

An Improved Upper Bound for SAT

Evgeny Dantsin and Alexander Wolpert

Roosevelt University,
430 S. Michigan Av.,
Chicago, IL 60605, USA
{edantsin, awolpert}@roosevelt.edu

Abstract. We give a randomized algorithm for testing satisfiability of Boolean formulas in conjunctive normal form with no restriction on clause length. Its running time is at most $2^{n(1-1/\alpha)}$ up to a polynomial factor, where $\alpha = \ln(m/n) + O(\ln \ln m)$ and n, m are respectively the number of variables and the number of clauses in the input formula. This bound is asymptotically better than the previously best known $2^{n(1-1/\log(2m))}$ bound for SAT.

1 Introduction

During the past few years there has been considerable progress in obtaining upper bounds on the complexity of solving the Boolean satisfiability problem. This line of research has produced new algorithms for k -SAT. They were further used to prove nontrivial upper bounds for SAT (no restriction on clause length).

Upper bounds for k -SAT. The best known upper bounds for k -SAT are based on two approaches: the satisfiability coding lemma [8, 7] and multistart random walk [10, 11]; both give close upper bounds on the time of solving k -SAT. The randomized algorithm in [10] has the $(2 - 2/k)^n$ bound where n is the number of variables in the input formula; the randomized algorithm in [7] has a slightly better bound. The multistart-random-walk approach is derandomized using covering codes in [2], which gives the best known $(2 - 2/(k + 1))^n$ bound for deterministic k -SAT algorithms. For small values of k , these bounds are improved: for example, 3-SAT can be solved by a randomized algorithm with the 1.324^n bound [6] and by a deterministic algorithm with the 1.473^n bound [1].

Upper bounds for SAT (no restriction on clause length). The first nontrivial upper bound for SAT is given in [9]: the $2^{n(1-1/2\sqrt{n})}$ bound for a randomized algorithm based on the satisfiability coding lemma. A close bound for a deterministic algorithm is proved in [3]. A much better bound for SAT is the $2^{n(1-1/\log(2m))}$ bound, where m is the number of clauses in the input formula. This bound is due to Schuler [12] who gives a randomized algorithm that solves SAT using the k -SAT algorithm [8] as a subroutine. Schuler's algorithm is derandomized in [4]. The derandomization gives a deterministic algorithm that solves SAT with the same bound.

In this paper we improve the $2^{n(1-1/\log(2m))}$ bound: we give a randomized algorithm that solves SAT with the following upper bound on the running time:

$$2^n \left(1 - \frac{1}{\ln\left(\frac{m}{n}\right) + O(\ln \ln m)} \right) \quad (1)$$

Idea of the algorithm. Our algorithm for SAT is basically a repetition of a polynomial-time procedure \mathcal{P} that tests satisfiability of an input formula F . If F is satisfied by a truth assignment A , the procedure \mathcal{P} finds A with probability at least p . As usual, repeating \mathcal{P} on the input formula $O(1/p)$ times, we can find A with a constant probability.

When describing \mathcal{P} , we view clauses as sequences (rather than sets) of literals. We divide each clause into *blocks* of length k . More exactly, for a given integer $k \geq 1$, a clause l_1, l_2, \dots, l_s is divided into $b = \lceil s/k \rceil$ blocks as follows:

$$\boxed{l_1, \dots, l_k}, \quad \boxed{l_{k+1}, \dots, l_{2k}}, \quad \dots \quad \boxed{l_{k(b-1)+1}, \dots, l_s}$$

where each block (except the last one) consists of k literals. By the *first block* we mean the block consisting of l_1, l_2, \dots, l_k . We say that a block is *true* under an assignment A if at least one literal in the block is true under A ; otherwise we say that the block is *false* under A . If A is fixed, we omit the words “under A ”.

Let F consist of clauses C_1, \dots, C_m ; let A be a fixed satisfying assignment to F . The procedure \mathcal{P} is based on the following dichotomy:

Case 1. The input formula F has “many” clauses in which the first block is false. We suppose that the number of such clauses is greater than or equal to some d . If we choose a clause C_i from C_1, \dots, C_m at random, the first block in C_i is false with probability at least d/m . Then we can simplify F by assigning “false” to all literals occurring in the first block of C_i .

Case 2. The input formula F has “few” clauses in which the first block is false. We suppose that the number of such clauses is less than d . Then we find A as follows. First, we guess those clauses in which the first block is false. Furthermore, we guess a true block in each such clause. Now we know a true block for each clause: it is either the first block or the block we have guessed. Let F' be the formula made up of these m true blocks. Obviously, F' is in k -CNF. Therefore we can use a k -SAT algorithm to find A .

This dichotomy suggests that \mathcal{P} is a recursive procedure that invokes a subroutine \mathcal{S} for processing Case 2. If the subroutine does not return a satisfying assignment, \mathcal{P} simplifies the input formula (Case 1) and recursively invokes itself on the simplified formula. Both \mathcal{P} and \mathcal{S} use k and d as parameters.

Clearly, the success probability of \mathcal{P} depends on values of the parameters k and d . What values of k and d maximize the success probability? We show that if we take $k \approx \log(m/n) + O(\log \log(m))$ and $d \approx n/\log^3 m$ then we obtain the following lower bound on the success probability:

$$2^{-n} \left(1 - \frac{1}{\ln\left(\frac{m}{n}\right) + O(\ln \ln m)} \right)$$

In Sect. 2 we describe the procedure \mathcal{P} and the subroutine \mathcal{S} . In Sect. 3 we define our algorithm for SAT and sketch a proof of bound (1) on its running time (see [5] for the full proof).

2 Procedure \mathcal{P} and Subroutine \mathcal{S}

2.1 Notation

We deal with Boolean formulas in conjunctive normal form (CNF). By a *variable* we mean a Boolean variable that takes truth values **t** (true) or **f** (false). A *literal* is a variable x or its negation $\neg x$. If l is a literal then $\neg l$ denotes the complement literal, i.e. if l is x then $\neg l$ denotes $\neg x$, and if l is $\neg x$ then $\neg l$ denotes x . Similarly, if v denotes one of the truth values **t** or **f**, we write $\neg v$ to denote the complement truth value. A *clause* C is a sequence of literals such that C contains no complement literals. A *formula* F is a set of clauses; n and m denote, respectively, the number of variables and the number of clauses in F . If each clause in F contains at most k literals, we say that F is a *k -CNF formula*.

An *assignment* to variables x_1, \dots, x_n is a mapping from $\{x_1, \dots, x_n\}$ to $\{\mathbf{t}, \mathbf{f}\}$. This mapping is extended to literals: each literal $\neg x_i$ is mapped to the truth value complement to the value assigned to x_i . We say that a clause C is *satisfied* by an assignment A (or, C is *true* under A) if A assigns **t** to at least one literal in C . Otherwise, we say that C is *falsified* by A (or, C is *false* under A). The formula F is *satisfied* by A if every clause in F is satisfied by A . In this case, A is called a *satisfying* assignment for F .

Let F be a formula and l_1, \dots, l_s be literals such that their variables occur in F . We write $F[l_1 = \mathbf{f}, \dots, l_s = \mathbf{f}]$ to denote the formula obtained from F by assigning the value **f** to all of l_1, \dots, l_s . This formula is obtained from F as follows: the clauses that contain any literal from $\neg l_1, \dots, \neg l_s$ are deleted from F , and the literals l_1, \dots, l_s are deleted from the other clauses. Note that $F[l_1 = \mathbf{f}, \dots, l_s = \mathbf{f}]$ may contain the empty clause or may be the empty formula.

Let A and A' be two assignments that differ only in the values assigned to a literal l . Then we say that A' is obtained from A by *flipping* the value of l .

By *SAT* we mean the following computational problem: Given a formula F in CNF, decide whether F is satisfiable. The *k -SAT* problem is the restricted version of SAT that allows only clauses consisting of at most k literals.

We write $\log_2 x$ to denote $\log_2 x$.

2.2 Description of \mathcal{S}

Both procedures \mathcal{S} and \mathcal{P} use the parameters k and d ; their values will be determined in the next section.

Suppose that an input formula F has a satisfying assignment such that F has only “few” ($< d$) clauses in which the first block is false under this assignment. Then the subroutine \mathcal{S} finds such an assignment (with some probability estimated in Sect. 3). The subroutine takes two steps:

1. Reduction of F to a k -CNF formula F' such that any satisfying assignment to F' satisfies F .
2. Use of a k -SAT algorithm to find a satisfying assignment to F' .

At the first step, \mathcal{S} guesses all “bad” clauses for some satisfying assignment A , i.e. clauses in which the first block is false under A . More exactly, the subroutine guesses a (possibly) larger set of clauses: a set $\{B_1, \dots, B_{d-1}\}$ such that all “bad” clauses are contained in this set. For each clause B_i , the subroutine guesses a true block in B_i . Thus, the subroutine gets a true block for each clause in F – the guessed true blocks for B_1, \dots, B_{d-1} and the first blocks for the other clauses in F . These true blocks make up F' . It is obvious that A satisfies F' .

To test satisfiability of k -CNF formulas at the second step, \mathcal{S} uses a randomized polynomial-time algorithm that finds a satisfying assignment with an exponentially small probability. We choose Schönig’s algorithm [10] to perform this testing (we could choose any algorithm that has at least the same success probability, for example the algorithm [7] based on the satisfiability coding lemma). More exactly, we use “one random walk” of Schönig’s algorithm, which has the success probability at least $(2 - 2/k)^{-n}$ up to a constant [11].

Note that if $d = 1$, i.e. there is no “bad” clause, then \mathcal{S} simply finds a satisfying assignment to the k -CNF formula made up of the m first blocks. Also note that the smaller d , the higher the probability of guessing a formula consisting of true blocks.

Subroutine \mathcal{S}

Input: Formula F with m clauses over n variables, integers k and d .

Output: Satisfying assignment or “no”.

1. Reduce F to a k -CNF formula F' as follows:
 - (a) Choose $d - 1$ clauses B_1, \dots, B_{d-1} in F at random.
Comment: Guess a set that contains all “bad” clauses.
 - (b) For each B_i , choose a block in B_i at random and replace B_i by the chosen block.
Comment: Guess a true block in each B_i and replace B_i by this true block.
 - (c) Replace each clause not belonging to $\{B_1, \dots, B_{d-1}\}$ by its first block.
Comment: The first block in each “good” clause is assumed to be true.
2. Test satisfiability of F' using one random walk of length $3n$ (see [10] for details):
 - (a) Choose an initial assignment a uniformly at random;
 - (b) Repeat $3n$ times:
 - i. If F' is satisfied by the assignment a then return a and stop;
 - ii. Pick any clause C in F' such that C is falsified by a . Choose a literal l in C uniformly at random. Modify a by flipping the value of l .
 - (c) Return “no”.

2.3 Description of \mathcal{P}

The procedure first calls Subroutine \mathcal{S} . The result of \mathcal{S} depends on which case of the dichotomy holds.

1. For every satisfying assignment A (if any), the input formula F has “many” ($\geq d$) clauses in which the first block is false under A . Then the subroutine never finds A .
2. For a satisfying assignment A , the input formula F has “few” ($< d$) clauses in which the first block is false. Then the subroutine returns a satisfying assignment with its success probability.

The “no” answer from \mathcal{S} is treated as Case 1 of the dichotomy. Therefore, the procedure \mathcal{P} simplifies F and recursively invokes itself on the simplified formula. To simplify F , the procedure chooses a clause C at random from the “long” clauses in F , i.e. the clauses that have more than one block. Then \mathcal{P} assigns f to all literals in the first block of C and reduces F to $F[l_1 = f, \dots, l_k = f]$. Why can we restrict the choice to “long” clauses? Because if the input formula is satisfiable then all one-block clauses must be true.

Procedure \mathcal{P}

Input: Formula F with m clauses over n variables, integers k and d .

Output: Satisfying assignment or “no”.

1. Invoke \mathcal{S} on F , k , and d .
Comment: If there are less than d “bad” clauses, the subroutine returns a satisfying assignment (with its success probability) and stops.
2. Choose clause C at random from those clauses in F that have more than one block.
Comment: We guess a “long” clause in which the first block is false.
3. Simplify F by assigning f to all literals of the first block in C . Namely, if the first block of C consists of l_1, \dots, l_k , reduce F to $F[l_1 = f, \dots, l_k = f]$.
Comment: We simplify F by eliminating the guessed variables.
4. Recursively invoke \mathcal{P} on $F[l_1 = f, \dots, l_k = f]$.
5. Return “no”.

Note that Schuler’s algorithm [12] can be viewed as a special case of \mathcal{P} : take $d = 1$, $k = \log(2m)$, and use a different method for testing k -CNF formulas at step 2 in \mathcal{S} (the algorithm based on the satisfiability coding lemma [8] instead of Schönig’s algorithm).

3 Main Result

Given an input formula F and some values of k and d , Procedure \mathcal{P} finds a fixed satisfying assignment A or returns “no”. What is the probability of finding A ? We prove the following lower bound on the success probability of \mathcal{P} :

$$\left(\frac{\sqrt{m}}{3e}\right) (emn)^{-(d-1)} 2^{-n\left(1-\frac{\log e}{k}\right)} \quad (2)$$

where n and m are respectively the number of variables and the number of clauses in F . The proof is more or less straightforward, but technical. We omit it here due to space limitation, see [5] for the full proof.

What values of k and d maximize bound (2) for given n and m ? The analysis in [5] shows that for n and m such that $10 \leq n < em \leq 2^{\sqrt[3]{n/2}}$, the maximum is attained when k and d are chosen as follows:

$$k_0 = \left\lceil \frac{\log\left(\frac{em}{n}\right) + 3 \log \log(em)}{1 + \frac{\log e}{\log^3(em)}} \right\rceil \quad d_0 = \left\lceil \frac{n}{\log^3(em)} \right\rceil \tag{3}$$

Using these values of the parameters, we define our main algorithm as a repetition of \mathcal{P} . The number r of repetitions is taken so that the success probability of \mathcal{P} is amplified to a constant probability. Namely, we substitute k_0 and d_0 in lower bound (2) and take the inverse:

$$r = \left\lceil \left(\frac{3en}{\sqrt{m}} \right) 2^{n\left(1 - \frac{\log e}{k_0 + 2}\right)} \right\rceil \tag{4}$$

Algorithm \mathcal{A}

Input: Formula F in CNF with m clauses over n variables.

Output: Satisfying assignment or “no”.

1. Compute k_0 and d_0 as in (3) and r as in (4).
2. Repeat the following r times:
 - (a) Run $\mathcal{P}(F, k_0, d_0)$;
 - (b) If a satisfying assignment is found, return it and stop.
3. Return “no”.

Theorem 1. *Algorithm \mathcal{A} runs in time*

$$O(n^3 m) 2^{n\left(1 - \frac{\log e}{k_0 + 2}\right)}$$

For any satisfiable input formula such that $10 \leq n < em \leq 2^{\sqrt[3]{n/2}}$, Algorithm \mathcal{A} finds a satisfying assignment with probability greater than $1/2$.

Proof (see [5] for details). To prove the first claim, we need to estimate the running time of \mathcal{P} . The procedure recursively invokes itself at most $\lfloor n/k_0 \rfloor$ times. Each call scans the input formula and eliminates at most k_0 variables. Therefore, the procedure performs $O(n)$ scans. Algorithm \mathcal{A} repeats \mathcal{P} at most r times, which gives the bound in the claim.

The success probability of \mathcal{A} is at least $1 - (1 - q(n, m))^r$ where $q(n, m)$ is the result of substituting k_0 and d_0 in lower bound (2), i.e. $q(n, m)$ is a lower bound on the success probability of \mathcal{P} for n, m, k_0 , and d_0 . Then the second claim follows from the inequality $(1 - x)^r \leq e^{-xr}$. □

Remark 1. Note that we can write the bound in Theorem 1 as

$$O(n^3 m) 2^{n \left(1 - \frac{1}{\ln\left(\frac{m}{n}\right) + O(\ln \ln m)} \right)}.$$

Remark 2. Our algorithm can be viewed as a generalization of Schuler's algorithm [12]. The latter is derandomized with the same upper bound on the running time [4]. It would be natural to try to apply the same method of derandomization to our algorithm. However, the direct application gives a deterministic algorithm with a much worse upper bound. Is it possible to derandomize Algorithm \mathcal{A} with the same or a slightly worse upper bound?

References

1. T. Brueggemann and W. Kern. An improved local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1-3):303–313, December 2004.
2. E. Dantsin, A. Goerd, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k -SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.
3. E. Dantsin, E. A. Hirsch, and A. Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science, STACS 2004*, volume 2996 of *Lecture Notes in Computer Science*, pages 141–151. Springer, March 2004.
4. E. Dantsin and A. Wolpert. Derandomization of Schuler's algorithm for SAT. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, pages 69–75, May 2004.
5. E. Dantsin and A. Wolpert. An improved upper bound for SAT. *Electronic Colloquium on Computational Complexity*, Report TR05-030, March 2005.
6. K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, page 328, January 2004. A preliminary version appeared in *Electronic Colloquium on Computational Complexity*, Report No. 53, July 2003.
7. 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.
8. 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.
9. P. Pudlák. Satisfiability – algorithms and logic. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 129–141. Springer-Verlag, 1998.
10. 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.

11. U. Schöning. A probabilistic algorithm for k -SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.
12. R. Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, January 2005. A preliminary version appeared as a technical report in 2003.