for the single branching rule experiments, we have counted aborted instances with 180sec.
The application of GRASP using DLIS as default branching rule leads to a total runtime of 409.14 seconds and 1 aborted instance (see Table 2, column DLIS). We refer to this experiment by GRASP-DLIS in the following.
Although the average time used by the different configurations is mostly less than the time of GRASP-DLIS, the average number of aborted instances ranges between good (see column Abort-Rank) and reasonable (see column Time-Rank) compared to GRASP-DLIS. But it is evident that our approach holds the potential to reduce time and aborted instances.
We have therefore performed a second series of experiments using the benchmarks from Table 4. These benchmarks were selected because they are ”hard” for GRASP (we refer the reader to [27, 28]). Since the preference initialization Abort-Rank gives the best results in Table 3, we only applied this one for the new experiments. Because of reasons of time we applied our approach only 10 times per benchmark instance, but with a time treshold of 600sec and counting aborted instances with 600sec. The results are given in Table 5. Please note that the results for GRASP-DLIS are in the first row. Now it becomes

Name	#var	#clauses	status
`bw_large.c`	3016	50457	satisfiable
`bw_large.d`	6325	131973	satisfiable
`e0ddr2-10-by-5-1`	19500	103887	satisfiable
`e0ddr2-10-by-5-4`	19500	104527	satisfiable
`enddr2-10-by-5-1`	20700	111567	satisfiable
`enddr2-10-by-5-8`	21000	113729	satisfiable
`ewddr2-10-by-5-1`	21800	118607	satisfiable
`ewddr2-10-by-5-8`	22500	123329	satisfiable
`hfo3.010.1`	215	920	satisfiable
`hfo3.022.1`	215	920	satisfiable
`hfo3.027.1`	215	920	satisfiable
`qg5-10`	1000	43636	unsatisfiable
`qg7-11`	1331	49534	unsatisfiable

Table 4: Selection of benchmarks from `blocksworld` [13], `bejing` [7, 11], `pader-hard` [25], and `quasigroup` [30].

Solver	Time		Time	
	$\oslash(T)$	$\sigma(T)$	$\oslash(A)$	$\sigma(A)$
GRASP-DLIS	3491.69	total	4	total
RW+Abort+uni	2988.97	487.72	3.60	0.92
RW+Abort+weight	2531.13	580.96	2.70	1.27
LR+Abort+uni	2281.15	467.34	2.20	0.75
LR+Abort+weight	2138.69	593.96	1.90	1.14
2T+Abort+uni	2293.54	593.64	2.70	1.19
2T+Abort+weight	2398.12	580.24	2.70	1.00

Table 5: Results for the benchmarks of Table 4.

evident that our approach is able to outperform GRASP-DLIS significantly. The average time and the average number of aborted instances are always less than the results of GRASP-DLIS. Also the standard deviations for time ($\sigma(T)$) and the aborted instances ($\sigma(A)$) are acceptable in the sense that we don't have to expect worse results than GRASP-DLIS. Our approach is up to a factor of 1.6 faster compared to GRASP-DLIS (see LR+Abort+weight) and aborts up to 50% less instances on the average than GRASP-DLIS (see also LR+Abort+weight). The impact of the different components of our approach is varying. As we have seen from Table 3 the initialization of the preference values influences primarily the average number of aborted instances. For the applied selection method Linear-Ranking (LR) seems to fit best in our approach resulting in the lowest search time and average number of aborted instances. Only the influence of the distribution mechanisms is ambigious, having a light bias for the weighted distribution.

9 Conclusions

In this paper we have presented an approach that improves propositional satisfiability algorithms by allowing not only one but several branching rules to be applied at some decision levels during search. Additionally, we use information from non-chronological backtracking to dynamically adapt the probabilities of the branching rules to be selected.
Our method can easily be integrated into existing Davis-Putnam-based SAT algorithms. Experimental results point out that our approach results in a faster and more robust behaviour of the SAT algorithm.

Acknowledgements

The authors want to thank Tobias Schubert, Christoph Scholl, and the participants of Dagstuhl Seminar 01051 for helpful discussions, and Matthew Lewis for proof-reading.

References

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

[2] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conf.*, 1999.

[3] A. Borälv and G. Stålmarck. Automated verification in railways. In M.G. Hinchey and J.P. Bowen, editors, *Industrial-Strength Formal Methods in Practice*. Springer, 1999.

[4] M. Buro and H. Kleine-Büning. Report on a SAT competition. Technical report, University of Paderborn, November 1992.

[5] P. Cheesemann, B. Kanefksy, and W.M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI*, 1991.

[6] S.A. Cook. The complexity of theorem proving procedures. In *3. ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[7] J.M. Crawford. International Competition and Symposium on Satisfiability Testing. International Competition on SAT Testing, Bejing, 1996.

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

[9] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:506–521, 1960.

[10] J.W. Freeman. *Improvements to Propositional Satisfiabilty Search Algorithms*. PhD thesis, University of Pennsylvania, Philadelphia (PA), May 1995.

[11] H.H. Hoos and T. Stützle. SAT2000: Highlights of Satisfiability Research in the year 2000. In *Frontiers in Artificial Intelligence and Applications*, pages 283–292. Kluwer Academic, 2000.

[12] R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[13] Henry A. Kautz and Bart Selman. Pushing the envelope : Planning, propositional logic, and stochastic search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201, 1996.

[14] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD*, 11:4–15, 1992.

[15] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 315–326, 1997.

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

[17] J.P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

[18] J.P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Design, Automation and Test in Europe*, pages 145–149, 1999.

[19] J.P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[20] J.P. Marques-Silva and K.A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Int'l Conf. on CAD*, pages 220–227, 1996.

[21] J.P. Marques-Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *Design Automation Conf.*, pages 675–680, 2000.

[22] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *National Conference on Artificial Intelligence (AAAI)*, 1992.

[23] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.

[24] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engeneering an efficient SAT solver. In *Design Automation Conf.*, 2001.

[25] available for downloading at http://sat.inesc.pt/benchmarks/cnf/uni-paderborn.

[26] L.G. Silva, J.P. Marques-Silva, L.M. Silveira, and K.A. Sakallah. Timing analysis using propositional satisfiability. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 1998.

[27] Laurent Simon. SAT-Ex. http://www.lri.fr/∼simon/satex/satex.php3, 2000.

[28] Laurent Simon and Philippe Chatalic. SATEx: a web-based framework for SAT experimentation. In Henry Kautz and Bart Selman, editors, *Electronic Notes in Discrete Mathematics*, volume 9. Elsevier Science Publishers, June 2001. http://www.lri.fr/∼simon/satex/satex.php3.

[29] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. Swedish Patent No. 467 076 (approved 1992), United States Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995)., 1989.

[30] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 155–160, July 1997.