CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Elsevier - Publisher Connector

A SIMPLE TEST IMPROVES CHECKING SATISFIABILITY

ELIEZER L. LOZINSKII

▷ In many practical cases satisfiability of a set of clauses can be decided before an interpretation is found that satisfies *all* clauses of the set. We present a test for such an early discovery of satisfiability, *EDS*, and develop an algorithm, *IDP*, incorporating EDS and a branching heuristics related to this test. IDP was implemented and tested on a wide variety of instances and showed a high performance and stability with respect to changing the proportion of non-Horn clauses. ◁

1. INTRODUCTION

Deciding whether a given logical formula F is satisfiable constitutes the *satisfiability problem*. Among the hard computational problems, the satisfiability problem is very important both theoretically and practically. It plays a central role in the complexity theory as the seed of the class of NP-complete problems [5], and deciding satisfiability presents an inevitable and most frequently employed process in automated reasoning, theorem proving, logic programming, deductive databases, etc. Since NP-complete problems, many of which find significant practical applications, are reducible to the satisfiability problem in polynomial time, efficient algorithms for deciding satisfiability are of high practical importance.

A basic form of the satisfiability problem, *SAT*, is to decide satisfiability of a propositional formula presented in the conjunctive normal form (*clausal form*). Let X be a set of *propositional variables* $\{x_1, \dots, x_n\}$, S denote a set $\{C_1, \dots, C_m\}$ of *clauses*, each clause C_i be a disjunction of *literals* (considered also as a set of literals), such that each literal is a variable or its negation. An *interpretation* I is a function $I: X \to \{true, false\}$. An interpretation I *satisfies* a clause C iff there is a literal $L \in C$ such that $I(L) = true$. S is *satisfiable* iff there exists an interpretation J satisfying all the clauses of S. Then, J is a *model* of S.

Address correspondence to Eliezer L. Lozinskii, Institute of Mathematics and Computer Science, The Hebrew University, Jerusalem 91904, Israel. Email: LOZINSKI@HUMUS.HUJI.AC.IL.
Received August 1990; revised March 1992; accepted April 1992.

©Elsevier Science Publishing Co., Inc., 1993
655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$5.00

2. DAVIS-PUTNAM ALGORITHM

A clause containing only one literal is a *unit clause*. A literal L is *pure* in a set of clauses S iff L appears in a clause of S, but $\neg L$ appears in no clause of S.

An early and still quite efficient method for solving SAT is the well-known algorithm proposed by Davis and Putnam [7], and improved in [8] (described also in [3, 15]). Due to [8], it can be expressed in the following recursive form.

Algorithm DP (Given a set of propositional clauses S, decides its satisfiability)

(1) If S is empty, then it is satisfiable;
(2) else if S contains an empty clause, then S is unsatisfiable;
(3) else (**Unit Clause Rule:**) if S contains a unit clause $C = \{L'\}$, then choose L' and do 3.1–3.3;
(3.1) delete from S all clauses containing L', obtaining a set of remaining clauses S_1';
(3.2) delete from the clauses of S_1' all occurrences of $\neg L'$, obtaining a set S_2';
(3.3) apply DP to S_2'; S is satisfiable iff S_2' is so;
(4) else (**Pure Literal Rule:**) if S contains a pure literal L'', then choose L'' and do 4.1–4.2;
(4.1) delete from S all clauses containing L'', obtaining a set S'';
(4.2) apply DP to S''; S is satisfiable iff S'' is so;
(5) else (**Splitting Rule:**) do 5.1–5.3;
(5.1) choose a literal L occurring in S;
(5.2) let Δ_L denote a set of all clauses of S containing L, $\Delta_{\neg L}$ stand for a set of all clauses of S containing $\neg L$, Δ be the rest of clauses of S containing neither L nor $\neg L$. Define

$$S_L = \Delta \cup \{(C - \{\neg L\}) | C \in \Delta_{\neg L}\},$$

$$S_{\neg L} = \Delta \cup \{(C - \{L\}) | C \in \Delta_L\};$$

(5.3) choose S_L or $S_{\neg L}$ and apply DP to it; S is satisfiable iff S_L or $S_{\neg L}$ is satisfiable. □

Algorithm DP has an exponential worst-case time complexity (in particular due to the splitting of S at Steps 5.2 and 5.3); however, in many practical cases the average time complexity of DP is polynomial. Cook stated in [5] that he had "not yet been able to find a series of examples showing that the procedure must require more than polynomial time." Goldberg, Purdom, and Brown [12, 13] provided an $O(n^2)$ upper bound on the expected time complexity of DP for a variety of probability distributions of clauses and literals. A polynomial average time complexity of DP has been shown also in experiments carried out by Gallo and Urbani [11]. On the other hand, it has been proved in [2] that a simplified version of DP requires an average time that is exponential in the number of variables n if the number of clauses m is at least a linear function of n and the probability that a literal appears in a clause is proportional to $1/n$. Recent analysis of best known algorithms for SAT [1, 2, 14, 18] shows that for each of them there is a "hard" distribution of the problem parameters for which the algorithm does not guarantee a polynomial average time. So, new developments and improvements of algorithms for SAT are intended to reduce the regions of unfavorable parameters.

3. LATER DEVELOPMENTS

A high practical efficiency of *DP* has inspired numerous efforts to further improve its performance.

Given a set of clauses S, an execution of *DP* induces a binary proof tree $T(S)$ with its root representing S. In the course of performing *DP*, the set S changes as its cardinality is being monotonously reduced (cf. *DP*, Steps 3.1, 3.2, 4.1, and 5.2). Each node ν of $T(S)$ represents a pair (S^{ν}, r^{ν}), where S^{ν} is the current state of S and r^{ν} is the rule applied to S^{ν}. If r^{ν} is the *unit clause* or *pure literal* rule, then there is only one arc going out of ν, labeled by the literal chosen by r^{ν} (that is, by L' or L'', respectively—see Steps 3 and 4). If the *splitting rule* is applied at ν (Step 5), then ν is a *branching node*, since there are two outgoing arcs labeled by L and $\neg L$. Each leaf of $T(S)$ represents either an empty set (Step 1) and is a *satisfiable leaf* or a set containing an empty clause (Step 2), and hence, is an *unsatisfiable leaf*. *DP* starts at the root of $T(S)$ and then examines its branches until either a satisfiable leaf is reached or *all* leaves of $T(S)$ are found unsatisfiable.

The configuration of $T(S)$, and hence the run-time of *DP*, are determined by the choices made at Steps 3, 4, 5.1, and 5.3, primarily by the branching due to the splitting rule (Step 5). However, the original algorithm by Davis and Putnam [7] provided no recommendation regarding the branching strategy, namely, what literal L should be chosen at Step 5.1, and to which subset, S_L or $S_{\neg L}$, should *DP* be applied first (at Step 5.2). So, a number of heuristics have been proposed for improving the efficiency of branching.

Let $S_L, S_{\neg L}$ be subsets of clauses formed at a branching node. To prevent the redundant work of developing the same interpretations along both branches, Purdom [17] suggested *complement searching*, that is, to consider for one of the two subsets only those interpretations that do not satisfy the other subset. So, if, for instance, the left son of the node is assigned S_L, then, its right son gets $S_{\neg L} \wedge \neg S_L$. It takes additional time to transform $\neg S_L$ into clausal form. As pointed out in [17], this branching strategy is efficient practically for sets with *almost pure* literals.

Monien and Speckenmeyer [16] introduced a *multiway branching* in the proof tree of *DP* by selecting a shortest clause $L_1 \vee \cdots \vee L_k$ and producing k branches by the following truth assignments to its literals: (1) $L_1 = true$; (2) $L_1 = false$, $L_2 = true, \ldots$, (k) $L_1 = L_2 = \cdots = L_{k-1} = false$, $L_k = true$. Let $l(S), r$ denote the number of literals in S and in the longest clause of S, respectively. Then the worst-case time complexity of the algorithm presented in [16] is $O(l(S) \cdot \alpha_r^n)$, where $\alpha_r < 2$, but approaches the value of 2 very closely as r grows; for example, $\alpha_3 \approx 1.62$, $\alpha_4 = 1.84$, $\alpha_6 = 1.97$.

Although in certain cases these branching heuristics reduce the size of the proof tree, the experiments reported in [11] show that on the average they do not save significant time over *DP*.

Gallo and Urbani [11] presented two new algorithms for SAT utilizing the fact that the satisfiability problem for a set of propositional Horn clauses, *HORN-SAT*, is solvable in linear time (e.g., by the algorithms given in [9, 19]). They proposed two relaxation schemes that map instances of SAT into instances of HORN-SAT. Due to one of these schemes, every non-Horn clause $\neg p_1 \vee \cdots \vee \neg p_k \vee q_1 \vee \cdots \vee q_m$ $(m > 1)$ is replaced by two clauses (the first is a Horn one): $p_1 \wedge \cdots \wedge p_k \rightarrow r$ and $r \rightarrow q_1 \vee \cdots \vee q_m$, where r is a new atomic proposition.

This replacement transforms a given set of clauses S into $S_H \cup S_N$, where S_H, S_N denote sets of Horn and non-Horn clauses, respectively. Now satisfiability

of S can be decided by the following algorithm (called *HORN*2 in [11]):

(1) Decide satisfiability of S_H (**in linear time**); If S_H is unsatisfiable, then so is S, else assign $S' = S_H$;
(2) if $S_N = \varnothing$, then S is satisfiable, else select the shortest clause C from S_N; assign $S_N = S_N - \{C\}$, $S' = S' \cup \{C\}$;
(3) decide satisfiability of S' (**S' contains non-Horn clauses**); If S' is unsatisfiable, then so is S, else go to Step 2. □

Performance of this algorithm is enhanced by detecting unsatisfiability of a subset of S, which can be done just in linear time in cases when S_H is unsatisfiable. Indeed, as reported in [11], *HORN*2 outperformed other known algorithms. However, the authors noted that the performance of *HORN*2 decreases as the proportion of non-Horn clauses in S grows.

4. EARLY DISCOVERY OF SATISFIABILITY

Let $T(S)$ be the proof tree of a set of clauses S. The Davis-Putnam algorithm (as well as its variants) terminates either if it discovers a satisfiable leaf or if all the leaves of $T(S)$ are proved unsatisfiable. Hence, the algorithm investigates most of $T(S)$ (or even the entire tree, if S is unsatisfiable) in order to reach the leaves. Clearly, the run time could be shortened significantly by pruning the proof tree due to a method that would detect satisfiability or unsatisfiability of S in advance, that is, before reaching the leaves of $T(S)$. The algorithms presented in [11] (*HORN*2 in particular) can be viewed as such a pruning device for an early discovery of unsatisfiability of S. However, if S is satisfiable, then still *HORN*2 computes an interpretation that satisfies *all* the clauses of $S_H \cup S_N$.

So, the efficiency of algorithms for SAT can be further improved by an *early discovery of satisfiability*, *EDS*, of a given set of clauses.

Consider a set S of m clauses over n propositional variables. S has 2^n interpretations. Let $J(S)$ denote the set of all interpretations falsifying S, then

$$J(S) = \bigcup_{C \in S} J(C),$$

where $J(C)$ is the set of all interpretations falsifying a clause C of S. So, S is satisfiable if and only if

$$|J(S)| < 2^n, \tag{1}$$

where $|set|$ denotes the cardinality of *set*.

Let σ^i be a subset of S containing i clauses and $K(\sigma^i)$ denote the set of all interpretations such that each one falsifies *every* clause of σ^i, and hence, falsifies the disjunction of all clauses of σ^i, as well as the union of all clauses of σ^i considered as sets of literals. Then,

$$|K(\sigma^i)| = \begin{cases} 0 \text{ if } \bigcup_{C \in \sigma^i} C \textit{ contains complementary literals} \\ 2^{n - l(\sigma^i)} \textit{ otherwise}, \end{cases}$$

where $\cup C$ denotes the union of clauses considered as sets of literals and $l(\sigma^i)$ is the number of different literals appearing in σ^i. Hence,

$$|J(S)| = \sum_{\sigma^1 \subseteq S} |K(\sigma^1)| - \sum_{\sigma^2 \subseteq S} |K(\sigma^2)| + \cdots$$

$$= \sum_{i=1}^{m} \left((-1)^{i-1} \sum_{\sigma^i \subseteq S} |K(\sigma^i)| \right). \qquad (2)$$

In [14], Iwama presented an algorithm for SAT based on computing of the inclusion-exclusion formula (2) and showed that this computation takes polynomial average time if $p^2 n \geq \ln m - c$ (p is the probability that a given literal appears in a given clause, and c is a constant), however, for different parameters polynomial average time is not guaranteed. We would like the time of *EDS* testing to be absorbed by the average complexity of *SAT*. Since the latter has been estimated for certain cases as $O(n^2)$ in [12, 13], we limit the computing of $|J(S)|$ to its linear component:

$$|J(S)| \leq \sum_{\sigma^1 \subseteq S} |K(\sigma^1)| \leq \sum_{C \in S} 2^{n - l(C)}, \qquad (3)$$

where $l(C)$ denotes the number of literals in clause C.

Now (1) and (3) imply a test for early discovery of satisfiability, expressed by the following proposition.

Proposition 4.1. (*EDS test*). A set of clauses $S = \{C\}$ is satisfiable if $\sum_{C \in S} 2^{-l(C)} < 1$. □

So, if at a node ν of the proof tree, the current set S^ν contains m^ν clauses such that $\sum_{i=1}^{m^\nu} 2^{-l(C_i)} < 1$, then the *EDS* test terminates the processing, while any algorithm not performing *EDS* has to proceed until proving satisifiability of S^ν. Thus, the size of the set S^ν pruned of the tree determines the run-time saving due to the *EDS*.

The EDS test subsumes checking emptiness of a given set of clauses as at Step 1 of Algorithm DP.

5. BRANCHING STRATEGY

A node ν of $T(S)$ is the root of a subtree T^ν. If T^ν contains a satisfiable leaf, then T^ν is a *satisfiable subtree*; otherwise, it is an *unsatisfiable* one.

If S is satisfiable, then a sensible strategy is to keep the proof process all the time (from the very start at the root of $T(S)$) within satisfiable subtrees. Indeed, should this goal be achieved, it would guarantee that a satisfiable leaf of $T(S)$ is reached, and so the proof is terminated, without backtracking.

Let w be a branching node representing a set of clauses S^w and a literal L be chosen by the splitting rule. Then, w has two son-nodes in $T(S)$ representing S_L^w and $S_{\neg L}^w$ such that (see the notation of Algorithm *DP*, Step 5.2):

$$S_L^w = \Delta^w \cup \{(C - \{\neg L\}) | C \in \Delta_{\neg L}^w\}, \qquad (4)$$

$$S_{\neg L}^w = \Delta^w \cup \{(C - \{L\}) | C \in \Delta_L^w\}. \qquad (5)$$

Consider a clause $C \in S$ containing $l(C)$ literals. There are $2^{n-l(C)}$ interpretations falsifying C, so, the probability $P(C)$ that an arbitrary interpretation satisfies C is

$$P(C) = \frac{2^n - 2^{n-l(C)}}{2^n} = 1 - 2^{-l(C)}. \tag{6}$$

Now, consider a subset $\sigma \subseteq S$ and the probability $P(\sigma)$ that an arbitrary interpretation satisfies all clauses of σ. If the clauses of σ are disjoint (contain no common variables) pairwise, then

$$P(\sigma) = \prod_{C \in \sigma} P(C) = \prod_{C \in \sigma} (1 - 2^{-l(C)}). \tag{7}$$

Formula (7) does not hold for an arbitrary set σ because of correlation among its clauses; however, the value of $P(\sigma)$ computed due to (7) can serve as a heuristic indicator of satisfiability of σ relative to other subsets of S. So, if, for instance, $P(S_L^w) > P(S_{\neg L}^w)$, then it is advisable to branch to S_L^w, or otherwise, to $S_{\neg L}^w$.

Given a set of clauses σ and a literal L, let us define the *ratio* of L in σ as

$$r(L, \sigma) = \frac{P(\sigma_L)}{P(\sigma_{\neg L})}.$$

Then for a branching node w (cf. (4) and (5)):

$$r(L, S^w) = \frac{P(S_L^w)}{P(S_{\neg L}^w)} = \frac{\prod_{C \in \Delta_{\neg L}^w} (1 - 2^{1-l(C)})}{\prod_{C \in \Delta_L^w} (1 - 2^{1-l(C)})}.$$

Our intention is to proceed from a branching node to a satisfiable subtree (if it exists), so, the larger $r(L, S^w)$ is, provided $r(L, S^w) > 1$, the stronger is our preference for branching to S_L^w rather then to $S_{\neg L}^w$. With this in mind we suggest the following heuristics.

Branching Strategy BR. If the Splitting rule is applied to a set S^v, then choose a literal L such that for all literals λ occurring in S^v, $r(L, S^v) \geq r(\lambda, S^v)$ and branch to S_L^v (cf. (4)).

6. IDP: AN IMPROVED DAVIS-PUTNAM ALGORITHM

The Davis-Putnam algorithm can be improved by incorporating the EDS test and the BR branching strategy in the following way.

Algorithm IDP (Given S, decides its satisfiability)

(1) If $\Sigma_{C \in S} 2^{-l(C)} < 1$, then S is satisfiable;
(2–4.2) the same as in Algorithm DP, but substitute IDP for DP;
(5) else do 5.1–5.5:
(5.1) find a literal L such that for all literals λ occurring in S, $r(L, S) \geq r(\lambda, S)$;
(5.2) the same as in Algorithm DP;

(5.3) first apply IDP to S_L; if S_L is satisfiable, then S is so;
(5.4) else apply IDP to $S_{\neg L}$; if $S_{\neg L}$ is satisfiable, then S is so;
(5.5) else S is unsatisfiable. □

The time complexity of EDS and BR is linear in the number of clauses and variables of S, respectively. The complexity of both these additions to DP is actually absorbed by that of DP: $\Sigma 2^{-l(C)}$ and $r(\lambda, S)$ are computed while looking for a unit clause or a pure literal (Steps 3 and 4) and while updating S (Steps 3.1, 3.2, and 4.1).

7. EXPERIMENTS

The IDP has been programmed in C and run on *GOULD POWER NODE* under *UNIX* (*UTX/32.2.1a*). To make performance estimation computer-independent and comparable, the run-time has been measured in number of accesses to a single literal, which is the most frequent elementary operation determining the run-time of any algorithm for SAT.

The following parameters and their ranges have been chosen for the experiments:

- number of propositional variables, $10 \leq n \leq 200$;
- number of clauses, $10 \leq m \leq 1000$;
- number of literals $l(C)$ in each clause C has been determined in three different ways:
  (1) $l(C)$ chosen randomly (uniform distribution) in the interval $1 \leq l(C) \leq n$ (Table 1, Figures 1 and 2);
  (2) $l(C)$ chosen randomly (uniform distribution) in the interval $5 \leq l(C) \leq n/3$ (Table 2, Figure 3);
  (3) constant $l(C)$ for all clauses (e.g., $l(C) = 7$ in Table 3, Figure 4).

For each combination of parameters, 100 instances have been randomly generated and average measures computed. A representative sample of results is given in Tables 1–3 showing the run-time T of DP and IDP measured in thousands of accesses to literals (cf. Figures 1–4).

At any step ν of DP or IDP, the satisfiability of the original set S of m clauses is determined by the satisfiability of a subset $S^{\nu} \subset S$. Suppose that at this step, m^{ν} clauses remained not yet satisfied. If the EDS test is performed at this step showing satisfiability of S^{ν}, then S is satisfiable and the processing terminates. So, m^{ν}

TABLE 1. T (for DP and IDP) in thousands of accesses; SC in %; $1 \leq l(C) \leq n$.

n		15	25	45	65	90	115	155	200
$m = 200$	DP	4	30	73	117	164	203	280	342
	IDP	3	17	44	43	36	29	22	14
	SC	0	3	18	34	50	64	76	85
$m = 1000$	DP	5	9	117	561	1,175	1,695	2,748	3,741
	IDP	4	8	103	484	997	1,446	1,570	1,592
	SC	0	0	0	1	2	5	11	17

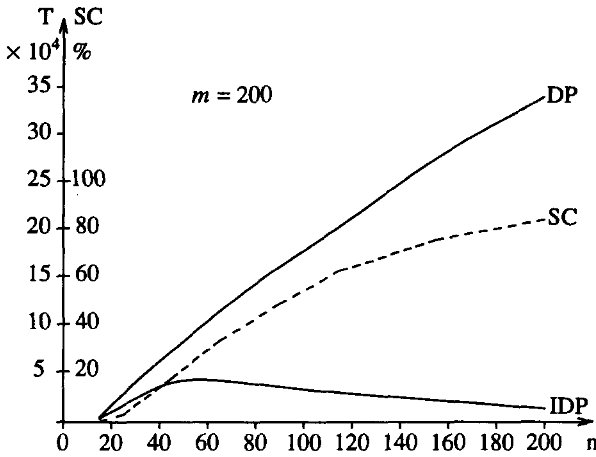

FIGURE 1. $1 \leq l(C) \leq n$.

clauses are saved from any further treatment. Let $SC = m^{\nu}/m$ denote the proportion of *saved clauses*. The larger is SC, the higher is the performance of IDP. Average values of SC in sample experiments are given in Tables 1–3. The longer are the clauses, the larger is SC. It is worth noting that if the length of clauses grows with the number of variables n, then the run-time of IDP is nonmonotonic in n and may even drop with increasing n, while SC approaches 100% (see Figures 1 and 3).

Algorithm IDP is equipped with EDS and BR, the means that are intended for a fast detection of satisfiability of a given set of clauses. However, if the set is unsatisfiable, then the overhead of computing EDS and BR is not justified and may decrease the performance of IDP. To investigate such cases, IDP was run on random sets containing many short clauses over relatively few variables, which are very likely to be unsatisfiable. (As it has been pointed out in [18], sets with m somewhat larger than n present particularly difficult SAT problems.) In the experiments, all clauses of a set had the same length l that varied for different sets such that $4 \leq l \leq 10$, while $100 \leq m \leq 500$ and $n = 50$ (see Table 4). Indeed, for

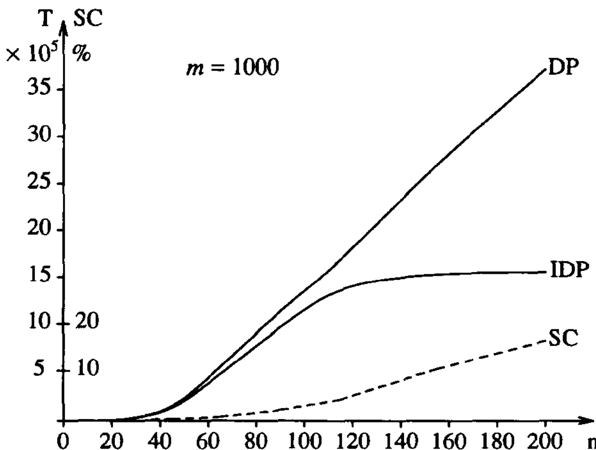

FIGURE 2. $1 \leq l(C) \leq n$.

TABLE 2. T (for DP and IDP) in thousands of accesses; SC in %; $5 \leq l(C) \leq n/3$.

n		10	15	25	35	45	65	90	115	155	200
	DP	17	32	51	103	144	184				
$m = 200$	IDP	13	18	22	4	1	0				
	SC	4	15	32	91	99	99				
	DP		176	403		1,048	1,856	2,804	3,819	5,605	
$m = 1000$	IDP		147	292		729	918	997	848	424	100
	SC		0	1		6	16	27	45	78	96

$m/n \geq 8$ and very short clauses, $l = 4$, IDP is slower than DP. As the clauses become longer (see Figure 5), the probability that a random set of clauses is satisfiable grows, so IDP becomes more and more efficient until it solves SAT in a very short constant time if $l > \log_2 m$ (cf. Proposition 4.1).

8. STABILITY OF IDP

It is known (e.g., see [11]) that existing algorithms for SAT are sensitive to the proportion of non-Horn clauses, nH, in the set being checked. These algorithms slow down while nH grows. This is true especially for algorithms that derive their efficiency from a Horn relaxation of SAT [11]. On the other hand, neither the EDS test nor the BR strategy depends on *non-Hornness* of the clauses, so IDP must exhibit a high stability with regard to changing nH. To investigate this property of IDP, it has been run with changing nH and fixed $n, m, l(C)$. Let $t_{\max}, t_{\min}$ denote, respectively, the maximum and the minimum run-time of an algorithm with nH changing in a certain range while other parameters are kept fixed. The larger is the ratio $t_{\max}/t_{\min}$, the more sensitive (less stable) is the algorithm with regard to non-Hornness of the given set of clauses. Table 5 compares the stability of three algorithms, DP, HORN2, and IDP by showing the value of $t_{\max}/t_{\min}$ for non-Horn-

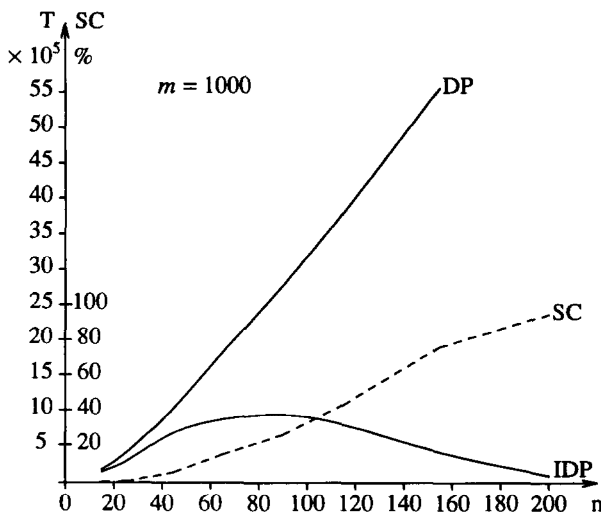

FIGURE 3. $5 \leq l(C) \leq n/3$.

TABLE 3. T (for DP and IDP) in thousands of accesses; SC in %; $l(C) = 7$.

n		15	25	45	65	90	115	155	200
$m = 200$	DP	21	54	180	316	458	592	701	727
	IDP	10	23	61	100	145	202	292	320
	SC	21	33	45	50	54	57	59	60
$m = 1000$	DP	147	393	1,325	2,278	3,845	5,887	10,739	16,119
	IDP	130	260	1,048	1,930	3,142	4,803	8,274	12,239
	SC	0	0	2	3	4	5	6	7

ness varied in the range $0 \leq nH \leq 50\%$, for $l(C) = 3$ and different combinations of n, m (the data for HORN2 are those published in [11]). Indeed, IDP turns out to be more stable than DP and HORN2.

9. SUMMARY

While the existing algorithms for SAT have an exponential worst-case time complexity [4, 6, 10], their average complexity has been shown polynomial in many practical cases [11–14, 18]. The latter indicates that computationally difficult cases of SAT requiring an exponential time are rather rare. So, to achieve a high performance, an algorithm for SAT must possess an ability to recognize special features of any particular instance which would allow a fast check of its satisfiability. In particular, in many cases it is possible to find out whether a given set of clauses is satisfiable (or unsatisfiable) by examining only a subset of it. We call this possibility an *early discovery of satisfiability*, *EDS* (or of *unsatisfiability*, *EDU*, respectively). The high performance of the algorithms presented in [11] is due to their ability to perform certain EDU testing. On the other hand, the known algorithms for SAT do not incorporate means for EDS. The DP and its variants, in

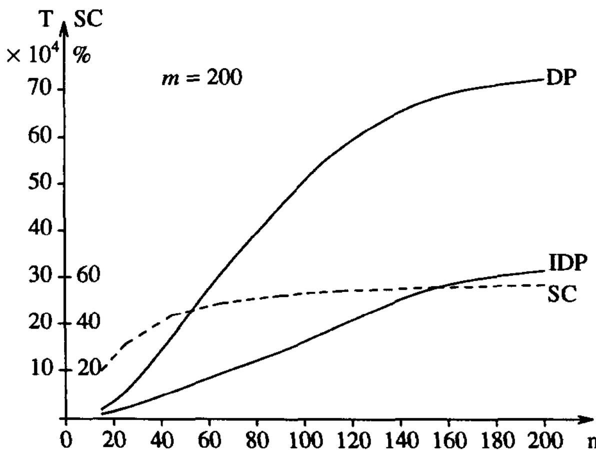

FIGURE 4. $l(C) = 7$.

TABLE 4. T (for DP and IDP) in thousands of accesses; $n = 50$.

l		4	5	6	7	8	9	10
$m = 100$	DP	82	76	73	69.0	65.0	61.0	58.0
	IDP	62	48	27	0.1	0.1	0.1	0.1
$m = 200$	DP	334	216	207	193	176	171.0	164.0
	IDP	197	166	131	72	0.2	0.2	0.2
$m = 300$	DP	705	407	396	321	299	287.0	271.0
	IDP	358	305	256	187	56	0.3	0.3
$m = 400$	DP	950	785	471	449	431	403	381.0
	IDP	1060	439	391	306	177	0.4	0.4
$m = 500$	DP	1170	1040	660	585	556	530.0	505.0
	IDP	1575	570	521	437	296	0.5	0.5

particular, decide that a set of clauses is satisfiable only when an interpretation is found that satisfies all clauses of the set (cf. Step 1 of DP, Section 2).

Another important factor determining run-time of an algorithm for SAT is the adopted branching strategy, since it strongly affects the shape and size of the proof-tree.

In this paper, we present an EDS test and suggest a branching heuristics, BR, related to the test. An algorithm, IDP, was developed that is an improvement of DP incorporating the EDS and BR. IDP was implemented and tested on a large number of randomly generated instances with varying dimensions: number of propositional variables (up to 200), number of clauses (up to 1,000), size of clauses, proportion of non-Horn clauses. In all these experiments (except a few extreme cases shown in Table 4) IDP outperformed DP, and the higher is the probability that a given set of clauses is satisfiable, the more efficient is IDP. It also showed a stability with respect to changing proportion of non-Horn clauses, which is higher than that of known algorithms.

Algorithms for SAT can be further improved by incorporating both EDU and EDS. So, a combination of HORN2 and IDP, for instance, is quite promising.

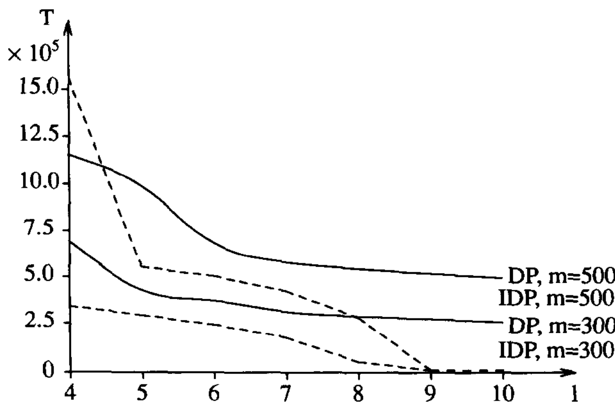

FIGURE 5. $n = 50$.

TABLE 5. The ratio t_{max}/t_{min} for non-Hornness varied in the range $0 \le nH \le 50\%$ and for $l(C) = 3$.

n	m	DP	HORN2	IDP
30	50	3.25	3.80	2.94
40	200	3.66	5.80	3.04
50	250	5.16	14.72	3.22

Many thanks to the anonymous referee for most apt, inspiring, and benevolent comments.

REFERENCES

1. Bugrara, K., and Brown, C., On the Average Case Analysis of Some Satisfiability Model Problems, *Information Science* 40:21–37 (1986).

2. Bugrara, K., Pan, Y., and Purdom, P., Exponential Average Time for the Pure Literal Rule, *SIAM. J. Comput.* 18:409–418 (1989).

3. Chang, C.-L., and Lee, R., *Symbolic Ligic and Mechanical Theorem Proving*, Academic Press, 1973.

4. Chvatal, V., and Szemeredi, E., Many Hard Examples for Resolution. *J. ACM* 35(4):759–768 (1988).

5. Cook, S., The Complexity of Theorem-Proving Procedures, in: *Proceedings of the 3rd ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

6. Cook, S., and Pitassi, T., A Feasibly Constructive Lower Bound for Resolution Proofs, *Inform. Process. Letters* 34:81–85 (1990).

7. Davis, M., and Putnam, H., A Computing Procedure for Quantification Theory, *J. ACM* 7:201–215 (1960).

8. Davis, M., Logemann, G., and Loveland, D., A Machine Program for Theorem-Proving, *Commun. ACM* 5:394–397 (1962).

9. Dowling, W., and Gallier, J., Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae, *J. Logic Programming* 3:267–284 (1984).

10. Galil, Z., On the Complexity of Regular Resolution and the Davis-Putnam Procedure, *Theoret. Comput. Sci.* 4:23–46 (1977).

11. Gallo, G., and Urbani, G., Algorithms for Testing the Satisfiability of Propositional Formulae, *J. Logic Programming* 7:45–61 (1989).

12. Goldberg, A., Average Case Complexity of the Satisfiability Problem, in: *Proceedings of the 4th Workshop on Automated Deduction*, Austin, Tex., 1979, pp. 1–6.

13. Goldberg, A., Purdom, P., and Brown, C., Average Time Analyses of Simplified Davis-Putnam Procedures, *Inform. Process. Letters* 15:72–75 (1982).

14. Iwama, K., CNF Satisfiability Test by Counting and Polynomial Average Time, *SIAM J. Comput.* 18:385–391 (1989).

15. Loveland, D., *Automated Theorem-Proving: A Logical Basis*, North-Holland, 1978.

16. Monien, B., and Speckenmeyer, E., Solving Satisfiability in Less Than 2^n Steps, *Discrete Appl. Math.* 10:287–295 (1985).
17. Purdom, P., Solving Satisfiability with Less Searching, *IEEE Trans. Pattern Anal. and Mach. Intell.* 6:510–515 (1984).
18. Purdom, P., and Brown, C., The Pure Literal Rule and Polynomial Average Time, *SIAM J. Comput.* 14:943–953 (1985).
19. Scutella, M., A Note on Dowling and Gallier's Top-Down Algorithm for Propositional Horn Satisfiability, *J. Logic Programming* 8:265–273 (1990).