New Worst-Case Upper Bounds for SAT*

Edward A. Hirsch†

Abstract

In 1980 Monien and Speckenmeyer [21] proved that satisfiability of a propositional formula consisting of K clauses (of arbitrary length) can be checked in time of the order $2^{K/3}$. Recently Kullmann and Luckhardt [16] proved the worst-case upper bound $2^{L/9}$, where L is the length of the input formula. The algorithms leading to these bounds are based on the *splitting method* which goes back to the Davis–Putnam procedure. *Transformation rules* (pure literal elimination, unit propagation etc.) constitute a substantial part of this method. In this paper we present a new transformation rule and two algorithms using this rule. We prove that these algorithms have the worst-case upper bounds $2^{0.30897K}$ and $2^{0.10299L}$ respectively.

1 Introduction.

SAT (the problem of satisfiability of a propositional formula in conjunctive normal form) can be easily solved in time of the order 2^N, where N is the number of variables in the input formula. In the early 1980s this trivial bound was reduced for formulas in 3-CNF by Monien and Speckenmeyer [20] (see also [22]) and independently by Dantsin [1] (see also [4] and [2]). After that, many upper bounds for SAT and its subproblems were obtained [21, 17, 25, 13, 14, 26, 16, 8, 23, 24]. Most authors consider bounds w.r.t. three main parameters: the length L of the input formula, the number K of its clauses and the number N of the variables occurring in it. Before the conference proceedings version of this paper [9], the best known worst-case upper bounds for SAT were:

- $p(L)2^{L/9}$ [16],
- $p(L)2^{K/3}$ [21] (see also [16]),

where p is a polynomial. Also, the upper bound $p(L)2^{L/4}$ for satisfiability problem for general Boolean formulas (i.e., not necessarily in CNF) is known [28]. Recently Paturi, Pudlak, Saks and Zane [24] proved that 3-SAT is checkable in the randomized time $O(2^{0.446N})$, but it is still

*Some of the work described in this paper was presented at the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA'98) [9].

†Steklov Institute of Mathematics at St.Petersburg, 27 Fontanka, 191011 St.Petersburg, Russia. Email: hirsch@pdmi.ras.ru, URL: http://logic.pdmi.ras.ru/~hirsch/index.html. Supported in part by grants from INTAS and RFBR.

unknown whether the trivial upper bound 2^N for the general SAT problem can be improved. Kullmann and Luckhardt in [16] simplified the algorithm of Monien and Speckenmeyer from [21] and presented also an improved algorithm of the same complexity $p(L)2^{K/3}$ which works better when the ratio of the number of clauses to the number of variables is greater than 2.64557. In the conference proceedings version of this paper [9] we presented two algorithms which improve the bounds $p(L)2^{K/3} = p(L)2^{0.33333...K}$ and $p(L)2^{L/9} = p(L)2^{0.11111...L}$ to $p(L)2^{0.30897K}$ and $p(L)2^{0.10537L}$ respectively. In this journal version we improve the second algorithm and prove the worst-case upper bound $p(L)2^{0.10299L}$. Our new algorithm uses the $p(L)2^{0.30897K}$-time algorithm as a subroutine; the proof of the corresponding upper bound is much simpler than the proof in the conference proceedings version [9].

The most popular methods for solving SAT are the *local search method* and the *splitting method*. Many experimental and average-case results show the significant power of the local search method (for references, see the survey [7]). However, most best known worst-case upper bounds for SAT and its $\mathcal{NP}$-complete subproblems are obtained using the splitting method [21, 16, 8] (for worst-case upper bounds for the local search method see [10, 11, 12]). The splitting method is also useful in proving worst-case upper bounds for exact and approximate MAXSAT solving [18, 3, 19].

The splitting method goes back to the Davis–Putnam procedure [6]. Let l be a literal occurring in a formula F. Let $F[l]$ be the formula obtained from F by assigning the value $True$ to the literal l, i.e. by removing all clauses containing l and deleting all occurrences of $\overline{l}$ from the other clauses. In short, the main idea of the Davis–Putnam procedure is that F is satisfiable iff at least one of the formulas $F[l]$ and $F[\,\overline{l}\,]$ is satisfiable. In a wider sense [5], a splitting algorithm constructs a tree by reducing satisfiability of a formula F to satisfiability of several formulas $F[I_1], F[I_2], \ldots, F[I_s]$ obtained from F by assignments $I_1, I_2, \ldots, I_s$ respectively. Then, a splitting algorithm simplifies each of the formulas $F[I_1], F[I_2], \ldots, F[I_s]$ according to *transformation rules* which do not change their satisfiability. These transformations take a polynomial time. Their role is to reduce some of the parameters L, K or N of the formulas $F[I_1], F[I_2], \ldots, F[I_s]$ and to simplify these formulas by eliminating pure literals (a literal is *pure* if its negation does not occur in the formula), 1-clauses (1-*clause* is a clause consisting of one literal) and other "easy pieces".

Recurrent equations are often used in complexity analysis of algorithms. Kullmann and Luckhardt in [16] described a very similar, but simpler technique which is very useful in the estimation of the running time of splitting algorithms. One can consider an execution of a splitting algorithm as a *branching tree*, i.e. a tree such that formulas labelling children are simpler than formulas labelling parents (leaves are labelled with the simplest formulas). With each node of this tree we associate a *branching vector* of non-negative numbers and a polynomial constructed from this vector. One can estimate the number of leaves in the tree using the largest of the positive roots of these polynomials. Precise definitions and formulations are given in Sect. 3.

Transformation rules play an important role in splitting algorithms. Two simplest rules were proposed in the original paper of Davis and Putnam [6]: we can eliminate pure literals and 1-clauses. All transformation rules we use are described in Sect. 4. This paper introduces a new transformation rule which goes up to the following simple observation: if each clause of our formula contains at least one negative literal, then this formula is trivially satisfiable

by the assignment in which all variables have the value $False$. Let $P(l, F)$ be a property of a literal and a formula, for example, "F contains exactly two occurrences of l and at least three occurrences of $\overline{l}$". In addition, we suppose that for each variable v in the formula F at most one of the literals v and $\overline{v}$ satisfies P. Given F, the literals that satisfy the property P will be referred as *P-literals*.

> *The black and white literals principle.* Let F be a formula in CNF. At least one of the following two alternatives holds.
>
> (1) There is a clause in F that contains a P-literal and does not contain the negations of any other P-literals.
>
> (2) Satisfiability of F is equivalent to satisfiability of the formula obtained from F by removing all clauses containing the negations of P-literals.

A formal proof of the black and white literals principle is given in Lemma 4.3. Figure 1 illustrates this principle. White circles ($\circ$) denote P-literals, black circles ($\bullet$) denote their negations, circles with dots ($\odot$) denote other literals (i.e., literals which are neither P-literals nor the negations of P-literals; note that the negation of such literal is again neither P-literal nor the negation of a P-literal). Columns correspond to clauses of the formula. The figure contains two formulas, the first one satisfies the condition (1), the second one satisfies the condition (2).

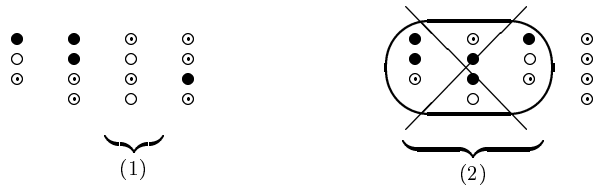

Figure 1: Two alternatives of the black and white literals principle.

This principle is the key point of our algorithm corresponding to the upper bound $p(L)2^{0.30897K}$. In this algorithm we use the presence of clauses that contain a P-literal and do not contain the negations of other P-literals for a certain property P. The black and white literals principle is a kind of insurance: it guarantees that if we cannot find a required clause, then we can replace F by a simpler formula, or F does not contain any P-literals at all. This principle is one of re-formulations of the following simple property: for any partial assignment A (i.e., a set of literals which does not contain simultaneously x and $\overline{x}$ for any variable x) for a formula F in CNF, either there is a clause in F containing only literals from A, or setting the values of all these literals to $False$ does not change the satisfiability of F.

Another known re-formulations of this property are the *generalized sign principle* [16] and the *autarkness principle* [20, 22, 17] (see also [16]). A comprehensive study of the autarkness principle can be found in [15].

The use of the black and white literals principle leads to two new bounds for SAT presented in this paper and also to several upper bounds for the satisfiability problem for formulas in CNF–(1,∞) [8] (a formula is in CNF–(1,∞) if each literal occurs in it positively at most once; the satisfiability problem for these formulas is $\mathcal{NP}$-complete).

In Sect. 2 we give basic definitions. Section 3 contains the technique that allows us to estimate the size of a branching tree. In Sect. 4 we explain transformation rules that we use in our algorithms. In Sect. 5 and Sect. 6 we describe the algorithms having the upper bounds $p(L)2^{0.30897K}$ and $p(L)2^{0.10299L}$ respectively, and the corresponding proofs.

2 Basic definitions.

Let V be a set of Boolean variables. The negation of a variable v is denoted by $\overline{v}$. Given a set U, we denote $\overline{U} = \{\overline{u} \,|\, u \in U\}$. *Literals* are the members of the set $W = V \cup \overline{V}$. *Positive literals* are the members of the set V. *Negative literals* are their negations. If w denotes a negative literal $\overline{v}$, then $\overline{w}$ denotes the variable v. A *clause* is a finite set of literals that does not contain simultaneously any variable together with its negation. The empty clause is interpreted as $False$. A *formula in CNF* (*CNF-formula*) is a finite set of clauses. The empty formula is interpreted as $True$. The *length of a clause* is its cardinality, the *length of a formula* is the sum of the lengths of all its clauses. The length of a clause C is denoted by $|C|$. A *k-clause* is a clause of the length k. A *k^+-clause* is a clause of the length at least k. A *k^--clause* is a clause of the length at most k. If we say that *a literal v occurs* in a clause or in a formula, we mean that this clause or this formula contains the literal v. However, if we say that *a variable v occurs* in a clause or in a formula, we mean that this clause or this formula contains the literal v, or it contains the literal $\overline{v}$.

An *assignment* is a finite subset of W that does not contain any variable together with its negation. Informally speaking, if an assignment I contains a literal w, it means that w has the value $True$ in I. To obtain $F[I]$ from F and an assignment $I = \{w_1, \ldots, w_s\}$, we

- remove from F all clauses containing the literals w_i,
- delete all occurrences of the literals $\overline{w}_i$ from the other clauses.

For short, we write $F[w_1, \ldots, w_s]$ instead of $F[\{w_1, \ldots, w_s\}]$.

An assignment I is *satisfying* for a formula F if $F[I]$ is the empty formula. A formula is *satisfiable* if there exists a satisfying assignment for it. We say that two formulas F and G are *equi-satisfiable* if both are satisfiable, or both are unsatisfiable.

Let w be a literal occurring in a formula F. This literal is a *i-literal* if F contains exactly i occurrences of w. It is a *(i,j)-literal* if F contains exactly i occurrences of w and exactly j occurrences of $\overline{w}$. It is a *(i,j^+)-literal* if F contains exactly i occurrences of w and at least j occurrences of $\overline{w}$. It is a *(i,j^-)-literal* if F contains exactly i occurrences of w and at most j occurrences of $\overline{w}$. Similarly, we define *(i^+,j^+)-literals*, *(i^+,j)-literals* etc.

We denote the number of occurrences of a literal w in a formula F by $\#_w(F)$. We denote the sum of the lengths of the clauses containing w by $\diamondsuit_w(F)$. When the meaning of F is clear from the context, we omit F. A literal w is a *$\diamondsuit i$-literal* if $\diamondsuit_w = i$. Similarly, we define *$\diamondsuit i^+$-literals*, *$(\diamondsuit i,\diamondsuit j)$-literals*, *$(\diamondsuit i^-,\diamondsuit j^+)$-literals* etc.

3 Estimation of the size of a branching tree.

Kullmann and Luckhardt introduced in [16] a notion of a branching tree. It is intended for estimating the time complexity of splitting algorithms, since a tree of formulas which such an algorithm splits, is a branching tree. One can consider an execution of a splitting algorithm as a tree whose nodes are labelled with CNF-formulas such that if some node is labelled with a CNF-formula F, then its sons are labelled with (simplified) formulas $F[I_1], F[I_2], \ldots, F[I_s]$ for some assignments $I_1, I_2, \ldots, I_s$.

Suppose we have a tree whose nodes are labelled with formulas in CNF. To each formula F we attach a non-negative integer $\mu(F)$ (*complexity of* F). The tree is a *branching tree* if, for each node, the complexity of the formula labelling this node is strictly greater than the complexity of each of the formulas labelling its sons. In this paper we use

(1) $\mu(F) = \mathcal{N}(F)$ is the number of variables in F;

(2) $\mu(F) = \mathcal{K}(F)$ is the number of clauses in F;

(3) $\mu(F) = \mathcal{L}(F)$ is the length of F.

We prove two upper bounds, w.r.t. $\mathcal{K}$ and w.r.t. $\mathcal{L}$. However, in this section we do not fix any concrete measure of complexity.

Let us consider a node in our tree labelled with a formula F_0. Suppose its sons are labelled with formulas F_1, F_2, ..., F_m. 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 *characteristic polynomial* of a branching vector $\vec{t}$ is defined by

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

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 exactly one positive root. We denote this root by $\tau(\vec{t}\,)$ and call it the *branching number*. We suppose $\tau(\emptyset) = 1$ for leaves. We omit one pair of parentheses and write $\tau(t_1, t_2, \ldots, t_m)$ instead of $\tau((t_1, t_2, \ldots, t_m))$.

Example 3.1 For example,

$\tau(1, 1) = 2$;

$\tau(1, 2) = 1.61803... = 2^{0.69424...}$ is the golden ratio;

$\tau(6, 7, 6, 7) = 1.23881... = 2^{0.30896...}$;

$\tau(10, 10) = 1.07177... = 2^{0.1}$.

The *branching number of a tree* T is the largest of the branching numbers $\tau(\vec{t}\,)$ of its nodes. We denote it by $\tau_{max,T}$. The following lemma proved by Kullmann and Luckhardt allows us to estimate the number of leaves in a branching tree using its branching number.

Lemma 3.1 (Kullmann, Luckhardt, [16]) *Let T be a branching tree, let its root be labelled with a formula F_0. Then the number of leaves in T does not exceed $(\tau_{max,T})^{\mu(F_0)}$.*

This lemma already allows us to estimate the running time of a splitting algorithm if we know the branching number of the splitting tree and the algorithm processes each its leaf in a polynomial time. However, our algorithm corresponding to the upper bound $p(L)\tau(6,7,6,7)^{L/3}$ calls the algorithm corresponding to the upper bound $p(L)\tau(6,7,6,7)^{K}$ as a subroutine and thus processes some of the leaves in an exponential time. To estimate the overall running time, we use the following simple generalization of Lemma 3.1.

Lemma 3.2 [8] *Let T be a branching tree, let its root be labelled with a formula F_0. Let G_l denote the object labelling a leaf l of the tree T. Let $f(m) = \alpha\lambda^m$, where $\lambda > 1$, $\alpha \geq 0$. Then*

$$\sum_{l \text{ is a leaf of } T} f(\mu(G_l)) \leq \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)}.$$

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

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

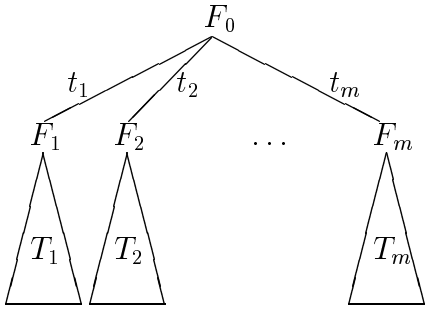

Figure 2: A splitting tree

$$
\begin{aligned}
&\leq \alpha \sum_{j=1}^{s} (\max(\lambda, \tau_{max,T}))^{\mu(F_0)-t_j} && (\text{since } \tau_{max,T_j} \leq \tau_{max,T}) \\
&= \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)} \cdot \sum_{j=1}^{s} (\max(\lambda, \tau_{max,T}))^{-t_j} \\
&= \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)}(1 - h_{\vec{t}}\,(\max(\lambda, \tau_{max,T}))) && (\text{by definition of } h_{\vec{t}}\,) \\
&\leq \alpha(\max(\lambda, \tau_{max,T}))^{\mu(F_0)} && (\text{by monotonicity of } h_{\vec{t}}\,)
\end{aligned}
$$

□

Now if we know the branching number of the tree corresponding to a splitting algorithm, then we can estimate its running time. We explicitly require our algorithms to perform splittings with branching numbers not greater than we wish (and we only need to prove that there always exists a splitting satisfying this condition). For this purpose, the algorithm has to compare branching numbers corresponding to different vectors. Of course, this can be done just by the examination of a constant number of cases and the use of the monotonicity of τ. However, a more general statement holds.

Lemma 3.3 (Kullmann, Luckhardt, [16]) *Let m, k be natural constants, $x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_k$ be positive rational numbers. The problem whether $\tau(x_1, x_2, \ldots, x_m)$ is not greater than $\tau(y_1, y_2, \ldots, y_k)$ is solvable in time polynomial of $\max(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_k)$.*

In the following, when estimating the running time of our algorithms, we use frequently inequalities like $\tau(7, 15) < \tau(5, 17)$ or $\tau(5, 17) < \tau(6, 7, 6, 7)^{1/3}$ without proofs. One can check these inequalities by approximate calculation of the branching numbers. However, there are several simple observations that may help to prove some of this inequalities easier.

Lemma 3.4 (Kullmann, Luckhardt, [16])

(1) Permutations of the components of a branching vector do not affect the corresponding branching number.

(2) The branching number strictly decreases as one or more components of the branching vector increase.

(3) Let $0 < \alpha < t_1$. Then $\tau(t_1, t_2, t_3 \ldots, t_m) \gtreqless \tau(t_1 - \alpha, t_2 + \alpha, t_2, \ldots, t_m)$ iff $t_1 \gtreqless t_2 + \alpha$.

4 Transformation rules.

In this section F denotes a formula in CNF.

Lemmas 3.1 and 3.2 allows us to take into consideration only the differences between the complexity of an input formula and the complexities of the formulas obtained from it by splitting. The higher these differences are, the smaller the number of leaves is. Thus, to obtain a good algorithm, we should reduce as much as possible the complexities of the

formulas obtained by splitting. Here we explain transformation rules that allow us to do it. More precisely, we explain that in certain cases we can find a simpler formula equi-satisfiable with F. In the following we use these rules so that they never increase the parameter under consideration (the number of clauses in a formula or the length of a formula).

Elimination of 1-clauses. If F contains a 1-clause $\{a\}$, then the formulas F and $F[a]$ are equi-satisfiable since all assignments that contain $\overline{a}$ are unsatisfying.

Subsumption. If F contains two clauses C and D such that $C \subseteq D$, then F and $F \setminus \{D\}$ are equi-satisfiable since each assignment that satisfies the clause C, satisfies also the clause D.

Resolution with subsumption. Suppose we are given a literal a and clauses C and D such that a is the only literal satisfying both conditions $a \in C$ and $\overline{a} \in D$. In this case, the clause $(C \cup D) \setminus \{a, \overline{a}\}$ is called the *resolvent by the literal* a of the clauses C and D. We denote it by $\mathcal{R}(C, D)$.

Let now F contain such clauses C and D. It is clear that adding $\mathcal{R}(C, D)$ to the formula does not change its satisfiability. However, it increases its size. To avoid this effect, we use this rule only if $\mathcal{R}(C, D) \subseteq D$. In this case we reduce the satisfiability problem for the formula F to the satisfiability problem for $(F \setminus \{D\}) \cup \{\mathcal{R}(C, D)\}$.

Elimination of a variable by resolution. Given a literal a, we construct the formula $DP_a(F)$ by

- adding to F all resolvents by a;
- removing from F all clauses containing a or $\overline{a}$.

Lemma 4.1 (Davis, Putnam, [6]) *The formulas F and $DP_a(F)$ are equi-satisfiable.*

This transformation can increase the size of a formula or/and the number of clauses in it, but we use this rule only if it does not increase the parameter under consideration (the number of clauses in a formula or the length of a formula). Note that in particular, this transformation does not increase any of the parameters of F if a is a pure literal, and thus it eliminates pure literals.

Elimination of blocked clauses. The clause C is *blocked* for a literal a w.r.t. F if C contains the literal a, and the literal $\overline{a}$ occurs only in the clauses of F that contain the negation of at least one of the literals occurring in $C \setminus \{a\}$. (Note that F may or may not contain C.) In other words, there are no resolvents by a of the clause C and any other clause of the formula F. For a CNF-formula F and a literal a occurring in it, we define the assignment

$$\mathcal{I}(a, F) = \{a\} \cup \{x \in W \setminus \{a, \overline{a}\} \mid \text{the clause } \{\overline{a}, x\} \text{ is blocked for } \overline{a} \text{ w.r.t. } F\}.$$

When the meaning of F is clear from the context, we omit F and write $\mathcal{I}(a)$.

The notion of a blocked clause was introduced and investigated by Kullmann in [13, 14]. We use the following two facts about blocked clauses.

Lemma 4.2 (Kullmann, [13, 14])

(1) If a clause C is blocked for a literal a w.r.t. F, then F, $F \setminus \{C\}$ and $F \cup \{C\}$ are equi-satisfiable.

(2) Given a literal a, the formula F is satisfiable iff at least one of the formulas $F[\overline{a}]$ and $F[\mathcal{I}(a)]$ is satisfiable.

Application of the black and white literals principle. Let P be a binary relation between literals and formulas in CNF, such that for a variable v and a formula F, at most one of $P(v, F)$ and $P(\overline{v}, F)$ holds.

Lemma 4.3 *Suppose that each clause of F that contains a literal w satisfying $P(w, F)$ contains also at least one literal b satisfying $P(\overline{b}, F)$. Then F and $F[\{l \mid P(\overline{l}, F)\}]$ are equi-satisfiable.*

Proof. This is a particular case of the autarkness principle [20, 22, 17, 16]. We denote $G = F[\{l \mid P(\overline{l}, F)\}]$. Suppose G is satisfiable. Consider a satisfying assignment I for the formula G. It is clear that the assignment $I \cup \{l \mid P(\overline{l}, F)\}$ satisfies F. On the other hand, $G \setminus F = \emptyset$, hence, each assignment satisfying F satisfies also G. □

5 A bound w.r.t. the number of clauses.

In this section we present Algorithm 5.1 which checks satisfiability of formula F in the time $p(\mathcal{L}(F))2^{0.30897\mathcal{K}(F)}$ (where p is a polynomial). This algorithm has two subroutines, Function $REDUCE_K$ and Function $SPLIT_K$. Function $REDUCE_K$ simplifies the input formula using the transformation rules (see Sect. 4). Function $SPLIT_K$ is intended for reducing the satisfiability problem for the input formula to the satisfiability problem for several simpler formulas. The execution of this algorithm can be viewed as follows: Function $REDUCE_K$ simplifies the input formula, then $SPLIT_K$ splits it into several formulas, $REDUCE_K$ simplifies each of these, and so on.

In the following, we denote by $REDUCE_K(F)$ the formula that Function $REDUCE_K$ outputs on input F. Similarly we define $SPLIT_K(F)$ etc.

Function $REDUCE_K$.

Input: *A formula F in CNF.*
Output: *A (simplified) formula in CNF.*

Method.

(KR1) **Elimination of 1-clauses.** If F contains a 1-clause $C = \{a\}$, then $F := F[a]$. Repeat this step while F contains 1-clauses.

(KR2) **Application of the black and white literals principle.** If each clause C in F that contains a (2,3$^+$)-literal, contains also a (3$^+$,2)-literal, then

$$F := F[\{a \mid a \text{ is a } (3^+,2)\text{-literal}\}].$$

(KR3) **Elimination of a variable by resolution.** Choose a literal a such that a or $\overline{a}$ occurs in F and $\Delta = \mathcal{K}(F) - \mathcal{K}(DP_a(F))$ is maximal. If there are several such literals, then choose a literal with $\#_a$ minimal. If $\Delta \geq 0$, then $F := DP_a(F)$. Repeat this step while F satisfies this condition.

(KR4) If F has been changed at steps (KR1)–(KR3), then go to step (KR1), otherwise return F.

□

Function $SPLIT_K$**.**

Input: *A formula F in CNF.*
Output: *If F is satisfiable, then $True$, otherwise $False$.*

Method.

(KS1) **The empty formula.** If $F = \emptyset$, then return $True$.

(KS2) **Formula containing the empty clause.** If $\emptyset \in F$, then return $False$.

(KS3) **Splitting into two subproblems.** For each literal a occurring in F, construct two formulas

$$F_1 = REDUCE_K(F[a]),$$
$$F_2 = REDUCE_K(F[\overline{a}]).$$

If there exists a literal a such that $\tau(\mathcal{K}(F) - \mathcal{K}(F_1), \mathcal{K}(F) - \mathcal{K}(F_2)) \leq \tau(6,7,6,7)$, then choose the formulas F_1 and F_2 corresponding to this literal. If $SPLIT_K$ returns $True$ for at least one of these, then return $True$, otherwise return $False$.

(KS4) **Splitting into four subproblems.** Choose a literal a occurring in F. For each two literals b and c occurring in $F[a]$ and $F[\overline{a}]$ respectively, construct four formulas

$$F_{11} = REDUCE_K(F[a,b]),$$
$$F_{12} = REDUCE_K(F[a,\overline{b}]),$$
$$F_{21} = REDUCE_K(F[\overline{a},c]),$$
$$F_{22} = REDUCE_K(F[\overline{a},\overline{c}]).$$

If there exist literals b and c such that

$$\tau(\mathcal{K}(F) - \mathcal{K}(F_{11}), \mathcal{K}(F) - \mathcal{K}(F_{12}), \mathcal{K}(F) - \mathcal{K}(F_{21}), \mathcal{K}(F) - \mathcal{K}(F_{22})) \leq \tau(6,7,6,7),$$

then choose the formulas F_{11}, F_{12}, F_{21} and F_{22} corresponding to these literals. If $SPLIT_K$ returns $True$ for at least one of these, then return $True$, otherwise return $False$.

□

Algorithm 5.1.

Input: *A formula F in CNF.*
Output: *If F is satisfiable, then $True$, otherwise $False$.*

Method.

(KM1) Return $SPLIT_K(REDUCE_K(F))$.

□

In Sect. 4 we explained that none of the steps of $REDUCE_K$ changes the satisfiability of the formula. The steps (KR1)–(KR4) cannot be repeated more than $\mathcal{K}(F) + \mathcal{N}(F)$ times since none of them increases the number of clauses or the number of variables, and during each iteration (KR1)–(KR4)–(KR1) at least one of these quantities decreases. Each of these steps takes a polynomial time. Thus, $REDUCE_K$ does not change the satisfiability of the formula and returns the answer in a polynomial time.

We now construct a tree which reflects the behaviour of Algorithm 5.1 (together with Functions $REDUCE_K$ and $SPLIT_K$). Its internal nodes are labelled with formulas that Algorithm 5.1 splits at steps (KS3) and (KS4). Its leaves are labelled with formulas that satisfy the conditions of the steps (KS1) and (KS2). Each internal node labelled with F has two sons labelled with F_1 and F_2 or four sons labelled with F_{11}, F_{12}, F_{21} and F_{22}. Since the conditions of the steps (KS3) and (KS4) guarantee that the corresponding branching numbers do not exceed $\tau(6,7,6,7)$, by Lemma 3.1 the number of leaves in the tree is at most $\tau(6,7,6,7)^{\mathcal{K}(F)}$, where F is the input formula. Thus, the Algorithm 5.1 returns the answer in the time $p(\mathcal{L}(F))\tau(6,7,6,7)^{\mathcal{K}(F)}$, where p is some polynomial.

It remains to prove that Algorithm 5.1 performs correctly, i.e., each formula in CNF reduced by $REDUCE_K$ satisfies at least one of the conditions of the steps (KS1)–(KS4). To prove this statement, we need two simple lemmas concerning the output of $REDUCE_K$.

Lemma 5.1 *Let F_1 be the value of F at step (KR1) of Function $REDUCE_K$ at some point of time (maybe after eliminating several 1-clauses), F_2 be the corresponding output formula of Function $REDUCE_K$, d and e be literals occurring in F_1.*

(1) If d is a $(1^-,1^+)$-literal, then $\mathcal{K}(F_2) \leq \mathcal{K}(F_1) - 1$.

(2) If d and e are $(1^-,1^+)$-literals, $\#_d + \#_e + \#_{\overline{d}} + \#_{\overline{e}} \geq 3$ and there are no clauses in F that contain d and e simultaneously, then $\mathcal{K}(F_2) \leq \mathcal{K}(F_1) - 2$.

Proof. (1). If $REDUCE_K$ modifies once more the formula at steps (KR1) or (KR2), then at least one clause will be deleted from it. Otherwise, at least one clause will be eliminated at step (KR3) since $\mathcal{K}(F_1) - \mathcal{K}(DP_d(F_1)) \geq 1$.
(2). W.l.o.g we suppose that F_1 contains at most one 1-clause, this clause (if any) contains a 1-literal and after eliminating this clause the formula F_1 does not contain 1-clauses at all. If F_1 contains exactly one 1-clause, then it will be eliminated at step (KR1), and one

more clause will be eliminated by (1) since at least one of the literals d and e remains a $(1^-,1^+)$-literal.

Now let F_1 contain no 1-clauses. If $REDUCE_K$ modifies this formula at step (KR2), then at least one clause, i.e., at least two occurrences will be eliminated.

Now F has the value F_1 before the step (KR3). W.l.o.g. we suppose that $\Delta = 1$ in terms of the step (KR3). At step (KR3) a 1^--literal will be chosen. After the first application of DP, at least one of the literals d and e remains a 1^--literal and its negation or the literal itself remains in the formula. Hence, by (1) at least one more clause will be eliminated. □

Lemma 5.2 *Let F be a formula in CNF. Then the formula $REDUCE_K(F)$ does not contain 1-clauses, 1^--literals and (2,2)-literals.*

Proof. Function $REDUCE_K$ eliminates 1-clauses at step (KR1). If d is a 1^--literal or a (2,2)-literal, then $\mathcal{K}(DP_d(F)) \leq \mathcal{K}(F)$. Function $REDUCE_K$ eliminates such literals at step (KR3). □

Theorem 5.1 *Algorithm 5.1 performs correctly and stops in the time $p(L)\tau(6,7,6,7)^K < p(L)2^{0.30897K}$, where L is the length of the input formula, K is the number of clauses in it, and p is a polynomial.*

Proof. We have shown above that it suffices to prove that each formula F in CNF reduced by $REDUCE_K$ satisfies at least one of the conditions of the steps (KS1)–(KS4). Suppose F does not satisfy the conditions of the steps (KS1)–(KS4). We now consider all possible cases. We prove that each of this cases is impossible. All symbols have the same meaning as they have at the corresponding steps of Function $SPLIT_K$. For $i,j = 1,2$, we denote

$$\begin{array}{ll} \Delta_i = \mathcal{K}(F) - \mathcal{K}(F_i), & \Delta_{ij} = \mathcal{K}(F) - \mathcal{K}(F_{ij}), \\ \mathrm{B}_1 = \mathcal{K}(F) - \mathcal{K}(F[a]), & \Gamma_1 = \mathcal{K}(F[a]) - \mathcal{K}(F_1), \\ \mathrm{B}_2 = \mathcal{K}(F) - \mathcal{K}(F[\overline{a}]), & \Gamma_2 = \mathcal{K}(F[\overline{a}]) - \mathcal{K}(F_2), \\ \Gamma_{1j} = \mathcal{K}(F[a]) - \mathcal{K}(F_{1j}), & \Gamma_{2j} = \mathcal{K}(F[\overline{a}]) - \mathcal{K}(F_{2j}). \end{array}$$

The key observation is that if the formula F contains few "frequent" variables, then after the transformation of F into $F[a], F[\overline{a}]$ for some literal a we can apply Lemma 5.1. We now prove this more formally.

Case 1: The formula F contains a $(3^+,4^+)$-literal a.

All clauses of the formula F containing the literal a disappear in $F[a]$, all its clauses containing the literal $\overline{a}$ disappear in $F[\overline{a}]$. Since a is a $(3^+,4^+)$-literal, $\mathrm{B}_1 \geq 3$, $\mathrm{B}_2 \geq 4$. Thus, in terms of the step (KS3), $\tau(\Delta_1,\Delta_2) \leq \tau(3,4) < \tau(6,7,6,7)$, i.e., the condition of the step (KS3) is satisfied.

Case 2: The formula F contains a clause that contains a (3,3)-literal a and a $(2,3^+)$-literal b.

All three clauses of the formula F containing a disappear in $F[a]$, all three clauses containing $\overline{a}$ disappear in $F[\overline{a}]$. In addition, b becomes a $(1^-,1^+)$-literal in $F[a]$. Thus, in terms of the step (KS3) we have $\Gamma_1 \geq 1$ by Lemma 5.1. Hence, $\Delta_1 = \mathrm{B}_1 + \Gamma_1 \geq 4$, $\Delta_2 \geq 3$, i.e., the condition of the step (KS3) is satisfied.

CASE 3: The formula F contains a 2-clause $C = \{a, b\}$ consisting of 2-literals.

All clauses of the formula F containing a disappear in $F[a]$, all its clauses containing $\overline{a}$ disappear in $F[\overline{a}]$. By Lemma 5.2, a is a (2,3$^+$)-literal. Thus, $\mathrm{B}_1 \geq 2$, $\mathrm{B}_2 \geq 3$. Similarly to Case 2, in terms of the step (KS3) we have $\Gamma_1 \geq 1$. In addition, C becomes a 1-clause in $F[\overline{a}]$ and will be eliminated at step (KR1), i.e. $\Gamma_2 \geq 1$. We now have $\Delta_1 = \mathrm{B}_1 + \Gamma_1 \geq 3$, $\Delta_2 = \mathrm{B}_2 + \Gamma_2 \geq 4$, i.e., the condition of the step (KS3) is satisfied.

CASE 4: The formula F contains a 3$^+$-clause $C = \{a_1, a_2, \ldots, a_s\}$ consisting of 2-literals.

Let $i \in \{1, 2, \ldots, s\}$. All clauses of the formula F containing a_i disappear in $F[a_i]$, all its clauses containing $\overline{a_i}$ disappear in $F[\overline{a_i}]$. By Lemma 5.2, a_i is a (2,3$^+$)-literal. Thus, $\mathcal{K}(F) - \mathcal{K}(F[a_i]) \geq 2$, $\mathcal{K}(F) - \mathcal{K}(F[\overline{a_i}]) \geq 3$. In addition, a_2 and a_3 become (1$^-$,1$^+$)-literals in $F[a_1]$. If C is the only clause that contains a_2 and a_3 simultaneously, then by Lemma 5.1 we have $\mathcal{K}(F[a_1]) - \mathcal{K}(\mathit{REDUCE}_K(F[a_1]))) \geq 2$. Otherwise, a_3 becomes a (0,3$^+$)-literal in $F[a_2]$, i.e. $\mathcal{K}(F[a_2]) - \mathcal{K}(\mathit{REDUCE}_K(F[a_2])) \geq 2$. Depending on which of these two alternatives takes place, we choose a_1 or a_2 as a in terms of the step (KS3). We now have $\Delta_1 = \mathrm{B}_1 + \Gamma_1 \geq 4$, $\Delta_2 \geq \mathrm{B}_2 \geq 3$, i.e., the condition of the step (KS3) is satisfied.

CASE 5: The conditions of Cases 1–4 are not satisfied.

Since the step (KR2) does not change F, it contains only 3-literals. Since F does not satisfy the condition of the step (KS3), for each literal a,

$$\mathcal{K}(F) = \mathcal{K}(F[a]) + 3 = \mathcal{K}(\mathit{REDUCE}_K(F[a])) + 3. \tag{5.1}$$

Let a be a literal occurring in F. We now prove that there exist literals b and c such that the condition of the step (KS4) is satisfied. We consider three sub-cases (the first two of them are similar to Cases 2–4).

CASE 5.1: There exists a clause $C = \{a_1, a_2, \ldots, a_s\}$ in $F[a]$, such that a_1 is a (3,3)-literal and a_2 is a 2-literal.

We suppose $b = a_1$ in terms of the step (KS4). Similarly to Case 2, $\Gamma_{11} \geq 4$, $\Gamma_{12} \geq 3$.

CASE 5.2: There exists a clause $C = \{a_1, a_2, \ldots, a_s\}$ in $F[a]$, such that a_1 is a (2,3)-literal and $a_2, \ldots, a_s$ are 2-literals.

We suppose $b = a_1$ in terms of the step (KS4). Similarly to Cases 3–4, $\Gamma_{11} \geq 3$, $\Gamma_{12} \geq 4$, or $\Gamma_{11} \geq 4$, $\Gamma_{12} \geq 3$. (Note that by (5.1) the literals a_i and a_j cannot occur together in two clauses.)

CASE 5.3: The conditions of Cases 5.1–5.2 are not satisfied.

Since by (5.1) the steps (KR1)–(KR3) do not reduce the number of clauses in the formula $F[a]$ and the conditions of Cases 5.1–5.2 are not satisfied, this formula consists of clauses that contain only (2,2)-literals and clauses that contain only (3,3)-literals. Since exactly three clauses of the formula F disappear in $F[a]$, all occurrences of any (3,3)-literal cannot disappear in $F[a]$. Hence, it contains at least one (2,2)-literal b_1. Let b_1 be a 2-literal. The formula F does not contain 2-clauses by (5.1). Similarly, it cannot contain two clauses, each containing the literals $\overline{a}$ and b_1 simultaneously. Thus, $F[a]$ contains a 3$^+$-clause $C = \{b_1, b_2, \ldots, b_s\}$, where $b_1, b_2, \ldots, b_s$ are 2-literals.

We choose the literal b_1 as b. Both clauses of the formula $F[a]$ that contain $\overline{b_1}$ disappear in $F[a,\overline{b_1}]$. In addition, these clauses contain only (2,2)-literals which become (1^-,1^+)-literals. Thus, by Lemma 5.1

$$\mathcal{K}(F[a]) - \mathcal{K}(REDUCE_K(F[a,\overline{b_1}])) \geq 3.$$

On the other hand, two clauses of $F[a]$ containing b_1 disappear in $F[a,b_1]$, and two more clauses disappear by Lemma 5.1 since b_2 and b_3 become (1^-,1^+)-literals. We note that by (5.1) the formula F contains only one clause that contains b_2 and b_3 simultaneously. Thus,

$$\mathcal{K}(F[a]) - \mathcal{K}(REDUCE_K(F[a,b_1])) \geq 4.$$

We now have that in each of Cases 5.1–5.3 there exists a literal b in $F[a]$ such that $\Delta_{11} = \mathrm{B}_1 + \Gamma_{11} \geq 6$, $\Delta_{12} = \mathrm{B}_1 + \Gamma_{12} \geq 7$. A literal c can be chosen similarly. Hence, the condition of the step (KS4) is satisfied. □

6 A bound w.r.t. the length of a formula.

In this section we present Algorithm 6.1 which checks satisfiability of formula F in the time $p(\mathcal{L}(F))2^{0.10299\mathcal{L}(F)}$ (where p is a polynomial). As in the previous section, we define two subroutines, Function $REDUCE_L$ and Function $SPLIT_L$. We use them similarly to $REDUCE_K$ and $SPLIT_K$: Function $REDUCE_L$ simplifies the input formula, then $SPLIT_L$ splits it into several formulas, $REDUCE_L$ simplifies each of these, and so on.

Function $REDUCE_L$**.**

Input: A formula F in CNF.
Output: A (simplified) formula in CNF.

Method.

(LR1) **Elimination of 1-clauses.** If F contains a 1-clause $C = \{a\}$, then $F := F[a]$. Repeat this step while F contains such a clause.

(LR2) **Subsumption.** If F contains two clauses C and D such that $C \subseteq D$, then $F := F \setminus \{D\}$. Repeat this step while F contains such clauses.

(LR3) **Elimination of blocked clauses.** If F contains a blocked clause C, then $F := F \setminus \{C\}$. Repeat this step while F contains blocked clauses.

(LR4) **Resolution with subsumption.** If F contains two clauses C and D such that $\mathcal{R}(C,D) \subseteq D$, then $F := (F \setminus \{D\}) \cup \{\mathcal{R}(C,D)\}$. Repeat this step while F contains such clauses.

(LR5) **Elimination of a variable by resolution.** Choose a literal a such that a or $\overline{a}$ occurs in F and $\Delta = \mathcal{L}(F) - \mathcal{L}(DP_a(F))$ is maximal. If $\Delta \geq 0$, then $F := DP_a(F)$. Repeat this step while F satisfies this condition.

(LR6) If F has been changed at steps (LR1)–(LR5), then go to step (LR1), otherwise return F.

□

Function $SPLIT_L$**.**

Input: A formula F in CNF.
Output: If F is satisfiable, then $True$, otherwise $False$.

Method.

(LS1) **The empty formula.** If $F = \emptyset$, then return $True$.

(LS2) **Formula containing the empty clause.** If $\emptyset \in F$, then return $False$.

(LS3) **Formula containing no 2-clauses.** If F does not contain 2-clauses, apply Algorithm 5.1 to F and return its answer.

(LS4) **Splitting into two subproblems.** For each literal a occurring in F, construct two formulas

$$F_1 = REDUCE_L(F[\mathcal{I}(a)]),$$
$$F_2 = REDUCE_L(F[\overline{a}]).$$

If there exists a literal a such that $\tau(\mathcal{L}(F) - \mathcal{L}(F_1), \mathcal{L}(F) - \mathcal{L}(F_2)) \leq \tau(5, 17)$, then choose the formulas F_1 and F_2 corresponding to this literal. If $SPLIT_L$ returns $True$ for at least one of these, then return $True$, otherwise return $False$.

□

Algorithm 6.1.

Input: A formula F in CNF.
Output: If F is satisfiable, then $True$, otherwise $False$.

Method.

(LM1) Return $SPLIT_L(REDUCE_L(F))$.

□

Similarly to $REDUCE_K$, Function $REDUCE_L$ does not change the satisfiability of the formula and returns the answer in a polynomial time.

We now construct a tree which reflects the behaviour of Algorithm 6.1 (together with Functions $REDUCE_L$ and $SPLIT_L$). Its leaves are labelled with formulas that satisfy the conditions of the steps (LS1)–(LS3). Algorithm 6.1 processes in a polynomial time formulas satisfying the conditions of the steps (LS1) and (LS2), and passes formulas satisfying the

condition of the step (LS3) to Algorithm 5.1 which processes such a formula F in the time $p(\mathcal{L}(F))\tau(6,7,6,7)^{\mathcal{K}(F)} \leq p(\mathcal{L}(F))\tau(6,7,6,7)^{\mathcal{L}(F)/3}$, where p is a polynomial (note that F contains no 2^--clauses since 1-clauses are eliminated at step (LR1)).

Internal nodes of our tree are labelled with formulas that Algorithm 6.1 splits at step (LS4). Each internal node labelled with F has two sons labelled with F_1 and F_2. Since the condition of the step (LS4) guarantees that the corresponding branching number does not exceed $\tau(5,17) < \tau(6,7,6,7)^{1/3}$, by Lemma 3.2 the running time of Algorithm 6.1 is upper bounded by $q(\mathcal{L}(G))\tau(6,7,6,7)^{\mathcal{L}(G)/3}$ for input formula G, where q is a polynomial.

It remains to prove that Algorithm 6.1 performs correctly, i.e., each formula in CNF reduced by $REDUCE_L$ satisfies at least one of the conditions of the steps (LS1)–(LS4). We need three simple lemmas concerning the output of $REDUCE_L$ to prove this statement.

Lemma 6.1 *Let F be a formula in CNF, d be a literal occurring in it.*

(1) If d is a $(1,2^-)$-literal occurring in a 3^--clause, then $\mathcal{L}(DP_d(F)) \leq \mathcal{L}(F) - 1$.

(2) If d is a $(1,1^+)$-literal occurring in a 2^--clause, then $\mathcal{L}(DP_d(F)) \leq \mathcal{L}(F) - 2$.

(3) If d is a $(1,1)$-literal, then $\mathcal{L}(DP_d(F)) \leq \mathcal{L}(F) - 2$.

Proof. By straightforward calculations. □

Lemma 6.2 *Let F_1 be the value of F at one of the steps (LR1)–(LR5) of Function $REDUCE_L$, F_2 be the corresponding output formula of Function $REDUCE_L$, d be a literal occurring in F_1.*

(1) If d is a $(1,2^-)$-literal occurring in a 3^--clause, then $\mathcal{L}(F_2) \leq \mathcal{L}(F_1) - 1$.

(2) If d is a $(1,1^+)$-literal occurring in a 2^--clause, then $\mathcal{L}(F_2) \leq \mathcal{L}(F_1) - 2$.

Proof. (1). Note that any change of the formula at steps (LR1)–(LR4) is a removal of some clauses and/or a change of other clauses by their subsets, and thus results in decreasing the length of the formula by at least one occurrence. If the formula does not change any more before the step (LR5), then at least one occurrence will be removed by Lemma 6.1(1).

(2). Suppose that after F gets the value F_1, Function $REDUCE_L$ modifies the formula before the step (LR4). This modifying cannot result in increasing the length of a clause or in increasing the number of occurrences of a literal. Moreover, if the formula changes at these steps, at least one occurrence is removed. If only one occurrence is removed, then at least one of the literals d, $\overline{d}$ remains in the formula. If this is the occurrence of d, then by Lemma 6.1(1) at least one more occurrence will be removed at step (LR5), and (2) holds. Otherwise, $\overline{d}$ becomes a pure literal and will be removed at step (LR5), thus (2) holds again.

Suppose now that $REDUCE_L$ does not modify the formula before the step (LR5). Then, (2) follows from Lemma 6.1(2). □

We use the following simple properties of the formula reduced by Function $REDUCE_L$ in the proof of 6.1 without explicit mentioning.

Lemma 6.3 *Let $F = REDUCE_L(G)$ for some formula G. Then*

(1) F does not satisfy any of the conditions of the steps (LR1)–(LR5);

(2) there are no 1-clauses in F;

(3) there are no pure literals and (1,1)-literals in F;

(4) there are no 2-clauses in F containing a 1-literal;

(5) for any literal d occurring in F, $\diamondsuit_d(F) \geq max(3, 2\#_d)$;

(6) for any 1-literal d occurring in F, $\diamondsuit_d + \#_{\overline{d}} \geq 7$;

(7) for any literal d occurring in F, $\diamondsuit_d + \#_{\overline{d}} \geq 5$.

Proof. (1) is trivial; (2)–(7) since F does not satisfy the conditions of the step (LR1)and (LR5); see also Lemma 6.1. □

Theorem 6.1 *Algorithm 6.1 performs correctly and stops in the time $p(L)\tau(6,7,6,7)^{L/3} < p(L)2^{0.10299L}$, where L is the length of the input formula and p is a polynomial.*

Proof. We have shown above that it is sufficient to prove that each formula F in CNF reduced by $REDUCE_L$ satisfies at least one of the conditions of the steps (LS1)–(LS4). Suppose F does not satisfy the conditions of the steps (LS1)–(LS3). We now consider the two possible cases: when F contains at least one 1-literal a, and when it contains no 1-literals. We prove that in each of these cases F satisfies the condition of the step (LS4).

Informally, this is done as follows. In the first case we show that the assignment $\mathcal{I}(a)$ contains many literals. The second case is handled by careful examination of sub-cases: we choose a 2-clause $\{c, d\}$ in F and examine the following sub-cases:

- there are many occurrences in the clauses containing the literal c (or d);
- F contains many occurrences of the literal $\overline{c}$ (or $\overline{d}$).

In these two sub-cases we show that many occurrences are eliminated during the transformation of F into $F[c], F[\overline{c}]$ (or $F[d], F[\overline{d}]$) and the subsequent elimination of 1-clauses obtained from the clauses containing the literals $\overline{c}, c$ (or $\overline{d}, d$). The third sub-case is:

- none of the previous sub-cases holds.

In this sub-case one of c and d is a (2,2)-literal occuring only in 2- and 3-clauses which allows us to use Lemma 6.1 after one occurrence of this literal is eliminated during the splitting by another literal.

In the following we denote $\Delta_1 = \mathcal{L}(F) - \mathcal{L}(F_1)$, $\Delta_2 = \mathcal{L}(F) - \mathcal{L}(F_2)$. Formulas F_1 and F_2 and a literal a have the same meaning as they have at step (LS4) of Algorithm 6.1.

CASE 1: The formula F contains a 1-literal a.

Let $D = \{a, a_2, \ldots, a_{|D|}\}$ be the only clause containing a. Let $r = \min\{5, |D|\}$. Since $\#_{\overline{a}} \geq 2$ and $\diamondsuit_a + \#_{\overline{a}} \geq 7$ by Lemma 6.3, we have $\#_{\overline{a}} \geq \max(7 - \diamondsuit_a, 2) \geq 7 - r$. Let γ be the number of 3^+-clauses among all clauses that contain $\overline{a}$.

During the transformation of the formula F into F_2, all occurrences of the literal a and all clauses containing $\overline{a}$ will be eliminated. Thus,

$$\Delta_2 \geq \diamondsuit_{\overline{a}} + \#_a \geq 2\#_{\overline{a}} + \gamma + \#_a \geq \\ 2(7 - r) + \gamma + 1 = 15 - 2r + \gamma \geq 5.$$

During the transformation of the formula F into F_1, the clause D and all occurrences of the literal $\overline{a}$ will be eliminated; all 2-clauses containing $\overline{a}$ will be eliminated too (at step (LR1)). Since D is not blocked for any a_i w.r.t. F, for each $i \in 2, \ldots, r$ there exists clause D_i in F such that $D_i \cap \{\overline{a}, \overline{a}_2, \ldots, \overline{a}_r\} = \{\overline{a}_i\}$. All these clauses will be eliminated since all clauses $\{\overline{a}, \overline{a}_i\}$ are blocked for $\overline{a}$ w.r.t. F (i.e. $\{a, \overline{a}_2, \ldots, \overline{a}_r\} \subseteq \mathcal{I}(a)$). All clauses containing $\overline{a}$, all clauses D_i and the clause D are distinct. Thus, we have

$$\Delta_1 \geq |D| + \#_{\overline{a}} + (\#_{\overline{a}} - \gamma) + 2(r - 1) \geq \\ r + (7 - r) + (\#_{\overline{a}} - \gamma) + 2(r - 1) = 5 + 2r + (\#_{\overline{a}} - \gamma) \geq 9.$$

We now have $\Delta_1 + \Delta_2 \geq 20 + \#_{\overline{a}} \geq 22$; $\Delta_1, \Delta_2 \geq 5$. Thus, $\tau(\Delta_1, \Delta_2) \leq \tau(5, 17)$, i.e., the condition of the step (LS4) is satisfied[1].

CASE 2: The formula F does not contain 1-literals.

Since F does not satisfy the condition of the step (LS3), it contains a 2-clause $D = \{c, d\}$. Let us note that

(6.2) The formula F does not contain other clauses that contain the literal d and the literal c, or the literal d and the literal $\overline{c}$, or the literal $\overline{d}$ and the literal c, simultaneously.

This proposition is true since F does not satisfy the conditions of the steps (LR2) and (LR4). Also, in this case $\mathcal{I}(c) = \{c\}$ and $\mathcal{I}(d) = \{d\}$ by (6.2) and the definition of $\mathcal{I}(\ldots)$.

Let us denote one of the literals c, d by a (we shall choose later which one). We denote the remaining literal by b. Let α be the number of 3^+-clauses among all clauses that contain a (note that $\#_a - \alpha \geq 1$), let γ be the number of 3^+-clauses among all clauses that contain $\overline{a}$. We now count the occurrences that disappear during the transformation of the formula F into $F[\mathcal{I}(a)]$ $(= F[a])$ and $F[\overline{a}]$ and subsequent elimination of 1-clauses resulted from the 2-clauses containing the literals $\overline{a}$ and a.

During the transformation of the formula F into F_1, all occurrences of the literal $\overline{a}$ and all clauses containing a will be eliminated; all 2-clauses that contain $\overline{a}$ will be eliminated too (at step (LR1)), there are $(\#_{\overline{a}} - \gamma)$ such clauses.

[1]Kullmann and Luckhardt [16] prove for a very similar algorithm that in this case $\tau(\Delta_1, \Delta_2) \leq \tau(5, 21)$; thus, $\tau(5, 17)$ in the condition of the step (LS4) can be replaced by $\tau(8, 12)$. However, this would not improve our upper bound while would make the proof longer; thus, we present here this simpler proof.

The formula $F[\overline{a}]$ contains the 1-clause $\{b\}$ obtained from the 2-clause D. It will be eliminated at step (LR1). If F contains other 2-clauses containing a, the clauses obtained from them will be eliminated at step (LR1) too. The number of such clauses is $(\#_a - 1 - \alpha)$. During the transformation of the formula F into F_2, all occurrences of the literal a and all clauses containing $\overline{a}$ or b will be eliminated. By (6.2), D is the only clause containing b in which the literals $a, \overline{a}$ can occur. Thus, even if we count only occurrences that disappear in clauses containing a or $\overline{a}$ before the first application of the step (LR2), we have

$$\Delta_1 \geq \diamondsuit_a + \#_{\overline{a}} + (\#_{\overline{a}} - \gamma), \tag{6.3}$$

$$\Delta_2 \geq \diamondsuit_{\overline{a}} + \#_a + (\diamondsuit_b - 1) + (\#_a - 1 - \alpha). \tag{6.4}$$

We consider several sub-cases.

Case 2.1: $\diamondsuit_c \geq 6$ and $\diamondsuit_d \geq 6$.
From (6.3) and (6.4) we have

$$\begin{aligned}
&\Delta_1 \geq 6 + 2 + (\#_{\overline{a}} - \gamma) \geq 8 + (\#_{\overline{a}} - \gamma) \geq 8,\\
&\Delta_2 \geq (2\#_{\overline{a}} + \gamma) + 2 + 5 = 11 + \gamma \geq 11,\\
&\Delta_1 + \Delta_2 \geq 19 + \#_{\overline{a}} \geq 21.
\end{aligned}$$

Thus, $\tau(\Delta_1, \Delta_2) \leq \tau(8, 13) < \tau(5, 17)$, i.e., the condition of the step (LS4) is satisfied.

Case 2.2: $\#_{\overline{c}} \geq 3$ or $\#_{\overline{d}} \geq 3$.
Let $a \in \{c, d\}$ be the literal for which $\#_{\overline{a}} \geq 3$ holds, let b be the remaining literal in $\{c, d\}$. Thus, we have

$$\begin{aligned}
&\Delta_1 \geq (2\#_a + \alpha) + 3 + (\#_{\overline{a}} - \gamma) \geq 7 + \alpha + (\#_{\overline{a}} - \gamma) \geq 7,\\
&\Delta_2 \geq (2\#_{\overline{a}} + \gamma) + 2 + 2\#_b - 1 + (\#_a - 1 - \alpha) \geq\\
&\qquad\qquad 10 + \gamma + (\#_a - \alpha) \geq 11,\\
&\Delta_1 + \Delta_2 \geq 17 + \#_a + \#_{\overline{a}} \geq 22.
\end{aligned}$$

Thus, $\tau(\Delta_1, \Delta_2) \leq \tau(7, 15) < \tau(5, 17)$, i.e., the condition of the step (LS4) is satisfied.

Case 2.3: $\#_{\overline{c}} = \#_{\overline{d}} = 2$, and $\diamondsuit_c \leq 5$ or $\diamondsuit_d \leq 5$.
Let $a \in \{c, d\}$ be a literal such that $\diamondsuit_a = \max(\diamondsuit_c, \diamondsuit_d)$, let b be the remaining literal in $\{c, d\}$. Then $4 \leq \diamondsuit_b \leq 5$ and $\diamondsuit_a \geq \diamondsuit_b$. In this case b is a 2-literal, it occurs in D and in one other 2- or 3-clause. We denote this clause by C_b.

The literal b becomes a (1,2)-literal in $F[\mathcal{I}(a)]$ $(= F[a])$. We now complete the proof by showing that several literal occurrences will be eliminated in addition to the occurrences counted by (6.3). There are three sub-cases.

CASE 2.3.1: At least two occurrences are eliminated before the first application of step (LR2) in addition to the occurrences counted by (6.3).

In this case,

$$\Delta_1 \geq \diamondsuit_a + \#_{\overline{a}} + (\#_{\overline{a}} - \gamma) + 2 \geq \\ (4 + \alpha) + 2 + (\#_{\overline{a}} - \gamma) + 2 \geq 8 + \alpha + (\#_{\overline{a}} - \gamma) \geq 8, \tag{6.5}$$

$$\Delta_2 \geq (4+\gamma)+2+3+(\#_a-1-\alpha) \geq 8+\gamma+(\#_a-\alpha) \geq 9, \tag{6.6}$$

$$\Delta_1 + \Delta_2 \geq 16 + \#_{\overline{a}} + \#_a \geq 20. \tag{6.7}$$

CASE 2.3.2: Exactly one occurrence is eliminated before the first application of step (LR2) in addition to the occurrences counted by (6.3).

We remind that (6.3) counts only occurrences to the clauses containing one of the literals a,$\overline{a}$. By (6.2) the occurrence of the literal b to the clause C_b or any occurrence of the literal $\overline{b}$ cannot be eliminated during the transformation of the formula F into the formula $F[a]$. If we counted any of these occurrences in the $\#_{\overline{a}} - \gamma$ term of (6.3), then it was an elimination of a 1-clause $\{\overline{b}\}$ obtained from a 2-clause $\{\overline{a}, \overline{b}\}$. Since it is impossible to have two distinct identical clauses $\{\overline{a}, \overline{b}\}$ in the formula, the other two occurrences (another occurrence of the literal $\overline{b}$ and the occurrence of the literal b in C_b) eliminated at this moment are not counted by (6.3); this fact contradicts the assumption of Case 2.3.2.

Thus in this case b remains a (1,2)-literal before the first application of the step (LR2). By Lemma 6.2(1) at least one more occurrence will be eliminated after that and thus (6.5)–(6.7) hold.

CASE 2.3.3: No occurrences are eliminated before the first application of step (LR2) in addition to the occurrences counted by (6.3).

Similarly to Case 2.3.2, b remains a (1,2)-literal before the first application of the step (LR2) during transformations of the formula $F[a]$ by Function $REDUCE_L$. Thus, if $|C_b| = 2$, then two more occurrences are eliminated by Lemma 6.2(2), and (6.5)–(6.7) hold.

If, however, $|C_b| = 3$, then we can apply only Lemma 6.2(1), but in this case $\diamondsuit_a \geq \diamondsuit_b = 5$, and thus

$$\Delta_1 \geq \diamondsuit_a + \#_{\overline{a}} + (\#_{\overline{a}} - \gamma) + 1 \geq \\ 5 + 2 + (\#_{\overline{a}} - \gamma) + 1 \geq 8 + (\#_{\overline{a}} - \gamma) \geq 8,$$
$$\Delta_2 \geq (4 + \gamma) + 2 + 4 \geq 10 + \gamma \geq 10,$$
$$\Delta_1 + \Delta_2 \geq 18 + \#_{\overline{a}} \geq 20,$$

Hence, in all three sub-cases 2.3.1–2.3.3, $\tau(\Delta_1, \Delta_2) \leq \tau(8, 12) < \tau(5, 17)$, i.e., the condition of the step (LS4) is satisfied. □

7 Conclusion and further work

In this paper we have improved the existing upper bounds for SAT with respect to K (the number of clauses) and L (the length of a formula). The key point in our algorithms and proofs is the *black and white literals principle*, a new transformation rule which can be viewed as a re-formulation of two previously known principles: the autarkness [20, 22, 17, 15] and the generalized sign principle [16].

Our proofs (as well as the proofs of the previous upper bounds [21, 16]) are neither short nor elegant. It would be a kind of breakthrough to find a *compact way* to present proofs of upper bounds for splitting algorithms. It is believable that such a way could lead to even *better bounds*, since currently our possibilities to create new heuristics and prove the corresponding upper bounds are limited by the length of a comprehensible proof. On the other hand, it is a challenging problem to prove (more) *tight lower bounds* for (some class of) splitting algorithms: currently, the exponential lower bounds for resolution proofs (see, e.g., [27]) are far enough from the known upper bounds for splitting algorithms. (Most of splitting algorithms can be viewed as resolution proofs and vice versa.)

Another direction for work is to find *randomized* algorithms that give better upper bounds for SAT (or to find a way how to apply modern randomized algorithms that have already been invented for 3-SAT in recent breakthrough papers [23, 24]). Also, it remains a challenging problem to find a *less-than*-2^N upper bound where N is the number of variables.

References

[1] E. Dantsin, *Tautology proof systems based on the splitting method*, Leningrad Division of Steklov Institute of Mathematics (LOMI), PhD Dissertation, Leningrad, 1982 (in Russian).

[2] E. Dantsin and L. O. Fuentes and V. Kreinovich, *Less than 2^n satisfiability algorithm extended to the case when almost all clauses are short*, Computer Science Department, University of Texas at El Paso, UTEP-CS-**91-5**, 1991.

[3] E. Dantsin, M. Gavrilovich, E. A. Hirsch, B. Konev, *Approximation algorithms for Max Sat: a better performance ratio at the cost of a longer running time*, PDMI preprint 14/1998, available from `ftp://ftp.pdmi.ras.ru/pub/publicat/preprint/1998/14-98.ps`

[4] E. Ya. Dantsin, V. Ya. Kreinovich, *Exponential upper bounds for the satisfiability problem*, Proc. of the IX USSR conf. on math. logic, Leningrad, 1988 (in Russian).

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

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

[7] J. Gu, P. W. Purdom, J. Franco, B. W. Wah, *Algorithms for Satisfiability (SAT) Problem: A Survey*, Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem, American Mathematical Society, 1997.

[8] E. A. Hirsch, *Separating the signs in the satisfiability problem*, Zap. nauchn. semin. POMI, vol. **241**, 1997, pp. 30-71 (in Russian). English translation of this collection is to appear in Journal of Mathematical Sciences in 1999.

[9] E. A. Hirsch, *Two new upper bounds for SAT*, Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 521–530.

[10] E. A. Hirsch, *Local Search Algorithms for SAT: Worst-Case Analysis*, Proc. of the 6th Scandinavian Workshop on Algorithm Theory, LNCS **1432**, 1998, pp. 246–254.

[11] E. A. Hirsch, *Hard formulas for SAT local search algorithms*, PDMI preprint 19/1998, available from `ftp://ftp.pdmi.ras.ru/pub/publicat/preprint/19-98.ps`

[12] E. A. Hirsch, *SAT local search algorithms: worst-case study*, Submitted to this volume.

[13] O. Kullmann, *Worst-case analysis, 3-SAT decision and lower bounds: approaches for improved SAT algorithms*, DIMACS Proc. SAT Workshop 1996, AMS, 1996, pp. 261–313.

[14] O. Kullmann, *New methods for 3-SAT decision and worst-case analysis*, to appear in Theoretical Computer Science, 1998, 68 pages; first version announced in Abstracts of contributed papers of the Logic Colloquium '93.

[15] O. Kullmann, *Investigations on autark assignments*, Submitted to Discrete Applied Mathematics, 1998, 19 pages.

[16] O. Kullmann, H. Luckhardt, *Deciding propositional tautologies: Algorithms and their complexity*, Preprint, 1997, 82 pages; the ps-file can be obtained at `http://mi.informatik.uni-frankfurt.de/people/kullmann/kullmann.html`. A journal version, *Algorithms for SAT/TAUT decision based on various measures*, is to appear in Information and Computation, 1998.

[17] H. Luckhardt, *Obere Komplexitätsschranken für TAUT-Entscheidungen,* Proc. Frege Conference 1984, Schwerine, Akademie-Verlag Berline, pp. 331–337.

[18] M. Mahajan and V. Raman, *Parametrizing above guaranteed values: MaxSat and MaxCut*, Technical Report TR97-033, Electronic Colloquium on Computational Complexity, 1997.

[19] R. Niedermeier and P. Rossmanith. *New upper bounds for MaxSat*, Technical Report KAM-DIMATIA Series 98-401, Charles University, Praha, Faculty of Mathematics and Physics, July 1998.

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

[21] B. Monien, E. Speckenmeyer, *Upper bounds for covering problems,* Bericht Nr. 7/1980, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn.

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

[23] R. Paturi, P. Pudlak, F. Zane, *Satisfiability Coding Lemma,* Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 1997, pp. 566–574.

[24] R. Paturi, P. Pudlak, M. E. Saks, F. Zane, *An Improved Exponential-time Algorithm for k-SAT,* Proceedings of the 39th Annual Symposium on Foundations of Computer Science, 1998, pp. 628–637.

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

[26] 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; University Köln, Report No. **96-230**.

[27] A. Urquhart, *The Complexity of Propositional Proofs,* Bulletin of Symbolic Logic **1**(4), 1995, pp. 425–467.

[28] A. Van Gelder, *A satisfiability tester for non-clausal propositional calculus,* Information and Computation **79**, 1988, pp. 1–21.