A Fast Deterministic Algorithm for Formulas That Have Many Satisfying Assignments

EDWARD A. HIRSCH, *Steklov Institute of Mathematics at St.Petersburg, 27 Fontanka, 191011 St.Petersburg, Russia.*
E-mail: hirsch@pdmi.ras.ru

Abstract

How can we find any satisfying assignment for a Boolean formula that has many satisfying assignments? There exists an obvious *randomized* algorithm for solving this problem: one can just pick an assignment at random and check the truth value of the formula for this assignment, this is iterated until a satisfying assignment occurs. Does there exist a polynomial-time *deterministic* algorithm that solves the same problem? This paper presents such an algorithm and shows that its worst-case running time is linear when input formulas are in k-CNF and a fraction of satisfying assignments (among all possible assignments) is greater than a constant. This algorithm is almost the same as the algorithm proposed by Monien and Speckenmeyer (and independently by Dantsin) in the early 1980s for less than 2^n steps 3-SAT decision.

Another result of this paper is that if a formula in k-CNF has many satisfying assignments, then there exists a *short* satisfying assignment, i.e. an assignment of truth values to a small number of variables. The proposed algorithm yields just such a short satisfying assignment. We also show that there exist formulas in general CNF having many satisfying assignments, that have no short satisfying assignments.

Keywords: Boolean formulas, the propositional satisfiability problem

1 Introduction

It is well-known that 3-SAT (the problem of satisfiability of a formula in 3-CNF) is NP-complete, all known decision algorithms for it (as well as for finding any satisfying assignment) have exponential upper bounds on time. Research on this problem has concentrated on three main directions.

The first direction is the improvement of the trivial bound 2^n on the complexity of 3-SAT (and general SAT) with respect to different parameters ([12, 2, 18, 13, 4, 14, 9, 15, 10, 7, 8]). Some recent bounds are:

- $2^{0.59n}$, where n is the number of variables in the input formula ([9, 15]),
- $2^{0.309K}$, where K is the number of clauses in the input formula ([7, 8]),
- $2^{0.106L}$, where L is the length of the input formula ([8]).

The second direction is the *experimental* study of the running time of various algorithms on various classes of formulas. Many (randomized) algorithms concerning SAT were proposed. Many reassuring experimental results about their performance were obtained during the last five years (TABLEAU [1], GSAT [17, 16]). Unfortunately, there is still a serious gap between experimental and theoretical results. Nobody can

describe a nontrivial class of formulas and guarantee that GSAT or TABLEAU works efficiently for formulas of this class. Known theoretical results concerning GSAT guarantee only that GSAT works efficiently for most formulas ([11]).

The third direction is the search for algorithms that are *guaranteed* to work efficiently on separate classes of formulas. For example, polynomial-time algorithms are known for formulas in 2-CNF, Horn formulas, formulas that represent systems of linear equations over Z_2 and some other classes of CNF-formulas. There is a survey of such results in [3]. In this paper we consider one family of such classes. Namely, we are interested in CNF-formulas that have many satisfying assignments with respect to the number of all possible assignments.

Consider, for example, $f = (a \vee b \vee \overline{c}) \,\&\, (\overline{a} \vee c) \,\&\, (\overline{b} \vee c)$: the total number of assignments to the variables a, b, c is $2^3 = 8$; four of these are satisfying ($\{a, b, c\}$, $\{\overline{a}, b, c\}$, $\{a, \overline{b}, c\}$, $\{\overline{a}, \overline{b}, \overline{c}\}$), so 50% of assignments are satisfying. There exists a very simple randomized algorithm that works efficiently on formulas of this class. This algorithm just picks an assignment at random and check the truth value of the formula for this assignment, this is iterated until a satisfying assignment occurs. For example, if at least 50% of assignments satisfy f, then this algorithm will stop after consideration of the first assignment with probability at least 1/2. After the second step it will stop with probability at least 3/4 and so on (with probability at least $1–\frac{1}{2^i}$ after the i-th step).

We now give a more formal description of this algorithm. We define a *complete* assignment (as opposed to a *partial* assignment) to be an assignment that fixes values of all variables appearing in the formula. For precise definitions we refer the reader to Section 2.

ALGORITHM 1.1
Input: A satisfiable Boolean formula f with variables $a_1, a_2, \ldots, a_n$.
Output: A satisfying assignment for f.

Method:

1) Pick a complete assignment S at random.
2) If S satisfies f, output S,
   otherwise go to step 1.

□

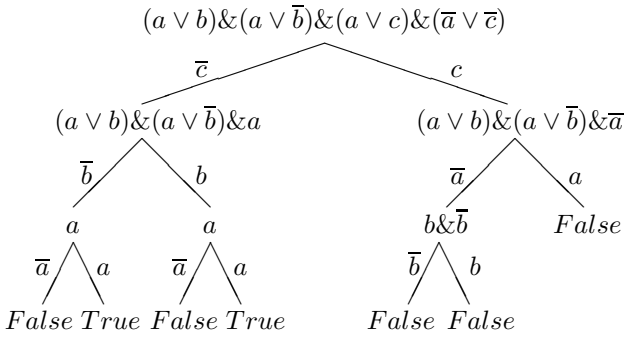
PROPOSITION 1.2
The mathematical expectation of the Algorithm 1.1 running time on formula f is $O(L\delta^{-1})$, where δ is the fraction of complete satisfying assignments for f, i.e.

$$\delta = \frac{\text{the number of complete satisfying assignments for } f}{\text{the number of all complete assignments, i.e. } 2^n}.$$

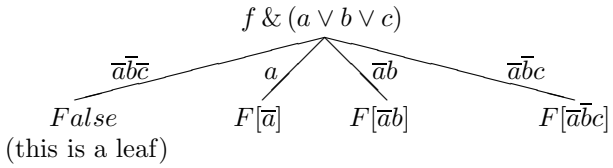
PROOF. Easy calculation. ∎

We show that there exists a *deterministic* algorithm taking a polynomial time on this class. This algorithm is given in Section 3. It can be applied to an arbitrary CNF-formula; it takes a polynomial time on CNF-formulas that have a sufficiently large number of satisfying assignments and consist of clauses bounded in length by some constant. It is based on the method proposed in [12] (see also [13]) and independently

in [2] (see also [4]) for solving the satisfiability problem in less than 2^n steps (here and below, n is the number of variables occurring in the input formula). This method is, in fact, a modification of the branching-tree algorithm ([5]) which is close to and often referred as the Davis-Putnam procedure ([6]). Satisfiability of a formula f reduces to satisfiability of the formulas $f[\overline{a}]$ and $f[a]$ (i.e. $f[a:=False]$ and $f[a:=True]$); it is enough to prove satisfiability of one of these to prove satisfiability of f. Satisfiability of each of these formulas reduces again to satisfiability of two formulas and so on. In this way we obtain a tree like the following:



We call this *a branching tree*. A full branching tree constructed by this procedure seems to contain up to 2^n leaves. However, as it was shown in [2] and [12], the number of leaves in a branching tree can be decreased by a special choice of the succession of variables the formula branches through. One can do it by choosing the variables for successive branchings from the same clause. We group these 2^k (k is the length of the clause) cases into k nodes. For example, the clause $(a \vee b \vee c)$ will produce the branching



A side effect of each branching is that we obtain many unsatisfying assignments. These are all assignments that evaluate the chosen clause to $False$. In the example above all assignments containing simultaneously $\overline{a}$, $\overline{b}$ and $\overline{c}$ (i.e. $a:=False$, $b:=False$, $c:=False$) are unsatisfying. Thus, if the input formula has many satisfying assignments, the limit of unsatisfying assignments will soon be reached. After that we cannot carry out the next branching, i.e. we obtain the empty formula. Owing to this fact, the algorithm must stop after a constant number of branchings, so it takes a linear time.

This algorithm produces a satisfying assignment that has one additional property: since it is found after constructing only a few levels of the branching tree, this assignment is short. In other words, it fixes the values of only a constant number of variables; the values of other variables are immaterial. This constant does not depend on the total number of variables appearing in the formula.

This paper is organized as follows. Section 2 contains basic definitions. In Section 3

we formulate main results and give a description of our deterministic algorithm. We discuss the results in Section 4, and then their proof is presented in Section 5 and Section 6.

2 Basic definitions

Let U be a finite set of Boolean variables. We use the following notations:

$\overline{v}$ for the logical negation of variable $v \in U$,
$\overline{V}$ for the set $\{\overline{v} \mid v \in V\}$, where $V \subseteq U$.

Literals are variables or their negations. A pair of *complementary* literals is a variable together with its negation.

A formula in *conjunctive normal form (CNF-formula)* is

$$\bigwedge_{i=1}^{c} \bigvee_{j=1}^{m_i} l_{ij},$$

where $l_{ij} \in U \cup \overline{U}$, $c \geq 0$, $m_i \geq 0$, $\forall j \, \forall h \, ((j \neq h) \rightarrow ((l_{ij} \neq l_{ih}) \& (l_{ij} \neq \overline{l}_{ih})))$. *The length of a clause* $\bigvee_{j=1}^{m_i} l_{ij}$ is the number of literals it contains (i.e. m_i). *The length of a CNF-formula* is the sum of the lengths of all its clauses.

Remark 2.1
The empty clause is interpreted as $False$; the empty formula is interpreted as $True$.

Our notion of assignment slightly differs from the usual one. We shall distinguish *partial* assignments and *complete* assignments. *A partial assignment* (or simply *assignment*) is a set $S \subset U \cup \overline{U}$ that does not contain complementary literals. *The size of an assignment* is its cardinality. *A complete assignment* for formula f with n variables is an assignment of size n (i.e. it fixes the values of all variables). An assignment (partial or complete) is *satisfying* for formula f if each clause of f contains at least one literal from S.

We write $g = f[S]$ for an assignment $S = \{v_1, v_2, \ldots, v_m\}$ and CNF-formulas f and g if g can be obtained from f by

- eliminating all clauses that contain the literals v_i, and
- deleting the literals $\overline{v}_i$ from the other clauses.

We say that g is obtained from f by assignment S. For short we write $f[v_1, v_2, \ldots, v_m]$ instead of $f[\{v_1, v_2, \ldots, v_m\}]$.

A branching tree is a tree whose nodes are labelled with CNF-formulas such that if some node is labelled with CNF-formula f, then its sons are labelled with formulas $f[S_1]$, $f[S_2]$, ..., $f[S_m]$ for some assignments S_1, S_2, ..., S_m.

Remark 2.2
This definition generalizes the notion of a branching tree introduced in Section 1.

Example 2.3
An example of a branching tree is shown in Figure 1.

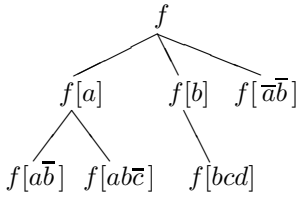


FIG. 1.

For the purpose of proving bounds on the size of the branching tree constructed by our algorithm, we introduce the following notation. *A CNF-function* is a pair (X, f) where f is a CNF-formula and X is a set that contains all variables occurring in f (but also may contain other variables).

EXAMPLE 2.4
An example of a CNF-function: $\left(\{a, b, c\}, (a \vee b)\&\overline{b}\right)$.

In fact, our algorithm constructs some branching tree. Its nodes each can be labelled with some CNF-function in a natural way. To estimate the number of nodes in this tree (and thus the time complexity of the algorithm) we use some auxiliary results concerning *branching tuples* introduced in [10]. These results are presented in Section 5, and here we define some relevant terminology.

Suppose we have a tree T whose nodes are labelled with objects associated with some characteristic of their complexity: to each object F we attach a non-negative integer $\mu(F)$ (*complexity of* F). Assume that this tree has the property that the complexity of an object labelling any node is greater than the complexity of each of the objects labelling its sons.

EXAMPLE 2.5 (Typical examples of μ)
1) objects are CNF-formulas, $\mu_1(f)$ is the number of variables in the formula f;
2) objects are CNF-formulas, $\mu_2(f)$ is the length of f;
3) objects are CNF-functions, $\mu_3((X, f))$ is the size $\#X$ of X.

In this paper we use only μ_3. However, until Section 6 we do not fix any concrete measure of complexity. It is even immaterial for us now that our objects are CNF-functions.

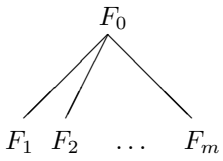


FIG. 2.

Let us consider an arbitrary node in our tree labelled with object F_0. Suppose its sons are labelled with $F_1, F_2, \ldots, F_m$ (see Figure 2). *A branching vector of a node* is an m-tuple $(t_1, \ldots, t_m)$ where t_i are positive numbers not exceeding $\mu(F_0) - \mu(F_i)$.

The branching number $\tau(\vec{t})$ *of a node* with branching vector $\vec{t} = (t_1, t_2, \ldots, t_m)$ is the only positive root of *the characteristic polynomial of the branching vector* $\vec{t}$

$$h_{\vec{t}}(x) = 1 - \sum_{i=1}^{m} x^{-t_i}.$$

We suppose $\tau(\vec{0}) = 1$ for leaves.

REMARK 2.6
The characteristic polynomial $h_{\vec{t}}(x)$ is a monotone function of x on the interval $(0, +\infty)$, and, $h_{\vec{t}}(0) = -\infty$, $h_{\vec{t}}(+\infty) = 1$. Hence, $h_{\vec{t}}(x) = 0$ has a unique solution on $(0, +\infty)$.

The largest of the branching numbers $\tau(\vec{t})$ of all nodes of the tree T we call *the branching number of the tree* T. We denote it $\tau_{max,T}$ (or simply τ_{max}).

Instead of the notion of a tree level we introduce the notion of *a floor* of our tree T. The *i-th floor* of the tree T is the set of nodes containing objects F with $\mu(F) = \mu(F_0) - i$, where F_0 is the object in the root.

3 Algorithm

We now consider an algorithm that finds in a linear time a satisfying assignment for formula in k-CNF that have a sufficiently large number of satisfying assignments. Below L denotes the length of the formula f, and n denotes the number of variables it contains.

The idea of Algorithm 3.1 is as follows. Let f be the input formula, and consider a clause $a \vee b$ in it. A satisfying assignment cannot contain $\overline{a}$ and $\overline{b}$ simultaneously. Therefore, having considered only one clause, we have excluded from consideration a quarter of all possible complete assignments (and so have found that at most 75% of complete assignments satisfy f). Now we can consider $f[a]$ and $f[\overline{a}, b]$ instead of f and apply the same argument to them, and so on.

Applying these operations, we obtain a branching tree for f. Algorithm 3.1 builds it in width successively constructing the floors $\Phi_0, \Phi_1, \ldots$ of this tree.

ALGORITHM 3.1
Input: CNF-formula f with variables $a_1, a_2, \ldots, a_n$.
Output: A satisfying assignment for f, if f is satisfiable, otherwise "No".

Method:

0) i:=0; Φ_0:=$\{f\}$; for all $m \in \{1, \ldots, n\}$ Φ_m:=$\emptyset$.
1) Take the shortest clause
$$l_1 \vee l_2 \vee \ldots \vee l_{s(i,j)}$$
in each $f_{i,j} \in \Phi_i = \{f_{i,1}, \ldots, f_{i,q_i}\}$.
2) For each $f_{i,j}$, construct new formulas $g_{i,j,1}, g_{i,j,2}, \ldots, g_{i,j,s(i,j)}$ obtained from $f_{i,j}$ by assignments
$$\{l_1\};$$

$\{\overline{l}_1, l_2\}$;
$\{\overline{l}_1, \overline{l}_2, l_3\}$;
...
$\{\overline{l}_1, \overline{l}_2, \overline{l}_3, \ldots, l_{s(i,j)}\}$.

If at least one of these formulas $g_{i,j,m}$ is the empty formula, stop with the answer corresponding to the chain of assignments leading to it.

3) For all $g_{i,j,m}$ that do not contain the empty clause,

$$\Phi_{i+m} := \Phi_{i+m} \cup \{g_{i,j,m}\}.$$

4) $i := i + 1$;
If $i > n$, stop with the answer "No",
otherwise go to step 1.

□

The main theorem of this paper states that Algorithm 3.1 works efficiently for formulas in k-CNF that have a sufficiently large number of satisfying assignments. We now define this class of formulas more precisely.

DEFINITION 3.2
Let $k \geq 2$ be an integer, and $0 < \delta < 1$. *Class* $C_{k,\delta}$ is the set of CNF-formulas such that CNF-formula f belongs to $C_{k,\delta}$ iff

1) each clause of f contains at most k literals;
2) $\dfrac{\text{the number of complete satisfying assignments for } f}{\text{the number of all complete assignments, i.e. } 2^n} \geq \delta$.

We introduce some notation before formulating the result.

DEFINITION 3.3
We denote by $\lambda(q)$ the branching number $\tau((1, 2, \ldots, q))$, i.e. the only positive root of the polynomial

$$(\star) \qquad h_{(1,2,\ldots,q)}(x) = 1 - x^{-1} - x^{-2} - \ldots - x^{-q}.$$

We define $B(k) = \left(\log_{\lambda(k)} 2 - 1\right)^{-1}$.

REMARK 3.4
An easy calculation shows that if $q_1 > q_2 > 1$, then $2 > \lambda(q_1) > \lambda(q_2) > 1$.

THEOREM 3.5 (Main Theorem)
Let $k \geq 2$ be an integer, $0 < \delta < 1$. Then

(a) Algorithm 3.1 finds a satisfying assignment for any satisfiable CNF-formula;
(b) if f belongs to $C_{k,\delta}$, then Algorithm 3.1 stops after having constructed at most $k\left(\frac{2}{\delta}\right)^{B(k)}$ nodes of the branching tree and takes time $O(L)$ on each of them;
(c) if f belongs to $C_{k,\delta}$, then the size of the output assignment does not exceed $\log_{2/\lambda(k)}(\frac{2}{\delta}) + k - 1$.

COROLLARY 3.6
Let $k \geq 2$ *be an integer,* $0 < \delta < 1$ *. If a CNF-formula belongs to* $C_{k,\delta}$*, then it has a satisfying assignment of size at most* $\log_{2/\lambda(k)}(\frac{2}{\delta}) + k - 1$.

The statement of Main Theorem will be proved in two steps. In Section 5 we prove two lemmas which generalize the results from [10] concerning branching tuples. In Section 6 these results are applied to our problem.

4 Remarks

1. If δ is considered to be a constant, then we obtain a linear bound on the running time of the Algorithm 3.1 and a constant bound on the size of the satisfying assignment. We can also view δ as a polynomial function of the formula parameters (the length, the number of variables, etc.). In this case we also obtain subexponential bounds since the time bound of the algorithm has only polynomial dependence on δ.

2. Approximate values of $B(k)$ are given in Table 1.

k	2	3	4	5	6
B	2.27	7.27	17.79	39.36	83.09

TABLE 1.

3. One might suspect that only very short formulas can have many satisfying assignments. To disprove this, we present a simple method of constructing formulas in $C_{3,0.5}$ having arbitrary lengths not greater than $O(n^2)$: take an arbitrary formula in 2-CNF and add a new variable to each of its clauses. Clearly, if one sets this new variable to true, the formula is satisfied irrespective of the values of other variables.

4. If some formula in k-CNF has a short satisfying assignment of size s, then this formula belongs to $C_{k,\delta}$, where $\delta = 2^{-s}$. Thus, if some other class of formulas in k-CNF has short satisfying assignments, then this class must be contained in $C_{k,\delta}$ for some appropriate δ. Therefore, Corollary 3.6 gives some characterization of classes $C_{k,\delta}$ via the size of satisfying assignments.

5. By the use of Corollary 3.6, one can show that a satisfying assignment can be obtained by a simple algorithm enumerating all assignments of the size mentioned in Corollary 3.6. However, the number of these assignments (and, hence, the worst-case running time) is much worse than the bound guaranteed by Main Theorem.

6. Surprisingly, formulas in general CNF (i.e. without the restriction on the length of clauses) that have many satisfying assignments, do not have short satisfying assignments. The following simple counting argument shows that for arbitrary n there is a formula in CNF with n variables that has at least 50% fraction of satisfying assignments, but has no satisfying assignments of size at most $\lceil \sqrt{n} \rceil$.

For each set S of assignments we can construct a formula F in CNF such that S is the set of all assignments satisfying F. Thus, it suffices to prove that there is a set S consisting of at least $2^{n/2}$ complete assignments, such that for each partial assignment I of size $\lceil \sqrt{n} \rceil$, there exists a complete assignment J such that $I \subset J$,

$J \notin S$. There are $r(n) = 2^{\lceil\sqrt{n}\rceil} n!/(\lceil\sqrt{n}\rceil!(n - \lceil\sqrt{n}\rceil)!)$ assignments of size $\lceil\sqrt{n}\rceil$. If n is sufficiently large, $r(n)$ is less than 2^n. Thus, if for each such partial assignment we remove an arbitrary complete assignment from the set of all complete assignments, the remaining set will satisfy the condition above.

We have now that there are short satisfying assignments for formulas in k-CNF (k is a constant) and there are no such assignments for formulas in CNF. This fact seems to be connected with the conjecture (see e.g. [10]) stating that there is no less–than–2^n algorithm for SAT (though there exists such an algorithm for k-SAT).

5 The number of nodes in a branching tree

LEMMA 5.1 (Kullmann, Luckhardt, [10])
The number of nodes of the i-th floor in a branching tree does not exceed $(\tau_{max})^i$.

PROOF. By induction on the number of nodes in the tree.

Base. Tree consisting of one node: $(\tau_{max})^i = 1$ which is equal to the number of nodes if $i = 0$ and greater if $i > 0$.

Step. Let us consider tree R presented in Figure 3.

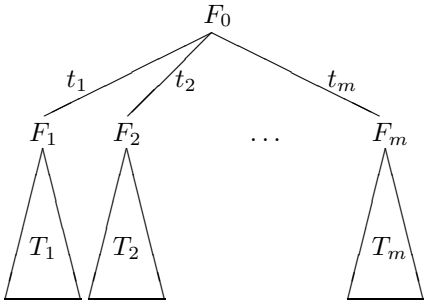


FIG. 3.

Let $\vec{t} = (t_1, \ldots, t_m)$ be the branching vector of its root, $\tau_{max} = \tau_{max,R}$. Then

$$\begin{aligned}
&\text{the number of nodes of the } i\text{-th floor of } R = \\
=\ & \sum_{j=1}^{m} (\text{the number of nodes of the } (i-t_j)\text{–th floor of } T_j) \\
\leq\ & \sum_{j=1}^{m} (\tau_{max,T_j})^{i-t_j} \quad \text{(by the induction hypothesis)} \\
\leq\ & \sum_{j=1}^{m} (\tau_{max})^{i-t_j} = (\tau_{max})^i \cdot \sum_{j=1}^{m} (\tau_{max})^{-t_j} \\
=\ & (\tau_{max})^i \cdot (1 - h_{\vec{t}}(\tau_{max})) \quad \text{(by the definition of } h_{\vec{t}}\text{)} \\
\leq\ & (\tau_{max})^i \quad \text{(by monotonicity of } h_{\vec{t}}\text{)}.
\end{aligned}$$

∎

LEMMA 5.2
Let R be a branching tree such that branching vectors of all its nodes take the form $(1, 2, \ldots, q)$, $q \leq k$. Let $i, m \geq 1$, $k \geq 2$. Then the number of nodes of the $(i+m)$-th floor that have parent in the $(i-1)$ –th or previous floors of the tree T, does not exceed $(\lambda(k))^i$.

PROOF. Each of the nodes we are interested in, is a son of some node that has a son of the i-th floor (see Figure 4). In addition, each node has at most one son of the $(i+m)$-th floor. Thus, the number of nodes that we are interested in is not greater than the number of nodes of the i-th floor. Lemma 5.1 then says that there are at most $(\tau_{max})^i$ such nodes. The branching number of this tree is $\lambda(k)$ (see Remark 3.4). So $(\tau_{max})^i \leq (\lambda(k))^i$. ∎

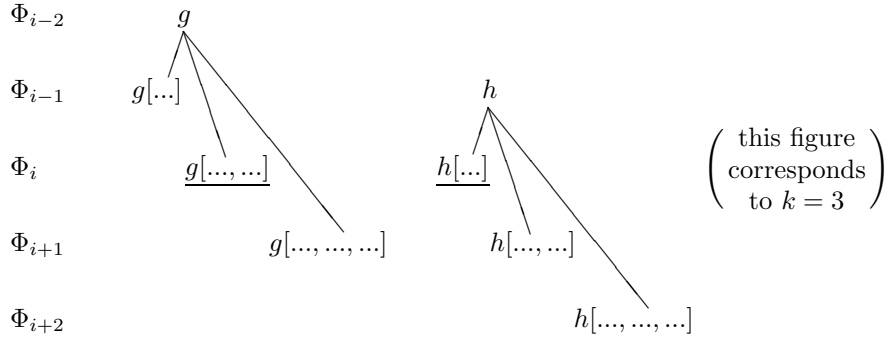


FIG. 4.

6 Proof of Main Theorem

Let us consider the tree constructed by Algorithm 3.1. We can label its nodes with CNF-functions in a natural way. If Algorithm 3.1 splits formula $f_{i,j}$ at step 2 as described in the algorithm, then the sons of the node labelled with CNF-function $(X_{i,j}, f_{i,j})$ are labelled with $(X_{i,j} - \{v_1\}, g_{i,j,1})$, $(X_{i,j} - \{v_1, v_2\}, g_{i,j,2})$, $\ldots$, $(X_{i,j} - \{v_1, v_2, \ldots, v_{s(i,j)}\}, g_{i,j,s(i,j)})$, where v_1, v_2, $\ldots$, $v_{s(i,j)}$ are variables that correspond to the literals l_1, l_2, $\ldots$, $l_{s(i,j)}$.

Note that Algorithm 3.1 constructs several first floors of some branching tree which differs from the full one (with 2^n leaves) only by the fact that some nodes labelled with $False$ (i.e. labelled with CNF-functions containing the empty clause) are excluded from it. (If CNF-formula f contains the clause

$$l_1 \vee l_2 \vee \ldots \vee l_q,$$

then every assignment containing

$$\{\overline{l}_1, \overline{l}_2, \ldots, \overline{l}_q\}$$

is unsatisfying, i.e. $\forall l'_1 \ldots l'_r \, (f[l'_1 \ldots l'_{n-q}, l_1 \ldots l_q] = \emptyset)$.)

The algorithm constructs this tree in width until a satisfying assignment occurs. Hence, the algorithm works correctly.

We denote by T the tree constructed by Algorithm 3.1. We shall bound its size using the results of the previous section. For complexity μ we use $\mu((X,f)) = \#X$. Branching vectors of all nodes of the tree T take the form $(1,2,\ldots,q)$, $q \leq k$. Characteristic polynomials of these branching vectors take the form

$$h_{(1,2,\ldots,q)}(x) = 1 - \sum_{i=1}^{q} x^{-i} \qquad (q \leq k).$$

The corresponding branching numbers are $\lambda(q)$. Thus, the branching number of this tree is the positive root of the polynomial $(\star)$, i.e. $\lambda(k)$. (See Remark 3.4.)

Let us consider several first floors of this tree obtained after step 3, at which point the algorithm has constructed all nodes of the i-th floor but has not yet constructed any of their sons. All assignments satisfying f also satisfy at least one formula of floors i, $i+1$, $\ldots$, $i+k-1$ constructed by this stage. There are at most 2^{n-j} such assignments for each node of the j-th floor. From Lemma 5.2, there are at most $(\lambda(k))^i$ nodes of the i-th floor, at most $(\lambda(k))^i$ nodes of the $(i+1)$-th floor etc. Hence, the original formula has not more than

$$(\lambda(k))^i \cdot 2^{n-i} + (\lambda(k))^i \cdot 2^{n-i-1} + \ldots + (\lambda(k))^i \cdot 2^{n-i-k+1} \leq 2 \cdot (\lambda(k))^i \cdot 2^{n-i}$$

complete satisfying assignments, i.e.

$$2 \cdot (\lambda(k))^i \, 2^{n-i} \geq \delta \cdot 2^n.$$

Hence

$$\left(\frac{\lambda(k)}{2}\right)^i \geq \frac{\delta}{2},$$

so

$$i \leq \log_{\lambda(k)/2}\left(\frac{\delta}{2}\right).$$

Therefore, Algorithm 3.1 stops (having found a satisfying assignment) having completed not more than $j \leq \log_{\lambda(k)/2}(\frac{\delta}{2})$ floors. Hence, owing to Lemma 5.1 and Lemma 5.2 it can construct not more than $k \cdot \lambda(k)^j + (\lambda(k))^{j-1} + \ldots + 1$ nodes. Easy calculation shows that

$$k \cdot \lambda(k)^j + (\lambda(k))^{j-1} + \ldots + 1 \; < \; \left(k + \frac{1}{\lambda(k)-1}\right) \cdot (\lambda(k))^j \; \leq$$

$$\leq \; \left(k + \frac{1}{\lambda(2)-1}\right) \cdot (\lambda(k))^j \; = \; \left(k + \frac{2}{\sqrt{5}-1}\right) \cdot (\lambda(k))^j \; =$$

$$= \; O\left(k \cdot (\lambda(k))^j\right) \; = \; O\left(k \cdot (\lambda(k))^{\log_{\lambda(k)/2}\left(\frac{\delta}{2}\right)}\right) \; =$$

$$= \; O\left(k \cdot (\lambda(k))^{\log_{\lambda(k)}\left(\frac{\delta}{2}\right) \big/ \log_{\lambda(k)} \frac{\lambda(k)}{2}}\right) \; = \; O\left(k \cdot \left(\frac{\delta}{2}\right)^{1 \big/ \log_{\lambda(k)} \frac{\lambda(k)}{2}}\right) \; =$$

$$= O\left(k \cdot \left(\frac{2}{\delta}\right)^{B(k)}\right)$$

and each node requires linear (with respect to the length of the formula) time.

This implies part (b) of Main Theorem. Since the number of the floor where the first satisfying assignment occurs is at most

$$j + k - 1 \leq \log_{2/\lambda(k)}\left(\frac{2}{\delta}\right) + k - 1,$$

we get part (c).

□

Remark 6.1

Table 2 presents the dependence between k, δ and the value of the size of the short satisfying assignment guaranteed by Main Theorem.

$1/\delta$ \ k	2	3	4	5	6
2	7	18	40	84	173
4	10	26	59	125	257
8	14	35	78	165	341
16	17	43	96	205	425
32	20	51	115	246	509
64	23	59	134	286	593
128	27	68	153	326	677
256	30	76	172	367	761
512	33	84	190	407	845
1024	36	93	209	448	930

Table 2.

7 Conclusion

In this paper we have considered Boolean formulas in k-CNF that have many satisfying assignments. These formulas are easy for the simple randomized algorithm that just picks a random assignment. We have shown that there exists a deterministic algorithm that solves this problem in a polynomial time. We have also shown that this algorithm finds a short satisfying assignment. There exist formulas in general CNF that have many satisfying assignments but do not have short satisfying assignments. The benefit of this paper is that it presents one more class of Boolean formulas for which a polynomial-time deterministic algorithm is known.

Acknowledgements

The author would like to thank Evgeny Dantsin for bringing this problem to his attention and for guidance. Also, many thanks to Oliver Kullmann for his helpful comments. This text would never have become readable without the remarks of anonymous referees. The author is partially supported by INTAS-RFBR project N95-0095.

References

[1] J. M. Crawford, L. D. Auton, *Experimental results on the Crossover Point in Satisfiability Problems*, Proceedings of the Eleventh Conference on Artificial Intelligence (AAAI-93), **1993**.

[2] E. Dantsin, *Tautology proof systems based on the splitting method* (in Russian), PhD thesis, Leningrad Division of Steklov Institute of Mathematics (LOMI), **1983**.

[3] E. Dantsin, *The algorithmics of the propositional satisfiability problem* (in Russian). In *Voprosy Kibernet. (Moscow), No. 131*, pp. 7–29, USSR Academy of Sciences, 1987. This collection was translated into English as *Problems of Reducing the Exhaustive Search* (V.Kreinovich and G.Mints, editors), American Mathematical Society Translations—Series 2, vol. 178, AMS, **1997**.

[4] E. Dantsin and V. Kreinovich. *Exponential upper bounds for the propositional satisfiability problem* (in Russian), Proceedings of the 9th National Conference on Mathematical Logic (Leningrad), p. 47, September **1988**.

[5] M. Davis, G. Logemann, D. Loveland, *A machine program for theorem-proving*, Comm. ACM, **5** (**1962**), pp. 394–397.

[6] M. Davis, H. Putnam, *A computing procedure for quantification theory*, J. Assoc. Comp. Mach., **7**, **1960**, pp. 201–215.

[7] E. A. Hirsch *Deciding satisfiability of formulas with K clauses in less than $2^{0.30897K}$ steps*, PDMI preprint **4/1997**, Steklov Institute of Mathematics in St.Petersburg, **1997**, `ftp://ftp.pdmi.ras.ru/pub/publicat/preprint/1997/97-04/`

[8] E. A. Hirsch *Two new upper bounds for SAT*, Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, **1998**. To appear.

[9] O. Kullmann, *Worst-case Analysis, 3-SAT Decision and Lower Bounds: Approaches for Improved SAT Algorithms,* DIMACS Proceedings SAT Workshop 1996, American Mathematical Society, **1996**, Eds.: Gu, Du, Pardalos.

[10] O. Kullmann, H. Luckhardt, *Deciding propositional tautologies: Algorithms and their complexity,* submitted to Information and Computation, **1997**.

[11] E. Koustsoupias, Ch. Papadimitriou, *On the greedy algorithm for satisfiability*, Information Processing Letters **43** (**1992**), pp. 53–55.

[12] B. Monien, E. Speckenmeyer, *3-satisfiability is testable in $O(1.62^r)$ steps*, Bericht Nr. **3/1979**, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn.

[13] B. Monien, E. Speckenmeyer, *Solving satisfiability in less than 2^n steps*, Discrete Applied Mathematics, **1985**, Vol. **10**, pp. 287–295.

[14] I. Schiermeyer, *Solving 3-satisfiability in less than 1.579^n steps*, LNCS **702**, **1993**, pp. 379–394.

[15] I. Schiermeyer, *Pure literal look ahead: An $O(1.497^n)$ 3-Satisfiability algorithm,* In: Workshop on the Satisfiability Problem, Technical Report, Siena, April, 29 – May, 3, **1996**, Eds.: J. Franco, G. Gallo, H. Kleine Büning, E. Speckenmeyer, C. Spera, University Köln, Report No. **96-230**.

[16] B. Selman, H. Kautz, B. Cohen, *Noise strategies for local search*, Proceedings of the Eleventh Conference on Artificial Intelligence (AAAI-93), **1993**, pp. 337–343.

[17] B. Selman, H. Levesque, D. Mitchell, *A new method for solving hard satisfiability problems*, Proceedings of the Tenth Conference on Artificial Intelligence (AAAI-92), **1992**, pp. 440–446.

[18] A. Van Gelder, *A satisfiability tester for non-clausual proposition calculus*, Information and Computation, **1988**, pp. 1–21. An early version appeared in LNCS **170**, **1984**, pp. 101–113.

Received 7 February 1996. Revised 10 December 1997