A Complete Adaptive Algorithm for Propositional Satisfiability

Renato Bruni and Antonio Sassano

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Buonarroti 12, I-00185 Rome, Italy

Abstract

An approach to propositional satisfiability using an adaptive procedure is described. Its main feature is a new branching rule, which is able to identify, at an early stage, hard sub-formulae. Such branching rule is based on a simple and easy computable criterion, whose merit function is updated by a learning mechanism, and guides the exploration of a clause based branching tree. Completeness is guaranteed. Moreover, a new search technique named core search is used to speed-up the above procedure while preserving completeness. This is inspired by the well known approaches of row and column generation used in mathematical programming. Encouraging computational results and comparisons are presented.

Key words: Backtracking, Search frameworks, Satisfiability.

1 Introduction

The problem of testing satisfiability of propositional formulae plays a main role in Mathematical Logic and Computing Theory. Satisfiability is fundamental in the fields of Artificial Intelligence, Expert Systems, and Deductive Databases, because propositional logic is able to formalize deductive reasoning. Satisfiability problems indeed are used for encoding and solving a wide variety of problems arising from different fields, e.g. VLSI logic circuit design and testing, computer aided design. Moreover, satisfiability for propositional logic formulae is a relevant member of the large family of NP-complete problems, which are nowadays identified as central to a number of areas in computing theory and engineering.

Email address: `bruni@dis.uniroma1.it` (Renato Bruni).

Logic formulae in CNF (conjunctive normal form) are logic conjunction ($\wedge$) of m clauses, which are logic disjunction ($\vee$) of literals, which can be either positive (α_k) or negative ($\neg\alpha_k$) propositions. A formula $\mathcal{F}$ has the following general structure:

$$(\alpha_{i_1} \vee ... \vee \alpha_{j_1} \vee \neg\alpha_{k_1} \vee ... \vee \neg\alpha_{n_1}) \wedge \ldots \wedge (\alpha_{i_m} \vee ... \vee \alpha_{j_m} \vee \neg\alpha_{k_m} \vee ... \vee \neg\alpha_{n_m})$$

Given a truth value (a value in the set $\{True, False\}$) for every proposition, a truth value for the whole formula is obtained. A formula is satisfiable if and only if there exists a truth assignment that makes the formula $True$, otherwise is unsatisfiable. Determining whether a formula is satisfiable or not is the *satisfiability problem*, SAT for short.

Many algorithms for solving the SAT problem have been proposed, based on different techniques (among others, [2,4,5,8–11,13,15,16,19,21,22,24,25,28–30,32], see also [3,7,14,17,18,23,31] for extensive references). A solution method is said to be complete if it is guaranteed (given enough time) to find a solution if it exists, or report lack of solution otherwise. Incomplete methods, on the contrary, cannot guarantee finding the solution, although they usually scale better then complete ones on many large problems. Most of complete methods are based on enumeration techniques. This paper is precisely concerned with such enumeration algorithms, also known as Davis-Putnam-Loveland variants [9,12,26,33]. They have the following general structure:

DPL scheme

1. Choose a variable α according to a branching rule [20]. Generally, priority is given to variables appearing in unit clauses (i.e. clauses containing only one literal).
2. Fix α to a truth value and cancel from the formula all satisfied clauses and all falsified literals, because they would not be able to satisfy the clauses where they appear.
3. If an empty clause is obtained (i.e. every literal is deleted from a clause which is still not satisfied) that clause would be impossible to satisfy. Therefore, backtrack and change former choices. Usually, a depth-first exploration of the search tree is performed.
The above is repeated until one of the two following conditions is reached:
- a satisfying solution is found: the formula is satisfiable.
- an empty clause is obtained and every truth assignment has been tried, i.e. the branching tree is completely explored: the formula is unsatisfiable.

Many different improvements to this procedure have been proposed, although,

of course, each of them performs well on some kind of formulae, while less well on another. One of the crucial choices seems to be the adopted branching rule. In fact, although it does not affect complexity of the worst case, it shows its importance in the average case, which is the one to deal with in real world.

A new technique to detect hard subsets of clauses is here proposed. Evaluation of clause hardness is based on the history of the search, and keeps improving throughout the computation, as illustrated in Section 2. Our branching rule consists in trying to satisfy at first such hard sets of clauses, while visiting a clause-based branching tree [6,20], as showed in Section 3. Moreover, a search technique that can speed-up enumeration is developed, as explained in Section 4. This new search technique is essentially based on the idea of considering only a hard subset of clauses (a *core*, as introduced in [27]), and solve it without propagating assignments to clauses out of this subset. Subsequently, such partial solution is extended to a larger subset of clauses, until solving the whole formula, or stopping at an unsatisfiable subformula. The proposed procedure is tested on problems from the DIMACS collection. Results and comparisons are in Section 5.

2 Individuation of Hard Clauses

Although a truth assignment S satisfies a formula $\mathcal{F}$ only when all C_j are satisfied, there are subsets $\mathcal{P} \subset \mathcal{F}$ of clauses which are more *difficult* to satisfy, i.e. which have a small number of satisfying truth assignments, and subsets which are rather *easy* to satisfy, i.e. which have a large number of satisfying truth assignments. In fact, every clause C_j actually *forbids* some of the 2^n possible truth assignments. Hardness of $\mathcal{F}$ is typically not due to a single clause in itself, but to a combination of several, or, in other words, to the combinations of any generic clause with the rest of the clauses in $\mathcal{F}$. Therefore, *hardness* of a clause C_j will hereafter mean *in the case when* C_j *belongs to the particular instance* $\mathcal{F}$ being solved. The following is an example of a $\mathcal{P} \subset \mathcal{F}$ constituted by short clauses containing always the same variables:

$$\ldots \quad C_p = (\alpha_1 \vee \alpha_2), \quad C_q = (\neg\alpha_1 \vee \neg\alpha_2), \quad C_r = (\alpha_1 \vee \neg\alpha_2), \quad \ldots$$

$\mathcal{P}$ restricts the set of satisfying assignment for $\mathcal{F}$ to those which have $\alpha_1 = True$ and $\alpha_2 = False$. Hence, $\mathcal{P}$ has the falsifying assignments $S_1 = \{\alpha_1 = False,\ \alpha_2 = False, \ldots\}$, $S_2 = \{\alpha_1 = True,\ \alpha_2 = True, \ldots\}$, $S_3 = \{\alpha_1 = False,\ \alpha_2 = True,\ \ldots\}$. Each S_i identifies 2^{n-2} (2 elements are fixed) different points of the solution space. Thus, $3(2^{n-2})$ points are forbidden. This number is as much as three forth of the number 2^n of points in the solution space. On the contrary, an example of $\mathcal{P} \subset \mathcal{F}$ formed by long clauses containing different

variables is:

$$\dots \quad C_p = (\alpha_1 \vee \neg\alpha_2 \vee \alpha_3), \quad C_q = (\alpha_4 \vee \neg\alpha_5 \vee \alpha_6), \quad C_r = (\alpha_7 \vee \neg\alpha_8 \vee \alpha_9), \quad \dots$$

In this latter case, $\mathcal{P}$ has the falsifying assignments $S_1 = \{\alpha_1 = False,\ \alpha_2 = True,\ \alpha_3 = False, \dots\dots\dots\ \}$, $S_2 = \{\dots, \alpha_4 = False,\ \alpha_5 = True,\ \alpha_6 = False, \dots\dots\}$, $S_3 = \{\dots\dots, \alpha_7 = False,\ \alpha_8 = True,\ \alpha_9 = False, \dots\}$. Each S_i identifies 2^{n-3} (3 elements are fixed) points of the solution space, but this time the S_i are not pairwise disjoint. 2^{n-6} of them falsifies 2 clauses at the same time (6 elements are fixed), and 2^{n-9} falsifies 3 clauses at the same time (9 elements are fixed). Thus, $3(2^{n-3}) - 3(2^{n-6}) + (2^{n-9})$ assignments are forbidden. This number, for values of n usually considered, is much less than before.

Starting assignment by satisfying the more difficult clauses, i.e. those which admit very few satisfying truth assignments, or, in other words, represent the more constraining relations, is known to be very helpful in reducing backtracks [20]. The point is how to find the hardest clauses. An *a priori* parameter is the length, which is quite inexpensive to calculate. In fact, unit clauses are universally recognized to be hard, and the procedure of unit resolution (almost universally performed) satisfies them at first. Other *a priori* parameters are not easy to formalize, and appear quite expensive to compute. Remember also that hardness is due both to the clause itself and to the rest of the instance.

In the course of our enumeration, a clause C_j is said to be *visited* during the exploration of the search tree if a truth assignment aimed at satisfying C_j is made. A *failure* occurs on a clause C_j either in the case when an empty clause is generated due to a truth assignment aimed at satisfying C_j, or in the case when C_j itself becomes empty due to some other truth assignment. The technique used here to evaluate the difficulty of a clause C_j is to count how many times C_j is visited, and how many times the enumeration fails on C_j. Visiting C_j many times shows that C_j is difficult, and failing on it shows even more clearly that C_j is difficult.

Clause hardness adaptive evaluation

Let v_j be the number of visits of clause C_j, f_j the number of failures due to C_j, p the penalty considered for failures, and l_j the length of C_j. An hardness evaluation of C_j in $\mathcal{F}$ is given by

$$\varphi(C_j) = (v_j + p f_j) \,/\, l_j$$

Such evaluation improves as the tree search proceeds. Counting visits and failures has the important feature of requiring very little overhead. As mentioned,

branching should be done in order to satisfy hard clauses first. Moreover, as widely recognized, unit clauses should be satisfied as soon as they appear in the formula, by performing all possible unit resolutions. Altogether, the following clause selection criterion is used:

Adaptive clause selection

(1) *Perform all unit resolutions.*
(2) *When no unit clauses are present, make a truth assignment satisfying the clause:*

$$C_{max} = \arg\max_{\substack{C_j \in \mathcal{F} \\ C_j \text{ still unsatisfied}}} \varphi(C_j)$$

The variable assignment will be illustrated in the next section, after introduction of a not binary tree search paradigm. Due to the above adaptive features, the proposed procedure can perform well on problems which are difficult for algorithms using *static* branching rules.

3 Clause based Branching Tree

Since our aim is to satisfy C_{max}, the choice for variable assignments is restricted to variables in C_{max}. A variable α_a appearing positive must be fixed at *True*, and a variable appearing negative must be fixed at *False* [6]. If such a truth assignment causes a failure, i.e. generates an empty clause, and thus backtrack is needed, the next assignment would not be just the opposite truth value for the same variable α_a, because this would not satisfy C_{max}. Instead, the procedure backtracks and selects another variable α_b in C_{max}. Moreover, since the former truth assignment for α_a was not successful, the opposite truth value for α_a can also be fixed. The resulting node structure is shown in Fig. 1. If there are no more free variables in C_{max}, or if all of them were tried without success, the procedure backtracks to the truth assignments made to satisfy the previous clause, and proceeds this way until no backtrack is possible. Therefore, a clause based branching tree is visited.

The above is a complete scheme: if a satisfying truth assignment exists, it will be reached, and, if the search tree is completely explored, the instance is unsatisfiable. Completeness would be guaranteed even in the case of branching only on all-positive clauses [20] (or, for instance, on all-negative). However, being our aim to select a set of hard clauses, as explained below, this could not be realized by selecting only all-positive clauses.

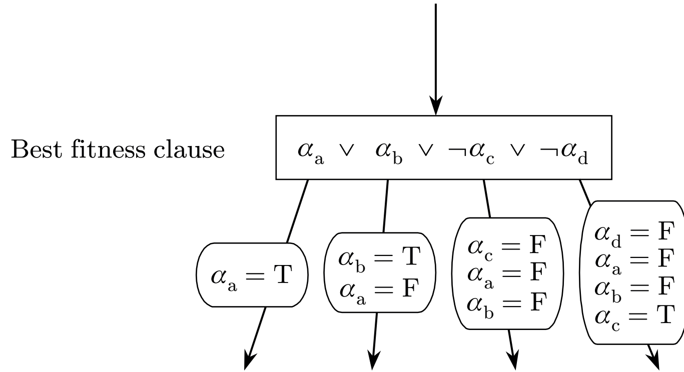

Fig. 1. Branching node structure. An example of selected clause appears in the rectangle, and the consistent branching possibilities appear in the ellipses

Our branching tree is not binary: every node has as many successors as the number of unassigned variables appearing in C_{max}. In practical case, however, a small part of this successors need to be explored [17]. On the other hand, useless truth assignments, namely those containing values not satisfying any still unsatisfied cause, are avoided. At present, variable assignment order is just their original order within C_{max}, because used reordering schemes seem not to improve computational times.

4 Adaptive Core Search

The above scheme can be modified in order to speed-up the entire procedure. Roughly speaking, the idea is that, when a hard subset of clauses, called a *core*, is detected, the search can work on it, just ignoring other clauses. After solving such core, if that is unsatisfiable, the whole formula is unsatisfiable. Conversely, if the core admits a satisfying solution, the extension of that solution to a larger subset of clauses is attempted, until solving the whole formula. The algorithm works as follows:

Adaptive core search

0. (Preprocessing) *Perform d branching iterations using just shortest clause rule. If the instance is already solved, Stop.*

1. (Base) *Select an initial collection of hardest clauses $\mathcal{C}_1$, by choosing the highest values of φ. This is the first core. Remaining clauses form $\mathcal{O}_1$.*

k. (Iteration) *Perform b branching iteration on $\mathcal{C}_k$, ignoring $\mathcal{O}_k$, using adaptive clause selection. One of the following* 1, 2, 3:

k.1. *$\mathcal{C}_k$ is unsatisfiable $\Rightarrow \mathcal{F}$ is unsatisfiable, then Stop.*

k.2. *No answer after b iterations $\Rightarrow$ select a new collection of hardest clauses $\mathcal{C}_{k+1}$ within $\mathcal{C}_k$, by choosing highest values of φ. Put $k := k+1$, goto* **k.**

k.3. *$\mathcal{C}_k$ is satisfied by solution $S_k \Rightarrow$ try S_k on $\mathcal{O}_k$.*
One of the following a, b, c:

k.3.*a* *All clauses are satisfied $\Rightarrow \mathcal{F}$ is satisfied, then Stop.*

k.3.*b* *There is a set $\mathcal{T}_k$ of falsified clauses $\Rightarrow$ add them to the core: put $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{T}_k$, $k := k+1$, goto* **k.**

k.3.*c* *No clauses are falsified, but there is a set $\mathcal{V}_k$ of still not satisfied clauses $\Rightarrow$ select a collection $\mathcal{C}'_k$ of hardest clauses in $\mathcal{V}_k$ by choosing the highest values of φ, put $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{C}'_k$, $k := k+1$, goto* **k.**

The preprocessing step has the aim to give initial values of visits and failures, in order to compute φ. The selection of hardest clauses in steps **1**. **k**.2 and **k**.3.*c* is done by choosing a fixed percentage c of the number of clauses contained in the set where the hardest clauses are being selected. When a satisfying solution S_k for the core is found, this has to be extended to the rest of the clauses $\mathcal{O}_k$. If some clauses of $\mathcal{O}_k$ are falsified by S_k, this means that they are difficult (together with the clauses of the core), and therefore they should be added to the core. In this case, since S_k falsifies some clauses now in the core, it is faster to completely rebuilt S_k. The iteration step is repeatedly applied to instances until their solution. In order to ensure termination to the above procedure, solution rebuilding is allowed only a finite number of times. After that, the solution is not entirely rebuilt, but modified by performing backtrack.

Core search framework has the important feature of considering only a subproblem (current core $\mathcal{C}_k$) at the nodes of the search tree. Hence, all operations performed, such like unit propagation consequent to any variable fixing, are performed only on a subset of clauses, with consequent speed up of node processing.

Moreover, finding a core is of great relevance in many practical applications. Typically, when a SAT instance encodes our application, this SAT instance should have a certain solution property (either to be satisfiable or to be unsatisfiable). When the instance does not have such property, the application should be modified in order to make the SAT instance as desired. Being the core a small unsatisfiable subformula, that helps in locating the part to modify if a satisfiable instance is desired, and the part to keep unchanged otherwise.

5 Computational Results

The algorithm was coded in C++. The following results are obtained on a Pentium II 450 MHz processor running MS Windows NT operating system.

In the tables, columns n and m shows respectively the number of variables and the number of clauses. Column lit is the number of all literals appearing in the formula, sol reports if satisfiable or unsatisfiable. Column labeled ACS reports times for solving the instance by Adaptive Core Search. Other table specific columns are described in following subsections. Times are in CPU seconds. Time limit was set at 600 sec. If exceeded, > 600 is reported. If not available, n.a. is reported. Parameter p appearing in hardness evaluation function φ was set at 10. Percentage c of hardest clauses selected in core search was set at 30%. During our experiments, in fact, such choices gave better and more uniform results. Parameter d and b range between 100 and 10000. Test problems are chosen from the DIMACS [1] collection, since they are widely-known, and the test instances, together with computational results, are easily available [2].

Running times of Adaptive Core Search are compared with those of other complete algorithms. In particular, the following tests are considered:

- ACS compared with two simpler versions of it (ABR and SCBR).
- ACS compared with two well known state-of-the-art SAT-solvers (SATO [32] and GRASP [28]).
- ACS compared with the four best complete algorithms of the second DIMACS challenge on Satisfiability [23].

In the first two comparisons, the algorithms run on our machine. In the third one, times reported in the original papers are "normalized" as if they were obtained on our same machine.

5.1 Comparison with two simpler versions of ACS

Adaptive Core Search is here compared with two simpler branching algorithm: Adaptive Branching Rule and Shortest Clause Branching Rule. Adaptive Branching Rule is a branching algorithm which does not use core search, but does use the adaptive branching rule based on φ. Its results are in column labeled ABR. Shortest Clause Branching Rule is a branching algorithm which does not use core search, and just uses shortest-clause-first branching rule. Its results are in column labeled $SCBR$. Such comparison is interesting because the three algorithms use the same data structure and share most of the code. Therefore, they perform a similar node processing. Number of backtracks, i.e. the total number of variable fixings undone during the whole search, is also

[1] NFS Science and Technology Center in Discrete Mathematics and Theoretical Computer Science - A consortium of Rutgers University, Princeton University, AT&T Bell Labs, Bellcore.

[2] Available from
`ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`

reported.

The comparison is on the DIMACS series *ii32*, since they are quite representative of real problems. Such instances encode inductive inference problems, and are contributed from M.G.C. Resende [24]. They essentially contain two kind of clauses: a set of binary clauses and a set of long clauses. Their size is quite big. Results are in table 1. ACS clearly is the fastest, and solves all problems in remarkably short times. ABR is generally faster than SCBR, although not always. The very simple SCBR is sometimes quite fast, but its results are very changeable, and in most of the cases exceeds the time limit. Moreover, ACS is only slightly slower in node processing, proving that our clause hardness evaluation requires a small computational overhead.

Problem features					Time			Backtracks		
Name	n	m	*lit*	*sol*	ACS	ABR	SCBR	ACS	ABR	SCBR
ii32a1	459	9 212	33 003	SAT	0.02	475.57	> 600	368	10 867 470	n.a.
ii32b1	228	1 374	6 180	SAT	0.00	20.65	356.74	244	227 150	6 584 353
ii32b2	261	2 558	12 069	SAT	0.03	36.56	> 600	967	401 242	n.a.
ii32b3	348	5 734	29 340	SAT	0.03	108.57	> 600	783	863 541	n.a.
ii32b4	381	6 918	35 229	SAT	1.53	311.62	> 600	12 895	2 361 343	n.a.
ii32c1	225	1 280	6 081	SAT	0.00	2.67	1.75	63	29 850	31 734
ii32c2	249	2 182	11 673	SAT	0.00	27.29	0.02	174	286 545	340
ii32c3	279	3 272	17 463	SAT	2.84	57.03	> 600	24 780	570 673	n.a.
ii32c4	759	20 862	114 903	SAT	5.07	> 600	> 600	16 311	n.a.	n.a.
ii32d1	332	2 730	9 164	SAT	0.01	409.21	> 600	350	3 285 423	n.a.
ii32d2	404	5 153	17 940	SAT	0.76	> 600	> 600	7 306	n.a.	n.a.
ii32d3	824	19 478	70 200	SAT	7.49	> 600	> 600	15 554	n.a.	n.a.
ii32e1	222	1 186	5 982	SAT	0.00	1.24	0.01	92	11 762	189
ii32e2	267	2 746	12 267	SAT	0.01	82.13	> 600	308	903 430	n.a.
ii32e3	330	5 020	23 946	SAT	0.08	131.38	> 600	1 259	1 051 899	n.a.
ii32e4	387	7 106	35 427	SAT	0.02	312.28	> 600	470	2 273 967	n.a.
ii32e5	522	11 636	49 482	SAT	1.03	382.36	> 600	3 955	1 995 834	n.a.

Table 1: Results of *ACS*, *ABR* and *SCBR* on the *ii32* series: inductive inference problems. From M.G.C. Resende.

5.2 Comparison with SATO and GRASP

A comparison is here given with the latest versions of two of the most representative state-of-the-art sat solver: *SATO* (SAtisfiability Testing Optimized)

version 3.2 [3], developed by H. Zhang [32], and *GRASP* (Generic seaRch Algorithm for the Satisfiability Problem) version 2000.00 [4], developed by J.P. Marques-Silva and K.A. Sakallah [28]. Both are very sophisticated DPL-variants, including several speed-up mechanisms. Sublinear time unit propagation, intelligent backjumping, mixed branching rule, restricted learning of new clauses for SATO [32], and conflict analysis procedure, non-chronological backtrack, recording of the causes of conflicts for GRASP [28].

For this comparisons, the focus is on computational times. Number of backtracks per second is also reported. Computational tree size does not appear meaningful because the different solvers do not perform similar node processing.

The comparison is on the DIMACS series *par16*, since they are representative of real problems, and have a useful peculiarity. Instances arise from the problem of learning the parity function, for a parity problem on 16 bits, and are contributed from J. Crawford. They contain clauses of different length: unit, binary and ternary. Their size is sometimes remarkably large. The peculiarity is that par16-x-c is an instance representing a problem equivalent to the corresponding par16-x, except that the first instance have been expressed in a somehow *compressed* form. This allows to test algorithms' effectiveness, giving less weight to the finishing touches of implementation.

Problem features					Time			Backtracks/second		
Name	*n*	*m*	*lit*	*sol*	ACS	SATO	GRASP	ACS	SATO	GRASP
par16-1	1015	3310	8788	S	10.10	24.16	340.02	29 621	375	20
par16-1-c	317	1264	3670	S	11.36	2.62	184.93	30 642	1 853	34
par16-2	1015	3374	9044	S	52.36	49.22	195.38	19 780	250	37
par16-2-c	349	1392	4054	S	100.73	128.15	3 251.10	15 622	318	8
par16-3	1015	3344	8924	S	103.92	40.81	45.46	13 875	255	70
par16-3-c	334	1332	3874	S	8.19	78.91	8.04	25 921	335	173
par16-4	1015	3324	8844	S	70.82	1.51	70.02	15 950	1 325	55
par16-4-c	324	1292	3754	S	5.10	133.07	216.24	27 353	252	31
par16-5	1015	3358	8980	S	224.84	4.92	8 385.03	11 423	694	4
par16-5-c	341	1360	3958	S	72.29	196.33	6 014.45	16 880	158	5

Table 2: Results of *ACS* 1.0, *SATO* 3.2 and *GRASP* 2000.00 on the *par16* series: instances arisen from the problem of learning the parity function. From J. Crawford.

Results are in table 2. They are extremely encouraging. With regards to SATO,

[3] Available from `ftp.cs.uiowa.edu/pub/sato/`

[4] Available from `http://sat.inesc.pt/~jpms/grasp/`

a sort of complementarity in computational time results can be observed: ACS is fast on the *compressed* versions of the problems, where SATO is slow. The converse happen on the *expanded* versions. Our hypothesis is that ACS is faster when it can take advantage of the identification of the hard part of the instances, but, due to an implementation and a data structure still not refined as in SATO, has more difficulties on larger instances. On the contrary, due to its very carefully implementation, which has been improved for several years, SATO 3.2 can handle more efficiently larger instances, but on smaller and harder instances, it cannot compensate the advantages of adaptive branching and core search. With regards to GRASP, no clear dependency from problem features can be observed. Nevertheless ACS results faster and more reliable. By analyzing number of backtracks per second, it results that ACS has a node processing which is orders of magnitude faster. In fact, ACS often sacrifice total number of nodes for a much faster node processing.

5.3 Comparison with algorithms from the second DIMACS challenge on Satisfiability

The series *aim100* is constituted by 3-SAT instances artificially generated by K. Iwama, E. Miyano and Y. Asahiro [1], and have the peculiarity that the satisfiable ones admit only one satisfying truth assignment. Such instances are not big in size, but can be very difficult. Some instances from this set were used in the test set of the Second DIMACS Implementation Challenge [23]. Our results are here compared with those of the four faster complete algorithms of that challenge. *C-sat*, presented by O. Dubois, P. Andre, Y. Boufkhad and J. Carlier [11], is a backtrack algorithm with a specialized branching rule and a local preprocessing at the nodes of search tree. It is considered a very fast algorithm. *2cl*, presented by A. Van Gelder and Y. K. Tsuji [13], consists in a combination of branching and limited resolution. *TabuS*, presented by B. Jaumard, M. Stan and J. Desrosiers [21], is an exact algorithm which includes a tabu search heuristic and reduction tests other than those of the Davis-Putnam-Loveland scheme. *BRR*, presented by D. Pretolani [29], makes use of directed hypergraph transformation of the problem, to which it applies a B-reduction, and of a pruning procedure.

In order to compare times taking into account machine performance, this is measured by using the DIMACS benchmark *dfmax* [5], although it had to be slightly modified to be compiled with our compiler. The measure of our machine performance in CPU seconds is therefore:

[5] Available from
`ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/volume/Machine/`

r100.5.b= 0.01 r200.5.b= 0.42 r300.5.b= 3.57 r400.5.b= 22.21 r.500.b= 86.63

Time reported in the original papers are therefore "normalized", as if all of the above solvers run on our same machine. Results are in table 3. A noticeable performance superiority of ACS can be observed, expecially on unsatisfiable problems.

Problem features					Time				
Name	n	m	*lit*	*sol*	ACS	C-sat	2cl	TabuS	BRR
aim-100-1_6-no-1	100	160	480	UNSAT	0.20	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-no-2	100	160	480	UNSAT	0.93	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-no-3	100	160	480	UNSAT	1.35	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-no-4	100	160	480	UNSAT	0.96	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-1	100	160	479	SAT	0.09	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-2	100	160	479	SAT	0.03	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-3	100	160	480	SAT	0.26	n.a.	n.a.	n.a.	n.a.
aim-100-1_6-yes1-4	100	160	480	SAT	0.01	n.a.	n.a.	n.a.	n.a.
aim-100-2_0-no-1	100	200	600	UNSAT	0.01	52.19	19.77	409.50	5.78
aim-100-2_0-no-2	100	200	600	UNSAT	0.38	14.63	11.00	258.58	0.57
aim-100-2_0-no-3	100	200	598	UNSAT	0.12	56.63	6.53	201.15	2.95
aim-100-2_0-no-4	100	200	600	UNSAT	0.11	0.05	11.66	392.23	4.80
aim-100-2_0-yes1-1	100	200	599	SAT	0.03	0.03	0.32	16.75	0.29
aim-100-2_0-yes1-2	100	200	598	SAT	0.09	0.03	0.21	0.24	0.43
aim-100-2_0-yes1-3	100	200	599	SAT	0.22	0.03	0.38	2.10	0.06
aim-100-2_0-yes1-4	100	200	600	SAT	0.04	0.12	0.11	0.03	0.03
aim-100-3_4-yes1-1	100	340	1019	SAT	0.44	n.a.	n.a.	n.a.	n.a.
aim-100-3_4-yes1-2	100	340	1017	SAT	0.53	n.a.	n.a.	n.a.	n.a.
aim-100-3_4-yes1-3	100	340	1020	SAT	0.01	n.a.	n.a.	n.a.	n.a.
aim-100-3_4-yes1-4	100	340	1019	SAT	0.12	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-1	100	600	1797	SAT	0.08	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-2	100	600	1799	SAT	0.07	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-3	100	600	1798	SAT	0.19	n.a.	n.a.	n.a.	n.a.
aim-100-6_0-yes1-4	100	600	1796	SAT	0.04	n.a.	n.a.	n.a.	n.a.

Table 3: Results of *ACS*, *C-SAT*, *2cl*, *DPL+tabu search*, *B-reduction*, on the *aim-100* series: 3-SAT artificially generated problems. From K. Iwama, E. Miyano and Y. Asahiro. Times are normalized according to *dfmax* results, as if they were obtained on the same machine.

6 Conclusions

A clause based tree search paradigm for satisfiability testing, which makes use of a new adaptive branching rule, and the original technique of core search, used to speed-up the procedure although maintaining the feature of complete method, are presented. The obtained enumeration technique is altogether denominated Adaptive Core Search, and is able to sensibly reduce computational times.

By using the above technique, a better performance improvement on instances which are "not uniformly" hard, in the sense that they contain subsets of clauses having different levels of "difficulty", can be observed. This is mainly due to the ability of our adaptive device in pinpointing hard sub-formulae during the branching tree exploration. Techniques to perform a fast complete enumeration are widely proposed in literature. Adaptive Core Search, on the contrary, can reduce the set that enumeration works on, and scales particularly well when such set is small.

Comparison of ACS with two simpler versions of it, one not using core search, and one not using neither core search nor the adaptive part of the branching rule, clearly reveals the great importance of this two strategies, and the reduced overhead required for their computation. Comparison with several published results shows the effectiveness of the proposed procedure. Comparison of ACS with state-of-the-art solvers is particularly encouraging. In fact, ACS, in its first implementation, is frequently faster then SATO 3.2 and GRASP 2000.00, whose implementation has evolved for several years. Running times can probably still improve on larger instances by further polishing our implementation, and by using several techniques available in the literature to perform a fast enumeration. Example of this could be to reduce clause revisiting by saving and reusing global inferences revealed during search, as many other modern solvers do [2,28,32].

References

[1] Y. Asahiro, K. Iwama and E. Miyano. Random Generation of Test Instances with Controlled Attributes. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26** (1996) 377–393.

[2] R.J. Bayardo and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. *In Proceedings AAAI-97* (1997).

[3] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A complexity index for Satisfiability Problems. *SIAM Journal on Computing* **23** (1994) 45–49.

[4] R. Bruni and A. Sassano. Finding Minimal Unsatisfiable Subformulae in Satisfiability Instances. *in proc. of 6th Internat. Conf. on Principles and Practice of Constraint Programming*, Lecture notes in Computer Science 1894 (Springer-Verlag, 2000) 500–505.

[5] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers* **C35**(8) (1986) 677–691.

[6] K.M. Bugrara and P.W. Purdom. Clause order backtracking. *Technical Report 311, Indiana University* (1990).

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

[8] J. Crawford and L. Auton. Experimental results on the crossover point in Satisfiability problems. *In Proceedings AAAI-93* (1993) 22–28.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. Assoc. for Comput. Mach.* **5** (1962) 394–397.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Jour. Assoc. for Comput. Mach.* **7** (1960) 201–215.

[11] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **26** (1996) 415–436.

[12] J. Franco and M. Paull. Probabilistic analysis of the Davis Putnam procedure for solving the Satisfiability Problem. *Discrete Applied Mathematics* **5** (1983) 77–87.

[13] A. Van Gelder and Y.K. Tsuji. Satisfiability testing with more reasoning and less guessing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26** (1996) 559–586.

[14] I.P. Gent, H. van Maaren, and T. Walsh editors. *SAT 2000* (IOS Press, Amsterdam, 2000).

[15] J. Groote and J. Warners. The propositional formula checker HeerHugo. In I.P. Gent, H. van Maaren, and T. Walsh eds. *SAT 2000* (IOS Press, Amsterdam, 2000).

[16] J. Gu. Optimization Algorithms for the Satisfiability (SAT) Problem. In Ding-Zhu Du ed. *Advances in Optimization and Approximation* (Kluwer Academic Publishers, 1994) 72–154.

[17] 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).

[18] F. Harche, J.N. Hooker, and G.L. Thompson. A computational study of Satisfiability Algorithms for Propositional Logic. *ORSA Journal on Computing* **6** (1994) 423–435.

[19] J.N. Hooker and C. Fedjki. Branch and Cut solution of Inference Problems in Propositional Logic. *Annals of Mathematics and AI* **1** (1990) 123–139.

[20] J.N. Hooker and V. Vinay. Branching Rules for Satisfiability. *Journal of Automated Reasoning* **15** (1995) 359–383.

[21] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the Satisfiability Problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26** (1996) 457–477.

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

[23] D.S. Johnson and M.A. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume **26** of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. (American Mathematical Society, 1996).

[24] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende A continuous approach to Inductive Inference. *Mathematical Programming* **57** (1992) 215–238.

[25] E. de Klerk, H. van Maaren, and J.P. Warners. Relaxations of the Satisfiability Problem using Semidefinite Programming. In I.P. Gent, H. van Maaren, and T. Walsh eds. *SAT 2000* (IOS Press, Amsterdam, 2000).

[26] D.W. Loveland. *Automated Theorem Proving: a Logical Basis*. (North Holland, 1978).

[27] C. Mannino and A. Sassano. Augmentation, Local Search and Learning. *AI-IA Notizie* XIII (2000) 34–36.

[28] J.P. Marques-Silva and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers* **48**(5) (1999) 506-521.

[29] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26** (1996) 479–498.

[30] G. Stålmarck and M. Säflund. Modeling and Verifying Systems and Software in Propositional Logic. *in proc. of Internat. Conf. on Safety of Computer Control Systems* (Pergamon Press, Oxford, 1990) 31-36.

[31] K. Truemper. *Effective Logic Computation*. (Wiley, New York, 1998).

[32] H. Zhang. SATO: An Efficient Propositional Prover. *in Proc. of International Conference on Automated Deduction (CADE-97)*, Lecture notes in Artificial Intelligence 1104 (Springer-Verlag, 1997) 308–312.

[33] H. Zhang and 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).