Derandomization of Schuler's Algorithm for SAT

Evgeny Dantsin and Alexander Wolpert

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

Abstract. Recently Schuler [17] presented a randomized algorithm that solves SAT in expected time at most $2^{n(1-1/\log_2(2m))}$ up to a polynomial factor, where n and m are, respectively, the number of variables and the number of clauses in the input formula. This bound is the best known upper bound for testing satisfiability of formulas in CNF with no restriction on clause length (for the case when m is not too large comparing to n). We derandomize this algorithm using deterministic k-SAT algorithms based on search in Hamming balls, and we prove that our deterministic algorithm has the same upper bound on the running time as Schuler's randomized algorithm.

1 Introduction

Known upper bounds

A natural way to evaluate a satisfiability-testing algorithm is to find an upper bound on its worst-case running time. Such bounds can be also used to compare algorithms with each other. Since the mid 80s there has been a "competition" for the "record" upper bounds for different versions of SAT. Typically, bounds for SAT have the form α^n up to a polynomial factor, where n is the number of variables in the input formula. The exponent's base α may be a constant ($\alpha < 2$) or may depend on parameters of input formulas (such as the number of variables or the number of clauses). A challenging problem is to lower α as much as we can.

The currently best known upper bounds are described below (we give only the exponential terms of the bounds, omitting polynomial factors). We discuss four groups of bounds: randomized algorithms for k-SAT and SAT, deterministic algorithms for k-SAT and SAT. We also mention main ideas in the underlying algorithms.

Figure 1 summarizes the "record" upper bounds.

Randomized algorithms for k-SAT. All "record" randomized algorithms for k-SAT use one (or both) of the following two approaches:

- Random-assignment generation combined with unit clause elimination and bounded resolution (Paturi, Pudlák, Saks, Zane [12, 11]);
- Multistart random walk (Schöning [15, 16]).

The best known bounds for 3-SAT and 4-SAT are obtained using an algorithm based on a combination of both methods, namely: 1.324^n for 3-SAT and 1.474^n for 4-SAT [9]. Other recent algorithms for 3-SAT, e.g. [1, 8, 14], follow up the multistart random walk approach. The bounds obtained using [11] are close: 1.362^n and 1.476^n for 3-SAT and 4-SAT respectively.

The best known bounds for $k > 4$ are due to the Paturi-Pudlák-Saks-Zane algorithm in [11]:

$$2^{n\left(1-\frac{\mu_k}{k-1}\right)+o(n)}$$

where $\mu_k \to \pi^2/6$ as $k \to \infty$. In particular, for $k = 5$ and $k = 6$, this gives 1.569^n and 1.637^n respectively. The multistart random walk algorithm [15] gives close bounds:

$$\left(2 - \frac{2}{k}\right)^n.$$

Deterministic algorithms for k-SAT. Until recently, “record” upper bounds for k-SAT were obtained usind DPLL-like algorithms [6, 5], for example the 1.505^n bound for 3-SAT [10]. Newer deterministic algorithms borrow ideas from randomized approaches to testing satisfiability. The algorithms in [3, 2] that have the best known upper bounds for k-SAT are based on the derandomization of multistart random walk. They cover the Boolean cube $\{0,1\}^n$ by Hamming balls and apply a local search method to find a satisfying assignment inside these balls. The “record” bound is

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

For $k = 3$, the bound can be improved to 1.481^n.

Randomized algorithms for SAT (no limit on clause length). The best known bound was proved by R. Schuler in [17]. His algorithm uses a combination of the Paturi-Pudlák-Saks-Zane algorithm [11] and “clause shortening” (see Sect. 2 for details). The bound is

$$2^{n\left(1-\frac{1}{\log(2m)}\right)}$$

where m is the number of clauses in the input formula and $\log x$ denotes $\log_2 x$. Also, there is another bound: $2^{n-c\sqrt{n}}$, where c is a constant. This bound is due to two different algorithms. One algorithm [13] uses the Paturi-Pudlák-Saks-Zane algorithm in combination with the DPLL approach. The second algorithm [4] is based on multistart search in Hamming balls: Generate a random assignment and use local search to find a solution within a certain Hamming distance around this assignment. Schuler’s bound [17] is more interesting because it is better than $2^{n-c\sqrt{n}}$ for the case when m is not too large comparing to n, namely when $m = o(2^{\sqrt{n}})$.

Deterministic algorithms for SAT (no limit on clause length). Up until now, the only non-trivial upper bound for deterministic SAT algorithms has been given in [4]:

$$2^{n\left(1-\frac{2}{\sqrt{n \log n}}\right)} .$$

The corresponding algorithm is a derandomized version of multistart search in Hamming balls. The derandomization is based on covering codes. In the case of deterministic algorithms for SAT, there are also other types of bounds that are “more” dependent on the number of clauses or other input parameters, e.g., 1.239^m [7].

In this paper we give a deterministic algorithm that has the same bound

$$2^{n\left(1-\frac{1}{\log(2m)}\right)}$$

as in the case of randomized algorithms for SAT.

	Randomized algorithms	Deterministic algorithms
3-SAT	1.324^n [9]	1.481^n [3]
4-SAT	1.474^n [9]	1.6^n [2]
k-SAT ($k > 4$)	$2^{n\left(1-\frac{\mu_k}{k-1}\right)+o(n)}$ where $\lim_{k\to\infty} \mu_k = \pi^2/6$ [11]	$\left(2 - \frac{2}{k+1}\right)^n$ [2]
SAT	$2^{n\left(1-\frac{1}{\log(2m)}\right)}$ [17]	$2^{n\left(1-\frac{1}{\log(2m)}\right)}$ [this paper]

Fig. 1. “Record” worst-case upper bounds for k-SAT and SAT.

Our result

We prove that SAT can be solved by a deterministic algorithm with the same upper bound on the running time as Schuler's randomized algorithm, i.e., with the bound $2^{n(1-1/\log_2(2m))}$ up to a polynomial factor.

Like Schuler's algorithm, our deterministic algorithm can be described in terms of two algorithms $\mathcal{M}$ (stands for *Main*) and $\mathcal{S}$ (stands for *Subroutine*). The algorithm $\mathcal{S}$ is used to test satisfiability of formulas with "short" clauses (of length at most $\log(2m)$). The algorithm $\mathcal{M}$ is the main algorithm that transforms an input formula F into to F' by "shortening" the clauses in F. Then $\mathcal{M}$ invokes $\mathcal{S}$ to check whether F' is satisfiable. If so, we are done. Otherwise, the algorithm $\mathcal{M}$ simplifies F and recursively invokes itself on the results of simplification.

Theorem 1 in Sect. 3 gives an upper bound on the running time of the algorithm $\mathcal{M}$ under an assumption on the running time of the subroutine $\mathcal{S}$. More exactly, the assumption is that $\mathcal{S}$ runs in time at most $2^{n(1-1/k)}$ up to a polynomial factor, where k is the maximum length of clauses in F. Then $\mathcal{M}$ runs in time at most $2^{n(1-1/\log(2m))}$ up to a polynomial factor. Does there exist any deterministic subroutine $\mathcal{S}$ that meets this assumption? The answer is positive (Theorem 2): the algorithms [2] have the required upper bound on the running time. Thus, taking any of them as the subroutine $\mathcal{S}$, we obtain a deterministic algorithm that solves SAT with the bound

$$2^{n\left(1-\frac{1}{\log(2m)}\right)}.$$

Notation

By a *formula* we mean Boolean formulas in conjunctive normal form (CNF) defined as follows. A *literal* is a Boolean variable x or its negation $\neg x$. A *clause* is a finite set C of literals such that C contains no opposite literals. The *length* of C (denoted by $|C|$) is the number of literals in C. A *formula* is a set of clauses. An *assignment* to variables $x_1, \ldots, x_n$ is a mapping from $\{x_1, \ldots, x_n\}$ to the truth values $\{\text{TRUE}, \text{FALSE}\}$. This mapping is extended to literals: each literal $\neg x_i$ is mapped to the truth value opposite to the value assigned to x_i. We say that a clause C is *satisfied* by an assignment A if A assigns TRUE to at least one literal in C. 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.

By *SAT* we mean the following computational problem: Given a formula F in CNF, decide whether F is satisfiable or not. The *k-SAT* problem is the restricted version of SAT that allows only clauses of length at most k.

Here is a summary of the notation used in the paper.

– F denotes a formula;
– n denotes the number of variables in F;
– m denotes the number of clauses in F;
– k denotes the maximum length of clauses in F;
– $|C|$ denotes the length of clause C;
– $\log x$ denotes $\log_2 x$.

2 Algorithms Based on Clause Shortening

Schuler's algorithm

We first sketch Schuler's algorithm [17]. More exactly, we describe a polynomial-time randomized procedure $\mathcal{R}$ that finds a satisfying assignment (if any) with probability at least $2^{-n(1-1/\log(2m))}$. This probability can be increased to a constant by repetitions in the usual way. The procedure $\mathcal{R}$ tests satisfiability in two steps:

1. Convert the input formula to a formula in k-CNF where $k = \log(2m)$;
2. Use a k-SAT algorithm to test satsfiability of the resulting formula.

Let F be an input formula consisting of clauses $C_1, \ldots, C_m$. Assuming that F is satisfied by an (unknown) assignment A, we show how $\mathcal{R}$ finds A. The procedure $\mathcal{R}$ starts with shortening the clauses in F as follows:

1. For each clause C_i such that $|C_i| > \log(2m)$, choose any $\log(2m)$ literals in C_i and delete the other literals.
2. Leave the shorter clauses as is.

Let $F' = \{D_1, \ldots, D_m\}$ be the result of the shortening. Obviously, any satisfying assignment for F' also satisfies F. The formula F' is in k-CNF where $k = \log(2m)$. Therefore, a satisfying assignment for F' (if any) can be found using a k-SAT algorithm. The procedure $\mathcal{R}$ uses the Paturi-Pudlák-Zane method from [12] to find a satisfying assignment for F' in polynomial time with probability at least $2^{-n(1-1/k)}$. There are two possible cases:

Case 1. A satisfies F'. Then the Paturi-Pudlák-Zane method finds A in polynomial time with probability at least $2^{-n(1-1/\log(2m))}$.

Case 2. A does not satisfy F'. Then there is a clause D_i such that all of its literals are false under A (but C_i is true under A). Therefore, if we "guess" this clause correctly, we may simplify F by assigning FALSE to all literals occurring in D_i. We choose a clause in F' uniformly at random. The probability that we have "guessed" the clause correctly (i.e., we have chosen D_i) is at least $1/m$. Then we simplify F' as follows:

   1. For each literal l in the chosen clause, remove all clauses that contain $\neg l$;
   2. Delete l from the remaining clauses.

   Finally, we recursively apply $\mathcal{R}$ to the result of the simplification.

The analysis of $\mathcal{R}$ in [17] shows that $\mathcal{R}$ finds A with the required probability. Note that the same bound holds if the Paturi-Pudlák-Zane method (used as a subroutine in $\mathcal{R}$) is replaced by another procedure that finds a satisfying assignment in polynomial time with the same or higher probability, for example by Schöning's random-walk method [15].

Algorithms $\mathcal{M}$ (for Main) and $\mathcal{S}$ (for Subroutine)

Schuler's algorithm invokes the Paturi-Pudlák-Saks-Zane procedure [11] for testing satisfiability of formulas with "short" clauses. Our derandomized version will also use a subroutine to check formulas with "short" clauses. However, we first describe our algorithm without specifying the invoked subroutine. That is, assuming that $\mathcal{S}$ is an arbitrary satisfiability-testing procedure, we define our main algorithm $\mathcal{M}$ as an algorithm that invokes $\mathcal{S}$ as a subroutine.

Algorithm $\mathcal{S}$ Input: Formula F (with no restriction on clause length).
Output: Satisfying assignment or "no".
Any method of testing satisiability.

Algorithm $\mathcal{M}$ Input: Formula F with clauses $C_1, \ldots, C_m$ over n variables.
Output: Satisfying assignment or "no".

1. Change each clause C_i to a clause D_i as follows: If $|C_i| > \log(2m)$ then choose any $\log(2m)$ literals in C_i and delete the other literals; otherwise leave C_i as is, i.e., $D_i = C_i$. Let F' denote the resulting formula.
2. Test satisfiability of F' using the algorithm $\mathcal{S}$.
3. If F' is satisfiable, return the satisfying assignment found in the previous step. Otherwise, for each clause D_i in F', do the following:
   (a) Convert F to F_i by assigning FALSE to all literals in D_i. Namely, for each literal l in D_i, remove all clauses containing $\neg l$ and delete l from the remaining clauses.
   (b) Recursively invoke $\mathcal{M}$ on F_i.
4. Return "no".

3 Bound for SAT

The choice of the subroutine $\mathcal{S}$ determines the main algorithm $\mathcal{M}$. In this section, we specify $\mathcal{S}$ so that the algorithm $\mathcal{M}$ solves SAT in time at most $2^{n(1-1/(\log(2m)))}$ up to a polynomial factor. First, we prove an upper bound for $\mathcal{M}$ assuming a specific upper bound on the running time of $\mathcal{S}$. Then we choose a subroutine that meets this assumption. As a result, we obtain the claimed upper bound for the main algorithm $\mathcal{M}$.

Theorem 1. *Suppose that for any formula F, the algorithm $\mathcal{S}$ runs on F in time at most*

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

up to a polynomial factor, where k is the maximum length of clauses in F. Then the running time of the algorithm $\mathcal{M}$ is at most

$$2^{n\left(1-\frac{1}{\log(2m)}\right)}$$

up to a polynomial factor.

Proof. Let $t_{\mathcal{S}}(F)$ and $t_{\mathcal{M}}(F)$ be, respectively, the running times of the algorithms $\mathcal{S}$ and $\mathcal{M}$ on a formula F. It is not difficult to see that $t_{\mathcal{M}}(F)$ can be estimated (up to a polynomial factor) as follows:

$$t_{\mathcal{M}}(F) \le t_{\mathcal{S}}(F') + m \cdot t_{\mathcal{M}}(F_i) \tag{1}$$

where F' and F_i are as described in the algorithm $\mathcal{M}$. Let $T_{\mathcal{M}}(n,m)$ denote the maximum of the running time of $\mathcal{M}$ on formulas with m clauses over n variables. For the subroutine $\mathcal{S}$, we define $T_{\mathcal{S}}(n,m)$ as the maximum running time on a different set of formulas, namely let $T_{\mathcal{S}}(n,m)$ be the maximum of the running time of $\mathcal{S}$ on the set of formulas F such that each F has m clauses over n variables and the maximum length of clauses is not greater than $\log(2m)$. Let L denote $\log(2m)$. Then for any n and m, the inequality (1) implies the following recurrence relation:

$$T_{\mathcal{M}}(n,m) \le T_{\mathcal{S}}(n,m) + m \cdot T_{\mathcal{M}}(n-L,m)$$

Iterating this recurrence and using the bound on $t_{\mathcal{S}}(F)$ with $k \le L$, we get (again up to a polynomial factor)

$$\begin{aligned} T_{\mathcal{M}}(n,m) &\le \sum_{i=0}^{n/L} m^i \cdot T_{\mathcal{S}}(n-iL,m) \\ &\le \sum_{i=0}^{n/L} m^i \cdot 2^{(n-iL)(1-1/L)} \;=\; 2^{n(1-1/L)} \sum_{i=0}^{n/L} \left(m \cdot 2^{1-L}\right)^i \end{aligned}$$

Since $L = \log(2m)$, we have $m\,2^{1-L} = 1$. Therefore,

$$t_{\mathcal{M}}(F) \le T_{\mathcal{M}}(n,m) \le 2^{n(1-1/L)}$$

up to a polynomial factor. □

Theorem 2 (based on [2]). *There exists a deterministic algorithm that tests satisfiability of an input formula F in time at most*

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

up to a polynomial factor, where n is the number of variables in F, and k is the maximum length of clauses in F.

Proof. Paper [2] defines two algorithms that can be applied to any formula. Their running times are estimated in terms of the maximum length of clauses in the input formula (thus, they can be viewed as algorithms for k-SAT). Both algorithms cover the Boolean cube $\{0,1\}^n$ by Hamming balls and search for a satisfying assignment inside these balls. The first algorithm runs in time at most $2^{n(1-\log(1+1/k))}$ up to a polynomial factor (Theorem 1 in [2]). Since

$$\log\left(1+\frac{1}{k}\right) = \frac{\log e}{k} + o\left(\frac{1}{k}\right),$$

this algorithm meets the claim. The second algorithm has a parameter δ; its running time is at most

$$2^{n\left(1-\log\left(1+\frac{1}{k}\right)+\delta\right)}$$

up to a polynomial factor (Theorem 2 in [2]). Taking $\delta \le \frac{\log(e/2)}{k}$, we have

$$2^{n\left(1-\log\left(1+\frac{1}{k}\right)+\delta\right)} \;\le\; 2^{n\left(1-\frac{\log e}{k}+\frac{\log(e/2)}{k}\right)} \;\le\; 2^{n\left(1-\frac{1}{k}\right)}.$$

Hence, the second algorithm also meets the claim.

The algorithms differ in the construction of the covering of $\{0,1\}^n$ by Hamming balls. The first algorithm uses a greedy method to construct the covering that is minimal up to a polynomial factor. The construction requires an exponential space (approximately $2^{n/6}$). The second algorithm constructs a "nearly minimal" covering, i.e., a covering that is minimal up to a factor of $2^{\delta n}$, where δ can be chosen arbitrary small.

To estimate the space used by the second algorithm, we have to consider details of how it constructs the covering of $\{0,1\}^n$. Each ball center is the concatenation of n/b blocks of length b (Lemma 7 in [2]). The algorithm constructs a covering code $\mathcal{C}$ of length b for blocks. Then, keeping this code in memory, the algorithm generates code words of length n (centers of balls) one by one. An upper bound on the space can be estimated as the cardinality of the covering code $\mathcal{C}$ for blocks. Using Lemma 4 in [2], we can estimate the cadrinality $|\mathcal{C}|$ as follows:

$$|\mathcal{C}| \;\le\; b\sqrt{b}\; 2^{b\left(1-H\left(\frac{1}{k+1}\right)\right)} \tag{2}$$

where $H(x) = -x\log x - (1-x)\log(1-x)$ is the binary entropy function. To obtain the desired upper bound on the running time, we should choose b so that

$$|\mathcal{C}|^{n/b} \;\le\; 2^{n\left(1-H\left(\frac{1}{k+1}\right)+\delta\right)}. \tag{3}$$

Using the bound (2) and the inequality (3), we get the following constraint on b:

$$\left(b\sqrt{b}\; 2^{b\left(1-H\left(\frac{1}{k+1}\right)\right)}\right)^{n/b} \;\le\; 2^{n\left(1-H\left(\frac{1}{k+1}\right)+\delta\right)} \tag{4}$$

which is equivalent to $(b\sqrt{b})^{1/b} \le 2^{\delta}$. Now we substitute $\delta = \frac{\log(e/2)}{k}$ and take $b = 4k\log k$. Then (4) holds for all sufficiently large k. Therefore, we can use blocks of length $4k\log k$. In fact, the algorithm will be applied to formulas with $k = \log(2m)$, which gives the upper bound

$$(2m)^{4\log\log(2m)}$$

on the space. □

Theorem 3. *Suppose that the algorithm $\mathcal{M}$ uses the algorithm from Theorem 2 as the subroutine $\mathcal{S}$. Then $\mathcal{M}$ tests satisfiability of an input formula F with m clauses over n variables in time at most*

$$2^{n\left(1-\frac{1}{\log(2m)}\right)}$$

up to a polynomial factor.

Proof. Immediately follows from Theorems 1 and 2. □

References

1. S. Baumer and R. Schuler. Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs. Electronic Colloquium on Computational Complexity, Report No. 10, February 2003.
2. E. Dantsin, A. Goerdt, 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, October 2002.
3. E. Dantsin, A. Goerdt, E. A. Hirsch, and U. Schöning. Deterministic algorithms for k-SAT based on covering codes and local search. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming, ICALP'2000*, volume 1853 of *Lecture Notes in Computer Science*, pages 236–247. Springer, July 2000.
4. 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.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
7. E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.
8. T. Hofmeister, U. Schöning, R. Schuler, and O. Watanabe. A probabilistic 3-SAT algorithm further improved. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Scienceg, STACS 2002*, volume 2285 of *Lecture Notes in Computer Science*, pages 192–202. Springer, March 2002.
9. K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. Electronic Colloquium on Computational Complexity, Report No. 53, July 2003.
10. O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.
11. 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.
12. 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.
13. 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.
14. D. Rolf. 3-SAT in $RTIME(O(1.32793^n))$ — improving randomized local search by initializing strings of 3-clauses. Electronic Colloquium on Computational Complexity, Report No. 54, July 2003.
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. U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.
17. R. Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. To appear in *Journal of Algorithms*, 2003.