

# Deterministic Algorithms for $k$ -SAT Based on Covering Codes and Local Search

Evgeny Dantsin<sup>1,\*</sup>, Andreas Goerdt<sup>2</sup>,  
Edward A. Hirsch<sup>3,\*\*</sup>, and Uwe Schöning<sup>4</sup>

<sup>1</sup> Steklov Institute of Mathematics, 27 Fontanka, 191011 St.Petersburg, Russia,  
dantsin@pdmi.ras.ru

<sup>2</sup> TU Chemnitz, Fakultät für Informatik, 09107 Chemnitz, Germany,  
goerdt@informatik.tu-chemnitz.de

<sup>3</sup> Steklov Institute of Mathematics, 27 Fontanka, 191011 St.Petersburg, Russia,  
hirsch@pdmi.ras.ru, <http://logic.pdmi.ras.ru/~hirsch>

<sup>4</sup> Universität Ulm, Abteilung Theoretische Informatik, 89069 Ulm, Germany,  
schoenin@informatik.uni-ulm.de

**Abstract.** We show that satisfiability of formulas in  $k$ -CNF can be decided deterministically in time close to  $(2k/(k+1))^n$ , where  $n$  is the number of variables in the input formula. This is the best known worst-case upper bound for deterministic  $k$ -SAT algorithms. Our algorithm can be viewed as a derandomized version of Schöning's probabilistic algorithm presented in [15]. The key point of our algorithm is the use of covering codes together with local search. Compared to other "weakly exponential" algorithms, our algorithm is technically quite simple.

We also show how to improve the bound above by moderate technical effort. For 3-SAT the improved bound is  $1.481^n$ .

## 1 Introduction

*Worst-Case Upper Bounds for SAT.* The satisfiability problem for propositional formulas (SAT) can be decided by an obvious algorithm in time<sup>1</sup>  $2^n$ , where  $n$  is the number of variables in the input formula. This worst-case upper bound can be decreased for  $k$ -SAT, i.e., if we restrict inputs to formulas in conjunctive normal form with at most  $k$  literals per clause ( $k$ -CNF). First upper bounds  $c^n$ , where  $c < 2$ , were obtained in 1980s [4,10], for example, the bound  $1.619^n$  for 3-SAT. Currently, much research in SAT algorithms is aimed at decreasing the base in exponential upper bounds, e.g., [16,14,8,13,5,12,7,15].

The best known bound for probabilistic 3-SAT algorithms is  $(4/3)^n$  due to U. Schöning [15]. For probabilistic  $k$ -SAT algorithms when  $k \geq 4$ , the best known bound is due to R. Paturi, P. Pudlák, M. E. Saks, and F. Zane [12]. This bound

---

\* Work done in part while visiting Department of Computer Science of University of Manchester. Supported by grants from TFR, INTAS, and RFBR.

\*\* Supported by INTAS project 96-0760 and RFBR grant 99-01-00113.

<sup>1</sup> In this paper, we write all complexity bounds up to a polynomial in the size of the input formula. For example, we write  $2^n$  instead of  $poly(n)2^n$ .

is not represented in compact form; the bound for 4-SAT is  $1.477^n$ , the bound for 5-SAT is  $1.569^n$ .

The best known bounds for deterministic  $k$ -SAT algorithms are as follows. For  $k = 3$ , O. Kullmann [7] gives the bound  $1.505^n$ . In [14], the bound  $1.497^n$  was announced ([6] sketched how this bound can be obtained by a refinement of [7]). For  $k = 4$ , the best known bound is still  $1.840^n$  due to B. Monien and E. Speckenmeyer [10]. For  $k \geq 5$ , R. Paturi, P. Pudlák and F. Zane [13] give the bound approaching  $2^{(1-1/(2k))n}$  for large  $k$ . In this paper we improve all these bounds for deterministic algorithms.

*Probabilistic Algorithm.* Our algorithm is inspired by the currently fastest probabilistic 3-SAT algorithm of [15] which uses a well-known heuristic search method, cf. [2,9]. This algorithm is based on random walks (like Papadimitriou's algorithm [11] for 2-SAT). Given a formula  $F$  in  $k$ -CNF with  $n$  variables, the algorithm chooses exponentially many initial assignments at random and runs local search for each of them. Namely, if the assignment does not satisfy  $F$ , then the algorithm chooses any unsatisfied clause, chooses a literal from this clause at random, and flips its value. If a satisfying assignment is not found in  $3n$  such steps, the algorithm starts local search from another random initial assignment.

Thus, the probabilistic algorithm of [15] includes two randomized components: (1) the choice of initial assignments, and (2) local search starting from these assignments.

*Deterministic Algorithm.* The deterministic  $k$ -SAT algorithm presented in this paper can be viewed as a derandomized version of the probabilistic algorithm above. The derandomization consists of two parts. To derandomize the choice of initial assignments, we cover the space of all possible  $2^n$  assignments by balls of some Hamming radius  $r$ . We use coding theory to find a good covering for given  $r$ . In each ball, we run a deterministic version of local search to check whether there is a satisfying assignment inside the ball.

The optimal value of  $r$  can be chosen so that the overall running time is minimal. Taking  $r = 1/(k+1)$ , we obtain the running time  $(2k/(k+1) + \varepsilon)^n$ . We also show how to decrease this bound by using a more complicated version of local search. For 3-SAT, the modified algorithm gives the bound  $1.481^n$ .

*Notation.* We consider propositional formulas in  $k$ -CNF ( $k \geq 3$  is a constant). These formulas are conjunctions of clauses of size at most  $k$ . A *clause* is a disjunction of literals. A *literal* is a propositional variable or its negation. The size of a clause is the number of its literals. An *assignment* maps the propositional variables to the truth values 0, 1, where 0 denotes *false* and 1 denotes *true*. A *trivial* formula is the empty formula (which is always true) or a formula containing the empty clause (which is always false).

For an assignment  $a$  and a literal  $l$ , we write  $a_{|l=1}$  to denote the assignment obtained from  $a$  by setting the value of  $l$  to 1 (more precisely, we set the value of the variable corresponding to  $l$ ). We also write  $F_{|l=1}$  to denote the formula obtained from  $F$  by assigning the value 1 to  $l$ , i.e., the clauses containing the

literal  $l$  itself are deleted from  $F$ , and the literal  $\bar{l}$  is deleted from the other clauses.

We identify assignments with binary words. The *Hamming distance* between two assignments is the number of positions in which these two assignments differ. The *ball* of radius  $r$  around an assignment  $a$  is the set of all assignments whose Hamming distance to  $a$  is at most  $r$ .

A *code* of length  $n$  is a subset of  $\{0, 1\}^n$ . The *covering radius*  $r$  of a code  $\mathcal{C}$  is defined by

$$r = \max_{u \in \{0,1\}^n} \min_{v \in \mathcal{C}} d(u, v),$$

where  $d(u, v)$  denotes the Hamming distance between  $u$  and  $v$ . The *normalized covering radius*  $\rho$  is defined by  $\rho = r/n$ .

*Organization of the Paper.* Section 2 describes the local search procedure. In Sect. 3 we specify the codes used in our algorithms. Section 4 contains the main algorithm and its analysis. We describe the technique yielding to the bound  $1.481^n$  in Sect. 5. We discuss further developments in Sect. 6.

## 2 Local Search

Suppose we are given an initial assignment  $a \in \{0, 1\}^n$  where  $n$  is the number of variables. Consider the ball of radius  $r$  around  $a$ . Obviously, the volume of this ball is

$$V(n, r) = \sum_{i=0}^r \binom{n}{i}.$$

If the normalized radius  $\rho = r/n$  satisfies  $0 < \rho < 1/2$ , the volume  $V(n, r)$  can be estimated as follows [1, page 121] or [3, Lemma 2.4.4, page 33]:

$$\frac{1}{\sqrt{8n\rho(1-\rho)}} \cdot 2^{h(\rho)n} \leq V(n, r) \leq 2^{h(\rho)n} \tag{1}$$

where  $h(\rho) = -\rho \log_2 \rho - (1-\rho) \log_2 (1-\rho)$  is the binary entropy function. These bounds show that for a constant  $\rho$ , the volume  $V(n, r)$  differs from  $2^{h(\rho)n}$  at most by a polynomial factor.

The efficiency of our algorithm relies on the following important observation. Suppose we need to find a satisfying assignment inside the ball of radius  $r$  around  $a$  (if one exists). Then it is not necessary to search through all assignments inside this ball. The given formula  $F$  can be used to prune the search tree. The following easy lemma captures this observation.

**Lemma 1.** *Let  $F$  be a formula and  $a$  be an assignment such that  $F$  is false under  $a$ . Let  $C$  be an arbitrary clause in  $F$  that is false under  $a$ . Then  $F$  has a satisfying assignment belonging to the ball of radius  $r$  around  $a$  iff there is a literal  $l$  in  $C$  such that  $F_{|l=1}$  has a satisfying assignment belonging to the ball of radius  $r - 1$  around  $a$ .*

*Proof.* For any clause  $C$  false under  $a$ , we have the following equivalence. The formula  $F$  has a satisfying assignment inside the ball of radius  $r$  around  $a$  iff there are a literal  $l$  in  $C$  and an assignment  $b$  such that  $l$  has the value 1 in  $b$  and the Hamming distance between  $b$  and  $a|_{l=1}$  is at most  $r - 1$ . The latter holds iff there is a literal  $l$  in  $C$  such that  $F|_{l=1}$  has a satisfying assignment inside the ball of radius  $r - 1$  around  $a$ .  $\square$

The recursive procedure  $\text{Search}(F, a, r)$  described below is the local search component of our algorithm. The procedure takes a formula  $F$ , an initial assignment  $a$ , and a radius  $r$  as input. It returns *true* if  $F$  has a satisfying assignment inside the ball of radius  $r$  around  $a$ . Otherwise, the procedure returns *false*.

Procedure  $\text{Search}(F, a, r)$

1. If all clauses of  $F$  are true under  $a$  then return *true*.
2. If  $r \leq 0$  then return *false*.
3. If  $F$  contains the empty clause then return *false*.
4. Pick (according to some deterministic rule) a clause  $C$  false under  $a$ . *Branch* on this clause  $C$ , i.e., for each literal  $l$  in  $C$  do the following: If  $\text{Search}(F|_{l=1}, a, r - 1)$  returns *false* then return *true*, otherwise return *false*.

The correctness of the procedure easily follows by induction on  $r$  with the invariant:  $\text{Search}(F, a, r)$  outputs *true* iff  $F$  has a satisfying assignment inside the ball of radius  $r$  around  $a$ . For the induction step the lemma above is used.

The recursion depth of  $\text{Search}(F, a, r)$  is at most  $r$ . If the input formula  $F$  is in  $k$ -CNF, the number of recursive calls generated at Step 4 is bounded by  $k$ . Therefore the recursion tree has at most  $k^r$  leaves and the complexity of the procedure is bounded by  $k^r$ .

Observe here that  $k^r$  can be much smaller than the volume of the ball of radius  $r$  around  $a$ . For example, for  $r = n/2$  and  $k = 3$  we obtain  $V(n, r) \geq 2^{n-1}$  whereas  $3^r < 1.733^n$ . This gives us a very simple deterministic SAT algorithm: Given an input formula  $F$  with  $n$  variables, run  $\text{Search}(F, a_0, n/2)$  and  $\text{Search}(F, a_1, n/2)$ , where  $a_0$  and  $a_1$  are  $n$ -bit assignments  $a_0 = (0, 0, \dots, 0)$  and  $a_1 = (1, 1, \dots, 1)$ . Since any assignment has the Hamming distance  $\leq n/2$  to either  $a_0$  or  $a_1$ , the algorithm is correct. Its running time is bounded by  $1.733^n$ .

The set  $\{a_0, a_1\}$  of assignments in the example above is nothing else than a very special covering code. In the next section we show how this example can be generalized and improved.

### 3 Good Coverings

By a *covering* of radius  $r$  for  $\{0, 1\}^n$  we mean a code  $\mathcal{C}$  of length  $n$  such that any word in  $\{0, 1\}^n$  belongs to at least one ball of radius  $r$  around a code word of  $\mathcal{C}$ .

For any covering  $\mathcal{C}$  of radius  $r$ , we have  $|\mathcal{C}| \cdot V(n, r) \geq 2^n$ , where  $V(n, r)$  is the volume of a ball of radius  $r$ . Using the upper bound on  $V(n, r)$  in (1), we obtain

$$|\mathcal{C}| \geq \frac{2^n}{V(n, r)} \geq \frac{2^n}{2^{h(\rho)n}} = 2^{(1-h(\rho))n}$$

where  $\rho = r/n$  is the normalized covering radius. Here the second inequality holds when  $0 < \rho < 1/2$ . This lower bound on  $|\mathcal{C}|$  is known as the *sphere covering bound*. The following lemma from coding theory implies the existence of coverings whose cardinalities “almost” achieve the sphere covering bound.

**Lemma 2 ([3, Theorem 12.1.2, page 320]).** *For any  $n$  and  $r$ , there exists a code  $\mathcal{C}$  of length  $n$  and covering radius  $r$  with*

$$|\mathcal{C}| \leq \lceil n2^n \ln 2/V(n, r) \rceil. \tag{2}$$

**Corollary 1.** *Let  $\delta > 0$  and  $0 < \rho < 1/2$ . There exist an integer  $\nu = \nu(\delta)$  and a code  $\Gamma$  of length  $\nu$  with the following properties:*

1. *The code  $\Gamma$  has the covering radius  $r \leq \rho\nu$ .*
2. *The code  $\Gamma$  achieves the sphere covering bound up to the “ $+\delta\nu$ ” in the exponent, namely:  $|\Gamma| \leq 2^{(1-h(\rho)+\delta)\nu}$ .*

*Proof.* We obtain the required bound on the cardinality by combining (2) and the lower bound on the volume in (1). □

The lemma and corollary above do not provide us with an efficient construction of good coverings. To construct them efficiently, we use the concept of the direct sum of two codes. Let  $\mathcal{C}_1$  and  $\mathcal{C}_2$  be two codes of lengths  $n_1, n_2$  and covering radii  $r_1, r_2$  respectively. The *direct sum* of  $\mathcal{C}_1$  and  $\mathcal{C}_2$  is the code  $\mathcal{C}$  of length  $n_1 + n_2$  that consists of all  $|\mathcal{C}_1| \cdot |\mathcal{C}_2|$  possible concatenations  $w_1w_2$ , where  $w_1 \in \mathcal{C}_1$  and  $w_2 \in \mathcal{C}_2$ . Obviously, the covering radius of  $\mathcal{C}$  is  $r_1 + r_2$ .

To generate a good covering  $\mathcal{C}$  for  $\{0, 1\}^n$ , we fix  $\varepsilon > \delta > 0$  and  $0 < \rho < 1/2$ . Let  $\Gamma$  be a code of length  $\nu$  satisfying Corollary 1. The following algorithm takes a (large) length  $n$  as input and uses  $\Gamma$  to generate the code  $\mathcal{C}$  of length  $n$ :

**Algorithm for Generating Good Coverings**

1. Find the smallest integer  $q$  such that  $n \leq q\nu$ .
2. Generate the direct sum  $\Gamma^q$ , i.e., generate words  $w_1w_2 \dots w_q$  one by one, where  $w_i$  ranges over  $\Gamma$ . If  $n$  is not divisible by  $\nu$ , we restrict the code words of  $\Gamma^q$  to the first  $n$  positions.

The resulting code is the required covering  $\mathcal{C}$ . The algorithm runs in time polynomial in  $|\mathcal{C}|$ . If  $n$  (and  $q$ ) is sufficiently large, its covering radius is at most  $q\rho\nu \leq \lfloor n(\rho + \varepsilon) \rfloor$  and the cardinality  $|\mathcal{C}|$  is bounded by:

$$2^{(1-h(\rho)+\delta)q\nu} \leq 2^{(1-h(\rho)+\varepsilon)n}.$$

Note that the sphere covering bound is “almost” (up to the “ $+\varepsilon n$ ” in the covering radius and in the exponent) achieved. Note also that the code  $\Gamma$  provided by Corollary 1 is “hard wired” into the algorithm.

## 4 Main Algorithm and Its Analysis

Given fixed  $\varepsilon > 0$  and  $0 < \rho < 1/2$ , our satisfiability algorithm takes as input a formula  $F$  with  $n$  variables and works as follows.

Main Algorithm

1. Generate a covering  $\mathcal{C}$  for  $\{0, 1\}^n$  by using the algorithm for generating good coverings in Sect. 3.
2. Calculate  $r = \lfloor n(\rho + \varepsilon) \rfloor$ .
3. For each code word  $a$  in  $\mathcal{C}$  run the procedure  $Search(F, a, r)$ . Return *true* if at least one procedure call returns *true*. Otherwise return *false*.

If inputs are formulas in  $k$ -CNF, the complexity of the algorithm is bounded by  $|\mathcal{C}| \cdot k^r$ . Since  $|\mathcal{C}| \leq 2^{(1-h(\rho))n}$  and  $k^r \leq k^{n\rho}$  up to a factor of  $c^{\varepsilon n}$ , the complexity is bounded by

$$2^{(1-h(\rho))n} \cdot k^{n\rho} = \left( \frac{2k^\rho}{2^{h(\rho)}} \right)^n = (2\rho^\rho(1-\rho)^{1-\rho}k^\rho)^n.$$

The choice  $\rho = 1/(k+1)$  minimizes (calculus required) the base value of this exponential function. Substituting  $1/(k+1)$  for  $\rho$ , we have

$$2 \cdot \left( \frac{1}{k+1} \right)^{1/(k+1)} \cdot \left( 1 - \frac{1}{k+1} \right)^{1-1/(k+1)} \cdot k^{1/(k+1)} = \frac{2k}{k+1}.$$

Thus, we obtain the following theorem.

**Theorem 1.** *For every  $\varepsilon > 0$  there is a deterministic algorithm for deciding satisfiability of propositional formulas in  $k$ -CNF whose running time is*

$$\left( \frac{2k}{k+1} + \varepsilon \right)^n$$

where  $n$  is the number of variables in the input formula.

## 5 Improved Local Search

The local search procedure  $Search(F, a, r)$  from Sect. 2 picks *any* false clause for branching. The complexity of this procedure is at most  $k^r$ . Choosing a clause for branching more carefully, we can improve this bound and, thereby, the overall running time of our main algorithm. In this section we show how to improve  $Search(F, a, r)$  for formulas in 3-CNF so that its complexity is bounded by  $2.848^r$  instead of  $3^r$ . We then obtain the bound  $1.481^n$  for 3-SAT.

Let  $F$  be formula and  $a$  an assignment. Let  $C$  be a clause in  $F$ . We classify  $C$  according to the number of its literals true under  $a$ . Namely, we call  $C$  a

$$\underbrace{(1, \dots, 1)}_p, \underbrace{(0, \dots, 0)}_m \text{-clause}$$

if  $C$  consists of exactly  $p$  literals true under  $a$ , and exactly  $m$  literals false under  $a$ . For example, a  $(1, 1, 0)$ -clause consists of two true and one false literals.

We say that  $F$  is  $i$ -false under  $a$  ( $i \geq 0$ ) iff  $F$  has no satisfying assignment within Hamming distance  $i$  from  $a$ . When  $i$  is fixed, we can test in polynomial time whether  $F$  is  $i$ -false under  $a$ , where  $F$  and  $a$  are given as input.

The following observation follows from the fact that a  $(1)$ -clause  $\bar{l}_i$  of  $F$  becomes the empty clause in  $F|_{l_i=1}$ .

**Lemma 3.** *Let  $F$  be a formula false under  $a$ . If  $l_1 \vee l_2 \vee l_3$  is a  $(0, 0, 0)$ -clause and  $\bar{l}_i$  is a  $(1)$ -clause of  $F$  then we have the following equivalence:  $F$  is satisfied within Hamming distance  $\leq r$  from  $a$  iff there is a  $j \neq i$  such that  $F|_{l_j=1}$  is satisfied within Hamming distance  $\leq r - 1$  from  $a$ .*

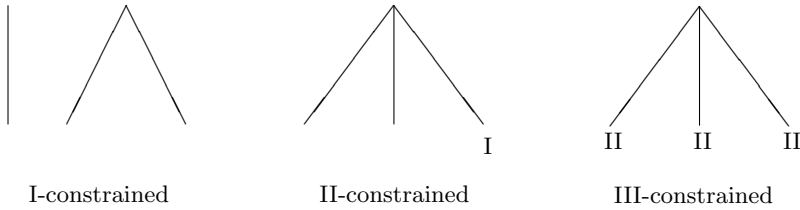


Fig. 1. Branching I, II, III-constrained formulas.

Let  $F$  be a non-trivial formula in 3-CNF. The following definitions describe types of formulas used in the new version of the procedure *Search*. Figure 1 illustrates these types.

1. Let  $F$  be false under  $a$ . We say that  $F$  is *I-constrained* with respect to  $a$  iff one of the following holds:
  - $F$  has a  $(0)$ - or  $(0, 0)$ -clause with respect to  $a$ .
  - $F$  has a  $(0, 0, 0)$ -clause containing a literal  $l$  such that  $\bar{l}$  is a  $(1)$ -clause of  $F$ .
2. Let  $F$  be 1-false under  $a$ . We say that  $F$  is *II-constrained* with respect to  $a$  iff one of the following holds:
  - $F$  is I-constrained with respect to  $a$ .
  - $F$  has a  $(0, 0, 0)$ -clause containing a literal  $l$  such that the formula  $F|_{l=1}$  is I-constrained.
3. Let  $F$  be 2-false under  $a$ . We say that  $F$  is *III-constrained* with respect to  $a$  iff one of the following holds:
  - $F$  is I-constrained or II-constrained with respect to  $a$ .
  - $F$  has a  $(0, 0, 0)$ -clause such that for *each* literal  $l$  in this clause,  $F|_{l=1}$  is II-constrained.

The next lemma and its corollary show that there is no need to make recursive calls for not III-constrained formulas, because such formulas turn out to be satisfiable.

**Lemma 4.** *Let  $F$  be non-trivial and 3-false under  $a$ . Let  $l$  be a literal false under  $a$ . Let  $F_{|l=1}$  still be non-trivial (it is 2-false under  $a$  by assumption). Then we have the implication:  $F_{|l=1}$  is III-constrained with respect to  $a$  then  $F_{|l=1}$  is II-constrained or  $F$  itself is III-constrained.*

*Proof.* First, a preparatory remark: If  $F$  is non-trivial, 1-false under  $a$  and not I-constrained then  $F_{|l=1}$  is non-trivial for any literal  $l$  belonging to a clause of  $F$  false under  $a$ . This is because the empty clause could only be generated if  $F$  contained the clause  $\bar{l}$ , which is not the case as  $F$  is not I-constrained under  $a$ .

We show that if  $F_{|l=1}$  is III-constrained with respect to  $a$  then  $F_{|l=1}$  is II-constrained or  $F$  is III-constrained.

By assumption  $F_{|l=1}$  is non-trivial and 2-false under  $a$ . If  $F_{|l=1}$  is I- or II-constrained, the lemma holds. The remaining case to consider is that  $F_{|l=1}$  has a  $(0, 0, 0)$ -clause  $l_1 \vee l_2 \vee l_3$  such that each  $F_{|l=1, l_i=1}$  is II-constrained. If an  $F_{|l=1, l_i=1}$  is I-constrained then  $F_{|l=1}$  is II-constrained and the lemma holds. We have to consider the situation that in each  $F_{|l=1, l_i=1}$  we have a  $(0, 0, 0)$ -clause  $k_i \vee k_{i,1} \vee k_{i,2}$  such that (without loss of generality) setting  $k_i = 1$  shows that  $F_{|l=1, l_i=1}$  is II-constrained. This means that  $G_i = F_{|l=1, l_i=1, k_i=1}$  is I-constrained for  $i = 1, 2, 3$ . As  $F$  is 3-false under  $a$ , we obtain that  $G_i$  is false under  $a$ .

It follows from the remark in the beginning of our proof that  $G_i$  is non-trivial.

The final idea is as follows. A clause that causes  $G_i$  to be I-constrained is present in  $F$  or generated in 1 or 2 (but not 3 because of  $F$  is in 3-CNF) of the steps  $l = 1, l_i = 1, k_i = 1$ . If the clause is present in  $F$  we are done. If the step  $l = 1$  is among the steps making at least one  $G_i$  I-constrained,  $F_{|l=1}$  is already II-constrained. If for no  $G_i$  the step  $l = 1$  is necessary to make  $G_i$  I-constrained, the clause  $l_1 \vee l_2 \vee l_3$  and other clauses already present in  $F$  cause each  $G_i$  to be I-constrained. This means that  $F$  itself is III-constrained. The missing details of this argument follow below.

We consider several cases depending on the types of clauses that make  $G_i$  I-constrained:

Case of  $(0, 0)$ -clause. If  $G_i$  has a  $(0, 0)$ -clause  $h_1 \vee h_2$  then  $h_1 \vee h_2$  is generated when setting  $k_i = 1$ . This is because we assume that  $F_{|l=1, l_i=1}$  is not I-constrained. Therefore  $F_{|l=1, l_i=1}$  has a  $(1, 0, 0)$ -clause  $\bar{k}_i \vee h_1 \vee h_2$  and the  $(0, 0, 0)$ -clause  $k_i \vee k_{i,1} \vee k_{i,2}$  where  $h_1, h_2 \neq k_i$ . As  $F$  is in 3-CNF these clauses are also present in  $F$  and we obtain that  $F$  is already II-constrained.

Case of 0-clause. If  $G_i$  has a 0-clause  $h_i$  then  $h_i$  is generated when setting  $k_i = 1$ . Again this is because we assume that  $F_{|l=1, l_i=1}$  is not I-constrained. Therefore  $F_{|l=1, l_i=1}$  has a  $(1, 0)$ -clause  $\bar{k}_i \vee h_i$  and the  $(0, 0, 0)$ -clause  $k_i \vee k_{i,1} \vee k_{i,2}$  where  $h_i \neq k_i$ . The clause  $k_i \vee k_{i,1} \vee k_{i,2}$  is present in  $F$  and  $F_{|l=1}$ . If  $\bar{k}_i \vee h_i$  is also present in  $F_{|l=1}$  then  $F_{|l=1}$  is II-constrained and we are done.

We have to assume that  $k_i \vee h_i$  is not present in  $F_{|l=1}$ . Then  $F_{|l=1}$  has the  $(0, 0, 0)$ -clauses

$$\begin{aligned}
 & l_1 \vee l_2 \vee l_3 \\
 & k_i \vee k_{i,1} \vee k_{i,2}
 \end{aligned}$$

and the (1,1,0)-clause  $\bar{l}_i \vee \bar{k}_i \vee h_i$ . Since  $F$  is in 3-CNF, these clauses also belong to  $F$ .

Case of (0,0,0)-clause and 1-clause. Let  $h'_i \vee h''_i \vee h'''_i$  be a (0,0,0)-clause with respect to  $a$  in  $G_i$ . Let  $\bar{h}'_i$  be a 1-clause in  $G_i$  (without loss of generality). If  $F_{|l=1, l_i=1}$  has the 1-clause  $\bar{h}'_i$  then  $F_{|l=1}$  is II-constrained and we are done.

So we assume that  $F_{|l=1, l_i=1}$  does not have the clause  $\bar{h}'_i$ . Then  $F_{|l=1, l_i=1}$  must have the clause  $\bar{k}_i \vee \bar{h}'_i$ . If this clause is also present in  $F_{|l=1}$  then  $F_{|l=1}$  is II-constrained and we are done.

We still need to assume that the clause  $\bar{k}_i \vee \bar{h}'_i$  is not in  $F_{|l=1}$ . Then  $F_{|l=1}$  has the (0,0,0)-clauses

$$\begin{aligned} & l_1 \vee l_2 \vee l_3 \\ & k_i \vee k_{i,1} \vee k_{i,2} \\ & h'_i \vee h''_i \vee h'''_i \end{aligned}$$

and the (1,1,1)-clause  $\bar{l}_i \vee \bar{k}_i \vee \bar{h}'_i$ . Since  $F$  is in 3-CNF, these clauses are also present in  $F$ .

If we cannot conclude that  $F$  or  $F_{|l=1}$  is II-constrained by now, we obtain that for each  $i = 1, 2, 3$ , one of the two remaining cases above applies. Then the (0,0,0)-clauses  $l_1 \vee l_2 \vee l_3$ ,  $k_i \vee k_{i,2} \vee k_{i,3}$  for  $i = 1, 2, 3$ , and the other clauses as specified above are present in  $F$ . These clauses show that  $F$  is III-constrained because when we branch on the clause  $l_1 \vee l_2 \vee l_3$  instead of considering  $F_{|l=1}$ , we can see that each  $F_{|l_i=1}$  is II-constrained.  $\square$

**Corollary 2.** *Let  $F$  be non-trivial and 2-false with respect to  $a$ . If  $F$  is not III-constrained with respect to  $a$  then  $F$  is satisfiable.*

*Proof.* If  $F$  is not 3-false under  $a$  then  $F$  is satisfiable and the corollary is proved. So assume  $F$  is 3-false and has clauses false under  $a$ . The only such clauses are (0,0,0)-clauses because  $F$  is not III-constrained and, therefore, not I-constrained with respect to  $a$ . Let  $l_1 \vee l_2 \vee l_3$  be a (0,0,0)-clause with respect to  $a$  in  $F$ . Then each  $F_{|l_i=1}$  is non-trivial and 2-false (cf. the remark in the beginning of the proof of Lemma 4). If each  $F_{|l_i=1}$  is III-constrained, we obtain from Lemma 4 that each  $F_{|l_i=1}$  is II-constrained or  $F$  itself is III-constrained. In any case  $F$  is III-constrained.

However  $F$  is not III-constrained by assumption. Therefore we obtain that at least one  $F_{|l_i=1}$  is non-trivial, 2-false under  $a$ , and not III-constrained. Moreover,  $F_{|l_i=1}$  has strictly less false clauses than  $F$ . Induction on the number of false clauses of  $F$  shows that finally we arrive at a formula that is not any more 3-false under  $a$  and hence satisfiable.  $\square$

Like the initial version of the procedure  $Search(F, a, r)$  defined in Sect. 2, the new version takes as input a formula  $F$  in 3-CNF with  $n$  variables, an initial assignment  $a$ , and a radius  $r$ . It returns *true* if  $F$  has a satisfying assignment inside the ball of radius  $r$  around  $a$ . If  $F$  is unsatisfiable, the procedure returns *false*.

Procedure  $Search(F, a, r)$

1. If  $F$  is not 2-false under  $a$  then return *true*.
2. If  $r \leq 0$  then return *false*.
3. If  $F$  contains the empty clause then return *false*.
4. If  $F$  is not III-constrained with respect to  $a$  then return *true*.
5. If  $F$  is I-constrained with respect to  $a$  then branch on a false clause that certifies that  $F$  is I-constrained.
6. If  $F$  is II-constrained with respect to  $a$  then branch on a false clause that certifies that  $F$  is II-constrained.
7. Branch on a false clause that certifies that  $F$  is III-constrained.

Here the notion of branching is modified as follows. If we branch on a  $(0, 0, 0)$ -clause  $l_1 \vee l_2 \vee l_3$  such that  $F$  has the 1-clause  $\bar{l}_i$  then we do not run  $Search(F_{l_i=1}, a, r - 1)$ .

The correctness of the procedure follows from Lemma 1 and 2 by induction on  $r$ . To estimate the number of leaves of the recursion tree, we use the function  $H$  defined by recursion as follows:  $H(0) = 1$ ,  $H(1) = 3$ ,  $H(2) = 9$ , and for  $r \geq 3$

$$H(r) = 6 \cdot (H(r - 2) + H(r - 3)).$$

**Lemma 5.** *Let  $L(F, a, r)$  be the number of leaves of the recursion tree of the procedure  $Search(F, a, r)$ . Then we have  $L(F, a, r) \leq H(r)$  for all  $r$ .*

*Proof.* We first show that  $2 \cdot H(r - 1) \leq H(r)$  for all  $r \geq 1$ . This holds for  $r = 1$ ,  $r = 2$  and  $r = 3$ . For  $r > 3$  the inequality is proved by easy induction. Second we show that  $2 \cdot (H(r - 1) + H(r - 2)) \leq H(r)$  for  $r \geq 2$ . This can be calculated directly for  $r = 2, 3, 4$ . For  $r > 4$  we use induction:

$$\begin{aligned} & 2 \cdot H(r - 1) + 2 \cdot H(r - 2) \\ &= 12 \cdot H(r - 3) + 12 \cdot H(r - 4) + 12 \cdot H(r - 4) + 12 \cdot H(r - 5) \\ &\leq H(r) \quad (\text{induction hypothesis}) \end{aligned}$$

The claim of the lemma now holds for  $r = 0, 1, 2$ . For induction on  $r$  we proceed as follows. If  $F$  is not 2-false under  $a$ , we have one leaf in the recursion tree of  $Search(F, a, r)$  and the claim holds. The same applies when  $F$  has the empty clause. If  $F$  is not III-constrained with respect to  $a$ , we again have only one leaf and the claim holds.

Otherwise  $F$  is I-, II-, or III-constrained. The claim follows by induction, applying the inequalities above and the definition of  $H$ . □

To obtain an explicit bound on the size of the recursion tree, we solve the recurrence for  $H$ . If  $\alpha$  satisfies the equation  $\alpha^3 = 6 \cdot \alpha + 6$ , we assume  $H(m) = \alpha^m$  for  $m \leq r$  and derive  $H(r + 1) = \alpha^{r+1}$ . The initial conditions are taken care of when we bound  $H(r) \leq 9 \cdot \alpha^r$  for all  $r$  where  $\alpha$  is the unique (some calculus required) positive solution of the cubic equation above. One can calculate that

$\alpha = \sqrt[3]{4} + \sqrt[3]{2}$  which is between 2.847 and 2.848. Hence the induction base holds. For the induction step observe that

$$\begin{aligned} H(r+1) &\leq 6 \cdot 9 \cdot \alpha^{r-1} + 6 \cdot 9 \cdot \alpha^{r-2} \quad (\text{induction hypothesis}) \\ &= 9 \cdot \alpha^{r+1} \end{aligned}$$

Choosing  $\rho = 0.26$ , we obtain a satisfiability algorithm running in time exponential in  $n$  where the base of the exponent is  $2^{\varepsilon n}$  times

$$\left(2.848^{0.26} \cdot 2^{1-h(0.26)}\right)^n.$$

Therefore, the running time of our 3-SAT algorithm is bounded by  $1.481^n$ .

## 6 Conclusion

In [15] the fourth author has shown that the idea of local search yields probabilistic algorithms whose running time is the best known for probabilistic 3-SAT algorithms. Here we show that local search can be also used to obtain fast deterministic algorithms for  $k$ -SAT. Similarly to [15], it is possible to extend our approach to the more general class of constraint satisfaction problems. In contrast to other deterministic  $k$ -SAT algorithms, our basic algorithm presented in Sect. 4 is technically very simple and has a better running time.

The improvement for 3-SAT given in Sect. 5 can be generalized to  $k$ -SAT. It is an open problem whether this method can be used to improve the complexity of the probabilistic algorithm from [15] (either for 3-SAT or for  $k$ -SAT).

## About This Paper

This paper resulted from merging two independent papers submitted simultaneously to two different conferences. Also, we recently learned that R. Kannan, J. Kleinberg, C. H. Papadimitriou and P. Raghavan independently obtained results similar to our results in Sect. 2–4.

U. Schöning wants to thank V. Arvind, S. Baumer, M. Bossert, J. Köbler, and P. Pudlák for valuable remarks and discussions. A. Goerdt thanks Oliver Kullmann for answering some questions. E. Dantsin and E. A. Hirsch are very grateful to Simon Litsyn who helped them to choose appropriate covering codes. They also thank Dima Grigoriev, Yuri Matiyasevich, Andrei Voronkov, and Maxim Vsemirnov for helpful discussions.

## References

1. R. B. Ash. *Information Theory*. Dover, 1965.
2. L. Bolc and J. Cytowski. *Search Methods for Artificial Intelligence*. Academic Press, 1992.

3. G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*, volume 54 of *Mathematical Library*. Elsevier, 1997.
4. E. Dantsin. Two propositional proof systems based on the splitting method (in Russian). *Zapiski Nauchnykh Seminarov LOMI*, 105:24–44, 1981. English translation: *Journal of Soviet Mathematics*, 22(3):1293–1305, 1983.
5. E. A. Hirsch. Two new upper bounds for SAT. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'98*, pages 521–530, 1998. A journal version is to appear in *Journal of Automated Reasoning*, 2000.
6. O. Kullmann. Worst-case analysis, 3-SAT decision and lower bounds: approaches for improved SAT algorithms. In D. Du, J. Gu, and P. M. Pardalos, editors, *Satisfiability Problem: Theory and Applications (DIMACS Workshop March 11-13, 1996)*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 261–313. AMS, 1997.
7. O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.
8. O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages, <http://www.cs.toronto.edu/~kullmann>. A journal version is submitted to *Information and Computation*, January 1997.
9. S. Minton, M. D. Johnston, A. B. Philips, and P. Lair. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
10. B. Monien and E. Speckenmeyer. Solving satisfiability in less than  $2^n$  steps. *Discrete Applied Mathematics*, 10:287–295, 1985.
11. C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS'91*, pages 163–169, 1991.
12. R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for  $k$ -SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS'98*, pages 628–637, 1998.
13. R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97*, pages 566–574, 1997.
14. I. Schiermeyer. Pure literal look ahead: An  $O(1, 497^n)$  3-satisfiability algorithm. Workshop on the Satisfiability Problem, Siena, April 29 – May 3, 1996. Technical Report 96-230, University Köln, 1996.
15. U. Schöning. A probabilistic algorithm for  $k$ -SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS'99*, pages 410–414, 1999.
16. W. Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155:277–288, 1996.