A Complete Adaptive Algorithm for Propositional Satis¯ability

Renato Bruni[z] and Antonio Sassano[x]

November 20, 2000

Abstract

We describe an approach to propositional satis¯ability which makes use of an adaptive technique. 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, we use a new search technique (Adaptive core search) to speed-up the procedure while preserving completeness. Encouraging computational results and comparisons are presented.

Keywords: Backtracking, NP-completeness, Satis¯ability.

1 Introduction

The problem of testing satis¯ability of propositional formulae plays a main role in Mathematical Logic and Computing Theory. Actually, it is fundamental in Arti¯cial Intelligence, Expert Systems, Deductive Database Theory, due to its ability of formalizing deductive reasoning, and thus solving logic problems by means of automated computation. Satis¯ability problems indeed are used for encoding and solving a wide variety of problems arisen from di®erent ¯elds, e.g. VLSI logic circuit design and testing, programming language project, computer aided design. Moreover, satis¯ability for

[z]Dipartimento di Informatica e Sistemistica, Universitµa di Roma \La Sapienza", via Buonarroti 12 - 00185 Roma, Italy. E-mail: bruni@dis.uniroma1.it

[x]Dipartimento di Informatica e Sistemistica, Universitµa di Roma \La Sapienza", via Buonarroti 12 - 00185 Roma, Italy. E-mail: sassano@dis.uniroma1.it

propositional logic formulae is a relevant member of the large family of NP-complete problems, which are nowadays identi¯ed as central to a number of areas in computing theory and engineering.

Logic formulae in CNF (conjunctive normal form) are logic conjunction (^) of m clauses, which are logic disjunction (_) of literals, which can be either positive ($®_k$) or negative ($:®_k$) propositions. A formula F has the following general structure:

$$(®_{i_1} \_ ::: \_ ®_{j_1} \_ :®_{k_1} \_ ::: \_ :®_{n_1}) \wedge ::: \wedge (®_{i_m} \_ ::: \_ ®_{j_m} \_ :®_{k_m} \_ ::: \_ :®_{n_m})$$

Given a truth values (a value in the set fTrue; Falseg) for every proposition, we have a truth value for the whole formula. A formula is satis¯able if and only if there exists a truth assignment that makes the formula True, otherwise is unsatis¯able. Deternining whether formula is satis¯able or not is called the satis¯ability problem, SAT for short.

Many algorithms for solving the SAT problem have been proposed. Examples are [9, ?, 4, 11]. A solution method is said to be complete if it is guaranteed (given enough time) to ¯nd a solution if it exists, or report lack of solution otherwise. Incomplete methods, on the contrary, cannot guarantee ¯nding the solution, but usually scale better then complete ones on many large problems. Most of complete methods are based on enumeration techniques, such as the Davis-Putnam-Loveland [8, 22, 29].

In this paper we are concerned with Davis-Putnam-Loveland variants, i.e. methods which have the following structure:

DPL scheme

1. Choose a variable ® according to a branching rule, e.g. [2, 8, 13, 19]. Generally, we give priority to variables appearing in unit clauses (i.e. clauses containing only one literal).

2. Fix ® to a truth value and cancel from the formula all satis¯ed clauses and all falsi¯ed 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 satis¯ed) that clause would be impossible to satisfy. We therefore need to backtrack and change former choices. Usually, we change the last truth assignment, by switching its truth value, or, if both of them are already tried, the last but one, and so on. This means a depth-¯rst exploration of the search tree.

The above is repeated until one of the two following conditions is reached:

- a satisfying solution is found: the formula is satis¯able.
- an empty clause is obtained and every truth assignment has been tried, i.e. the branching tree is completely explored: the formula is unsatis¯able.

There have been proposed many di®erent improvements to this procedure, and of course each of them performs well on some kind of formulae while bad on another. A crucial choice seems to be the adopted branching rule. In fact, although it does not a®ect complexity of the worst case, it shows its importance in the average case, which is the one we have to deal with in real world.

We propose a technique to detect hard subsets of clauses. 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 ¯rst such hard sets of clauses, while visiting a clause-based branching tree [5, 17], as showed in section 3. Moreover, we develop a search technique that can speed-up enumeration, as explained in section 4. It is essentially based on the idea of considering only a hard subset of clauses (a core, as introduced in [23]), and solve it without propagating assignments to clauses out of this subset. Subsequently, we extend such partial solution to a bigger subset of clauses, until solving the whole formula. The proposed procedure is tested on a set of arti¯cially generated hard problems from the Dimacs collection. Results are in section 5.

2 Individuation of hard clauses

Although a truth assignment S satis¯es a formula F only when all C_j are satis¯ed, there are subsets P ½ F of clauses which are more di±cult to satisfy, i.e. which have a small number of satisfying truth assignment S, and subsets which are rather easy to satisfy, i.e. which have a large number of satisfying truth assignment S. In fact, every clause C_j actually forbids some of the 2^n possible truth assignments.

Hardness of 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 F. Therefore, we will speak of hardness of a clause C_j in the case when C_j belongs to the particular instance F we are solving, and this would often be implicit. The same clause can, in fact, make difficult an instance A, because it combines in an unfortunate way with other clauses, while let another instance B be easy.

The following is an example of a $P \subset F$ constituted by short clauses containing always the same variables:

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

P restricts the set of satisfying assignment for F to those which have $\alpha_1 = \text{True}$ and $\alpha_2 = \text{False}$. Hence, P has the falsifying assignments:

$$S_1 = \{\alpha_1 = \text{False};\ \alpha_2 = \text{False}; \ldots\}$$

$$S_2 = \{\alpha_1 = \text{True};\ \alpha_2 = \text{True}; \ldots\}$$

$$S_3 = \{\alpha_1 = \text{False};\ \alpha_2 = \text{True}; \ldots\}$$

Each S_i identifies 2^{n-2} (2 elements are fixed) different points of the solution space. Thus, we forbid $3(2^{n-2})$ points. 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 $P \subset F$ constituted by long clauses containing different variables is:

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

In this latter case, P has the falsifying assignments:

$$S_1 = \{\alpha_1 = \text{False};\ \alpha_2 = \text{True};\ \alpha_3 = \text{False}; \ldots\ldots\ldots\}$$

$$S_2 = \{\ldots;\alpha_4 = \text{False};\ \alpha_5 = \text{True};\ \alpha_6 = \text{False}; \ldots\ldots\}$$

$$S_3 = \{\ldots\ldots;\alpha_7 = \text{False};\ \alpha_8 = \text{True};\ \alpha_9 = \text{False}; \ldots\}$$

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, we forbid $3(2^{n-3}) - 3(2^{n-6}) + (2^{n-9})$ assignments. This number, for values of n we deal with, is much less then before. Hence, this P does not restrict too much the set of satisfying assignment for F.

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 [3, 17]. This holds because, if such clauses are considered somewhere deep in the branching tree, where many possible truth assignments are already dropped, they would probably result impossible to satisfy, and would cause to backtrack far. If, on the contrary, such clauses are considered at the beginning of the branching tree, they would cause to drop a lot of truth assignments, but they would be satisfied earlier, or, if this is not possible (because they are an unsatisfiable set), unsatisfiability would be detected faster. Indeed, there would be no need to backtrack far. As for clauses considered deep in the branching tree, they should be the easier ones, which would probably not cause any backtrack.

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 propagation, which is universally performed, satisfies them at first. Other *a priori* parameters could be the observations made before, not exactly formalized, but probably quite expensive to compute. Remember also that hardness is due both to the clause itself and to the rest of the instance. For the above reasons, a merely *a priori* evaluation is not easy to carry on.

We say that a clause C_j is visited during the exploration of the tree if we make a truth assignment aimed at satisfying C_j. The technique we used to evaluate the difficulty of a clause C_j when appearing in the particular instance F, is to count how many times C_j is visited during the exploration of the tree, and how many times the enumeration fails on C_j. Failures can be either because an empty clause is generated due to truth assignment made on C_j, or because C_j itself becomes empty. Visiting C_j many times shows that C_j is difficult, and failing on it shows even more clearly that C_j is difficult. Counting visits and failures has the important feature of requiring very little overhead.

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 F is given by

$$\varphi(C_j) = (v_j + pf_j)/l_j$$

Therefore, during the elaborations performed by a DPL-style procedure, we can evaluate clause hardness. As told, we choose to branch in order to satisfy hard clauses ¯rst. Moreover, as widely recognized, unit clauses should be satis¯ed as soon as we have them in the formula, by performing all unit resolutions. Altogether, we use the following branching rule:

Adaptive clause selection

1. Perform all unit resolutions.

2. When no unit clauses are present, make a truth assignment satisfying the clause:

$$C_{max} = \arg \max_{\substack{C_j \ 2 \ F \\ C_j \text{ still unsat.}}} \text{'}(C_j)$$

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

3 Clause based Branching Tree

Being our aim to satisfy C_{max}, the choice is restricted to variables in C_{max}. A variable ®$_a$ appearing positive must be ¯xed at True, and a variable appearing negative must be ¯xed at False [5]. If such a truth assignment causes a failure, i.e. generates an empty clause, and thus we need to backtrack and change it, the next assignment would not be, as usual, the opposite truth value for the same variable ®$_a$, because it would not permit to satisfy C_{max}. Instead, we backtrack and select another variable ®$_b$ in C_{max}. Moreover, since the former truth assignment for ®$_a$ was not successful, we can also ¯x the opposite truth value for ®$_a$. The resulting node structure is shown in ¯gure 1. If we have no more free variables in C_{max}, or if we tried all of them without success, we backtrack to the truth assignments made to satisfy the previous clause, until we have another choice.

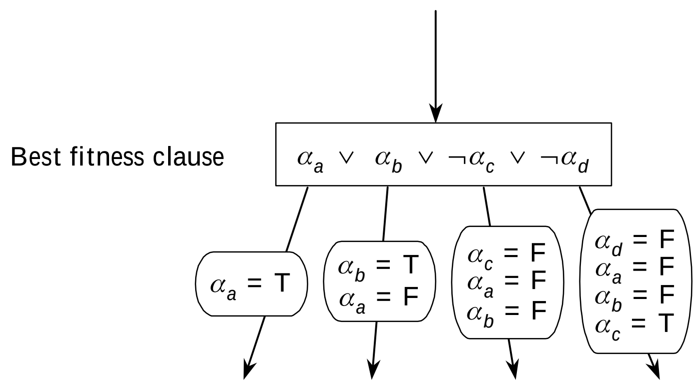
Best fitness clause
$\alpha_a \vee \alpha_b \vee \neg\alpha_c \vee \neg\alpha_d$
$\alpha_a = T$
$\alpha_b = T$, $\alpha_a = F$
$\alpha_c = F$, $\alpha_a = F$, $\alpha_b = F$
$\alpha_d = F$, $\alpha_a = F$, $\alpha_b = F$, $\alpha_c = T$

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

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 unsatis¯able. Completeness is guaranteed by being this just a branch-and-bound scheme. Completeness would be guaranteed even in the case of branching only on all-positive clauses [17] (or on all-negative). However, being our aim to select a set of hard clauses, as explained below, this could not be reached by selecting only all-positive clauses.

This scheme leads to explore a branching tree that is not, in general, binary: every node has as many successors as the number of unassigned variables appearing in C_{max}. In practical case, however, very few of this successors need to be explored. On the other hand, we can avoid even to try some truth assignments: the useless ones, namely those containing values which do not satisfy any still unsatis¯ed cause.

As usual in branching techniques, the solution that satis¯es the entire set of the clauses may contain some variables that are still free, i.e. not assigned. This happens when such variables were not used to satisfy clauses, so their value can be called "don't care". If d is the number of variables put to "don't care", the number of satisfying solutions trivially is 2^d. They are explicitly obtainable by substitution of each "don't care" with True and False. At present, variable assignment order is just their original order within C_{max}, because reordering seems not to improve computational times.

4 Adaptive core search

The above scheme can be modi¯ed in order to speed-up the entire procedure. Roughly speaking, the idea is that, when we have a hard subset of clauses, that we call a core, we can at ¯rst work on it, just ignoring other clauses. After solving such core, if that is unsatis¯able, the whole formula is unsatis¯able. Conversely, if the core admits a satisfying solution, we try to extend such solution to a bigger subset of clauses, until solving the whole formula. Selection of hardest clauses within a clause-set of cardinality m is always intended as the selection of the top $c¢m$ values for ', with $0 < c < 1$. The algorithm works as follows:

Adaptive core search

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

1. (Base) Select an initial collection of hardest clauses C_1. This is the ¯rst core. Remaining clauses form O_1.

k. (Iteration) Perform b branching iteration on C_k, ignoring O_k, using adaptive clause rule. We have one of the following:

k.1. C_k is unsatis¯able) F is unsatis¯able, then Stop.

k.2. No answer after b iteration) select a new collection of hardest clauses C_{k+1} within C_k, put $k := k + 1$, goto k.

k.3. C_k is satis¯ed by solution S_k) try S_k on O_k. One of the following:

k.3.a All clauses are satis¯ed) F is satis¯ed, then Stop.

k.3.b There is a set T_k of falsi¯ed clauses) add them to the core: put $C_{k+1} = C_k$ [T_k, $k := k + 1$, goto k.

k.3.c No clauses are falsi¯ed, but there is a set V_k of still not satis¯ed clauses) select a collection $C^0{}_k$ of hardest clauses in V_k, put $C_{k+1} = C_k$ [$C^0{}_k$, $k := k + 1$, goto k.

The preprocessing step has the aim to give initial values of visits and failures, in order to compute '. After that, we select the clauses that resulted hard during this branching phase, and try to solve them as if they were our entire instance. If they really are an unsatis¯able instance, we have done. If, after b branching iterations we cannot solve them, our instance is still too big, and it must be reduced more. Finally, if we ¯nd a satisfying solution for them, we try to extend it to the rest of the clauses. If some clauses are falsi¯ed, this means that they are di±cult (together with the clauses of the core), and therefore they should be added to the core. In this case, since the current solution falsi¯es some clauses now in the core, it results faster to rebuilt it completely. 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 ¯nite number of times. After that, the solution is not entirely rebuilt, but modi¯ed by performing backtrack. This choice makes the above algorithm a complete one.

Core Search has the important feature of solving, in average case, smaller subproblems at the nodes of the search tree, hence the operation performed, such like unit propagation consequent to any truth assignment, are performed only on the current C_k. Such idea of delaying (at least partially) the unit propagation subsequent to any variable ¯xing is recently recognized to be successful [29], and nowadays state of the art solvers (as Sato [28]) try in di®erent ways to incorporate it.

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 labeled n and m shows respectively number of variables and number of clauses. Column labeled literals shows the number of all literals appearing in the formula, hence the sum of the lengths of the clauses. Column labeled sol reports if satis¯able or unsatis¯able. Column labeled ACS reports times for solving the instance by Adaptive Core Search. Other table speci¯c columns are described in following subsection. Times are in CPU seconds. We set a time limit of 600 sec. When this is exceeded, we report > 600. When a running time is not available, we report n.a.

Computational tree size was not considered because the di®erent solvers compared here do not perform similar node processing, hence times to perform such node processing can greatly vary. It would not help to know that a procedure needs to explore only a small number of nodes if their exploration requires a very long time. Therefore, for the following comparisons, we consider meaningful only computational time.

Parameter p appearing in hardness evaluation function ' was set at 10. During our experiments, in fact, such choice seems to give better and more uniform results.

We choose the test problems available from the DIMACS [1], since they are widely-known, and the test instances [2], together with computational results, are easily available. Some problems are randomly generated instances, such like the series aim, jnh, while some other are encoding of real logic problems, such like the series ii, par, ssa. In addition, we solved some real-life problems arisen from a cryptography application, the des series.

Running times of Adaptive Core Search are compared with those of other complete algorithms. In such comparisons, either we could make the algorithms run on our machine, or we considered times reported in literature but, when possible, normalized as if they were run on our machine. In order to compare times taking into account such machine performance, we measure it by using the DIMACS benchmark dfmax [3], although it had to be slightly modi¯ed to be compiled with our compiler. The measure of our machine performance in CPU seconds is therefore:

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

5.1 The series ii32

The series ii32 is constituted by instances encoding inductive inference problems, contributed from M.G.C. Resende [21]. They essentially contain two kind of clauses: a set of binary clauses and a set of long clauses. Their size is quite big. On this problem we compare the algorithm of Adaptive Core Search 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 uses the adaptive branching rule based on '. Its times are in column labeled ABR. Shortest

[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/

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

Clause Branching Rule is a branching algorithm which does not use core search, and just uses shortest-clause-¯rst branching rule. Its times are in column labeled SCBR. Results on this set are in table 1. ACS distinctly 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.

Problem	n	m	literals	sol	ACS	ABR	SCBR
ii32a1	459	9212	33003	SAT	0.02	475.57	> 600
ii32b1	228	1374	6180	SAT	0.00	20.65	356.74
ii32b2	261	2558	12069	SAT	0.03	36.56	> 600
ii32b3	348	5734	29340	SAT	0.03	108.57	> 600
ii32b4	381	6918	35229	SAT	1.53	311.62	> 600
ii32c1	225	1280	6081	SAT	0.00	2.67	1.75
ii32c2	249	2182	11673	SAT	0.00	27.29	0.02
ii32c3	279	3272	17463	SAT	2.84	57.03	> 600
ii32c4	759	20862	114903	SAT	5.07	> 600	> 600
ii32d1	332	2730	9164	SAT	0.01	409.21	> 600
ii32d2	404	5153	17940	SAT	0.76	> 600	> 600
ii32d3	824	19478	70200	SAT	7.49	> 600	> 600
ii32e1	222	1186	5982	SAT	0.00	1.24	0.01
ii32e2	267	2746	12267	SAT	0.01	82.13	> 600
ii32e3	330	5020	23946	SAT	0.08	131.38	> 600
ii32e4	387	7106	35427	SAT	0.02	312.28	> 600
ii32e5	522	11636	49482	SAT	1.03	382.36	> 600

Table 1: Results of ACS on the ii32 series: inductive inference problems. From M.G.C. Resende

5.2 The series par16

The series par16 is constituted by instances arisen from the problem of learning the parity function, for a parity problem on 16 bits. Contributed from J. Crawford. They contain clauses of di®erent length: unit, binary and ternary. Their size is sometimes remarkably big. par16-x-c denotes an instance which represent a problem equivalent to the corresponding par16-x, except that the ¯rst instance have been expressed in a compressed form. For this set, we compare with the latest version (3.2) [4] of the state-of-the-art sat solver Sato [28]. Results are in table 2. They are extremely encouraging. We can observe a sort of complementarity in computational time results: ACS

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

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 identi¯cation of the hard part of the instances, but, due to an implementation and a data structure still not re¯ned as Sato's ones, has more di±culties on bigger instances. On the contrary, due to its very carefully implementation, which has been improved for several years, Sato 3.2 can handle more e±ciently bigger instances, but on smaller and harder instances, cannot compensate the advantages of adaptive branching and core search.

Problem	n	m	literals	sol	ACS 1.0	Sato 3.2
par16-1	1015	3310	8788	SAT	10.10	24.16
par16-1-c	317	1264	3670	SAT	11.36	2.62
par16-2	1015	3374	9044	SAT	52.36	49.22
par16-2-c	349	1392	4054	SAT	100.73	128.15
par16-3	1015	3344	8924	SAT	103.92	40.81
par16-3-c	334	1332	3874	SAT	8.19	78.91
par16-4	1015	3324	8844	SAT	70.82	1.51
par16-4-c	324	1292	3754	SAT	5.10	133.07
par16-5	1015	3358	8980	SAT	224.84	4.92
par16-5-c	341	1360	3958	SAT	72.29	196.33

Table 2: Results of ACS and Sato 3.2 on the par16 series: instances arisen from the problem of learning the parity function. From J. Crawford.

5.3 The series aim100

The series aim100 is constituted by 3-SAT instances arti¯cially generated by K. Iwama, E. Miyano and Y. Asahiro [1], and have the peculiarity that the satis¯able ones admit only one satisfying truth assignment. Such instances are not big in size, but can be very di±cult. Results on these sets are reported in table 3.

Some instances from this set were used in the test set of the Second DIMACS Implementation Challenge [20]. We also report the results of the four faster complete algorithms of that challenge, normalizing their times according to the results with dfmax declared in the original papers, in order to compare them in a machine-independent way.

C ¡ sat, presented by O. Dubois, P. Andre, Y. Boufkhad and J. Carlier [10], is a backtrack algorithm with a specialized branching rule and a local preprocessing at nodes of search tree. It is considered one of the fastest algorithms for SAT. Its times are in column labeled C-SAT. 2cl, presented by A. Van Gelder and Y. K. Tsuji [13], consists in a combination of branching and limited resolution. Its times are in column labeled 2cl. TabuS, presented by B. Jaumard, M. Stan and J. Desrosiers [18], is an exact algorithm which includes a tabu search heuristic and reduction tests other than those of the Davis-Putnam-Loveland scheme. Its times are in column labeled TabuS. BRR, presented by D. Pretolani [25], makes use of directed hypergraph transformation of the problem, to which it applies a B-reduction, and of a pruning procedure. Its times are in column labeled BRR.

A noticeable performance superiority of ACS can be observed, expecially on unsatis¯able problems.

Problem	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(limited resolution), DPL with Tabu Search, B-reduction, on the aim-100 series: 3-SAT arti¯cially 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.

5.4 The series jnh

The series jnh is constituted by random instances generated by J. Hooker. As stated in [14], the parameter were carefully chosen to result in hard problems [24], because otherwise random problem tend to be too easy. Each variable occurs in a given clause with probability p, and it occurs direct or negated with equal probability. The probability is chosen so that the expected number of literals per clause is 5. Empty clauses and unit clauses are rejected. Such problems are hardest [16] when the number of variable is 100 and the number of clauses is between 800 and 900. Results on this set are reported in table 4 (part a and b).

Problem	n	m	lit	sol	ACS	DPL	JW	GU	B&C	CS
jnh1	100	850	4392	SAT	0.03	10.5	18.6	53.1	20.8	107.9
jnh2	100	850	4192	UNSAT	0.05	1007.2	15.4	363.1	26.3	37.0
jnh3	100	850	4168	UNSAT	0.28	672.4	239.1	970.1	148.0	195.0
jnh4	100	850	4160	UNSAT	0.07	661.0	50.3	2746.9	108.9	36.0
jnh5	100	850	4164	UNSAT	0.06	670.8	42.8	120.2	88.3	39.1
jnh6	100	850	4155	UNSAT	0.32	1274.5	84.2	23738.2	149.9	217.0
jnh7	100	850	4160	SAT	0.03	5.9	7.2	160.3	51.4	69.2
jnh8	100	850	4147	UNSAT	0.05	165.6	62.8	624.4	58.7	95.8
jnh9	100	850	4156	UNSAT	0.09	345.2	78.9	1867.9	82.6	81.3
jnh10	100	850	4164	UNSAT	0.08	340.4	36.9	313.6	82.0	160.2
jnh11	100	850	4132	UNSAT	0.19	2280.6	135.1	4182.2	165.0	134.6
jnh12	100	850	4171	SAT	0.03	120.6	5.1	398.0	28.8	70.1
jnh13	100	850	4132	UNSAT	0.06	776.8	45.3	503.1	34.6	139.7
jnh14	100	850	4163	UNSAT	0.04	184.2	69.0	2610.4	76.7	39.9
jnh15	100	850	4126	UNSAT	0.08	1547.2	83.1	585.3	65.3	130.5
jnh16	100	850	4172	UNSAT	4.92	13238.7	542.4	20112.2	573.6	434.4
jnh17	100	850	4133	SAT	0.03	140.1	10.8	32.3	58.1	143.1
jnh18	100	850	4169	UNSAT	0.62	2261.0	158.2	2980.6	132.0	191.5
jnh19	100	850	4148	UNSAT	0.07	294.5	87.5	4184.4	153.8	132.3
jnh20	100	850	4154	UNSAT	0.07	648.6	124.5	203.7	126.3	187.3

Table 4a: Results of ACS, Davis-Putnam-Loveland, Jeroslow-Wang, Gallo-Urbani, Branch and Cut, Column Subtraction on the jnh series: randomly generated hard problems. From J.N. Hooker. In this table only, the last ¯ve columns show times on a di®erent machine, hence times cannot be directly compared.

For most of them we have also results obtained by several other complete algorithms coded in Fortran and run on a Sun Sparc Station 330 in Unix environment, as shown in [14]. In this case only, we cannot calculate the exact computational performance relationship between their and our machine (probably our is at least an order of 10 faster), so we simply report the original times for Davis-Putnam-Loveland [22] (column labeled DPL), Jeroslow-Wang [19] (column labeled JW), Gallo-Urbani [12] (column labeled GU), Branch and Cut [16] (column labeled B&C), Column Subtraction [15] (column labeled CS) methods.

Problem	n	m	lit	sol	ACS	DPL	JW	GU	B&C	CS
jnh201	100	800	4154	SAT	0.02	8.0	6.3	5.9	28.4	40.2
jnh202	100	800	3962	UNSAT	0.03	3515.2	47.4	710.5	42.7	34.6
jnh203	100	800	3906	UNSAT	0.18	939.8	66.6	294.6	186.3	241.6
jnh204	100	800	3914	SAT	0.41	1109.9	8.4	8905.8	78.1	220.0
jnh205	100	800	3911	SAT	0.05	309.1	12.7	1176.5	57.2	149.3
jnh206	100	800	3905	UNSAT	0.18	1556.6	126.6	3863.9	96.4	85.0
jnh207	100	800	3936	SAT	0.03	3.2	119.8	1037.0	65.1	48.0
jnh208	100	800	3908	UNSAT	0.17	388.3	51.6	958.0	63.6	33.8
jnh209	100	800	3902	SAT	0.11	4.2	50.8	1239.2	77.9	175.4
jnh210	100	800	3915	SAT	0.04	6.1	9.3	576.0	37.6	39.6
jnh211	100	800	3888	UNSAT	0.08	n.a.	n.a.	n.a.	n.a.	n.a.
jnh212	100	800	3932	SAT	0.26	n.a.	n.a.	n.a.	n.a.	n.a.
jnh213	100	800	3900	SAT	0.04	n.a.	n.a.	n.a.	n.a.	n.a.
jnh214	100	800	3896	UNSAT	0.12	n.a.	n.a.	n.a.	n.a.	n.a.
jnh215	100	800	3898	UNSAT	0.08	n.a.	n.a.	n.a.	n.a.	n.a.
jnh216	100	800	3888	UNSAT	0.19	n.a.	n.a.	n.a.	n.a.	n.a.
jnh217	100	800	3939	SAT	0.23	n.a.	n.a.	n.a.	n.a.	n.a.
jnh218	100	800	3905	SAT	0.01	n.a.	n.a.	n.a.	n.a.	n.a.
jnh219	100	800	3889	UNSAT	0.24	n.a.	n.a.	n.a.	n.a.	n.a.
jnh220	100	800	3923	SAT	0.06	n.a.	n.a.	n.a.	n.a.	n.a.
jnh301	100	900	4654	SAT	0.12	12528.6	65.8	271.3	116.0	77.5
jnh302	100	900	4441	UNSAT	0.03	161.6	13.0	380.4	17.0	84.3
jnh303	100	900	4380	UNSAT	0.16	388.6	111.4	307.1	98.2	40.0
jnh304	100	900	4417	UNSAT	0.15	132.0	27.3	409.2	43.4	43.0
jnh305	100	900	4406	UNSAT	0.06	652.7	68.0	138.9	101.7	196.2
jnh306	100	900	4425	UNSAT	1.25	4202.2	195.7	32270.0	221.9	205.1
jnh307	100	900	4365	UNSAT	0.04	6.7	32.1	19.7	25.6	180.3
jnh308	100	900	4410	UNSAT	0.20	1196.5	127.7	6188.3	159.7	164.6
jnh309	100	900	4415	UNSAT	0.03	131.0	14.7	298.2	48.1	42.7
jnh310	100	900	4369	UNSAT	0.03	262.8	25.9	406.7	9.8	43.9

Table 4b: Results of ACS, Davis-Putnam-Loveland, Jeroslow-Wang, Gallo-Urbani, Branch and Cut, Column Subtraction on the jnh series: randomly generated hard problems. From J.N. Hooker. In this table only, the last ¯ve columns show times on a di®erent machine, hence times cannot be directly compared.

5.5 The series ssa

The series ssa is constituted by instances generated by A. Van Gelder and Y. Tsuji. They are encoding of application problems of circuit fault analysis, used in checking for circuit "single-stuck-at" fault. These instances are large in size but not particularly hard. Results on this set are reported in table 5.

The series were used in the test set of the Second DIMACS Implementation Challenge [20]. We also report the results of the four faster complete algorithms of that challenge: C ¡ sat, 2cl, TabuS, and BRR, already described in 5.3. Times are normalized according to their result with dfmax, in order to compare them in a machine-independent way.

Problem	n	m	literals	sol	ACS	C-sat	2cl	TabuS	BRR
ssa7552-038	1501	3575	8248	SAT	0.19	0.49	0.86	0.01	0.25
ssa7552-158	1363	3034	6827	SAT	0.08	0.33	0.53	5.41	0.17
ssa7552-159	1363	3032	6822	SAT	0.15	0.36	0.53	0.75	0.20
ssa7552-160	1391	3126	7025	SAT	0.20	0.36	0.67	0.75	0.21

Table 5: Results of ACS on the ssa series: circuit fault analysis problems.

5.6 The series des

The series des is constituted by instances [5] arising from a practical application: veri¯cation and Cryptanalysis of Cryptographic Algorithms [27]. Such problems, which are nowadays showing their importance, can be encoded into instances which can be also very large. They are always satis¯able by construction, but we are interested in ¯nding the satisfying truth assignment. Results on this set are reported in table 9.

Problem	n	m	literals	sol	ACS 1.0	SATO 3.2
des-1-1	316	1687	5186	SAT	0.11	1.94
des-1-4	1010	6446	20016	SAT	0.98	0.11
des-2-1	600	3531	10746	SAT	0.66	0.09
des-2-4	2062	13387	41224	SAT	2.45	0.19

Table 6: Results of ACS on the des series: cryptography problems.

[5] Available upon request.

6 Conclusions

We present a clause based tree search paradigm for Satis¯ability testing, which makes use of a new adaptive branching rule, and the original techniques of core search, used to speed-up the procedure although maintaining the feature of complete method. We therefore obtain an enumeration technique altogether denominated Adaptive Core Search, which is able to sensibly reduce computational times.

By using the above technique, we observed a better performance improvement on instances which are not uniformly hard, in the sense they contain subsets of clauses having di®erent di±culty degrees. This is mainly due to the ability of our adaptive device in pinpointing hard sub-formulae during the branching tree exploration earlier than other methods. We stress that 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.

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. Comparison with several published results shows the e®ectiveness of the proposed procedure. Comparison of ACS with the state-of-the-art solver Sato is particularly encouraging. In fact, ACS, in its ¯rst release 1.0, is sometimes faster that Sato 3.2, which has evolved for several years. In particular, Sato is faster mainly when the instances are big and f lat, due to its very carefully implementation. We belive running times can further improve on big-sized instances by further polishing our implementation, and by using several techniques available in literature to perform a fast enumeration. Example of this could be to reduce clause revisits by saving and reusing global inferences revealed during search, as some other modern solvers do. This could be suitably introduced in our core search scheme, by evaluating our ¯tness function for the global inferences as well, and using this as a criterion to discard them. Future work will explore the introduction of similar tighter bounds in presented scheme, in order to reduce branching tree exploration.

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, 26:377{393, 1996.

[2] A. Billionnet and A. Sutter. An e±cient algorithm for the 3-Satis¯ability Problem. Operations Research Letters, 12:29{36, 1992.

[3] J.R. Bitner and E.M. Reingold. Backtrack programming techniques. Comm. of ACM, 18(11):651{656, Nov. 1975.

[4] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A complexity index for Satis¯ability Problems. SIAM Journal on Computing, 23:45{49, 1994.

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

[6] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. Proc. IJCAI-91, pages 331{337, 1991.

[7] J. Crawford and L. Auton. Experimantal results on the crossover point in Satis¯ability problems. In Proc. AAAI-93, pages 22{28, 1993.

[8] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. Comm. Assoc. for Comput. Mach., 5:394{397, 1962.

[9] M. Davis and H. Putnam. A computing procedure for quanti¯cation theory. Jour. Assoc. for Comput. Mach., 7:201{215, 1960.

[10] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:415{436, 1996.

[11] J. Franco and M. Paull. Probabilistic analysis of the Davis Putnam procedure for solving the Satis¯ability Problem. Discrete Applied Mathematics, 5:77{87, 1983.

[12] G. Gallo and G. Urbani. Algorithms for testing the Satis¯ability of Propositional Formulae. Journal of Logic Programming, 7:45{61, 1989.

[13] A. Van Gelder and Y.K. Tsuji. Satis¯ability testing with more reasoning and less guessing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:559{586, 1996.

[14] F. Harche, J.N. Hooker, and G.L. Thompson. A computational study of Satis¯ability Algorithms for Propositional Logic. ORSA Journal on Computing, 6:423{435, 1994.

[15] F. Harche and G.L. Thompson. The Column Subtraction algorithm, an exact method for solving weighted Set Covering, Packing and Partitioning Problems. Computers and Operations Research, 21:689{705, 1990.

[16] J.N. Hooker and C. Fedjki. Branch and Cut solution of Inference Problems in Propositional Logic. Annals of Mathematics and AI, 1:123{139, 1990.

[17] J.N. Hooker and V. Vinay. Branching Rules for Satis¯ability. Journal of Automated Reasoning, 15:359{383, 1995.

[18] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the Satis¯ability Problem. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:457{477, 1996.

[19] R.E. Jeroslow and J. Wang. Solving Propositional Satis¯ability Problems. Annals of Mathematics and AI, 1:167{187, 1990.

[20] D.S. Johnson and M.A. Trick, editors. Cliques, Coloring, and Satis¯ability, volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.

[21] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende A continuous approach to Inductive Inference. Mathematical Programming, 57:215{238, 1992.

[22] D.W. Loveland. Automated Theorem Proving: a Logical Basis. North Holland, 1978.

[23] C. Mannino and A. Sassano. Augmentation, Local Search and Learning. AI IA Notizie, XIII:34{36, Mar. 2000.

[24] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT Problems. In Proceedings of AAAI'92, pages 459{465, Jul. 1992.

[25] D. Pretolani. E±ciency and stability of hypergraph SAT algorithms. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:479{498, 1996.

[26] M.G.C. Resende and T.A. Feo. A GRASP for Satis¯ability. DIMACS Series in Discrete Mathematics, 26:499{520, 1996.

[27] B. Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, 1994.

[28] H. Zhang. SATO: An E±cient Propositional Prover. in Proc. of International Conference on Automated Deduction (CADE-97), Lecture notes in Arti¯cial Intelligence 1104, Springer-Verlag, 308{312, 1997.

[29] H. Zhang and M.E. Stickel. Implementing the Davis-Putnam Method. Technical Report, The University of Iowa, 1994.