A Deterministic $\left(2 - \frac{2}{k+1}\right)^n$ Algorithm for k-SAT Based on Local Search

Evgeny Dantsin* Andreas Goerdt† Edward A. Hirsch‡ Ravi Kannan§
Jon Kleinberg¶ Christos Papadimitriou‖ Prabhakar Raghavan** Uwe Schöning††

Abstract

Local search is widely used for solving the propositional satisfiability problem. Papadimitriou [16] showed that randomized local search solves 2-SAT in polynomial time. Recently, Schöning [20] proved that a close algorithm for k-SAT takes time $(2 - \frac{2}{k})^n$ up to a polynomial factor. This is the best known worst-case upper bound for *randomized* 3-SAT algorithms (cf. also [21]).

We describe a *deterministic* local search algorithm for k-SAT running in time $(2 - \frac{2}{k+1})^n$ up to a polynomial factor. The key point of our algorithm is the use of covering codes instead of random choice of initial assignments. Compared to other "weakly exponential" algorithms, our algorithm is technically quite simple. We also describe an improved version of local search. For 3-SAT the improved algorithm runs in time 1.481^n up to a polynomial factor. Our bounds are better than all previous bounds for deterministic k-SAT algorithms.

1 Introduction

Worst-Case Upper Bounds for SAT. The satisfiability problem for propositional formulas (SAT) can be solved by an obvious algorithm in time $poly(n) \cdot 2^n$, where n is the number of variables in the input formula, and $poly(n)$ is a polynomial. 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). The first upper bounds $poly(n) \cdot c^n$, where $c < 2$, were obtained in 1980s [4, 15],

*Department of Computer Science, University of Manchester, Oxford Road, M13 9PL, UK. On leave from Steklov Institute of Mathematics, St.Petersburg, Russia. Email: dantsin@pdmi.ras.ru. Supported in part by grants from EPSRC, INTAS and RFBR.

†TU Chemnitz, Fakultät für Informatik, 09107 Chemnitz, Germany. Email: goerdt@informatik.tu-chemnitz.de.

‡Steklov Institute of Mathematics, 27 Fontanka, 191011 St.Petersburg, Russia. Email: hirsch@pdmi.ras.ru, Web: http://logic.pdmi.ras.ru/~hirsch. Supported in part by INTAS Fellowship YSF 99-4044 and grants from NATO and RFBR.

§Department of Computer Science, Yale University, New Haven CT 06520. Email: kannan@cs.yale.edu. Supported in part by NSF grant CCR-9820850.

¶Department of Computer Science, Cornell University, Ithaca NY 14853. Email: kleinber@cs.cornell.edu. Supported in part by a David and Lucile Packard Foundation Fellowship, an ONR Young Investigator Award, and NSF Faculty Early Career Development Award CCR-9701399.

‖Computer Science Division, Soda Hall, UC Berkeley, CA 94720. Email: christos@cs.berkeley.edu. Research supported by an NSF grant.

**Verity Inc., 892 Ross Drive, Sunnyvale CA 94089. Email: praghava@verity.com. Portions of this work were done while the author was at the IBM Almaden Research Center.

††Universität Ulm, Abteilung Theoretische Informatik, 89069 Ulm, Germany. Email: schoenin@informatik.uni-ulm.de.

for example, the bound $poly(n) \cdot 1.619^n$ for 3-SAT. Currently, much research in SAT algorithms is aimed at decreasing the base in exponential upper bounds, e.g., [23, 19, 13, 18, 17, 12, 20, 7].

The best known bound for randomized 3-SAT algorithms is $poly(n) \cdot (4/3)^n$ due to U. Schöning [20]. For randomized k-SAT algorithms when $k \geq 4$, the best known bound is due to R. Paturi, P. Pudlák, M. E. Saks, and F. Zane [17]. This bound is not represented in compact form; the bound for 4-SAT is $poly(n) \cdot 1.477^n$, the bound for 5-SAT is $poly(n) \cdot 1.569^n$.

The best known bounds for deterministic k-SAT algorithms are as follows. For $k = 3$, O. Kullmann [12] gives the bound $poly(n) \cdot 1.505^n$. In [19], the bound $poly(n) \cdot 1.497^n$ was announced ([11] sketched how this bound can be obtained by a refinement of [12]). For $k = 4$, the best known bound is still $poly(n) \cdot 1.840^n$ due to B. Monien and E. Speckenmeyer [15]. For $k \geq 5$, R. Paturi, P. Pudlák and F. Zane [18] give the bound approaching $poly(n) \cdot 2^{(1-1/(2k))n}$ for large k. In this paper we improve all these bounds for deterministic algorithms.

Randomized Local Search. Local search is a well-known heuristic search method [2, 14, 22]. A huge amount of experiments has demonstrated that local search is often very good in solving SAT for many classes of formulas, see [6] for survey. Also, there is a few average-case and worst-case bounds for local search in SAT, e.g., [16, 10, 5, 20, 8]. C. H. Papadimitriou showed [16] that 2-SAT can be solved in polynomial time by a randomized local search procedure. Our algorithm is inspired by Schöning's algorithm [20] which is close to Papadimitriou's algorithm and runs in time $poly(n) \cdot (2 - \frac{2}{k})^n$ for k-SAT. This is the best known worst-case upper bound for randomized 3-SAT algorithms.

Given a formula F in k-CNF with n variables, Schöning's 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 algorithm in [20] includes two randomized components: (1) the choice of initial assignments, and (2) local search starting from these assignments.

Deterministic Local Search. The deterministic k-SAT algorithm presented in this paper can be viewed as a derandomized version of the randomized 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 describe two algorithms that generate a good covering for given r. The first one constructs a covering whose cardinality is minimum up to a polynomial factor, but takes exponential space. The second one constructs an "almost" minimal covering and runs in polynomial space. In each ball of the covering, 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 $poly(n) \cdot (2 - \frac{2}{k+1})^n$. 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 $poly(n) \cdot 1.481^n$.

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 set of these words of length n is the *Hamming space* denoted by $H_n = \{0,1\}^n$. 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 simply a subset of H_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* ρ is defined by $\rho = r/n$.

By a *covering code* of radius r we mean a code understood as a covering of H_n by balls of radius r, i.e., every word in H_n belongs to at least one ball of radius r centered in a code word.

The paper is organized as follows: Section 2 describes the local search procedure. In Section 3 we show how to construct the codes used in our algorithms. Section 4 contains the main algorithm and its analysis. We describe the technique yielding the bound $poly(n) \cdot 1.481^n$ in Section 5. We discuss further developments in Section 6.

2 Local Search

Suppose we are given an initial assignment $a \in H_n$. Consider the ball of radius r around a. The number of assignments in this ball (the *volume* of the ball) is

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

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

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

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 ρ in the interval $0 < \rho \le 1/2$, 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 wish to check whether a satisfying assignment exists inside the ball of radius r around a. Then it is not necessary to search through all $V(n,r)$ 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 a satisfying 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 *Search*(F, a, r) performs a depth-first search as described below. It 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 *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 *Search*$(F_{|l=1}, a, r-1)$ returns *true* then return *true*, otherwise return *false*.

Lemma 2. If F has a satisfying assignment within distance r of a, then *Search*(F, a, r) will find a satisfying assignment.

Proof. The lemma easily follows by induction on r with the invariant: *Search*(F, a, r) outputs *true* iff F has a satisfying assignment inside the ball of radius r around a. For the induction step Lemma 1 is used. $\square$

Lemma 3. *Search*(F, a, r) runs in time $poly(n) \cdot k^r$.

Proof. The recursion depth of *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. $\square$

We observe that the exponential part k^r of this bound 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 *Search*$(F, a_0, n/2)$ and *Search*$(F, a_1, n/2)$, where a_0 and a_1 are n-bit assignments $a_0 = (0, 0, \ldots, 0)$ and $a_1 = (1, 1, \ldots, 1)$. 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 $poly(n) \cdot 1.733^n$. The set $\{a_0, a_1\}$ in this example is nothing else than a covering code of radius bounded by $n/2$.

3 Construction of Covering Code

For any covering code $\mathcal{C}$ 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, and $0 < \rho < 1/2$. This lower bound on $|\mathcal{C}|$ is known as the *sphere covering bound* [3]. The following lemma known in coding theory [3, Theorem 12.1.2, p. 320] shows the existence of covering codes whose size achieves the sphere covering bound up to a factor of n. We include the proof for completeness.

Lemma 4. For any $n \geq 1$ and r, there exists a covering code $\mathcal{C}$ of length n, covering radius at most r, and size at most

$$\lceil n \cdot 2^n / V(n,r) \rceil. \tag{2}$$

Proof. We show the existence of $\mathcal{C}$ by a probabilistic argument. Choose $n \cdot 2^n/V(n,r)$ elements of H_n uniformly at random with replacement. We now show that these elements form a covering code of radius at most r with non-zero (and, in fact, high) probability. Let a be a fixed element of H_n. It belongs to the Hamming ball of radius r around a randomly chosen element b of H_n with probability $V(n,r)/2^n$. The probability that a belongs to none of the balls of radius r around our randomly chosen elements is therefore

$$(1 - V(n,r)/2^n)^{n \cdot 2^n/V(n,r)} \leq e^{-n}$$

using $1 + x \leq e^x$ for all x. Thus the chosen elements form a covering code of radius at most r with probability at least $1 - 2^n \cdot e^{-n}$ which tends to 1 for $n \to \infty$. □

Corollary 1. Let $0 < \rho < 1/2$ and let $\beta(n) = \sqrt{n\rho(1-\rho)}$. For every n there exists a covering code $\mathcal{C}$ of length n, radius at most ρn, and size at most $n\beta(n) \cdot 2^{(1-h(\rho))n}$.

Proof. Follows from Lemma 4 and bound (1) on the volume:

$$n \cdot 2^n / V(n,r) \leq n\beta(n) \cdot 2^{(1-h(\rho))n}$$

□

The corollary provides the existence of a covering code of nearly minimum size and a randomized algorithm for constructing such a code (just choose it at random). We now describe two deterministic algorithms. The first one is a greedy algorithm that generates a covering code achieving the sphere covering bound up to a polynomial factor. However, this algorithm takes exponential space. The second algorithm constructs in polynomial space a covering code that "almost" achieves the sphere covering bound ("almost" means "up to the factor of $2^{\epsilon n}$" for arbitrary $\epsilon > 0$). Although the algorithms can be easily derived from results in the literature, the algorithms as such with the purpose of generating covering codes seem to be new.

First Algorithm. We can view the construction of a covering code as an instance of the *Set Cover* problem: We wish to cover the elements of H_n using as few balls of radius ρn as possible. The following greedy algorithm [9] approximates the minimum number of balls needed to within a factor of n: at each step, choose a ball that covers as many as-yet-uncovered elements as possible. Here is a naive estimate of the running time of this algorithm. We can associate with each ball the number of as-yet-uncovered elements it contains; in one iteration, we choose a ball for which this number is maximum, and update the numbers for all other balls. In this way, we can run each iteration in time $poly(n) \cdot 2^{2n}$, and there are at most 2^n iterations. Thus we have

Lemma 5. Let $n \geq 1$, $0 < \rho < 1/2$, and $\beta(n) = \sqrt{n\rho(1-\rho)}$. Then a covering code of length n, radius at most ρn, and size at most $n^2\beta(n) \cdot 2^{(1-h(\rho))n}$ can be constructed in time $poly(n) \cdot 2^{3n}$.

Unfortunately, the running time bound here is much too large. The following result trades off the size of the covering code for an improved running time.

Lemma 6. Let $d \geq 2$ be a divisor of $n \geq 1$, and $0 < \rho < 1/2$. Then there is a polynomial q_d such that a covering code of length n, radius at most ρn, and size at most $q_d(n) \cdot 2^{(1-h(\rho))n}$ can be constructed in time $q_d(n) \cdot (2^{3n/d} + 2^{(1-h(\rho))n})$.

Proof. We partition the n bits in the words of H_n into d *blocks* of length n/d each. We use Lemma 5 to construct a covering code $\mathcal{C}'$ of radius at most $\rho n/d$ for $H_{n/d}$. We then define $\mathcal{C}$ to be the direct sum of d instances of $\mathcal{C}'$, i.e., the set of all concatenations of d words from $\mathcal{C}'$.

It is easy to check that $\mathcal{C}$ is a covering code of radius at most ρn. For every $a \in H_n$, we can divide it into blocks $a_1, \ldots, a_d$ of length n/d each, and identify a word w_i in $\mathcal{C}'$ within distance $\rho n/d$ of each block a_i. The concatenation $w_1 w_2 \cdots w_d$ is then an element of $\mathcal{C}$ within distance ρn of a.

The time taken to construct $\mathcal{C}'$ is $O(q(n) \cdot 2^{3n/d})$. The size of $\mathcal{C}$ is at most $\left(n^2 \beta(n) \cdot 2^{(1-h(\rho))n/d}\right)^d = n^{2d} \beta(n)^d \cdot 2^{(1-h(\rho))n}$. □

Second Algorithm. In Lemma 6, each code word consists of a constant number d of blocks of length n/d. The blocks are *constructed* by the greedy algorithm. In the following lemma, a code word consists of a linear number n/b of blocks of a constant length b. These blocks are *hardwired* into the algorithm.

Lemma 7. Let $\delta > 0$ and $0 < \rho < 1/2$. There is a constant $b = b(\delta, \rho)$ such that for any $n = bl$, a covering code of length n, radius at most ρn, and size at most $2^{(1-h(\rho)+\delta)n}$ can be constructed in polynomial space using polynomial time per code word.

Proof. Corollary 1 implies that there is a code $\mathcal{C}' = \mathcal{C}'(\delta, \rho)$ of length $b = b(\delta, \rho)$, radius at most ρb, and size at most $2^{(1-h(\rho)+\delta)b}$. Clearly, the direct sum of n/b instances of $\mathcal{C}'$ is the required code. □

4 Main Algorithm and Its Analysis

Our main algorithm takes as input a formula F in k-CNF of n variables. The idea behind the algorithm is as follows. We cover H_n by balls of radius ρn using Lemma 6. The covering consists of $B(n, \rho, d) = q_d(n) \cdot 2^{(1-h(\rho))n}$ balls and its construction takes $T_1(n, \rho, d) = q_d(n) \cdot (2^{3n/d} + 2^{(1-h(\rho))n})$ time, where q_d is a polynomial. In each ball we invoke the local search procedure which takes $T_2(n, \rho) = p(n) \cdot k^{\rho n}$ time per ball, where p is a polynomial. The overall running time is therefore

$$T_1(n, \rho, d) + B(n, \rho, d) \cdot T_2(n, \rho). \tag{3}$$

Straightforward differentiation shows that we can minimize the exponential part of the product by choosing $\rho = 1/(k+1)$. The first term of (3) can be made smaller than the second one by choosing a suitable d, say, take $d = 6$.

For simplicity, we assume that n is divisible by d. Clearly, introducing a constant number of extra variables increases the running time bound by at most a constant factor.

Main Algorithm.

1. Set $\rho = 1/(k+1)$.

2. Use Lemma 6 with $d = 6$ to generate a covering code $\mathcal{C}$ of length n and radius at most ρn.

3. For each code word a in $\mathcal{C}$, run $\mathit{Search}(F, a, \rho n)$. Return *true* if at least one procedure call returns *true*. Otherwise return *false*.

Theorem 1. Main Algorithm solves k-SAT in time $\mathit{poly}(n) \cdot (2 - \frac{2}{k+1})^n$, where n is the number of variables in the input formula.

Proof. We calculate (3) as follows.

$$\begin{aligned}
&T_1(n,\rho,d) + B(n,\rho,d) \cdot T_2(n,\rho) \\
&= q_d(n) \cdot (2^{3n/d} + 2^{(1-h(\rho))n}) + q_d(n) \cdot 2^{(1-h(\rho))n} \cdot p(n) \cdot k^{\rho n} \\
&= \mathit{poly}(n) \cdot 2^{n(1+\frac{1}{k+1}\log_2\frac{1}{k+1}+\frac{k}{k+1}\log_2\frac{k}{k+1}+\frac{1}{k+1}\log_2 k)} \\
&= \mathit{poly}(n) \cdot 2^{n(1-\frac{1}{k+1}\log_2(k+1)+\frac{k}{k+1}\log_2 k-\frac{k}{k+1}\log_2(k+1)+\frac{1}{k+1}\log_2 k)} \\
&= \mathit{poly}(n) \cdot 2^{n(1+\log_2\frac{k}{k+1})} \\
&= \mathit{poly}(n) \cdot (2 - \tfrac{2}{k+1})^n.
\end{aligned}$$

□

Theorem 2. For any $\epsilon > 0$ and integer k, Main Algorithm can be modified so that it runs in time $\mathit{poly}(n) \cdot (2 - \frac{2}{k+1} + \epsilon)^n$ using polynomial space.

Proof. We modify Main Algorithm as follows: Instead of Lemma 6, we use Lemma 7 with $\delta = \log_2(\frac{k+1}{2k}\epsilon + 1)$. As above, we can assume that n is divisible by $b(\delta,\rho)$. Then the overall running time can be estimated as follows (up to a polynomial factor):

$$\begin{aligned}
&2^{(1-h(\rho)+\delta)n} \cdot k^{\rho n} \\
&= 2^{(1+\frac{1}{k+1}\log_2\frac{1}{k+1}+\frac{k}{k+1}\log_2\frac{k}{k+1}+\frac{1}{k+1}\log_2 k+\delta)n} \\
&= 2^{(1-\frac{1}{k+1}\log_2(k+1)+\frac{k}{k+1}\log_2 k-\frac{k}{k+1}\log_2(k+1)+\frac{1}{k+1}\log_2 k+\delta)n} \\
&= 2^{(1+\log_2\frac{k}{k+1}+\delta)n} \\
&= (2 - \tfrac{2}{k+1} + \epsilon)^n.
\end{aligned}$$

□

5 Improved Local Search

The local search procedure $\mathit{Search}(F, a, r)$ from Section 2 picks *any* false clause for branching. The complexity of this procedure is $\mathit{poly}(n) \cdot k^r$. 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 $\mathit{Search}(F, a, r)$ for formulas in 3-CNF so that its complexity is bounded by $\mathit{poly}(n) \cdot 2.848^r$ instead of $\mathit{poly}(n) \cdot 3^r$. We then obtain the bound $\mathit{poly}(n) \cdot 1.481^n$ for 3-SAT.

Let F be a 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

$$(\underbrace{1,\ldots,1}_{p},\underbrace{0,\ldots,0}_{m})\text{-}\mathit{clause}$$

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.

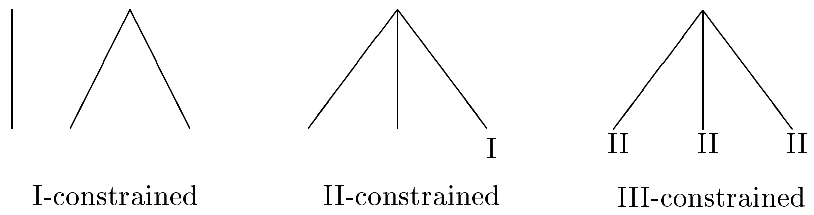

Figure 1: Branching I, II, III-constrained formulas.

We say that F is *d-false* under a ($d \geq 0$) iff F has no satisfying assignment within Hamming distance d from a. When d is fixed, we can test in polynomial time whether F is d-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 8. Let F be a formula false under a. If F contains a $(0,0,0)$-clause $l_1 \vee l_2 \vee l_3$ and a (1)-clause $\bar{l_i}$ where $i \in \{1,2,3\}$, then we have the following equivalence: F is satisfied within Hamming distance $\leq r$ from a iff there is a $j \in \{1,2,3\}$ such that $j \neq i$ and $F_{|l_j=1}$ is satisfied within Hamming distance $\leq r-1$ from a.

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 the types of branching that can occur:

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 9. 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: If $F_{|l=1}$ 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. Indeed, first, 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. Second, $F_{|l=1}$ is not the empty formula because F is 1-false 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 $\bar{k}_i \vee h_i$ is not present in $F_{|l=1}$. Then $F_{|l=1}$ has the (0,0,0)-clauses

$$l_1 \vee l_2 \vee l_3$$
$$k_i \vee k_{i,1} \vee k_{i,2}$$

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

$$l_1 \vee l_2 \vee l_3$$
$$k_i \vee k_{i,1} \vee k_{i,2}$$
$$h_i' \vee h_i'' \vee h_i'''$$

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 $l_1 \vee l_2 \vee l_3$, $k_i \vee k_{i,1} \vee k_{i,2}$ 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 9). If each $F_{|l_i=1}$ is III-constrained, we obtain from Lemma 9 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. If F has a satisfying assignment inside the ball of radius r around a then the procedure returns *true*. 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 $\bar{l_i}$ for some $i \in \{1, 2, 3\}$, 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 10. Let $L(F,a,r)$ be the number of leaves of the recursion tree of $\mathit{Search}(F,a,r)$. 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:

$$\begin{aligned}
&2 \cdot H(r-1) + 2 \cdot H(r-2) \\
&= 12 \cdot H(r-3) + 12 \cdot H(r-4) + 12 \cdot H(r-4) + 12 \cdot H(r-5) \\
&\leq 6H(r-2) + 6H(r-3) \\
&= H(r) \quad \text{(induction hypothesis)}
\end{aligned}$$

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 $\mathit{Search}(F,a,r)$ 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. Note that the inequalities above correspond to the cases of I- and II-constrained formulas, and the definition of H corresponds to the case of a III-constrained formula. Therefore, the claim follows by induction. $\square$

To obtain an explicit bound on the size of the recursion tree, we solve the recurrence for H. If α 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 α 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 $\rho = 0.26$ to minimize the exponential part of the product in (3), we obtain an algorithm which solves 3-SAT in time

$$\mathit{poly}(n) \cdot \left(2.848^{0.26} \cdot 2^{1-h(0.26)}\right)^n \leq \mathit{poly}(n) \cdot 1.481^n.$$

6 Conclusion

The algorithm in [20] uses local search to obtain the best known running time for randomized 3-SAT algorithms. A small improvement was recently obtained in [21]. Here we show that local search can be also used to obtain a fast deterministic algorithm for k-SAT. Similarly to [20], it is possible to extend our approach to the more general class of constraint satisfaction problems. Compared to other known deterministic k-SAT algorithms, our basic algorithm presented in Section 4 is technically very simple and has a better running time.

The improvement for 3-SAT given in Section 5 can be generalized to k-SAT. It is an open problem whether this method can also be used to improve the complexity of the randomized algorithm from [20] (either for 3-SAT or for k-SAT).

Acknowledgements

The authors thank V. Arvind, S. Baumer, M. Bossert, D. Grigoriev, J. Köbler, S. Litsyn, Yu. Matiyasevich, P. Pudlák, A. Voronkov, and M. Vsemirnov for valuable remarks and 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] J. Gu and Q.-P. Gu. Average time complexity of the sat1.2 algorithm. In *Proceedings of the 5th Annual International Symposium on Algorithms and Computation, ISAAC'94*, volume 834 of *Lecture Notes in Computer Science*, pages 146–154. Springer, 1994.

[6] J. Gu, P. Purdom, J. Franco, and B. W. Wah. *Algorithms for the Satisfiability Problem.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, January 2000.

[7] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.

[8] E. A. Hirsch. SAT local search algorithms: Worst-case study. *Journal of Automated Reasoning*, 24(1/2):127–143, 2000.

[9] D. S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, 1997.

[10] E. Koutsoupias and C. H. Papadimitriou. On the greedy algorithm for satisfiability. *Information Processing Letters*, 43(1):53–55, 1992.

[11] 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.

[12] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] U. Schöning, R. Schuler, and O. Watanabe. On choosing initial assigments for a probabilistic 3-sat algorithm. Preprint, submitted for publication, July 2000.

[22] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In W. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI-92*, pages 440–446. MIT Press, 1992.

[23] W. Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155:277–288, 1996.