SEPARATING SIGNS IN THE PROPOSITIONAL SATISFIABILITY PROBLEM

E. A. Hirsch

§1. Introduction.

It is well-known that the satisfiability problem for formulas in CNF (in the sequel denoted by SAT) is NP-complete (see, for example, [1]); a polynomial-time algorithm for solving it is unknown. If we consider restrictions on some parameters of the formula, we obtain new problems which are subproblems of general SAT; many such subproblems were also studied. For example, if we restrict the length of clause, we obtain formulas in k–CNF (each clause contains at most k literals). In [11], H. Luckhardt defined classes of formulas in CNF with restriction on the number of occurrences of each literal denoted by CNF–(p,m)(each positive literal occurs at most p times, and each negative literal occurs at most m times). For some of subclasses of CNF that were studied, the satisfiability problem is proved to be solvable in a polynomial time. These are formulas in 2–CNF, CNF–(1,1), Horn's CNF, formulas representing systems of linear equations over $\mathbb{Z}_2$, and some other classes of formulas in CNF (a review of such results is given in [3]). The satisfiability problem for formulas in 3–CNF, CNF–(1,∞), CNF–(1,2) and many others turned out to be still NP-complete (in the sequel we denote these problems by 3-SAT, SAT-(1,∞), SAT-(1,2) etc. respectively). Nevertheless, upper bounds for these subproblems as well as for the satisfiability problem for CNF in general, are being improved starting from [12] (see also [13]) and [2] (see also [4]), where the evident upper bound $p(L)2^N$ for general SAT was reduced for the first time (hereafter p is an appropriate polynomial, L is the length of the input formula, N is the number of variables in it, K is the number of clauses). Table 1 contains recent results in this field including the results of this paper.

Partially supported by INTAS–RFBR grant 95–0095 and by Pro Mathematica foundation (France)

	number of clauses K	length of formula L	number of variables N
CNF	1.2388^K	1.0801^L	2^N
k–CNF			$\psi(k)^N$
3–CNF			1.4963^N
CNF–(1,∞)	1.1939^K	1.0644^L	1.4423^N
k–CNF–(1,∞)			$(\psi(k)+1)^{N/3}$
3–CNF–(1,∞)			1.3566^N
CNF–(1,2)		1.0416^L	1.1299^N
3–CNF–(1,2)	1.1299^K		

TABLE 1. Upper bounds for the satisfiability problem of formulas of different classes up to a polynomial factor $p(L)$. Definition of $\psi(k) \in (1;2)$ is given in §9.

The upper bound $p(L)2^N$ for the satisfiability problem of formulas in general CNF is trivial, the bound $p(L)1.2389^K$ was obtained in [6]. The bound $p(L)\psi(k)^N$ for formulas in k–CNF was published by B. Monien and E. Speckenmeyer in [13] (definition of $\psi(k) \in (1;2)$ is given in §9), the bound $p(L)1.4963^N$ was published by I. Schiermeyer in [14], and also by O. Kullmann in [9]. O. Kullmann and H. Luckhardt proved in [10] that one can check the satisfiability of a formula in CNF in the time $p(L)1.0801^L$. In the same paper they proved that SAT-(1,2) can be solved in the time $p(L)1.0416^L \approx p(L)1.1299^N$, 3-SAT-(1,2) can be solved in the time $p(L)1.1299^K$, and SAT-(1,∞) can be solved in the time $p(L)1.4423^N$ (the last result was published earlier by H. Luckhardt in [11] where formulas in CNF-(1,∞) were introduced). This paper also concerns upper bounds for the satisfiability problem of formulas in CNF–(1,∞) and k–CNF–(1,∞). The upper bound $p(L)1.1939^K$ for SAT-(1,∞) is proved in §7, the bound $p(L)1.0644^L$ is proved in §8, the bounds $p(L)1.3566^N$ for 3-SAT-(1,∞) and $p(L)\psi(k)^{N/3}$ for k-SAT-(1,∞) are proved in §9.

If one reduces the satisfiability problem of a formula in CNF to the satisfiability problem of a formula in some subclass of CNF, usually the size of the formula increases considerably. Thus, no wonder that one does not know examples of applying an algorithm for some subclass with a good upper bound, to improving upper bounds for general SAT. However, in this paper we give an example of such applying, namely, using one of given algorithms one can check the satisfiability of a formula in CNF with $L > 8.49N$ faster than the best known algorithm from [10] (1.0801^L). On the other hand, the increase of the size of the formula obtained after reduction allows to hope for the existence of algorithms that perform inverse reduction of the satisfiability problem for some subclass of CNF to general SAT with considerable reduction of the formula size. However, the author does not know any publications in which such algorithms are used. In this paper we give an example of such a reduction for formulas in CNF–(1,∞)–(+/–); we discuss the definition and the importance of this class for SAT-(1,∞) below.

All algorithms presented in this paper use *the splitting method* originating from the Davis-Putnam procedure ([5]). In short, its key point is the fact that a formula F is satisfiable if and only if at least one of the formulas $F[a]$ and $F[\overline{a}]$ is satisfiable. In a wider sense, the splitting method proceeds as follows. The algorithm constructs a tree by reducing satisfiability of a formula to satisfiability of several formulas obtained from it by assignments. It also performs some *polynomial-time transformations* which do not change satisfiability of the formula and take in each case a polynomial time (the elimination of pure literals, 1-clauses and others). A procedure that performs these transformations is common for all algorithms presented in this paper. This procedure is given in §4 (Procedure *REDUCE*). A general method for the estimation of the size of such a tree (and thus the running time of algorithm) was proposed by O. Kullmann and H. Luckhardt in [10]. We associate with each node of this tree a tuple of positive numbers and a polynomial corresponding to this tuple. The number of leaves in the tree may be estimated using the greatest positive root of the polynomials associated with nodes of the tree. This method as well as its generalization in the case when the leaves of the tree are formulas processed by another algorithm is described in §3.

The main method used in this paper for solving SAT-(1,∞) is reducing this problem to the satisfiability problem of formulas in CNF–(1,∞) that do not include clauses containing simultaneously positive and negative literals (formulas in CNF–(1,∞)–(+/–)). The key point is that the satisfiability problem of formulas that are rather "difficult" to process by the ordinary splitting method turns out to admit easy reduction to the satisfiability problem of formulas of this form. This form is rather special and formulas of this form can be splitted faster. The class of formulas that do not contain clauses with occurrences of positive and negative literals simultaneously is good also because we obtain such formulas when reducing SAT$\rightarrow$SAT-(1,∞). That is why it is easier to construct the inverse reduction.

Splittings that result in disappearing of clauses containing literals that occur only once (1-literals) are very advantageous because after this, in addition, all clauses containing the literal complementary to the removed 1-literal can be removed. It is well-known that an unsatisfiable formula must contain at least one clause consisting of positive literals. In the case of CNF–(1,∞) this fact guarantees the existence of a clause consisting of 1-literals only, this idea is a key point of H. Luckhardt's algorithm from [11] which solves SAT-(1,∞) in the time $p(L)1.4423^N$. We note in addition that if the variable with respect to which we split occurs in another clause together with a 1-literal, this makes the splitting even more advantageous. It turns out that we can guarantee the existence of such clauses in all cases except when the satisfiability of a formula can be reduced in a polynomial time to the satisfiability of a formula in CNF–(1,∞)–(+/–). Generalizing this idea, we can formulate it as the following principle.

Separating signs principle. If for each variable v of a formula F exactly one of the literals v and $\overline{v}$ satisfies a property P, then there are only two possibilities:

1) F contains clauses C and D and a variable v such that the literal v occurs

in C, the literal $\overline{v}$ occurs in D and at least one of literals occurring in D and all literals occurring in C satisfy the property P;

2) the satisfiability of the formula F is equivalent to the satisfiability of its part consisting of the clauses that do not contain simultaneously literals satisfying the property P and not satisfying it.

The separating signs principle is a particular case of the black and white literals principle (see [6],[7]).

Black and white literals principle. If for each variable v of a formula F at most one of the literals v and $\overline{v}$ satisfies a property P, then there are only two possibilities:

1) there is a clause in F that contains at least one literal satisfying the property P and does not contain the negations of the literals satisfying the property P;
2) if we exclude from F all clauses that contain the negations of the literals satisfying the property P, then its satisfiability does not change.

This principle is a development of the generalized sign principle proposed in [10]. Its correctness is due to the fact that if we change the values of literals satisfying the property P to $False$ in any satisfying assignment for a formula F that does not satisfy 1), then this assignment will remain satisfying.

In this paper we give several algorithms for the same class of formulas CNF–(1,∞). In order to solve SAT-(1,∞) more efficiently, one should run them in parallel. On the other hand, it is clear that the guaranteed upper bound on the complexity of an algorithm whose running time is estimated as a function of L is less than such bounds for other algorithms in case when the input formula is short enough. For example, the algorithm solving SAT-(1,∞) in the time $p(L)1.0644^L \approx p(L)2^{0.08992L}$ is better than the algorithm solving SAT-(1,∞) in the time $p(L)1.4423^N \approx p(L)2^{0.5284N}$ in case $L < 0.5284/0.08992 \approx 5.88N$. The same is true for other parameters of the input formula. Respectively, the algorithm solving SAT-(1,∞) in the time $p(L)1.1939^K$ is given in §7 (Algorithm for formulas with small number of clauses), the algorithm solving SAT-(1,∞) in the time $p(L)1.0644^L$ is given in §8 (Algorithm for short formulas), the algorithm solving 3-SAT-(1,∞) in the time $p(L)1.3566^N$ is given in §9 (Algorithm for formulas with small number of variables).

The paper is organized as follows:

§2. Basic definitions.

Let V be a set of Boolean variables. We denote by $\overline{v}$ the negation of a variable v. For an arbitrary set U we use the notation $\overline{U} = \{\overline{u} \,|\, u \in U\}$. *Literals* are elements of the set $W = V \cup \overline{V}$. A literal is *positive* if it is a variable. A literal is *negative* if it is the negation of a variable. If w denotes a negative literal $\overline{v}$, then $\overline{w}$ denotes the variable v. We identify a clause $w_1 \vee w_2 \vee \cdots \vee w_s$ with the set of literals $\{w_1, w_2, \ldots, w_s\} \subset W$. No clause contains any variable together with its negation, i.e. $w_i \neq \overline{w}_j$ if $i \neq j$. We identify the empty clause with $False$. A clause is said to be *positive* if it contains only positive literals. A clause is said to be *negative* if it contains only negative literals. *Formula in CNF* (or *CNF-formula*) is a finite set of clauses interpreted as their conjunction. The empty formula is interpreted as $True$. *The length of a clause* is its cardinality as a set, *the length of a formula* is the sum of cardinalities of all its clauses. A clause of the length exactly k is called a *k-clause*. If we say that *a variable* $v \in V$ occurs in a clause or in a formula, we mean that this clause or this formula contains the literal v or the literal $\overline{v}$. On the contrary, if we say that *a literal* v occurs in a clause or in a formula, we mean that it contains this literal and not its negation. In the following F denotes a formula in CNF with variables from V.

CNF–(1,s) is a class of formulas in CNF defined by the following condition: a formula F belongs to this class if each variable $v \in V$ occurs in F at most once as v and at most s times as $\overline{v}$. (Here $s \in \mathbb{N} \cup \{+\infty\}$.)

k–CNF–(1,s) is a class of formulas in CNF defined by the condition: a formula F belongs to this class if it belongs to the class CNF–(1,s) and each its clause contains at most k literals. (Here $k, s \in \mathbb{N} \cup \{+\infty\}$. If $k = +\infty$, the class k–CNF–(1,s) coincides with the class CNF–(1,s).)

CNF–(1,s)–(+/–) is a class of formulas in CNF defined by the condition: a formula F belongs to this class if it belongs to the class CNF–(1,s) and contains only positive and negative clauses.

k–CNF–(1,s)–(+/–) is a class of formulas in CNF defined by the condition: a formula F belongs to this class if it belongs to the class k–CNF–(1,s) and contains only positive and negative clauses.

A variable v is a *(i,j)-variable* if the literal v occurs in F exactly i times, and the literal $\overline{v}$ occurs in F exactly j times. A variable v is a *(i,j+)-variable* if the literal v occurs in F exactly i times, and the literal $\overline{v}$ occurs in F at least j times.

Assume that we have a literal l and clauses C_1 and C_2 such that $l \in C_1$, $\overline{l} \in C_2$ and no other literal $l' \in C_1$ satisfies $\overline{l}' \in C_2$. In this case the clause $C_1 \cup C_2 \setminus \{l, \overline{l}\}$ is called the *l-resolvent* of the clauses C_1 and C_2.

Let $l \in W$. The formula $DP_l(F)$ is obtained by

- adding in F all l-resolvents and
- removing from F all clauses containing l or $\overline{l}$.

An assignment is a finite subset of W that does not contain simultaneously a variable and its negation.

We define the formula $F[S]$ obtained from a CNF-formula F by assignment $S = \{v_1, \ldots, v_s\}$ as the formula obtained from F by deleting all clauses containing the literals v_i and deleting the literals $\overline{v}_i$ from the other clauses. For short, we write $F[v_1, \ldots, v_s]$ instead of $F[\{v_1, \ldots, v_s\}]$. Formula F is *satisfiable* if there exists an assignment S such that $F[S] = True$. Formulas F and G are *equi-satisfiable* if they are both satisfiable, or they are both not satisfiable.

A branching tree is a tree whose nodes are labelled with CNF-formulas such that if a node is labelled with a CNF-formula F, then its sons are labelled with the formulas $F[I_1]$, $F[I_2]$, ... , $F[I_s]$ for some assignments I_1, I_2, ... , I_s.

Example of a branching tree is given in Fig. 1.

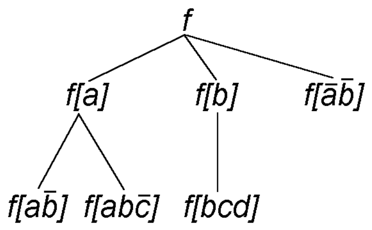


Fig. 1

To estimate the number of leaves in such a tree (and thus the complexity of an algorithm) we use auxiliary results concerning *branching tuples* introduced in [10]. These results are presented in §3. We introduce below some necessary terms.

Let T be a branching tree whose nodes are labelled with objects having some characteristic of their complexity: with each object F we associate a non-negative number $\mu(F)$ (*complexity of* F or *measure of complexity of* F) in such a way that for every node the complexity of the object labelling it is greater than the complexity of each object labelling its sons. In our case objects are CNF-formulas, and μ is one of the following three functions:

1) $\mathfrak{K}(F)$ is the number of clauses in F;
2) $\mathfrak{L}(F)$ is the length of F;
3) $\mathfrak{N}(F)$ is the number of variables in F.

For simplicity, we assume that each internal node of our branching tree is of degree at least two (we do not use trees that do not satisfy this condition).

The notions of the branching tuple and the branching number defined below assume the choice of a measure of a formula complexity. That is why we use these notions with comments "...with respect to the measure of complexity $\mathfrak{K}$ ($\mathfrak{L}$, $\mathfrak{N}$)" when it is not clear from the context. Nevertheless, when proving general results concerning branching numbers, we do not fix any concrete measure of complexity. It is even immaterial for this purpose that objects are CNF-formulas.

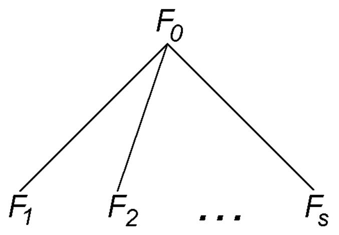


FIG. 2

Consider an arbitrary internal node of our tree labelled with an object F_0. Let its sons be labelled with objects F_1, F_2, ... , F_s (see Fig. 2). *The branching tuple* in node F_0 is an s-tuple $(t_1, \ldots, t_s)$, where $s \geqslant 2$ and each of t_i is a positive number that does not exceed $\mu(F_0) - \mu(F_i)$.

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

$$(\star) \qquad \mathfrak{h}_{\vec{t}}(x) = 1 - \sum_{i=1}^{s} x^{-t_i}.$$

We assume $\tau(\emptyset) = 1$ if the node is a leaf.

Remark. The characteristic polynomial $\mathfrak{h}_{\vec{t}}(x)$ is a monotone function of x on the interval $(0, +\infty)$. Moreover, $\mathfrak{h}_{\vec{t}}(0) = -\infty$, $\mathfrak{h}_{\vec{t}}(+\infty) = +1$. Hence, the equation $\mathfrak{h}_{\vec{t}}(x) = 0$ has the unique solution on the interval $(0, +\infty)$.

The greatest of the branching numbers $\tau(\vec{t})$ in all nodes of a tree T is called *the branching number of a tree T*. We denote it by $\tau_{max,T}$ (or simply τ_{max}).

Now we present simple properties of branching numbers. In the sequel we often use them without explicit mentioning.

Properties of branching numbers.

1) *A branching number is strictly greater than one.*
2) *A branching number does not change if we permute the components of the branching tuple.*
3) *If each coordinate of a branching tuple $\vec{t_1}$ does not exceed the corresponding coordinate of another tuple $\vec{t_2}$ of the same dimension, then the branching number associated with the tuple t_1 does not exceed the branching number of the tuple t_2.*
4) *Let a, b, c, d be natural numbers. If $|a - b| < |c - d|$ and $a + b = c + d$, then $\tau((a, b)) < \tau((c, d))$.*

§3. Estimation of the number of leaves in a branching tree

Lemma 3.1 (O. Kullmann, H. Luckhardt, [10]). *The number of leaves in a branching tree with an object F_0 in the root does not exceed $(\tau_{max})^{\mu(F_0)}$.*

The proof of Lemma 3.1 is given in [10]. In the sequel we need also the following generalized version of this lemma (Lemma 3.1 is a corollary of this generalization).

Lemma 3.2. *Let $f(m) = \alpha\lambda^m$, where $\lambda > 1$, $\alpha \geqslant 0$. Let G_l denote the object labelling a leaf l of the tree T, F_0 denote the object labelling the root of the tree T. Then*

$$\sum_{l - T} f(\mu(G_l)) \leqslant \alpha(\max(\tau_{max}, \lambda))^{\mu(F_0)}.$$

Proof is the induction on the construction of the tree.

Base. The tree consisting of the unique node. In this case $\tau_{max} = 1$, $\alpha(\max(\lambda, \tau_{max}))^{\mu(F_0)} = \alpha\lambda^{\mu(F_0)} = f(\mu(F_0))$.

Step. Consider the tree T presented in Fig. 3.

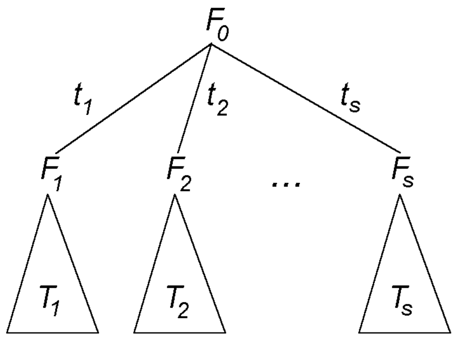


FIG. 3

Let $\vec{t} = (t_1, t_2, \ldots, t_s)$ be the branching tuple in its root.

$$\begin{aligned}
&\sum_{l \text{ is a leaf of } T} f(\mu(G_l)) = \\
&= \sum_{j=1}^{s} \left(\sum_{l \text{ is a leaf of } T_j} f(\mu(G_l)) \right) \\
&\leqslant \alpha \sum_{j=1}^{s} (\max(\lambda, \tau_{max,T_j}))^{\mu(F_j)} \quad \text{(by induction hypothesis)} \\
&\leqslant \alpha \sum_{j=1}^{s} (\max(\lambda, \tau_{max,T_j}))^{\mu(F_0)-t_j} \quad \text{(by definition of the tuple } \vec{t}\text{)} \\
&\leqslant \alpha \sum_{j=1}^{s} (\max(\lambda, \tau_{max,T}))^{\mu(F_0)-t_j} \quad \text{(since } \tau_{max,T_j} \leqslant \tau_{max,T}\text{)} \\
&= \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)} \cdot \sum_{j=1}^{s} (\max(\lambda, \tau_{max,T}))^{-t_j} \\
&= \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)} (1 - \mathfrak{h}_{\vec{t}}(\max(\lambda, \tau_{max,T}))) \text{ (by definition of } \mathfrak{h}_{\vec{t}}\text{)} \\
&\leqslant \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)} \quad \text{(by monotonicity of } \mathfrak{h}_{\vec{t}}\text{)}
\end{aligned}$$

□

§4. POLYNOMIAL-TIME TRANSFORMATIONS.

All algorithms of this paper split a formula and then perform polynomial-time transformations described in Procedure *REDUCE* given in this section. The result is eliminating from the formula pure literals (literals whose negations do not occur in the formula), 1-clauses, etc. In particular, at step (R4) we eliminate blocked clauses. The notion of a blocked clause was introduced by O. Kullmann in [8]. A blocked clause is a clause D containing a literal whose negation does not occur in clauses not containing the negation of another literal of D. In our case, we eliminate blocked clauses of a special form, namely, we eliminate each clause D such that in the formula F there is a clause C and variables a and b satisfying the conditions

- $\{a, b\} \subset C$,
- $\{\overline{a}, \overline{b}\} \subset D$.

By analogy with blocked clauses, we eliminate also each clause E such that in F there are clauses C and D and variables a, b and c satisfying the conditions

- $\{a, b\} \subset C$,
- $\{\overline{a}, c\} \subset D$,
- $\{\overline{b}, \overline{c}\} \subset E$.

We show below (Lemma 4.1) that none of these transformations change the satisfiability of the formula. They result in eliminating "simple" parts of the formula. After these transformations the formula is "in a general position", i.e. each variable occurs in it at least three times, each clause contains at least two literals, etc.

Procedure *REDUCE*.

Input:

- a number $k \in \mathbb{N} \cup \{+\infty\}$,
- a formula F in k–CNF–(1,∞).

Output: a formula in k–CNF–(1,∞) equi-satisfiable to the input formula F.

Method:

(R1) Eliminate 1-clauses: if there is a clause containing the only literal l, change F to $F[l]$.

(R2) Eliminate pure literals: if a literal l does not occur in F, change F to $F[\overline{l}]$.

(R3) If there is a variable $d \in V$ such that
- $DP_d(F)$ is in k–CNF–(1,∞),
- the number of clauses in $DP_d(F)$ does not exceed the number of clauses in F,
- the length of $DP_d(F)$ does not exceed the length of F,

assume $F = DP_d(F)$.

(R4) Eliminate blocked clauses: if there are clauses C and D and variables a and b such that $\{a, b\} \subset C$ and $\{\overline{a}, \overline{b}\} \subset D$, then eliminate from F the clause D.

(R5) If there are clauses C, D and E and variables a, b and c such that $\{a, b\} \subset C$, $\{\overline{a}, c\} \subset D$ and $\{\overline{b}, \overline{c}\} \subset E$, then eliminate from F the clause E.

(R6) If there are clauses $C, D \in F$ and a literal $l \in C \cap \overline{D}$ such that $C \backslash \{l\} \subset D \backslash \{\overline{l}\}$, then assume $F = (F \backslash \{C\}) \cup (\{D\} \backslash \{l\})$.

(R7) If the formula F has been changed at steps (R1)–(R6), then go to step (R1). Otherwise, output F.

$\square$

Lemma 4.1. *Given a formula F in k–CNF–$(1,\infty)$, Procedure REDUCE stops in a time polynomial on the length of the input formula and outputs a formula in k–CNF–$(1,\infty)$ equi-satisfiable to F.*

Proof. We repeat the steps (R1)–(R6) at most $L + N$ times, because no one of them increases the number of variables in the formula and its length, and during each iteration at least one of these steps decreases at least one of these parameters. Since each step takes a polynomial time, the same holds for the whole Procedure *REDUCE*. It is evident that the output formula is in k–CNF–$(1,\infty)$.

We now prove that each step does not change the satisfiability of the formula F.

(R1) Elimination of 1-clauses: if a clause contains only one literal, then every satisfying assignment contains this literal.

(R2) Elimination of pure literals: if a literal l does not occur in F, then in any assignment satisfying F we may replace l by $\overline{l}$ obtaining again a satisfying assignment.

(R3) Let us show that the formulas F and $DP_d(F)$ are equi-satisfiable. Consider a satisfying assignment for F. Since for each pair of clauses whose resolvent has been added, at least one clause is satisfied even if we exclude the occurrence of the variable d, also the resolvent is satisfied. Conversely, consider an assignment I satisfying $DP_d(F)$. We now show that we may add to it d or $\overline{d}$ obtaining a satisfying assignment for F. Assume that F has two clauses D_1 and D_2, such that D_1 contains the literal d, D_2 contains the literal $\overline{d}$, and I satisfies neither $D_1 \setminus d$ nor $D_2 \setminus \overline{d}$. Then I also does not satisfy the resolvent of D_1 and D_2. But this resolvent occurs in $DP_d(F)$, and this contradicts the assumption that I satisfies $DP_d(F)$.

(R4) Elimination of blocked clauses. Assume that the formula F was unsatisfiable and it has become satisfiable. Consider a satisfying assignment. It does not satisfy the clause D, i.e. it contains the literals a and b. Let us replace in it a by $\overline{a}$. Now the clauses C and D are satisfied and the satisfiability of the other clauses has not been changed (since F is a formula in CNF–$(1,\infty)$, the literal a occurs only in the clause C). Thus we have obtained a satisfying assignment. The converse statement is evident.

(R5) Assume that the formula F was unsatisfiable and it has become satisfiable. Consider a satisfying assignment. It does not satisfy the clause E, i.e. it contains the literals b and c. If it contains also the literal a, replace in it b by $\overline{b}$. Otherwise replace c by $\overline{c}$. Now the clauses C, D and E are satisfied, and the satisfiability of the other clauses has not been changed (the literals a, b and c occur in no clauses except C, D and E). Thus we have obtained a satisfying assignment. The converse statement is evident.

(R6) Let F_1 be the formula F_1 before this step, $F_2 = (F_1\backslash\{C\}) \cup (\{D\}\backslash\{l\})$. Note that the formula $DP_l(F_1)$ is equi-satisfiable to the formula $DP_l(F_2)$. Hence, F_1 and F_2 are equi-satisfiable too.

□

We now give a list of some useful properties of the formula $REDUCE(k, F)$.

Lemma 4.2.

1) *If $k = +\infty$, then each variable v in $REDUCE(k, F)$ occurs exactly once as*

the literal v and at least twice as the literal $\overline{v}$.

2) *The formula $REDUCE(k, F)$ does not contain clauses C and D such that $\exists a_1, a_2 \in V\ ((\{a_1, a_2\} \subset C)\ \&\ (\{\overline{a}_1, \overline{a}_2\} \subset D))$.*
3) *The formula $REDUCE(k, F)$ does not contain clauses C, D and E such that $\exists a_1, a_2, b \in V\ ((\{a_1, a_2\} \subset C)\ \&\ (\{\overline{a}_1, b\} \subset D)\ \&\ (\{\overline{a}_2, \overline{b}\} \subset E))$.*
4) *If the formula $REDUCE(k, F)$ contains a positive clause $C = \{a_1, a_2, \dots, a_s\}$, then it contains also two clauses containing $\overline{a}_i$ for each $i \in \overline{1, s}$, and all these clauses are distinct.*
5) *If the formula $REDUCE(k, F)$ contains a positive clause $C = \{a_1, a_2, \dots, a_s\}$ and a clause D containing the literal $\overline{a}_1$ and at least one positive literal a_{s+1}, then the formula $REDUCE(k, F)$ contains at least two clauses containing $\overline{a}_i$ for each $i \in \overline{2, s+1}$, and all these clauses are distinct.*

Proof.

1) All other variables are eliminated at steps (R2)–(R3) of Procedure $REDUCE$.
2) Such clauses are eliminated at step (R4) of Procedure $REDUCE$.
3) Such clauses are eliminated at step (R5) of Procedure $REDUCE$.
4) Follows from 1) and 2).
5) Follows from 1), 2) and 3).

□

§5. Transformation of a formula in CNF–(1,∞) into formulas in CNF–(1,∞)–(+/–).

Three main algorithms of this paper are very similar. These algorithms base on Algorithm 5.1 described in this section. The key point of the method is the following. Suppose that the input formula in CNF–(1,∞) contains a clause $C = \{a_1, a_2, \dots, a_s\}$ and a clause D satisfying the following conditions

- C is a positive clause,
- D contains the literal $\overline{a}_1$,
- D contains at least one positive literal.

Then splitting with respect to the variable a_1 is very advantageous, because both assignments $\{a_1\}$ and $\{\overline{a}_1\}$ imply disappearing of the only occurrence of at least one literal. If there is no such pair of clauses in the input formula, then by separating signs principle one may keep in the formula only positive and negative clauses without changing its satisfiability. The satisfiability problem of the resulting formula in CNF–(1,∞)–(+/–) is to be solved by different algorithms called by Algorithm 5.1. To obtain an algorithm that works efficiently for formulas with small number of clauses, we apply Algorithm 7.1 to formulas in CNF–(1,∞)–(+/–); we use Algorithm 8.1 for short formulas; and we use Algorithm 9.1 for formulas with small number of variables. Some of these algorithms work independently; splitting in others generates formulas with clauses containing both positive and negative literals, these formulas are processed again by Procedure $SPLIT$ which is the main part of Algorithm 5.1.

Procedure $SPLIT$**.**

Input:

- a number $k \in \mathbb{N} \cup \{+\infty\}$,

- a formula F in k–CNF–(1,∞) processed by Procedure *REDUCE*,
- an algorithm $\mathcal{M}$ solving the satisfiability problem of formulas in k–CNF–(1,∞)–(+/–).▌

Output: "Satisfiable", if F is satisfiable; "Unsatisfiable" otherwise.

Method:

(S1) If F contains no positive clauses, assume $F = True$.

(S2) If for each positive clause C of the formula F and each clause D of the formula F containing a positive literal, the condition $\overline{C} \cap D = \emptyset$ holds, then assume $G = F[\{\overline{v} \in \overline{V} \mid \exists C \in F\ \exists u \in V\ (\{\overline{v}, u\} \subset C)\}]$ and output the result of the call $\mathcal{M}(k, G)$.

(S3) Choose in F a clause $C = \{a_1, a_2, \ldots, a_s\}$ and a clause D satisfying the following conditions:
- C is a positive clause,
- D contains the literal $\overline{a}_1$,
- at least one positive literal b occurs in D.

Construct formulas

$$F_1 = REDUCE(k, F[a_1]),$$
$$F_2 = REDUCE(k, F[\overline{a}_1]).$$

If Procedure *SPLIT* outputs "Satisfiable" for at least one of these, then output "Satisfiable"; otherwise output "Unsatisfiable".

□

Lemma 5.1. *Procedure SPLIT always stops outputting an answer. It solves correctly the satisfiability problem of the input formula F in k–CNF–(1,∞) if the latter is processed by Procedure REDUCE.*

Proof. By Lemma 4.1 Procedure *REDUCE* does not change the satisfiability of a formula. If the formula does not contain positive clauses, then it is satisfied by the assignment $\{\overline{v} \mid v \in V\}$ consisting of only negative literals. Thus, the step (S1) is correct too. Note that the formula satisfying the condition of this step is replaced by $True$ and is immediately passed at step (S2) to the algorithm $\mathcal{M}$.

Let P_+ be the set of all positive clauses of the formula F, P_- be the set of all its negative clauses, $M = F \backslash (P_+ \cup P_-)$. If the condition of the step (S2) is satisfied, then there is no variables occurring in P_+ among the variables whose negations occur in M. Hence, if an assignment I satisfies F, then also the assignment I' obtained from I by replacing all negative literals occurring in M with the corresponding positive literals satisfies F. Really, since these variables occur without negation neither in P_+ nor in P_-, only clauses from M can become false after this replacing. But each of these clauses contains the negative literal that we have just included into the assignment I'. Thus, if F is satisfiable, then $G = F[\{\overline{v} \in \overline{V} \mid \exists C \in F\ \exists u \in V\ (\{\overline{v}, u\} \subset C)\}] = F[\{\overline{v} \in \overline{V} \mid \exists C \in M(\overline{v} \in C))\}]$ is satisfiable too. The inverse statement is trivial. Moreover, G is a formula in k–CNF–(1,∞)–(+/–). Hence, the step (S2) is correct too.

If the condition of the step (S2) is not satisfied, then the condition of the step (S3) must be satisfied. At step (S3), the satisfiability of F is reduced to the satisfiability

of the formulas
$$F_1 = REDUCE(k, F[a_1]),$$
$$F_2 = REDUCE(k, F[\overline{a}_1]),$$
and each of these formulas has less variables than F. Since none of the steps of this algorithm increases the number of variables in F, this recursion must finish in a finite time.
□

Algorithm 5.1

Input:

- a number $k \in \mathbb{N} \cup \{+\infty\}$,
- a formula F in k–CNF–(1,∞),
- an algorithm $\mathcal{M}$ solving the satisfiability problem of formulas in k–CNF–(1,∞)–(+/–).

Output: "Satisfiable", if F is satisfiable; "Unsatisfiable" otherwise.

Method:

(M1) Assume $F = REDUCE(k, F)$.
(M2) Output the result of the call $SPLIT(k, F, \mathcal{M})$.

□

Correctness of Algorithm 5.1 follows from Lemma 4.1 and Lemma 5.1. □

Running time of Algorithm 5.1

We now construct in a natural way the tree $T_{5.1}$ of formulas splitted by this algorithm. The internal nodes of this tree are labelled with the formulas splitted at step (S3), and the leaves are labelled with the formulas that Algorithm 5.1 processes without splitting but passing them to the algorithm $\mathcal{M}$.

To obtain from Algorithm 5.1 an algorithm that solves the satisfiability problem, we must present an algorithm $\mathcal{M}$ solving the satisfiability problem of formulas in k–CNF–(1,∞)–(+/–). We describe such algorithms $\mathcal{M}_K$, $\mathcal{M}_L$, $\mathcal{M}_N$ in §§7–9. To minimize the running time of Algorithm 5.1, one should choose $\mathcal{M}$ depending on the type of the input formula. Respectively, we obtain upper bounds with respect to different parameters: if we use the algorithm $\mathcal{M}_K$, Algorithm 5.1 takes the time $p(L)1.1939^K$ for formulas in CNF–(1,∞); if we use $\mathcal{M}_L$, Algorithm 5.1 takes the time $p(L)1.0644^L$ for formulas in CNF–(1,∞), if we use $\mathcal{M}_N$, Algorithm 5.1 takes the time $p(L)1.3566^N$ for formulas in CNF–(1,∞). More exact formulations are given in §§7–9.

The proof of each of these bounds is divided into two steps. At first step (Lemmas 6.1–6.3) we prove an upper bound for the branching number of the tree $T_{5.1}$, that is in some sense an upper bound for the running time of Algorithm 5.1. At the second step we describe the algorithms $\mathcal{M}_K$, $\mathcal{M}_L$, $\mathcal{M}_N$ and show that each of them processes formulas in 3–CNF–(1,∞)–(+/–) rather fast. Using Lemma 3.2, we unite these bounds in Theorems 7.1–9.1 to prove the general bound on the running time of the whole Algorithm 5.1 which applies different algorithms $\mathcal{M}$.
□

§6. Estimation of the branching number for the algorithm of reducing formulas in CNF–(1,∞) to CNF–(1,∞)–(+/–).

To estimate the running time of Algorithm 5.1, we need upper bounds on the branching number of the tree that it constructs, i.e. the tree $T_{5.1}$ defined in §5. Below we give such upper bounds for different measures of a formula complexity (recall that $\mathfrak{K}$ is the number of clauses in a formula, $\mathfrak{L}$ is the length of a formula, $\mathfrak{N}$ is the number of variables in a formula). In this section C, D, s, a_1, a_2, ... , a_s, b, F_1, F_2 denote the same objects as in Algorithm 5.1.

Lemma 6.1. *The branching number of the tree* $T_{5.1}$ *with respect to the measure of complexity* $\mathfrak{K}$ *does not exceed* 1.1939.

Proof. We consider two cases.

Case 1. $s = 2$.

Note that the number of clauses in the formula F_1 is less than the number of clauses in the formula F by at least three. Namely, besides the clause C which is eliminated immediately, Procedure *REDUCE* at step (R2) eliminates all clauses containing $\overline{a}_2$ (since after elimination of the clause C, the only occurrence of the literal a_2 disappears from F). By Lemma 4.2, there are at least two such clauses.

The number of clauses in the formula F_2 is less than the number of clauses in the formula F by at least five. First, the clauses containing $\overline{a}_1$ are eliminated immediately. By Lemma 4.2, there are at least two such clauses. Second, at step (R1) we delete the clause obtained from C by deleting a_1. Moreover, since the only occurrence of the literal b disappears, all clauses containing $\overline{b}$ will be deleted at step (R2). Since by Lemma 4.2 there are at least two such clauses, we have that at least five clauses will be deleted.

Thus, in this case the node has the branching tuple $(3, 5)$ and the branching number $\tau((3, 5)) < 1.1939$.

Case 2. $s \geqslant 3$.

Note that the number of clauses in the formula F_1 is less than the number of clauses in the formula F by at least five. Namely, besides the clause C eliminated immediately, Procedure *REDUCE* at step (R2) eliminates all clauses containing $\overline{a}_2$ and $\overline{a}_3$ (since after elimination of the clause C, the only occurrences of the literals a_2 and a_3 disappear from F). By Lemma 4.2, there are at least four such clauses.

The number of clauses in the formula F_2 is less than the number of clauses in the formula F by at least three. First, the clauses containing $\overline{a}_1$ are eliminated immediately. By Lemma 4.2, there are at least two such clauses. Second, since the only occurrence of the literal b disappears, we eliminate at step (R2) all clauses containing $\overline{b}$. Since by Lemma 4.2 there are at least two such clauses, and D is not such clause, we obtain that at least three clauses will be eliminated.

Thus, in this case the node also has the branching tuple $(3, 5)$ and the branching number $\tau((3, 5)) < 1.1939$.

□

Lemma 6.2. *The branching number of the tree* $T_{5.1}$ *with respect to the measure of complexity* $\mathfrak{L}$ *does not exceed* 1.0644.

Proof.

We denote $\Delta_1 = \mathfrak{L}(F) - \mathfrak{L}(F_1)$, $\Delta_2 = \mathfrak{L}(F) - \mathfrak{L}(F_2)$, $\Delta = \Delta_1 + \Delta_2$. We now consider the following cases.

Case 1. The literal $\overline{a}_1$ occurs in the formula F at least three times.

Case 1.1. $s = 2$.

First, in both F_1 and in F_2 all occurrences of the variables a_1 and a_2 disappear (in comparison with F). This is a result of eliminating the variable a_1 and subsequent transformations at steps (R1) and (R2) of Procedure *REDUCE*.

Hence $\Delta_1, \Delta_2 \geqslant 7$, $\Delta \geqslant 14$.

Each of the clauses containing $\overline{a}_1$ or $\overline{a}_2$ (the total number of such clauses is at least five by Lemma 4.2 and since the number of occurrences of $\overline{a}_1$ in the formula F is greater than two), except the literals $\overline{a}_1$ and $\overline{a}_2$ already taken into account, contributes at least two to Δ. Namely, each of the 2-clauses contributes 1 to Δ_1 and Δ_2 because of transformations (R1) and (R2); each of the 3-clauses contributes 2 to Δ, since when a 3-clause disappears two occurrences are eliminated at once (not including $\overline{a}_1$ or $\overline{a}_2$). Thus, Δ increases by at least ten, and each of Δ_1 and Δ_2 increases by at least two.

Among all branching tuples of the form (Δ_1, Δ_2) such that $\Delta_1 + \Delta_2 = \Delta \geqslant 24$ and $\Delta_1, \Delta_2 \geqslant 9$, the greatest branching number corresponds to the tuple (9,15). This number is $\tau((9,15)) < 1.0609$.

Case 1.2. $s \geqslant 3$.

First, all occurrences of the variables a_1, a_2 and a_3 disappear in F_1. This is a result of eliminating the variable a_1 and subsequent transformations at step (R2) of Procedure *REDUCE*, at least ten occurrences in total. In F_2 all occurrences of the variable a_1 disappear, i.e. at least four occurrences.

Hence $\Delta_1 \geqslant 10, \Delta_2 \geqslant 4$, $\Delta \geqslant 14$.

The clauses containing $\overline{a}_2$ and $\overline{a}_3$ are eliminated when passing from F to F_1 at step (R2) of Procedure *REDUCE*. By Lemma 4.2, there are at least four such clauses. Each of them contains at least one literal except $\overline{a}_2$ and $\overline{a}_3$. This contributes at least four to Δ_1 (and to Δ respectively).

Each of the clauses containing $\overline{a}_1$ (there are at least three such clauses) contributes at least two to Δ except the literal $\overline{a}_1$ itself. Namely, each of the 2-clauses contributes 1 to Δ_1 and to Δ_2 because of transformations (R1) and (R2); each of the 3-clauses contributes 2 to Δ, since when a 3-clause disappears, two occurrences (not including $\overline{a}_1$) are eliminated. Thus, Δ increases by at least six, and Δ_2 increases by at least three.

Thus we have $\Delta \geqslant 24$, $\Delta_1 \geqslant 14$, $\Delta_2 \geqslant 7$.

Among all branching tuples of the form (Δ_1, Δ_2) such that $\Delta_1 + \Delta_2 = \Delta \geqslant 24$ and $\Delta_1, \Delta_2 \geqslant 7$, the greatest branching number corresponds to the tuple (7,17). This number is $\tau((7,17)) < 1.0637$.

Note that in Case 1 we do not use the fact that one of the clauses containing $\overline{a}_1$ contains the positive literal b. We shall use it when we apply this argument once more in the proof of Theorem 8.1.

Case 2. The literal $\overline{a}_1$ occurs in the formula F exactly twice.

Case 2.1. $s = 2$.

First, all occurrences of the variables a_1 and a_2 (at least six occurrences in total) disappear in F_1, and all occurrences of the variables a_1, a_2 and b (at least nine occurrences in total) disappear in F_2. This is a result of eliminating the variable a_1 and subsequent transformations at steps (R1) and (R2) of Procedure *REDUCE*.

Thus we have $\Delta_1 \geqslant 6$, $\Delta_2 \geqslant 9$, $\Delta \geqslant 15$.

Each of the clauses containing $\overline{a}_2$ (there are at least three such clauses) except the literal $\overline{a}_2$ itself, contributes to Δ at least two. Namely, each of the 2-clauses contributes 1 to Δ_1 and Δ_2 because of transformations (R1) and (R2); each of the 3-clauses contributes 2 to Δ, since when a 3-clause disappears, two occurrences (not including $\overline{a}_2$) are eliminated at once. Thus, Δ increases by at least four, and Δ_2 increases by at least two.

Note that among the clauses containing the literal $\overline{b}$, there is at least one clause that contains neither the variable a_1 (there are only three clauses containing the variable a_1, and one of these is positive), nor the variable a_2 (by Lemma 4.2). It contains at least one more literal, at it will be eliminated together with the whole clause at step (R2) when passing from F to F_2.

Thus, at the moment we have that $\Delta_1 \geqslant 8$, $\Delta_2 \geqslant 10$, $\Delta \geqslant 20$.

The literals occurring in the clauses containing the literal $\overline{a}_1$ are also eliminated when passing from F to F_2. By Lemma 4.2, in these clauses we have already counted only the literals corresponding to the variables a_1 and b.

Case 2.1.1. There is a 2-clause among the clauses containing $\overline{a}_1$.

Note that in this case none of the clauses containing $\overline{a}_1$ contains $\overline{b}$ (otherwise, they would be both eliminated earlier at step (R6)), hence one additional literal is eliminated (or two literals if one of them contains more than two literals). Moreover, this means that besides the clause containing $\overline{b}$ that we have already counted, we should take into account one more clause, and it contains at least one more literal except $\overline{b}$. Thus, if one of these clauses contains more than two literals, then $\Delta \geqslant 23$. Otherwise, we can take into account that when passing from F to F_1, the clause $\{\overline{a}_1, b\}$ becomes a 1-clause, and thus all occurrences of the variable b are eliminated, hence, in this case we have $\Delta \geqslant 25$.

Case 2.1.2. Each of the clauses containing $\overline{a}_1$ contains at least three literals.

In this case, these clauses contribute to Δ at least two more occurrences, i.e. $\Delta \geqslant 22$.

Case 2.1.2.1. None of the clauses containing $\overline{a}_1$ contains $\overline{b}$, or one of these clauses contains at least four literals.

In this case, we can take into account one more occurrence, i.e. $\Delta \geqslant 23$.

Case 2.1.2.2. A clause containing $\overline{b}$ but not containing $\overline{a}_1$ contains more than two literals, or there are at least two such clauses.

In this case, we can also take into account one more occurrence, i.e. $\Delta \geqslant 23$.

Case 2.1.2.3. The formula F takes the form

$$(\overline{a}_1 \vee b \vee l_1) \,\&\, (\overline{a}_1 \vee \overline{b} \vee l_2) \,\&\, (\overline{b} \vee l_3) \,\&\, F',$$

where F' does not contain occurrences of b.

In this case, if l_1 is a negative literal, then when passing to F_2 all these clauses (more exactly, the clauses obtained from them by deleting a_1) will be eliminated at step (R3) of Procedure *REDUCE* (by operation DP_b), i.e. $\Delta \geqslant 25$. If l_1 is a positive literal, then when passing to F_2, all occurrences of this variable will be eliminated; by Lemma 4.2, it does not occur in the clauses containing b. Assume that we have already taken into account all these occurrences; this means that all of them are in the 2-clauses containing $\overline{a}_2$. But all such clauses are identical (and the formula is a set!) Thus, we have not taken into account at least one occurrence of $\overline{l}_1$, i.e. $\Delta \geqslant 23$.

Among all branching tuples of the form (Δ_1, Δ_2) such that $\Delta_1 + \Delta_2 \geqslant 23$ and $\Delta_1, \Delta_2 \geqslant 8$, the greatest branching number corresponds to the tuple (8,15). This number is $\tau((8,15)) < 1.0644$.

Case 2.2. $s \geqslant 3$.

As in Case 2.1, we now prove that $\Delta \geqslant 23$, $\Delta_1, \Delta_2 \geqslant 8$.

First, all occurrences of the variables a_1, a_2 and a_3 (at least nine occurrences in total) disappear in F_1 in comparison with F, and all occurrences of the variables a_1 and b (at least six occurrences in total) disappear in F_2. This is a result of eliminating the variable a_1 and subsequent transformations at step (R2) of Procedure *REDUCE*.

Hence $\Delta_1 \geqslant 9$, $\Delta_2 \geqslant 6$, $\Delta \geqslant 15$.

In addition, when passing from F to F_1, all clauses containing $\overline{a}_2$ or $\overline{a}_3$ are eliminated, i.e. at least four occurrences of the literals different from $\overline{a}_2$ and $\overline{a}_3$.

Note that among the clauses containing the literal $\overline{b}$ there is at least one clause that contains neither the variable a_1 (there are only three clauses containing the variable a_1, and one of them is positive), nor the variables a_2 and a_3 (by Lemma 4.2). It contains at least one more literal besides $\overline{b}$, and it will be eliminated together with the whole clause at step (R2) when passing from F to F_2.

Note that $\Delta_2 \geqslant 8$, since $\Delta_2 = 7$ only if both clauses containing $\overline{a}_1$ are 2-clauses, but in this case there are at least two clauses containing $\overline{b}$ but not containing $\overline{a}_1$, i.e. $\Delta_2 \geqslant 8$.

Thus, at the moment we have that $\Delta_1 \geqslant 13$, $\Delta_2 \geqslant 8$, $\Delta \geqslant 20$. The further argument (taking into account the literals that occur in the clauses containing $\overline{a}_1$) is similar to Case 2.1.

□

Lemma 6.3. *The branching number of the tree $T_{5.1}$ with respect to the measure of complexity $\mathfrak{N}$ does not exceed* 1.3248.

Proof. We consider two cases.

Case 1. $s = 2$.

Note that the number of variables in the formula F_1 is less than the number of variables in the formula F by at least two. Namely, besides the variable a_1 eliminated immediately, Procedure *REDUCE* at step (R2) eliminates the variable a_2 (since after the elimination of the clause C, the only occurrence of the literal a_2 disappears from the formula F).

The number of variables in the formula F_2 is less than the number of variables in the formula F by at least three. First, the variable a_1 is eliminated immediately. Second, at step (R1) the variable a_2 will be eliminated, because it occurs in a 1-clause obtained from C by deleting a_1. Moreover, since the only occurrence of the literal b disappears, at step (R2) the variable b will be eliminated too.

Thus, in this case the node has the branching tuple $(2,3)$ and the branching number $\tau((2,3)) < 1.3248$.

CASE 2. $s \geqslant 3$.

Note that the number of variables in the formula F_1 is less than the number of variables in the formula F by at least three. Namely, besides the variable a_1 eliminated immediately, Procedure *REDUCE* at step (R2) eliminates the variables a_2 and a_3 (since after elimination of the clause C, the only occurrences of the literals a_2 and a_3 disappear from the formula F).

The number of variables in the formula F_2 is less than the number of variables in the formula F by at least two. First, the variable a_1 is eliminated immediately. Second, since the only occurrence of the literal b disappears, at step (R2) the variable b will be eliminated.

Thus, in this case the node also has the branching tuple $(2,3)$ and the branching number $\tau((2,3)) < 1.3248$.

□

§7. Algorithms for formulas with small number of clauses.

To obtain from Algorithm 5.1 an algorithm that works efficiently for formulas with small number of clauses, we apply as $\mathcal{M}$ the following algorithm.

Algorithm 7.1 (Algorithm $\mathcal{M}_K$)

Input:

- a number $k \in \mathbb{N} \cup \{+\infty\}$,
- a formula F in k–CNF–(1,∞)–(+/–) processed by Procedure *REDUCE*.

Output: "Satisfiable" if F is satisfiable; "Unsatisfiable" otherwise.

Method:

(K1) Check if F contains positive clauses containing more than two literals. If there are none of these, check if F is a formula in CNF–(1,3). If it is, perform the following operations:
- Assume $H = F$.
- Assume $P_+ = \{$the set of all positive clauses of $H\}$.
- For each $C = \{u_1, u_2\} \in P_+$ perform the following operation: eliminate C from H, and then replace all occurrences of $\overline{u}_2$ in H by u_1.
- Output the result of applying to H some algorithm $\mathcal{A}_K$ that solves the satisfiability of k–CNF in a time not exceeding $p(L)1.3289^{\mathfrak{K}(H)}$, where p is a polynomial (for example, such algorithms are given in [6] and [13]).

(K2) Let F contain a positive clause containing at least three literals, and let this clause contain a variable v_1 occurring in F at least four times. In this case, construct the formulas

$$G_1 = REDUCE(k, F[v_1]),$$
$$G_2 = REDUCE(k, F[\overline{v}_1]).$$

If Procedure *SPLIT* outputs "Satisfiable" for at least one of these, output "Satisfiable"; otherwise, "Unsatisfiable".

(K3) If F is not a formula in CNF–(1,3), choose the variable v_1 with the maximal number of occurrences in F. Construct the formulas

$$G_1 = REDUCE(k, F[v_1]),$$
$$G_2 = REDUCE(k, F[\overline{v}_1]).$$

If Procedure $SPLIT$ outputs "Satisfiable" for at least one of these, output "Satisfiable"; otherwise, "Unsatisfiable".

(K4) Let F contain a positive clause $C = \{v_1, v_2, \ldots, v_s\}$ and a negative clause D such that
- $s \geqslant 3$;
- $\overline{v}_1 \in D$;
- the clause D contains a (1,2)-variable u_1.

In this case construct the formulas

$$G_1 = REDUCE(k, F[v_1]),$$
$$G_2 = REDUCE(k, F[\overline{v}_1]).$$

If Procedure $SPLIT$ outputs "Satisfiable" for at least one of these, output "Satisfiable"; otherwise, "Unsatisfiable".

(K5) Find in F clauses $C = \{v_1, v_2\}$ and D such that
- v_1 is a (1,3)-variable;
- $\overline{v}_1 \in D$;
- the clause D contains a (1,2)-variable u_1.

In this case construct the formulas

$$G_1 = REDUCE(k, F[v_1]),$$
$$G_2 = REDUCE(k, F[\overline{v}_1]).$$

If Procedure $SPLIT$ outputs "Satisfiable" for at least one of these, output "Satisfiable"; otherwise, "Unsatisfiable".

□

Lemma 7.1. *Algorithm 7.1 performs correctly (i.e. at least one of the conditions (K1)–(K5) is always satisfied and the answer the algorithm outputs is correct).*

Proof. The matter is that if the conditions of the steps (K1) and (K3) are not satisfied, then there are in F positive clauses of length at least three. If also the condition of the step (K2) is not satisfied, then these clauses consist of only (1,2)-variables; consider one of these clauses. If the condition of the step (K4) is not satisfied either, then one of the variables of this clause occurs in a negative clause that contains besides it only (1,3)-variables (the formula F does not contain (1,4+)-variables, since the condition of the step (K3) is not satisfied). Each of these (1,3)-variables occurs in a positive 2-clause (since the condition of the step (K2) is not satisfied). The latter negative and positive clauses do for the clauses D and C at step (K5). Thus, the condition of at least one of the steps (K1)–(K5) is satisfied for any formula processed by Procedure $REDUCE$. Since each step decreases the number of clauses in the formula, the algorithm must stop.

The only place in the algorithm that may seem confusing is the step (K1), i.e. the inverse reduction of the formula in k–CNF–(1,∞) to k–CNF. Note that each satisfying assignment for the original formula corresponds to a satisfying assignment for the formula passed to the algorithm $\mathcal{A}_K$: for each positive clause $\{v_1, v_2\}$ we keep in this assignment exactly one of the variables v_1 or v_2 (it must contain one of them because it is satisfiable, and the other one without negation occurs once just in this clause), and then perform the same changes of literals as in the formula itself. Conversely, given an assignment satisfying the formula passed to the algorithm $\mathcal{A}_K$, we can construct an assignment satisfying F by making these changes in the opposite direction: each of arising positive 2-clauses will have exactly one positive and negative literal.
□

Consider the tree $T_{7.1}$ whose internal nodes are the formulas splitted at steps (K2)–(K5) of Algorithm 7.1 and at step (S3) of Procedure *SPLIT* called from Algorithm 7.1. Each its internal node has two sons corresponding to the formulas G_1 and G_2, or F_1 and F_2 respectively. The leaves of the tree $T_{7.1}$ are the formulas that Algorithm 7.1 does not split but passes to the algorithm $\mathcal{A}_K$ or outputs the answer immediately.

Lemma 7.2. *The branching number of the tree $T_{7.1}$ with respect to the measure of complexity $\mathfrak{K}$ does not exceed* 1.1939.

Proof. By Lemma 6.1, it suffices to prove that the branching numbers corresponding to the steps (K2)–(K5) do not exceed 1.1939.

Case 1. Splitting takes place at step (K2).
In this case the number of clauses in the formula G_1 is less than the number of clauses in the formula F by at least five owing to the disappearance of the positive clause C containing v_1 and four clauses containing the negations of other variables occurring in C (their existence follows from Lemma 4.2).

As to the formula G_2, the number of clauses in it is less than the number of clauses in F by at least three owing to the disappearance of the clauses containing $\overline{v}_1$.

In this case the node has the branching tuple $(3, 5)$ and the branching number $\tau((3, 5)) < 1.1939$.

Case 2. Splitting takes place at step (K3).
In this case the formula F is not in CNF–(1,3). Hence, at step (K3) a (1,4+)-variable will be chosen as v_1. Assume it occurs in clause

$$\{v_1, v_2, \ldots, v_s\}. \tag{7.2}$$

Case 2.1. $s = 2$.
In this case $\mathfrak{K}(G) - \mathfrak{K}(G_1) \geqslant 3$, since besides the clause (7.2) eliminated immediately, the clauses containing $\overline{v}_2$ are eliminated at step (R2) (by Lemma 4.2, there are at least two such clauses). When passing from G to G_2, the algorithm eliminates the clause (7.2) (at step (R1)) as well as the clauses containing $\overline{v}_1$. Since v_1 is a (1,4+)-variable, there are at least four such clauses. Thus we have $\mathfrak{K}(G) - \mathfrak{K}(G_2) \geqslant 5$.

In this case the node has the branching tuple $(3,5)$ and the branching number $\tau((3,5)) < 1.1939$.

CASE 2.2. $s \geqslant 3$.
In this case $\mathfrak{K}(G)-\mathfrak{K}(G_1) \geqslant 5$, since besides the clause (7.2) eliminated immediately, the clauses containing $\overline{v}_2$ and $\overline{v}_3$ are eliminated at step (R2) (by Lemma 4.2, there are at least four such clauses). On the other hand, $\mathfrak{K}(G) - \mathfrak{K}(G_2) \geqslant 4$. When passing from G to G_2, the clauses containing $\overline{v}_1$ will be eliminated. Since v_1 is a (1,4+)-variable there are at least four such clauses. In this case the node has the branching tuple $(4,5)$ and the branching number $\tau((4,5)) < 1.1939$.

CASE 3. SPLITTING TAKES PLACE AT STEP (K4).
In this case $\mathfrak{K}(G) - \mathfrak{K}(G_1) \geqslant 5$, since besides the clause C eliminated immediately, the clauses containing $\overline{v}_2$ and $\overline{v}_3$ are eliminated at step (R2) (by Lemma 4.2, there are at least four such clauses). On the other hand, $\mathfrak{K}(G) - \mathfrak{K}(G_2) \geqslant 3$. When passing from G to G_2, two clauses containing $\overline{v}_1$ will be eliminated. In addition, the occurrence of the (1,2)-variable u_1 will be eliminated. This means that at step (R3) (or at step (R2)) at least one more clause will be eliminated. In this case the node has the branching tuple $(3,5)$ and the branching number $\tau((3,5)) < 1.1939$.

CASE 4. SPLITTING TAKES PLACE AT STEP (K5).
In this case $\mathfrak{K}(G) - \mathfrak{K}(G_1) \geqslant 3$, since besides the clause C eliminated immediately, the clauses containing $\overline{v}_1$ are eliminated at step (R2) (by Lemma 4.2, there are at least two such clauses). On the other hand, $\mathfrak{K}(G) - \mathfrak{K}(G_2) \geqslant 5$. When passing from G to G_2, three clauses containing $\overline{v}_1$ will be eliminated. In addition, the occurrence of the (1,2)-variable u_1 will be eliminated. This means that at step (R3) (or at step (R2)) at least one more clause will be eliminated. In this case the node has the branching tuple $(3,5)$ and the branching number $\tau((3,5)) < 1.1939$.
□

Theorem 7.1. *If we use Algorithm 7.1 as the algorithm $\mathcal{M}$ in Algorithm 5.1 , then Algorithm 5.1 solves the satisfiability problem for a formula in k-CNF-(1,∞) in a time not exceeding $p(L)1.1939^K$, where K is the number of clauses in the input formula, L is its length, q is a polynomial.*

Proof. It follows from Lemma 5.1, Lemma 5.2 and Lemma 7.1 that the algorithm solves the satisfiability problem correctly. The branching number of the tree $T_{5.1}$ with respect to the measure of complexity $\mathfrak{K}$ does not exceed 1.1939 by Lemma 6.1. Hence, by Lemma 3.2, it suffices to prove that Algorithm 7.1 outputs the answer in a time at most $p(L)1.1939^K$, where p is a polynomial, K is the number of clauses in the formula given as input to Algorithm 7.1, L is its length.

When the formula is passed to the algorithm $\mathcal{A}_K$, the number of clauses in it decreases considerably. Namely, if H is the formula given to the algorithm $\mathcal{A}_K$ as the input, then $\mathfrak{K}(H) = \mathfrak{K}(F) - \mathfrak{N}(F)/2$. The algorithm $\mathcal{A}_K$ stops in a time not exceeding $p_1(L)1.2664^{\mathfrak{K}(H)}$, where p_1 is a polynomial. Since F is a formula in CNF–(1,3), and each its clause contains at least two literals, $\mathfrak{N}(F) \geqslant \mathfrak{K}(F)/2$. Therefore

$$1.2664^{\mathfrak{K}(H)} = 1.2664^{\mathfrak{K}(F)-\mathfrak{N}(F)/2} \leqslant 1.2664^{\frac{3}{4}\mathfrak{K}(F)} < 1.1938^{\mathfrak{K}(F)}.$$

Thus, it takes a time at most $p_2(L)1.1938^m$ to process each leaf of the tree $T_{7.1}$, where m is the number of clauses in the formula labelling this leaf. Let $f(m) = p_2(L)1.1938^m$. Algorithm 7.1 takes a time not exceeding

$$p_3(L) \sum_{l \text{ is a leaf of } T_{7.1}} f(\mathfrak{K}(l)). \tag{7.1}$$

to process the input formula. By Lemma 7.2, the branching number of the tree $T_{7.1}$ does not exceed 1.1939. Hence, by Lemma 3.2 the sum (7.1) does not exceed $p_3(L)p_2(L)\max(1.1939, 1.1938)^K = p(L)1.1939^K$.
□

§8. Algorithm for short formulas.

To obtain from Algorithm 5.1 an algorithm that works efficiently for short formulas, we apply as $\mathcal{M}$ the following algorithm.

Algorithm 8.1 (Algorithm $\mathcal{M}_L$)

Input:

- a number $k \in \mathbb{N} \cup \{+\infty\}$,
- a formula F in k–CNF–(1,∞)–(+/–) processed by Procedure *REDUCE*.

Output: "Satisfiable" if F is satisfiable; "Unsatisfiable" otherwise.

Method:

(L1) Check if F is a formula in CNF–(1,2). If it is, then output the answer returned for this formula by the algorithm of O. Kullmann and H. Luckhardt from [10] for formulas in CNF–(1,2) whose running time is $p_1(L)1.0416^L$.

(L2) Choose the variable v_1 with the maximal number of occurrences in F. Construct formulas

$$G_1 = REDUCE(k, F[v_1]),$$
$$G_2 = REDUCE(k, F[\overline{v}_1]).$$

If Procedure *SPLIT* outputs "Satisfiable" for at least one of these, output "Satisfiable"; otherwise, "Unsatisfiable".

□

Theorem 8.1. *If we use Algorithm 8.1 as the algorithm $\mathcal{M}$ in Algorithm 5.1 then Algorithm 5.1 solves the satisfiability problem for a formula in k–CNF–(1,∞) in a time not exceeding $q(L)1.0644^L$, where L is the length of the input formula, q is a polynomial.*

Proof. It follows from Lemma 5.1, Lemma 5.2 and the correctness of the algorithm from [10] used in Algorithm 8.1 that the algorithm solves the satisfiability problem correctly. The branching number of the tree $T_{5.1}$ with respect to the measure of complexity $\mathfrak{L}$ does not exceed 1.0644 by Lemma 6.2. Thus, by Lemma 3.2 it suffices to prove that Algorithm 8.1 outputs the answer in a time not exceeding $p(L)1.0644^L$, where p is a polynomial, L is the length of the formula given as input to Algorithm 7.1.

Consider the tree $T_{8.1}$ whose internal nodes are the formulas splitted at step (L2) of Algorithm 8.1 and at step (S3) of Procedure *SPLIT* called from Algorithm 8.1. Each its internal node has two sons corresponding to the formulas G_1 and G_2, or F_1 and F_2 respectively. The leaves of the tree $T_{8.1}$ are the formulas that Algorithm 7.1 does not split but passes to the algorithm by O. Kullmann and H. Luckhardt from [10] or outputs the answer immediately.

The running time of the algorithm from [10] does not exceed $p_1(L)1.0416^L$. Thus, by Lemma 3.2, to prove the desired bound it suffices to show that the branching number of the tree $T_{8.1}$ does not exceed 1.0644. The branching numbers corresponding to the step (S3) satisfy this condition by Lemma 6.2. Thus, we only have to prove the same for the branching numbers corresponding to the step (L2). But as was noted, consideration of Case 1 in Lemma 6.2 does not use the fact that one of the clauses containing $\overline{a}_1$ contains a positive literal b. The condition of Case 1 of Lemma 6.2 is also satisfied at step (L2), hence we may use this argument for the step (L2), too.

Theorem follows.

□

§9. Algorithm for formulas with small number of variables.

To obtain from Algorithm 5.1 an algorithm that works efficiently for formulas with small number of variables, we apply as $\mathcal{M}$ the following algorithm. It uses as a procedure an algorithm that works efficiently for formulas in k-CNF. We do not fix a concrete algorithm because the case of its application is a critical point: the better algorithm we take, the better bound we obtain.

Algorithm 9.1 (Algorithm $\mathcal{M}_N$)

Input:

- a number $k \in \mathbb{N} \cup \{+\infty\}$,
- a formula F in k-CNF-(1,∞)-(+/-).

Output: "Satisfiable" if F is satisfiable; "Unsatisfiable" otherwise.

Method:

(N1) Let F contain a positive clause consisting of at least four literals $C = \{v_1, v_2, \dots, v_s\}$. In this case, construct the formulas

$$G_1 = REDUCE(k, F[v_1]),$$
$$G_2 = REDUCE(k, F[\overline{v}_1]).$$

If Procedure *SPLIT* outputs "Satisfiable" for at least one of these, output "Satisfiable"; otherwise, "Unsatisfiable".

(N2) Let F contain a positive clause consisting of three literals $C = \{v_1, v_2, v_3\}$. In this case, construct the formulas

$$G_1 = F[v_1],$$
$$G_2 = F[\overline{v}_1].$$

If the algorithm $\mathcal{M}_N$ outputs "Satisfiable" for at least one of these, then output "Satisfiable"; otherwise, "Unsatisfiable".

(N3) Assume $H_1 = F$.

(N4) Assume $\mathcal{P} = \{$the set of all positive clauses $H\}$.

(N5) For each $C = \{u_1, u_2\} \in \mathcal{P}$ perform the following operation: eliminate C from H_1, and then replace all occurrences of $\overline{u}_2$ in H_1 by u_1.

(N6) Assume $H = H_1$. Output the answer returned for the formula $REDUCE(k, H)$▌ by some algorithm $\mathcal{A}_N$ that solves the satisfiability problem of formulas in k–CNF.

□

Lemma 9.1. *Algorithm 9.1 solves the satisfiability problem correctly.*

Proof. By Lemma 4.1 and Lemma 5.1, the only steps of Algorithm 9.1 that may seem confusing are (N3)–(N6), i.e. the inverse reduction of the formula in k–CNF–(1,∞) to k–CNF. Note that any satisfying assignment for the original formula generates a satisfying assignment for the formula H, and a formula passed to the algorithm $\mathcal{A}_N$ is equi-satisfiable to H. Namely, for each positive clause $\{v_1, v_2\}$ we keep in this assignment exactly one of the variables v_1 or v_2 (it must contain one of them because it is satisfiable, and the other one without negation occurs once just in this clause), and then perform the same changes of literals as in the formula H_1. Conversely, given an assignment satisfying H, we can construct an assignment satisfying F by making these changes in the opposite direction: each of arising positive 2-clauses will have exactly one positive and negative literal.
□

Lemma 9.2. *Let F be a formula in k–CNF–(1,∞)–(+/–) processed by Procedure $REDUCE$, let L be its length, N be the number of variables occurring in it, $\mathcal{T}(n) = p_1(L)\,\gamma^n$ be an upper bound on the running time of the algorithm $\mathcal{A}_N$ for a formula containing n variables, where $1 < \gamma \leqslant 2$, p_1 is a polynomial. Then*

(1) *if $k = 3$, then Algorithm 9.1 computes the answer for the formula F in the time $p_2(L)\,(\gamma+1)^{N/3}$ for some polynomial p_2;*

(2) *if $k > 3$, then Algorithm 9.1 computes the answer for the formula F in the time $q_2(L) \max\left(1.3803, (\gamma+1)^{1/3}\right)^N$ for some polynomial q_2.*

Proof. (1) Note that in this case Algorithm 9.1 changes nothing at step (N1). Let F contain $t \leqslant N/3$ positive clauses consisting of three literals. Consider the tree $T_{9.1}$ whose internal nodes are the formulas splitted at step (N2). Each its internal node has two sons corresponding to the formulas G_1 and G_2. We call the corresponding edges the edges of type 1 and type 2 respectively. The leaves of the tree $T_{9.1}$ are the formulas that Algorithm 9.1 does not split but passes to the algorithm $\mathcal{A}_N$.

When we pass an edge of type (2), one variable is eliminated. When we pass an edge of type (1), one variable is eliminated, and the only positive occurences of two more variables are eliminated whose negative occurences are eliminated then at step (R2) of Procedure $REDUCE$ called at step (N6). The number of the rest variables (occuring in positive 2-clauses) is reduced twice at step (N5). Hence, if the path from the root (the original formula passed from Algorithm 5.1) to a leaf contains exactly i_1 edges of type 1 and exactly i_2 edges of type 2, then the formula labelling this leaf has at most $(N - 3i_1 - i_2)/2$ variables. Note that $i_1 + i_2 = t$, since splitting can not result in disappearance of positive 3-clauses different of the splitted clause itself, and in arising of new 3-clauses. For given i_1 and i_2, in the tree $T_{9.1}$ there are at most $\binom{t}{i_2}$ different leaves with such a path from the root. Since processes of splitting the formula, calling the procedures and the steps (N3)–(N6) take a polynomial time, Algorithm 9.1 outputs the answer in a time not exceeding

$$p_3(L)\sum_{i_2=0}^{t}\binom{t}{i_2}\mathcal{T}((N-3i_1-i_2)/2) =$$

$$p_3(L)\sum_{i_2=0}^{t}\binom{t}{i_2}\gamma^{(N-3i_1-i_2)/2}p_1(L) =$$

$$= p_1(L)p_3(L)\sum_{i_2=0}^{t}\binom{t}{i_2}\gamma^{(N+2i_2-3t)/2} =$$

$$\gamma^{(N-3t)/2}p_2(L)\sum_{i_2=0}^{t}\binom{t}{i_2}\gamma^{i_2} =$$

$$= \gamma^{(N-3t)/2}p_2(L)(\gamma+1)^t =$$

$$\gamma^{N/2}p_2(L)\left(\frac{\gamma+1}{\gamma^{3/2}}\right)^t \leqslant$$

$$= \gamma^{N/2}p_2(L)\left(\frac{\gamma+1}{\gamma^{3/2}}\right)^{N/3} =$$

$$p_2(L)(\gamma+1)^{N/3},$$

where $p_3(L)$ is a polynomial, $p_2 = p_1p_3$. The inequality holds because for $1 < \gamma \leqslant 2$, we have $\gamma + 1 > \gamma^{3/2}$, $t \leqslant N/3$.

(2) Construct the tree $T'_{9.1}$ whose internal nodes are labelled with formulas splitted at step (N1) of Algorithm 9.1 and at step (S3) of Procedure *SPLIT* called from Algorithm 9.1. Each its internal node has two sons corresponding to the formulas G_1 and G_2, or F_1 and F_2 respectively. The leaves of the tree $T'_{9.1}$ are the formulas that Algorithm 9.1 splits not at step (N1) but at step (N2), or passes to the algorithm $\mathcal{A}_N$. Each of these leaves is a formula G in k–CNF–(1,∞)–(+/–) such that each of its positive clauses contains at most three literals. By (1), such formula is processed in a time at most $p_2(L)(\gamma+1)^{\mathfrak{N}(G)/3}$ (one can easily see that in the proof of (1), the restriction $k = 3$ applies only to positive clauses). The step (N1) has the branching tuple $(1, 4)$ and the branching number $\tau((1,4)) < 1.3803$. By Lemma 6.3, the branching numbers of the tree $T'_{9.1}$ corresponding to other nodes do not exceed 1.3803, either. Thus, by Lemma 3.2 the desired statement follows.
□

Theorem 9.1. *Let F be a formula in k–CNF–(1,∞), L be its length, N be the number of variables occuring in it, $\mathcal{T}(n) = \gamma^n p_1(L)$ be the upper bound on the run-*

ning time of the algorithm $\mathcal{A}_N$ for a formula containing n variables, $1 < \gamma \leqslant 2$, p_1 is a polynomial. If we use Algorithm 9.1 as the algorithm $\mathcal{M}$ in Algorithm 5.1 then Algorithm 5.1 solves the satisfiability problem for a formula in k–CNF–(1,∞) and outputs the answer in a time not exceeding

(1) $p(L)(\max(1.3248, (\gamma+1)^{1/3}))^N$ *if* $k = 3$*;*
(2) $p(L)(\max(1.3803, (\gamma+1)^{1/3}))^N$ *if* $k > 3$*,*

where p is a polynomial.

Proof. It follows from Lemma 5.1, Lemma 5.2 and Lemma 9.1 that the algorithm solves the satisfiability problem correctly.

(1) Note that the running time of Algorithm 5.1 depends polynomially on the time that Algorithm 9.1 takes to process *all* formulas passed to it. This time equals $\sum_{l \text{ is a leaf of } T_{5.1}} f(\mu(G_l))$, where $\mu(J)$ denotes the number of variables in the formula J, G_l is the formula labelling the leaf l of the tree $T_{5.1}$, $f(\nu)$ is the time taken by Algorithm 9.1 on a formula containing ν variables. By Lemma 9.2, $f(\nu) = p_2(L)\,(\gamma+1)^{\nu/3}$. By Lemma 6.3, we have $\tau_{max} \leqslant 1.3248$. Thus, by Lemma 3.2, the desired sum equals $p(L)(\max(1.3248, (\gamma+1)^{1/3}))^N$ for some polynomial p.

(2) Similar to (1).
□

Corollary 9.1. *Let the number k given as input to Algorithm 5.1 equals three. If we take as $\mathcal{A}_N$ the algorithm from* [9]*, then the running time of Algorithm 5.1 is* $p(L)1.3566^N$*.*

Proof. O. Kullmann proved in [9] that his algorithm solves the satisfiability problem for formulas in 3-CNF[1] in the time $p(L)1.49625^{\mathfrak{N}(F)}$. By Theorem 9.1, the desired bound $p(L)(\max(1.3248, 2.49625^{1/3}))^N \leqslant p(L)(\max(1.3248, 1.3566))^N = p(L)1.3566^N$ follows.
□

Corollary 9.2. *Let the number k given as input to Algorithm 5.1 is greater than three. Then if we take as $\mathcal{A}_N$ the algorithm from* [13]*, then the running time of Algorithm 5.1 is* $p(L)(\psi(k)+1)^{N/3}$*, where* $\psi(k) = \tau((1,2,\ldots,k-1))$*.*

Proof. B. Monien and E. Speckenmeyer showed in [13] that their algorithm solves the satisfiability problem of the formula in k–CNF in the time $p(L)\psi(k)^N$, where $\psi(k) = \tau((1,\ldots,k-1))$. Easy computations show that $\tau((1,4)) < (\tau((1,2,3))+1)^{1/3} \leqslant \ldots \leqslant (\tau((1,2,\ldots,k-1))+1)^{1/3}$. The desired bound $p(L)(\max(1.3803, (\psi(k)+1)^{1/3}))^N = p(L)(\psi(k)+1)^{N/3}$ follows now from Theorem 9.1.
□

The following table contains approximate values of $\psi(k)$ and $(\psi(k)+1)^{1/3}$.

[1] In [9], the algorithm was formulated in terms of the dual problem, i.e. the tautology problem of 3-DNF.

k	$\psi(k)$	$(\psi(k)+1)^{1/3}$
4	1.8393	1.4161
5	1.9276	1.4306
6	1.9660	1.4368
7	1.9836	1.4397

TABLE 2. Approximate values of $\psi(k)$ and $(\psi(k)+1)^{1/3}$.

§10. Corollary for general SAT.

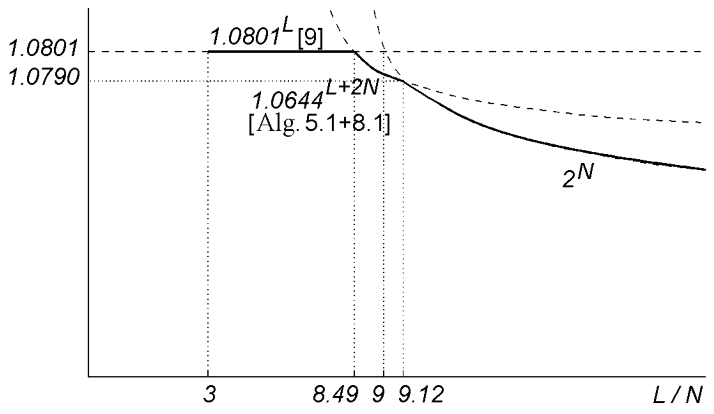
For the satisfiability problem for formulas in CNF, as well as for its subproblems similar to that considered in this paper, there are different upper bounds depending on parameters of the input formula (the length, the number of clauses, the number of variables). These upper bounds are obtained by using different algorithms. To obtain the answer to the question about the satisfiability of a given formula more efficiently, one should run these algorithms in parallel, or run the algorithm such that the bound on its running time is the best for the given formula. Thus, we obtain bounds that depend on additional parameters – $\frac{\mathfrak{K}}{\mathfrak{N}}$, $\frac{\mathfrak{L}}{\mathfrak{N}}$ etc., "uniting" the bounds of different algorithms. Some bounds of this kind for different classes of Boolean formulas are considered in [10]. We now consider one upper bound for formulas in CNF arising from application of the results of this paper. Besides evident bounds for formulas in CNF–(1,∞) arising from uniting the bounds §§7–9, it turns out that these results imply a bound for arbitrary formulas in CNF.

Consider a formula F in CNF. We can transform it into a formula G in CNF–(1,∞) in a polynomial time. Its parameters change as follows:

$$\begin{aligned}
\mathfrak{K}(G) &= \mathfrak{K}(F) + \mathfrak{N}(F),\\
\mathfrak{L}(G) &= \mathfrak{L}(F) + 2\cdot\mathfrak{N}(F),\\
\mathfrak{N}(G) &= 2\cdot\mathfrak{N}(F).
\end{aligned}$$

One can do it by replacing all occurrences of each positive literal $v \in V$ with the negation of a new variable x_v and adding the clause $v \vee x_v$. The equivalence of the satisfiability of F and G is evident.

If we now apply the bound of Theorem 8.1 to our formula, then we obtain that the satisfiability problem of the formula F in CNF can be solved in the time $p(\mathfrak{L}(G))1.0644^{\mathfrak{L}(G)} \leqslant p'(\mathfrak{L}(F))1.0644^{(\mathfrak{L}(F)+2\cdot\mathfrak{N}(F))}$, where p, p' are polynomials. Note that for certain values of $\frac{\mathfrak{L}(F)}{\mathfrak{N}(F)}$ this expression gives a bound better than the $1.0801^{\mathfrak{L}(F)}$-algorithm from [10] and the trivial $2^{\mathfrak{N}(F)}$-algorithm. The most optimal is an algorithm that processes formulas with $\mathfrak{L}(F) < 8.49\mathfrak{N}(f)$ using the $1.0801\mathfrak{L}(F)$-algorithm from [10], processes formulas with $\mathfrak{L}(F) > 9.12\mathfrak{N}(f)$ using the trivial $2^{\mathfrak{N}(F)}$-algorithm, and processes all other formulas using Algorithm 5.1 that applies Algorithm 8.1. The following graph presents the dependence of the base α of the exponent $\alpha^{\mathfrak{L}(F)}$ on $\frac{L}{N} = \frac{\mathfrak{L}(F)}{\mathfrak{N}(F)}$. We do not give the data for formulas with $\frac{L}{N} < 3$ because such formulas contain variables occurring at most twice, and can be transformed in a polynomial time into formulas with $\frac{L}{N} \geqslant 3$. It seems that there exists even more effective algorithm for solving the satisfiability problem of such formulas.



References

1. M. R. Garey, D. S. Johnson, *Computers and intractability. A guide to the theory of NP-completeness*, W. H. Freeman and Company, San Francisco, 1979.
2. E. Dantsin, *Tautology proof systems based on the splitting method (in Russian)* PhD thesis, Leningrad Division of Steklov Institute of Mathematics (LOMI), Leningrad, 1983.
3. E. Dantsin, *The algorithmics of the propositional satisfiability problem*, 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, "Nauka", Leningrad, 1988.
5. M. Davis, H. Putnam, *A computing procedure for quantification theory*, J. Assoc. Comp. Mach. **7** (1960), 201–215.
6. 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.
7. E. A. Hirsch, *Two New Upper Bounds for SAT*, Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (1998), to appear.
8. O. Kullmann, *A systematical approach to 3-SAT-decision, yielding 3-SAT-decision in less than* 1.5045^n *steps*, Theoretical Computer Science, 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.
10. O. Kullmann and H. Luckhardt, *Deciding propositional tautologies: Algorithms and their complexity*, submitted to Information and Computation, 1997.
11. H. Luckhardt, *Obere Komplexitätsschranken für TAUT-Entscheidungen*, Proc. Frege Conference 1984, Schwerine, Akademie-Verlag Berline, 331–337.
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 then* 2^n *steps*, Discrete Applied Mathematics **10** (1985), 287–295.
14. I. Schiermeyer, *Pure literal look ahead: An* $O(1.497^n)$ *3-Satisfiability algorithm*, Workshop on the Satisfiability Problem Technical Report, Siena, April, 29 – May, 3, **1996**.