The Impact of Branching Heuristics in Propositional Satisfiability Algorithms

João Marques-Silva

Technical University of Lisbon,
IST/INESC, Lisbon, Portugal
jpms@inesc.pt
URL: http://algos.inesc.pt/~jpms

Abstract. This paper studies the practical impact of the branching heuristics used in Propositional Satisfiability (SAT) algorithms, when applied to solving real-world instances of SAT. In addition, different SAT algorithms are experimentally evaluated. The main conclusion of this study is that even though branching heuristics are crucial for solving SAT, other aspects of the organization of SAT algorithms are also essential. Moreover, we provide empirical evidence that for practical instances of SAT, the search pruning techniques included in the most competitive SAT algorithms may be of more fundamental significance than branching heuristics.

Keywords: Propositional Satisfiability, Backtrack Search, Branching Heuristics.

1 Introduction

Propositional Satisfiability (SAT) is a core problem in Artificial Intelligence, as well as in many other areas of Computer Science and Engineering. Recent years have seen dramatic improvements in the real world performance of SAT algorithms. On one hand, local search algorithms have been used for solving large random instances of SAT and some classes of practical instances of SAT [21, 20, 15, 7]. On the other hand, systematic backtrack search algorithms, based on new and effective search pruning techniques, have been used for solving large structured real-world instances of SAT, a significant fraction of which requires proving unsatisfiability. Among the many existing backtrack search algorithms, rel_sat [2], GRASP [18] and SATO [24] have been shown, on a large number of real-world instances of SAT, to be among the most competitive backtrack search SAT algorithms. There are of course other backtrack search SAT algorithms, which are competitive for specific classes of instances of SAT. Examples include satz [17], POSIT [9], NTAB [5], 2cl [11] and CSAT [8], among others. It is interesting to note that the most competitive backtrack search SAT algorithms share a few common properties, which have empirically been shown to be particularly useful for solving hard real-world instances of SAT. Relevant examples of these techniques are non-chronological backtracking search strategies and clause (*nogood*) identification and recording [3, 10, 22].

One key aspect of backtrack search SAT algorithms is how assignments are selected at each step of the algorithm, i.e. the *branching heuristics*. Over the years many branching heuristics have been proposed by different authors [5, 9, 13, 17]. In this paper we propose to study several of the branching heuristics that have been shown to be more effective in practice. For this purpose we apply different backtrack search SAT algorithms and different branching heuristics to different classes of real-world practical applications of SAT. One interesting result of this study is that even though branching heuristics are indeed of importance in solving SAT, other aspects of the organization of backtrack search algorithms turn out to be of far more significance when the objective is to reduce the amount of search and the running time. This empirical result motivates the development of new search pruning techniques, in particular when the objective is to solve large, structured and hard instances of SAT.

The paper is organized as follows. First, Section 2 introduces the notational framework used in the remainder of the paper. Afterwards, in Section 3 current state-of-the-art backtrack search SAT algorithms are briefly reviewed. The next step is to describe the different branching heuristics evaluated in this paper. Section 5 provides and analyzes experimental results on instances of SAT from different application domains. Finally, Section 6 concludes the paper.

2 Definitions

This section introduces the notational framework used throughout the paper. Propositional variables are denoted $x_1, \ldots, x_n$, and can be assigned truth values *false* (also, F or 0) or *true* (also, T or 1). The truth value assigned to a variable x is denoted by $\nu(x)$. A literal l is either a variable x_i or its negation $\neg x_i$. A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. A clause is said to be *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is unsatisfied if at least one clause is unsatisfied. The SAT problem is to decide whether there exists a truth assignment to the variables such that the formula becomes satisfied.

It will often be simpler to refer to clauses as sets of literals, and to the CNF formula as a set of clauses. Hence, the notation $l \in \omega$ indicates that a literal l is one of the literals of clause ω, whereas the notation $\omega \in \varphi$ indicates that clause ω is one of the clauses of CNF formula φ.

In the following sections we shall address backtrack search algorithms for SAT. Most if not all backtrack search SAT algorithms apply extensively the *unit clause rule* [6]. If a clause is unit, then the sole free literal must be assigned value

1 for the formula to be satisfiable. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation (BCP) [23]. For implementing some of the techniques common to some of the most competitive backtrack search algorithms for SAT [2, 18, 24], it is necessary to properly *explain* the truth assignments to the propositional variables that are implied by the clauses of the CNF formula. For example, let $x = v_x$ be a truth assignment implied by applying the unit clause rule to a unit clause clause ω. Then the explanation for this assignment is the set of assignments associated with the remaining literals of ω, which are assigned value 0.

Let $\omega = (x_1 \vee \neg x_2 \vee x_3)$ be a clause of a CNF formula φ, and assume the truth assignments $\{x_1 = 0, x_3 = 0\}$. Then, for the clause to be satisfied we must necessarily have $x_2 = 0$. We say that the implied assignment $x_2 = 0$ has the explanation $\{x_1 = 0, x_3 = 0\}$. A more formal description of explanations for implied variable assignments in the context of SAT, as well as a description of mechanisms for their identification, can be found for example in [18].

3 Backtrack Search SAT Algorithms

The overall organization of a generic backtrack search SAT algorithm is shown in Figure 1. This SAT algorithm captures the organization of several of the most competitive algorithms [2, 9, 18, 24]. The algorithm conducts a search through the space of the possible truth assignments to the problem instance variables. At each stage of the search, a truth assignment is selected with the `Decide()` function. (Observe that selected variable assignments are characterized by having no explanation.) A decision level d is also associated with each selection of an assignment. Moreover, a decision level $\delta(x)$ is associated with each assigned variable x, that denotes the decision level at which the variable is assigned.

Implied necessary assignments are identified with the `Deduce()` function, which in most cases corresponds to the BCP procedure [23]. Whenever a clause becomes unsatisfied, the `Deduce()` function returns a conflict indication which is then analyzed using the `Diagnose()` function. The diagnosis of a given conflict returns a backtracking decision level, which corresponds to the decision level to which the search process can provably backtrack to. The `Erase()` function clears implied assigned variables that result from each assignment selection. Different organizations of SAT algorithms can be modeled by this generic algorithm, examples of which include POSIT [9] and NTAB [5].

Currently, and for solving large, structured and hard instances of SAT, all of the most efficient SAT algorithms implement a number of the following key properties:

1. The analysis of conflicts can be used for implementing Non-Chronological Backtracking search strategies. Hence, assignment selections that are deemed irrelevant can be skipped during the search [2, 18, 24].
2. The analysis of conflicts can also be used for identifying and recording new clauses that denote implicates of the Boolean function associated with the

```
SAT(d, &β)
{
  if (Decide(d) != DECISION)
    return SATISFIABLE;
  while (TRUE) {
    if (Deduce(d) != CONFLICT) {
      if (SAT(d + 1, β) == SATISFIABLE)
        return SATISFIABLE;
      else if (β != d || d == 0) {
        Erase(d); return UNSATISFIABLE;
      }
    }
    if (Diagnose(d, β) == CONFLICT) {
      return UNSATISFIABLE;
    }
  }
}
```

Fig. 1. Generic SAT Algorithm

CNF formula. Clause recording plays a key role in recent SAT algorithms, despite in most cases large recorded clauses being eventually deleted [2, 18, 24].

3. Other techniques have been developed. Relevance-Based Learning [2] extends the life-span of large recorded clauses that will eventually be deleted. Conflict-Induced Necessary Assignments [18] denote assignments of variables which are necessary for preventing a given conflict from occurring again during the search.

Before running the SAT algorithm, different forms of preprocessing can be applied [8, 18, 17]. This in general is denoted by a `Preprocess()` function that is executed before invoking the search process.

4 Branching Heuristics

This section describes the branching heuristics that are experimentally evaluated in Section 5. The most simple heuristic is to randomly select one of the yet unassigned variables, and to it assign a randomly chosen value. We shall refer to this heuristic as RAND. Most, if not all, of the most effective branching heuristics take into account the dynamic information provided by the backtrack search algorithm. This information can include, for example, the number of literals of each variable in unresolved clauses and the relative sizes of the unresolved clauses that contain literals in a given variable.

In the following sections we describe several branching heuristics which utilize dynamic information provided by the backtrack search SAT algorithm. These heuristics are described in the approximate chronological order in which they have been applied to solving SAT. We should note that additional branching heuristics exist and have been studied in the past. See for example [9, 12, 13] for detailed accounts.

4.1 BOHM's Heuristic

Bohm's heuristic is briefly described in [4], where a backtrack search algorithm using this branching heuristic was shown to be the most competitive algorithm (at the time), for solving randomly generated instances of SAT.

At each step of the backtrack search algorithm, the BOHM heuristic selects a variable with the maximal vector $(H_1(x), H_2(x), \ldots, H_n(x)$ in lexicographic order. Each $H_i(x)$ is computed as follows:

$$H_i(x) = \alpha\ max(h_i(x), h_i(\neg x)) + \beta\ min(h_i(x), h_i(\neg x)) \quad (1)$$

where $h_i(x)$ is the number of unresolved clauses with i literals that contain literal x. Hence, each selected literal gives preference to satisfying small clauses (when assigned value true) or to further reducing the size of small clauses (when assigned value false). The values of α and β are chosen heuristically. In [4] the values suggested are $\alpha = 1$ and $\beta = 2$.

4.2 MOM's Heuristic

One of the most well-known and utilized branching heuristics is the Maximum Occurrences on clauses of Minimum size (MOM's) heuristic [8, 9, 19, 23].

Let $f^*(l)$ be the number of occurrences of a literal l in the smallest non-satisfied clauses. It is widely accepted that a *good* variable to select is one that maximizes the function,

$$[f^*(x) + f^*(\neg x)] * 2^k + f^*(x) * f^*(\neg x) \quad (2)$$

Intuitively, preference is given to variables x with a large number of clauses in x or in $\neg x$ (assuming k is chosen to be sufficiently large), and also to variables with a large number of clauses in *both* x and $\neg x$. Several variations of MOM's heuristic have been proposed in the past with heuristic functions related to but different from (2). A detailed description of MOM's heuristics can be found in [9]. We should also note that, in general, we may also be interested in taking into account not only the smallest clauses, but also clauses of larger sizes.

In this paper, the implementation of MOM's heuristic we experimented with has the following definition:

- Select only variables in the clauses of smallest size, V.
- Among the variables in V, give preference to those with the largest number of *smallest* clauses, V_c. If a variable appears in many of the smallest clauses, then it is likely to induce other implied assignments, thus constraining the search.

– For those variables in V_c, give preference to those that appear the the largest number of *small* clauses. Again the previous intuitive justification applies.

To the selected variable assign value true if the variable appears in more smallest clauses as a positive literal, and value false otherwise.

4.3 Jeroslow-Wang Heuristics

Two branching heuristics were proposed Jeroslow and Wang in [13], and are also analyzed in [1, 12]. For a given literal l, let us compute:

$$J(l) = \sum_{l \in \omega \wedge \omega \in \varphi} 2^{-|\omega|} \tag{3}$$

The one-sided Jeroslow-Wang (JW-OS) branching heuristic selects the assignment that satisfies the literal with the largest value $J(l)$. The two-sided Jeroslow-Wang (JW-TS) heuristic identifies the variable x with the largest sum $J(x) + J(\neg x)$, and assigns to x value true, if $J(x) \geq J(\neg x)$, and value false otherwise.

4.4 Literal Count Heuristics

Besides the heuristics proposed in the previous sections, others are certainly possible. In this section we describe three simple branching heuristics that only take into account the number of literals in unresolved clauses of a given variable at each step of the backtrack search algorithm.

Literal count heuristics count the number of unresolved clauses in which a given variable x appears as a positive literal, C_P, and as negative literal, C_N. These two numbers can either be considered individually or combined. When considered combined, i.e. $C_P + C_N$, we select the variable with the largest sum $C_P + C_N$, and assign to it value true, if $C_P \geq C_N$, or value false, if $C_P < C_N$. Since the C_P and C_N figures are computed during the search, we refer to this heuristic as dynamic largest combined sum (of literals), or DLCS.

When the values C_P and C_N are considered separately, we select the variable with the largest individual value, and assign to it value true, if $C_P \geq C_N$, or value false, if $C_P < C_N$. We refer to this heuristic as dynamic largest individual sum (of literals), or DLIS.

As we shall show in Section 5, branching heuristics can sometimes yield bad branches because they are simply too greedy. A variation of DLIS, referred to as RDLIS, consists in *randomly selecting the value* to be assigned to a given selected variable, instead of comparing C_P with C_N. The random selection of the value to assign is in general a good compromise to prevent making too many bad decisions for a few specific instances. Clearly, we could also use DLCS for implementing a RDLCS branching heuristic.

Table 1. The classes of instances analyzed

Benchmark Class	# Instances
aim-200	24
jnh	50
pret	8
dubois	13
ucsc-bf	223
ucsc-ssa	102

5 Experimental Results

In this section we compare different SAT algorithms and different branching heuristics. Where possible, we have concentrated on analyzing practical instances of SAT, i.e. instances derived from real-world applications.

The classes of instances used for the experimental evaluation are shown in Table 1. All classes of instances are obtained from the DIMACS suite and from the UCSC suite [14]. The number of instances for each class is also shown in the table. Of these classes of instances, *bf* and *ssa* represent practical applications of SAT models to Design Automation [16]. The others were proposed by different authors for the 1993 DIMACS Satisfiability Challenge [14].

For the instances considered, we ran rel_sat, GRASP, SATO, POSIT and NTAB. While rel_sat, GRASP and SATO implement the search pruning techniques described in Section 3, POSIT and NTAB are mainly fast implementations of the Davis-Putnam procedure, with different branching heuristics implemented. As will be suggested by the experimental results, fast implementations of the Davis-Putnam procedure are in general inadequate for solving real-world instances of SAT. This conclusion has actually been reached by other researchers in the past [2, 18, 24]. On the other hand, rel_sat, GRASP and SATO implement a similar set of search pruning techniques which are shown to be very effective. After this experiment, we concentrate on evaluating GRASP when run with different branching heuristics, in particular those described in Section 4. It should be mentioned that either rel_sat or SATO could be used for this purpose, but these algorithms only provide a single branching heuristic, whereas GRASP incorporates a significant number of the branching heuristics proposed in the literature in recent years.

For the experimental results given below, the CPU times were obtained on a Pentium-II 350MHz Linux machine, with 128 MByte of RAM. In all cases the maximum CPU time that each algorithm was allowed to spend on any given instance was 500 seconds. The SAT algorithms POSIT and NTAB were run with the default options. rel_sat was run with learning degree of 3, whereas GRASP and SATO were run allowing recorded clauses of size no greater than 20 to be recorded. Furthemore, GRASP implemented relevance-based learning with degree 5 [2] and was run with the RDLIS branching heuristic. The additional options of rel_sat, GRASP and SATO were set to their default values.

Table 2. CPU times for different SAT algorithms

Class	rel_sat	sato	grasp	posit	ntab
aim-200	0.20	0.51	3.45	0.14	6787.03
bf	1.08	1.33	2.59	6.64	1585.26
dubois	0.09	0.57	17.69	364.68	5317.37
ii16	135.34	2.25	120.36	26.67	1022.33
ii32	399.32	4.95	580.65	2.89	10.70
jnh	0.62	0.93	8.86	0.18	14.90
pret	1.07	1.46	3.30	173.26	2460.31
ssa	17.02	1.71	2.87	15.90	1006.46
ucsc-bf	194.22	68.79	115.14	2642.99	89616.31
ucsc-ssa	149.12	24.23	32.32	518.31	6519.14

Table 3. Number of aborted instances for different SAT algorithms

Class	rel_sat	sato	grasp	posit	ntab
aim-200	0	0	0	12	13
bf	0	0	0	2	3
dubois	0	0	0	9	9
ii16	0	0	0	1	2
ii32	0	0	0	0	0
jnh	0	0	0	0	0
pret	0	0	0	4	4
ssa	0	0	0	0	2
ucsc-bf	0	0	0	100	174
ucsc-ssa	0	0	0	2	12
Total	0	0	0	130	219

5.1 Results for Different SAT Algorithms

The experimental results for rel_sat, GRASP, SATO, POSIT and NTAB, on selected classes from the DIMACS and the UCSC instances, are shown in Tables 2, 3 and 4. Table 2 includes the CPU times for each class of instances. Table 3 indicates the number of instances each algorithm was unable to solve in the allowed CPU time. Finally, Table 4 indicates the total number of decisions made by each algorithm for each class of instances. This figure provides an idea of the amount of search actually conducted by the different algorithms. For instances aborted by any given algorithm, the number of decisions accounted for is 0. For example, for POSIT and for class aim-200, the number of decisions shown are solely for the instances POSIT was able to solve (i.e. 12), which represent 50% of all instances in class aim-200.

It should be emphasized that the results for the UCSC benchmarks are particularly significant, since they result from actual practical applications of SAT [16].

From the above results several conclusions can be drawn:

Table 4. Number of branches for different SAT algorithms

Class	rel_sat	sato	grasp	posit	ntab
aim-200	1074	1597	4098	580	937782
bf	992	1708	1083	14914	48143
dubois	883	1854	18399	9662919	15728668
ii16	36841	3993	6644	5607	3946
ii32	84123	8321	11566	415	1268
jnh	1135	1284	3651	280	482
pret	4584	6012	15241	4187100	4194300
ssa	16960	2904	1780	56519	3180
ucsc-bf	147540	108469	43890	3006755	1334638
ucsc-ssa	138997	44001	19018	1428196	367679

- Backtrack search SAT algorithms, based on plain implementations of the Davis-Putnam procedure, are clearly inadequate for solving a significant fraction of the instances studied.
- The search pruning techniques included in more recent algorithms, e.g. rel_sat, SATO and GRASP, are clearly effective in most classes of instances, and for some classes of instances they are essential.

From these results, and since rel_sat, SATO and GRASP use different branching heuristics, one might be tempted to extrapolate that the branching heuristic used is irrelevant when effective search pruning techniques are implemented by SAT algorithms. In general, this is is not the case, as we shall see in the following sections. Nevertheless, we will be able to provide evidence that in most cases, and for practical instances of SAT, the set of search pruning techniques used by a backtrack search SAT algorithm plays a more significant role than the actual branching heuristic used.

5.2 Results for Different Branching Heuristics

In this section we use GRASP for evaluating the effect of branching heuristics in recent SAT algorithms. (GRASP was selected because it is the only algorithm that implements several branching heuristics proposed by different authors [13, 9, 4].)

The experimental results for GRASP, on the same classes of instances of the previous section, and for the branching heuristics described in Section 4, are shown in Tables 5, 6, 7, 8 and 9, which respectively present the number of aborted instances, the CPU times, the total number of decisions, the total number of backtrack steps taken, and the percentage of backtrack steps that were taken *non-chronologically*.

From the experimental results, the following conclusions can be drawn:

- GRASP obtains *similar* results with most branching heuristics. A more detailed analysis of the experimental data actually reveals substantial differences *only* for a few instances of the few hundred instances evaluated.

Table 5. Number of aborted instances for GRASP with different branching heuristics

Class	BOHM	DLCS	DLIS	JW-OS	JW-TS	MOM	RAND	RDLIS
aim-200	0	0	0	0	0	0	0	0
bf	0	0	0	0	0	0	0	0
dubois	0	0	0	0	0	0	1	0
ii16	2	2	1	1	1	1	0	0
ii32	1	1	0	1	1	1	1	0
jnh	0	0	0	0	0	0	0	0
pret	0	0	0	0	0	0	4	0
ssa	1	0	0	0	0	1	0	0
ucsc-bf	2	0	0	0	0	2	0	0
ucsc-ssa	6	0	0	0	0	6	0	0
Total	12	3	1	2	2	11	6	0

Table 6. GRASP CPU times with different branching heuristics

Class	BOHM	DLCS	DLIS	JW-OS	JW-TS	MOM	RAND	RDLIS
aim-200	1.03	3.20	3.42	1.60	1.39	0.78	1	3.18
bf	5.98	2.25	2.25	3.95	3.31	6.07	59.73	2.65
dubois	0.35	12.65	11.97	31.87	20.57	0.36	111.23	17.49
ii16	2043.32	970.78	431.16	696.94	940.92	1178.23	13.21	120.16
ii32	2179.03	858.19	2.01	345.39	334.21	1250.04	204.74	574.31
jnh	2.10	3.99	6.02	3.14	2.42	1.98	71.01	8.75
pret	1.88	3.58	3.58	3.89	3.87	1.88	809.00	3.34
ssa	167.86	4.07	2.56	4.62	7.29	168.35	2.47	2.87
ucsc-bf	550.43	100.57	83.54	88.08	93.17	526.06	159.35	115.24
ucsc-ssa	1195.58	34.54	24.87	47.21	225.21	1196.44	65.27	32.05

- For a few classes of instances, the branching heuristics most often used by SAT algorithms (i.e. BOHM and MOM) end up yielding worse results. Our interpretation is that these heuristics are simply too greedy, and can for some instances make too many bad branches.
- The plain, straightforward, randomized branching heuristic, RAND, compares favorably with the other heuristics, and actually performs better (in GRASP and for the classes of instances considered) than BOHM's or MOM's heuristics.
- Randomization can actually be a powerful branching mechanism. For example, while DLIS aborts one instance, RDLIS, by not being so greedy as DLIS, aborts none. In general the run times for RDLIS are slightly larger than those for DLIS, but RDLIS is less likely to make bad branches, that can cause a SAT algorithm to eventually quit on a given instance.

Another interesting result is the percentage of non-chronological backtrack steps (see Table 9). In general the percentages are similar for different heuristics, but differences do exist. Qualitatively, the percentages tend to be similar for three

Table 7. Number of decisions for GRASP with different branching heuristics

Class	BOHM	DLCS	DLIS	JW-OS	JW-TS	MOM	RAND	RDLIS
aim-200	1312	4009	4629	2480	1992	1199	5867	4080
bf	2086	865	1004	1142	1139	2150	13808	1083
dubois	1288	15117	15662	16481	15858	1288	71177	18399
ii16	53908	32167	20171	31367	42753	42003	5564	6644
ii32	29099	15549	648	19949	17616	24828	51376	11566
jnh	1181	2422	3198	1967	1454	1150	12102	3651
pret	4904	16584	16645	16312	16346	4892	93437	15241
ssa	14672	2587	1869	2755	3074	14674	3529	1780
ucsc-bf	81991	40792	35993	33424	31391	77818	104304	43890
ucsc-ssa	114049	23398	18193	26644	37940	114168	59245	19018

Table 8. Number of backtracks for GRASP with different branching heuristics

Class	BOHM	DLCS	DLIS	JW-OS	JW-TS	MOM	RAND	RDLIS
aim-200	830	1901	1996	1228	1183	744	1844	1934
bf	1178	404	422	641	725	1219	6360	472
dubois	902	3917	3935	5473	4851	902	15994	4267
ii16	48839	27320	13838	20799	24895	38674	1099	4563
ii32	25892	13868	59	11265	10848	22161	26464	8439
jnh	969	1919	2502	1416	1121	951	9552	2959
pret	1684	1870	1765	1847	1825	1688	43398	1684
ssa	10203	986	739	1139	1410	10202	1159	662
ucsc-bf	40598	12509	9393	11536	12364	38737	31872	14307
ucsc-ssa	61179	5273	4026	6822	16231	61170	15827	5087

main groups of heuristics. First for BOHM and MOM, second for DLIS, DLCS, JW-OS, JW-TS, and RDLIS, and finally, for RAND. In general the highest percentage of non-chronological backtrack steps is largest in RAND, which is to be expected.

6 Conclusions

This paper analyzes different backtrack search SAT algorithms and their use of branching heuristics. The obtained experimental results provide evidence that, even though branching heuristics play an important role in solving SAT, more significant performance gains are in general possible, for real-world instances of SAT, by using techniques for pruning the amount of search. Among these, we have studied the effect of non-chronological backtracking and clause recording, among others. Further validation of the conclusions presented in this paper can be obtained by extending the experimental analysis to other real-world applications of SAT, for which relevant sets of instances exist.

Techniques for pruning the amount of search have been proposed over the years in many different areas. The results presented in this paper motivate ex-

Table 9. Percentage of non-chronological backtracks in GRASP

Class	BOHM	DLCS	DLIS	JW-OS	JW-TS	MOM	RAND	RDLIS
aim-200	18.19	30.35	33.92	27.44	19.61	19.35	51.08	30.51
bf	28.35	28.22	38.86	22.62	16.14	28.22	50.25	36.86
dubois	4.32	42.53	45.67	23.61	25.44	4.32	69.48	47.11
ii16	5.41	5.34	7.46	15.73	18.69	4.53	66.33	10.65
ii32	6.46	5.65	42.37	26.18	20.00	6.41	42.22	18.63
jnh	4.64	10.89	13.47	13.56	7.58	4.63	17.94	12.77
pret	36.22	64.12	69.18	65.62	65.75	35.96	50.04	68.59
ssa	23.12	34.99	33.15	29.68	29.08	23.14	58.67	36.10
ucsc-bf	33.61	40.06	44.01	35.84	32.78	32.87	60.90	48.42
ucsc-ssa	33.35	37.78	37.98	33.35	30.69	33.35	64.12	40.26

ploring the application of additional search pruning techniques to SAT, with the goal of allowing state-of-the-art SAT algorithms to solve an ever increasing number of real-world instances of SAT.

References

1. P. Barth. A Davis-Putnam enumeration procedure for linear pseudo-boolean optimization. Technical Report MPI-I-2-003, MPI, January 1995.
2. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.
3. M. Bruynooghe. Analysis of dependencies to improve the behaviour of logic programs. In *Proceedings of the 5th Conference on Automated Deduction*, pages 293–305, 1980.
4. M. Buro and H. Kleine-Büning. Report on a SAT competition. Technical report, University of Paderborn, November 1992.
5. J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 22–28, 1993.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
7. D. Du, J. Gu, and P. M. Pardalos, editors. *Satisfiability Problem: Theory and Applications*, volume 35. American Mathematical Society, 1997.
8. O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.
9. J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, May 1995.
10. J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1979.
11. A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.

12. J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.
13. R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
14. D. S. Johnson and M. A. Trick, editors. *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.
15. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, 1996.
16. T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.
17. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, 1997.
18. J. P. Marques-Silva and K. A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, November 1996.
19. D. Pretolani. Efficiency and stability of hypergraph sat algorithms. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.
20. B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 290–295, 1993.
21. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 440–446, 1992.
22. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, October 1977.
23. R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, 1988.
24. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.