

AVAL: an enumerative method for SAT

Gilles Audemard, Belaid Benhamou and Pierre Siegel

Laboratoire d'Informatique de Marseille
Centre de Mathématiques et d'Informatique
39, Rue Joliot Curie - 13453 Marseille cedex 13 - France
Tel : 04 91 11 36 25 - Fax : 04 91 11 36 02
email : {audemard, benhamou, siegel}@lim.univ-mrs.fr

Abstract. We study an algorithm for the SAT problem which is based on the Davis & Putnam procedure. The main idea is to increase the application of the unit clause rule during the search. When there is no unit clause in the set of clauses, our method tries to produce one occurring in the current subset of binary clauses. A literal deduction algorithm is implemented and applied at each branching node of the search tree. This method AVAL is a combination of the Davis & Putnam principle and of the mono-literal¹ deduction procedure. Its efficiency comes from the average complexity of the literal deduction procedure which is linear in the number of variables. The method is called “AVAL” (avalanch) because of its behaviour on hard random SAT problems. When solving these instances, an avalanche of mono-literals is deduced after the first success of literal production and from that point, the search effort is reduced to unit propagations, thus completing the remaining part of enumeration in polynomial time.

keywords : Satisfiability, deduction, enumeration...

1 Introduction

Some progresses have been realized in solving SAT problem. In particular, application of local search methods [13] to hard satisfiable SAT instances gives satisfying results. However, they can not deal with unsatisfiable instances.

To solve such instances, one usually uses systematic methods based on the Davis & Putnam procedure [3] (DP). DP efficiency comes from the property of unit clause propagation. This method, and its well known improvements (SATO [14], C-SAT [4], POSIT [6] [7], SATZ [10]...) find their limits when applied to SAT random instances located in the transition phase.

This paper introduces the method AVAL based on DP procedure to improve search efficiency and uses it as a base method to study the behaviour of enumeration methods on hard SAT instances. The improvement consists in maximizing the use of unit clause propagation during the search. To do that, a literal production algorithm (LP) is implemented. This procedure is called when there is no

¹ a mono-literal means a unit clause

explicit mono-literal in the set of clauses, to produce new ones from the subset of binary clauses and to use them as new propagations. Such propagations are not done by the classical method of Davis & Putnam. The AVAL method is a combination of DP and of the literal production procedure. Its efficiency comes from the average complexity of the literal production process which is linear in the number of variables.

The literal production procedure deduces literals occurring in the subset of binary clauses. Such literals are more likely to be logical consequence of the current set of clauses than any others. The LP procedure is called at each branching node when the classic unit clause rule of DP does not apply. This minimizes the number of branching nodes (choice nodes) and provides a robust algorithm which is less sensitive to heuristics. This algorithm can be used in practice to analyze the limits of systematic methods in solving hard SAT instances.

The paper is organized as following: In section 2, we study the literal production algorithm. Section 3 describes the avalanche method (AVAL), the heuristics and the pre-processing used. Section 4 shows experimental results on a large variety of problems: hard random instances and problems of both challenges DIMACS and Beijing. A comparison of AVAL with other algorithms among the most powerful ones like POSIT [6] and SATZ [11] is done. section 5 concludes.

2 Literal Production

Let's give some definitions we shall use. Let $V = \{x_1 \dots x_n\}$ be a set of boolean variables, a literal l is a variable x_i or its negation \bar{x}_i . A monotone literal is a literal occurring exclusively either in its positive or negative form. A clause is a disjunction of literals $c_i = l_1 \vee l_2 \dots \vee l_{n_i}$. A unit clause (a mono-literal) is a clause of one literal. The conjunctive normal form of a propositional formula C is a conjunction of clauses $C = c_1 \wedge c_2 \dots \wedge c_m$. We can consider C as a set of clauses $C = \{c_1 \dots, c_m\}$. The SAT decision problem is defined as follows: is there an assignment of the variables so that the formula C is satisfied, i.e all the clauses of the set C are satisfied?

If l is logically implied by the set of clauses C , then we write $C \models l$. When a system of clauses C is unsatisfiable, we note it by $C \models \square$, where \square denotes the empty clause. The k -SAT problem is the problem SAT where all clauses have exactly k literals. 3-SAT is known to be the simplest form of k -SAT which remains NP-Complete.

The Davis & Putnam procedure is a real improvement of the Quine method [12] thanks to both unit clause and monotone literal rules (cf proposition 1). Methods like C-SAT [4] and SATZ [10] do more unit propagations. In the same spirit, our work consist in revealing, at lower cost, unit clauses that DP does not consider in exploiting them in order to reduce the size of the search tree. The main property used in DP is the following.

Proposition 1. *Let S be a SAT problem and x be a mono-literal or a monotone literal then S is satisfiable if and only if $S \wedge \{x\}$ is satisfiable.*

If there is no mono-literal in the current set of clauses at a given node of the search tree, our algorithm tries to produce one. For efficiency reasons we restrict the production process to literals occurring in binary clauses. These literals are more likely to be produced.

Let I be the current instantiation and C_I be the set of the clauses C simplified by the instantiation I . If l is a literal occurring in a binary clause of C_I , then producing l from C_I ($C_I \models l$) is equivalent to proving the unsatisfiability of $C_I \wedge \{\neg l\}$. Two cases are possible:

1. If $C_I \wedge \{\neg l\} \models \square$ then l is produced by C_I and considered as a mono-literal. Thus, $I = I \cup \{l\}$
2. If $C_I \wedge \{\neg l\} \not\models \square$ then l is not produced by C_I , and this failure deduction highlights several literals which can not be deduced by C_I . It will be useless to consider them as candidates for production.

The efficiency of the literal production is due to this elimination of useless variables. Formally:

Proposition 2. *Let B_I be the set of binary clauses of C_I and V_{B_I} be its set of literals. Let $l \in V_{B_I}$, if $C_I \cup \{\neg l\} \models a_{i \in \{1..n\}}$ such that $C_I \cup \{\neg l\} \not\models \square$ then $\forall a_{i \in \{1..n\}} \in V_{B_I}, C_I \cup \{a_i\} \not\models \square$.*

Proof. Let $l \in V_{B_I}$, suppose that $C_I \cup \{\neg l\} \models a_{i \in \{1..n\}}$ and $C_I \cup \{\neg l\} \not\models \square$. If there exists $a_{j \in \{1..n\}}$ such that $C_I \cup \{a_j\} \models \square$, then, $C_I \cup \{\neg l\} \cup \{a_j\} \models \square$. But $C_I \cup \{\neg l\} \cup \{\neg a_j\} \models \square$, hence $C_I \cup \{\neg l\} \models \square$. This makes a contradiction with the hypothesis. \square

The literals $\{\neg a_1, \dots, \neg a_n\}$ can not be logical consequences of C_I . Thus, considering them for production is irrelevant. The previous proposition gives the literal production algorithm (LP) described in figure 1. In the following, we show the mechanism algorithm.

Example 1. Let the set of clauses $C = \{x_1 \vee \neg x_2 \vee \neg x_3, x_1 \vee x_2, \neg x_2 \vee x_3\}$. The literals $V_{B_I} = \{x_1, x_2, \neg x_2, x_3\}$ appears in the binary clauses.

We try to produce x_3 . $C \wedge \{\neg x_3\} \models \neg x_2, x_1$. So, $C \wedge \{\neg x_2\} \not\models \square$ then x_2 can't be deduced by C and $Candidate[x_2] = False$. Now, literals candidate to production are $\{x_1, \neg x_2\}$.

We try to produce x_1 . $C \wedge \{\neg x_1\} \models x_2, x_3, \square$. So, $C \models x_1$. x_1 is produced by C .

Remark 1. In the algorithm of figure 1, $Candidate[x] = True$ expresses the fact that x is a candidate to production.

The LP algorithm deduces literals among those appearing in binary clauses. Its termination, correctness, completeness and complexity are studied in the following propositions.

```

Procedure LP( $C$  : set of clauses ;  $I$  : instantiation)
Return : a literal  $l$  if  $C \models l$ 
        0 if  $\nexists l \in V_{B_I}$  such that  $C \models l$ 

Begin
For All  $l \in V_{B_I}$  do Candidate[ $l$ ] = True
For All  $l \in V_{B_I}$  such that Candidate[ $l$ ] = True Do Begin
  Candidate[ $l$ ] = False
   $I' = I \cup \{-l\}$ 
  While  $C_{I'} \neq \emptyset$  and  $\exists x \in V_{C_{I'}}$ ,  $x$  is a unit clause and  $\square \notin C_{I'}$  Do
    Begin
       $I' = I' \cup \{x\}$ 
      Candidate[ $\neg x$ ] = False
    End
  If  $\square \in C_{I'}$  Then Return  $l$ 
End
Return 0
End

```

Algorithm 1. Literal Production Algorithm (LP)

Proposition 3. *Let C be the current set of clauses, and l a literal occurring in a binary clause. If LP produces l ($LP(C) \models l$) then $C \equiv C \cup \{l\}$.*

Proof. Let l be a literal produced by LP. Then, $(C \cup \{-l\}) \models \square$.

But $C \equiv (C \wedge l) \vee (C \wedge \{-l\})$. Thus, $C \equiv C \cup \{l\}$ and LP correctness is proved. \square

Proposition 4. *If C_I is the current set of clauses, and B_I the subset of binary clauses, then the algorithm terminates and its complexity in worst case is in $O(|V_{B_I}| \times |V_{C_I}|)$.*

Proof. When, LP tries to deduce l ($C_I \models l$?), it propagates in the worst case V_{C_I} unit clauses. If l is not deductible ($C_I \not\models l$), the LP procedure tries to deduce another literal among those of V_{B_I} . The worst case is when all the propagated literals occurring in the failure of producing l ($C_I \models l$?) are not in V_{B_I} . In this case LP tries V_{B_I} literals. Thus its complexity in the worst case is in order of $O(|V_{B_I}| \times |V_{C_I}|)$. \square

The C-SAT method [4] does some local deduction on selected variables, but does not take advantage of the literals that are not potential candidates for production (see proposition 2) despite Boufkhad remarked these irrelevant literals in his thesis [1]. In [7] (POSIT) Freeman uses a kind of literal production on selected variables but does not suppress the irrelevant literals. Elimination of these useless literals allows to obtain a literal production algorithm whose average time complexity is linear in practice (see algorithm 1).

3 The Avalanche Method (AVAL)

The combination of the literal production algorithm (LP) and the DP procedure yields the enumeration method AVAL.

The difference between DP and AVAL is that AVAL calls the procedure LP to produce a mono-literal when no explicit unit clause exists in the current set of clauses. This prevents for visiting some nodes that DP visits. When LP succeeds, the returned literal is considered as a mono-literal. In case of failure we choose via heuristics the next literal, thus creating a new choice point in the search tree. Exploiting the literals produced by LP, leads to minimizing the number of choice points in the search tree (for a given heuristic). The algorithm 2 sketches the AVAL method. The first call to AVAL is made with the parameter values : $I = \emptyset$ and $C = C_\emptyset$.

```
Procedure AVAL( $C$  : Set of Clause ;  $I$  : Instantiation)
Return : True if  $C$  is satisfiable by  $I$ 
        False otherwise

Begin
If  $C = \emptyset$  Then Return True
If  $C$  contains an empty clause Then Return False
If  $C$  contains an unit clause  $l$  Then Return AVAL( $C_{\{l\}}$ ,  $I \cup \{l\}$ )
q=LP()
If  $q \neq 0$  Then Return AVAL( $C_{\{q\}}$ ,  $I \cup \{q\}$ )
Choose by heuristic a literal  $p \in C$ 
If AVAL( $C_{\{p\}}$ ,  $I \cup \{p\}$ )
    Then Return True
    Else Return AVAL( $C_{\{\neg p\}}$ ,  $I \cup \{\neg p\}$ )
End
```

Algorithm 2. AVAL Method

3.1 Heuristics

Heuristics are used when there is no mono-literal in the current set of clauses and when LP does not produce one. We use the MOM heuristic (Maximum Occurrence in minimum size clauses) (Freeman [6]) which chooses the variable with the greatest number of occurrences in the minimum size clauses and the UP (Unit Propagation) heuristic which takes advantage of unit propagation. These heuristics allow to produce new binary clauses which favor the production of mono-literals when calling LP. Let us summarize them.

Mom Heuristic: If l is a literal then $w(l) = \sum_{-l \in C_i} 5^{-|C_i|}$ is its weight. MOM heuristic chooses the variable which maximizes the function used by Freeman in [6]: $H(x) = 1024.w(x).w(\neg x) + w(x) + w(\neg x)$.

UP heuristic: UP heuristic (see [6] and [10]) exploits unit propagation more than MOM heuristic does and gives more binary clauses which help LP to produce literals as soon as possible during the search. Let C_I be the current set of clauses, x a variable of C_I , $C'_I = C_I \wedge x$ and $C''_I = C_I \wedge \neg x$. After unit propagation on both C'_I and C''_I , UP chooses the variable which shortens a maximum number of clauses in C'_I and C''_I . We use this heuristic to select the variables for the first choice points of the search tree where the procedure LP usually fails to produce literals.

3.2 Pre-processing

Before starting the search, we do some pre-processings which consists in adding resolvents to the set of clauses. This usually reduces the search space. We use the same technique as the one of Chu Min Lee in [10]. Only resolvents of size less than 3 are considered and can to be used to produce other resolvents. The resolvents technique consists in the following rules: Two binary clauses can create only unary one. A binary and a ternary clauses can create only a binary one. Two ternary clauses produce a resolvent of size less or equal to 3. The process is maintained until saturation. This pre-processing allows a gain of nearly 10% for the hard random problems.

4 Experiments

We first compare our method AVAL with the classic Davis & Putnam Method. We study the behaviour of AVAL during search and compare it with two known algorithms (SATZ [10] and POSIT [6]) on different problems: instances of both challenges DIMACS and Beijing, and random 3-SAT instances. Random problems are generated as follows: Let c be the number of clauses and v be the number of variables, we generate randomly c clauses among the $2^3 \binom{v}{3}$ possible ones. Many research works showed that there is transition phase for the random k -SAT problems : there is a critical value of the ratio $\frac{c}{v}$ before which problems are few constrained (have many models) and after which problems are very constrained (have no models). The hard problems are in the neighborhood of this critical value. The existence of the threshold for 3-SAT was proven by Friedgut [8] but the critical value is still not known. We only know bounds ($3.03 \leq \frac{c}{v} \leq 4.64$), (Dubois and al [5]). For 2-SAT, the value is equal to one (Chvatal [2], Goerdt [9]).

The results are measured on a PC Pentium 200 with 64 Mo of RAM. The code of the program is written in C and includes 1300 lines. All CPU times are given in seconds.

4.1 Comparison between DP and AVAL

We compared the AVAL and DP methods augmented by the MOM and UP heuristics on hard random SAT instances. The samples of each test are 50 randomly generated instances with a ratio ($\frac{c}{v} = 4.25$). Tables 1 and 2 show the results obtained.

We can see that AVAL surpasses DP in both the number of search nodes and the CPU time. This confirms the efficiency of the LP procedure in producing literals and the advantage in combining it with DP. The gain increases as the number of variable grows giving a promising way to solve large scale problems. We can also see the benefit of using UP heuristic to define the variable assignment order. For this reason we use it in AVAL to compare the method with both SATZ and POSIT methods which out perform C-SAT [4].

Using the LP procedure in an enumerative method leads to minimizing the number of branchings. Indeed, LP efficiently produces literals, thus avoiding some choice points in the search tree. This optimizes in some way, the number of branching for the enumerative method with respect to the heuristic used for the variable ordering.

Table 1. Comparison between DP and AVAL (Nodes)

Number of Variables	140	160	180	200	220	240
DP + MOM	756	1165	3193	6736	18997	51733
AVAL + MOM	69	98	245	474	1212	2672
DP + UP + MOM	654	1047	2709	5271	15373	33000
AVAL + UP + MOM	58	82	189	335	923	2072

Table 2. Comparison between DP and AVAL (Time)

Number of variables	140	160	180	200	220	240
DP + MOM	0.219	0.355	1.01	2.269	7.055	19.8
AVAL + MOM	0.176	0.273	0.717	1.554	4.508	12.1
DP + UP + MOM	0.261	0.409	1.039	2.131	6.15	14.8
AVAL + UP + MOM	0.228	0.339	0.79	1.535	4.50	10.5

4.2 Success Rate of the Literal Production

Results of table 3 confirm that about 90% of calls to LP succeed to produce a literal. This is a promising result and explains the gain in number of nodes

Table 3. Success rate of literal production

<i>Nb of variables</i>	100	140	180	220	260
<i>Nb of tests</i>	170	860	3795	14136	62672
<i>Nb of success</i>	152	784	3494	13105	58372
<i>%</i>	89	91	92	92	93

when comparing AVAL to DP. Experiments on random instances show that after the depth $\frac{v}{21}$ in the search tree, AVAL search efforts only consist in unit propagations. Indeed, all the calls to LP succeed to produce literals. Literal production failures (about 10%) correspond to calls to LP in the top part of the search tree. In practice, an avalanche of mono-literals is observed after the first literal production. Such phenomenon occurs after nearly the depth $\frac{v}{21}$ and the search process is achieved in linear time complexity. This phenomenon is observed for the classical method DP too, but later, after a depth of $\frac{v}{13}$ in the search tree. This explains the difference between the efficiency of AVAL and DP. The more early the avalanche, the more efficient the algorithm. Thus, one can think that a minimal bound of the number of nodes that an enumerative method (w.r.t a given heuristic) has to explore is reached with the method AVAL.

4.3 Efficiency of our Method

Theoretical complexity in the worst case of the LP algorithm is in $O(|V_{B_I}| |V_{C_I}|)$. But in practice the average complexity is linear in the number of variables. This is confirmed by the experimental results of table 4. The ratio $\frac{b}{a}$ gives the average number of unit propagations for one call to the literal production procedure LP. Its variation is linear with respect to the number of variables. This allows to perform LP at each node of the search tree which explains the efficiency of AVAL.

Table 4. Complexity of the LP algorithm

<i>Nb of variables</i>	100	140	180	220	260
<i>a=nb call of LP</i>	170	860	3795	14136	62672
<i>b=nb Unit Prop</i>	9919	73303	420363	1930100	10216322
<i>ratio $\frac{b}{a}$</i>	58	85	110	136	163

4.4 Threshold of unit clauses production

The number of binary clauses in the current set of clauses has a great impact on literal production. The more the number of binary clauses, the more the

chance to succeed in producing a literal. As AVAL performances depend on the efficiency of literal production, it is important to know in practice how many binary clauses are necessary to produce a literal. In theory, the threshold of the satisfiability problem for 2-SAT random instances is reached when the ratio number of clauses to the number of variable is equal to one. This means that literal production shall succeed when the number of binary clauses is equal to the number of variables. But in practice literal production is guaranteed with a fewer number of binary clauses. Table 5 reports experiments on random 3-SAT instances which show that for a ratio $\frac{c}{v} \geq 0.7$ the procedure LP always succeeds in producing a literal. This means that $0.7 \times v$ binary clauses always produce a literal. This number of binary clauses is maintained at each node of the search tree after a depth of $\frac{v}{21}$ and make an avalanch of unit clauses. It will be interesting to study the existence of a theoretical threshold for literal production.

Table 5. Ratio nb binary / nb prop

<i>Nb of Variables</i>	100	140	180	220	260	300
$\frac{c}{v}$	0.702	0.708	0.747	0.748	0.744	0.757

4.5 Comparison with SATZ and POSIT

We compared both SATZ and POSIT methods to our method AVAL on hard random SAT instances, on the basis of the CPU times and the number of nodes. The number of variables stands from 100 to 400 (by a step of 50) and the samples of each test are 100 random instances when the number of variables is greater than 300, 200 otherwise. The instances are generated in the transition phase ($\frac{c}{v} = 4.25$). Table 6 shows the results. We can see that AVAL solves problems with 400 variables in the hard region in less than 2 hours in average. AVAL is better than both SATZ and POSIT in the number of nodes. Because of sophisticated heuristics (application of UP to selected variables) SATZ gives the best CPU times, AVAL and POSIT CPU times are comparable.

Table 6. Random Problems.

Nb of variables	100	150	200	250	300	350	400
AVAL Time (sec)	0.069	0.268	1.681	12,1	100	610	5278
Nodes	14	72	382	2182	13 946	70 405	502 803
SATZ Time (sec)	0.068	0.205	0.856	4,399	30.6	189	1096
Nodes	18	111	590	3089	18 371	100 014	521 349
POSIT Time (sec)	0.016	0.148	1.074	8.605	65.7	407	3698
Nodes	39	264	1502	10094	65 505	334 847	2 898 510

4.6 Challenges Beijing and DIMACS

We also compared the three methods on problems of the DIMACS and Beijing challenges. The maximum time that an algorithm can spend in solving an instance is limited to two hours (7200 seconds). Problems of the challenge Beijing are listed individually and those of DIMACS are gathered into classes. *Time* denotes the total time spent in solving all the problems of a class. When a problem is not solved in two hours, its CPU time is considered as 2 hours. The symbol $\#M$ indicates the number of problems in a class and $\#S$ the number of solved problems. Tables 7 and 8 show the results.

Problems of the challenge Beijing, listed in table 7, are mostly planning and scheduling problems. We resolve one more instance than SATZ and four more instances than POSIT. Except the problem 2_bit_add_12, AVAL CPU times are comparable to those of both SATZ and POSIT.

For the DIMACS challenge, AVAL solved less instances than SATZ for the classes: dubois, ii16 and ssa, and less instances than POSIT for the class ii16. But only AVAL is able to solve the whole problems in class ii32. POSIT solved half of the class aim200. These results show the advantage of combining literal deduction algorithm LP with DP.

Table 7. Challenge Beijing.

<i>Problem</i>	AVAL		SATZ		POSIT	
	<i>Time</i>	<i>Nodes</i>	<i>Time</i>	<i>Nodes</i>	<i>Time</i>	<i>Nodes</i>
2bitadd_10	3706	2 116 944	¿7200	-	¿7200	-
2bitadd_11	6.4	4 838	113	120 982	¿7200	-
2bitadd_12	35.2	28 843	0.265	99	0.04	35
2bitcomp_5	0.02	11	0.01	6	0.01	34
2bitmax_6	0.06	14	0.05	7	0.05	12
3bitadd_31	¿7200	-	¿7200	-	¿7200	-
3bitadd_32	¿7200	-	3101	297 652	¿7200	-
3blocks	2.2	16	1.6	7	2.38	669
4blocks	1184	18 823	930	228 040	¿7200	-
4blocksb	11.5	97	8	8	70	8424
e0ddr2-10-by-5-1	2657	705	86	35	¿7200	-
e0ddr2-10-by-5-4	617	661	86	32	2726	34759
enddr2-10-by-5-1	38	10	¿7200	-	¿7200	-
enddr2-10-by-5-8	66	15	81	30	¿7200	-
ewddr2-10-by-5-1	41	15	124	40	143	250
ewddr2-10-by-5-8	861	238	92	39	¿7200	-

Table 8. Challenge DIMACS

		AVAL		SATZ		POSIT	
Pb Class	#M	#S	Time	#S	Time	#S	Time
aim-50	24	24	14	24	14	24	0.3
aim-100	24	24	3.55	24	3.4	24	320
aim-200	24	24	4.2	24	3.85	12	86400
dubois	13	8	43274	12	38665	8	45500
hole	5	5	180	5	213	5	444
ii8	14	14	15.4	14	5	14	0.77
ii16	10	8	14967	10	104	9	7268
ii32	17	17	2060	16	7638	15	14410
jnh	50	50	12.84	50	11	50	0.25
par8	10	10	0.7	10	0.66	10	0.05
par1	10	10	288	10	403	10	33
ssa	8	7	10500	8	826	7	7231

5 Conclusion

Systematic search methods based on the Davis and Putnam procedure use unit propagation. But, only the explicit mono-literals occurring in the set of clauses are considered. AVAL does more propagation than these methods. Indeed, when there is no explicit mono-literal in the current set of clauses, AVAL calls the procedure LP to produce one before branching. Produced literals are propagated as mono-literals, thus minimizing the number of branching nodes. The LP algorithm produces literals appearing in binary clauses with a linear time complexity ($O(v)$ in practice). This allows to use LP at each branching node of the search tree, thus increasing the efficiency of the AVAL method.

We studied the behaviour of AVAL on hard random SAT instances generated in the neighbourhood of the threshold area and we observed two different parts in the search tree when solving such instances by AVAL. The hard part is the one formed by the first levels of the search tree. AVAL shows that enumerative methods have to spend a lot of time in this part of the search tree which contains all the branching nodes. Future work will consist to aim at finding variable ordering heuristics and techniques in order to reduce the search space of this part.

References

1. Y. Boufkhad. *Aspects probabilistes et algorithmiques du problme de satisfaisabilit*. PhD thesis, Univertsit de Jussieu, 1996.
2. V. Chvátal and B. Reed. Mick Gets Some (the odds are on this side). In *33rd IEEE Symposium on Foundation of Computers Science*, 1992.
3. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 1960.

4. O. Dubois, P. Andr, Y. Boufkhad, and J. Carlier. Sat versus unsat. *AMS, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26, 1996.
5. O. Dubois and Y. Boufkhad. A General Upper Bound for the Satisfiability Threshold of random r -sat formulae. *Journal of Algorithms*, 1996.
6. J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Univ. of Pennsylvania, Philadelphia, 1995.
7. J.W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81(2):183–198, 1996.
8. E. Friedgut. Necessary and sufficient conditions for sharp thresholds of graphs properties and the k -sat problem. Technical report, Institute of Mathematics, The Hebrew University of Jerusalem, 1997.
9. A. Goerdt. A threshold for unsatisfiability. In *Mathematical Foundations of Computer Science*, volume 629, pages 264–274. Springer, 1992.
10. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problem. In *proceedings of IJCAI 97*, 1997.
11. Chu Min Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *proceedings of CP97*, pages 341–355, 1997.
12. W.V. Quine. Methods of logics. *Henry Holt, New York*, 1950.
13. B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence AAAI'94*, 1994.
14. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated deduction*, 1997.