

# Algorithmics in Exponential Time

Uwe Schöning

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

**Abstract.** Exponential algorithms, i.e. algorithms of complexity  $O(c^n)$  for some  $c > 1$ , seem to be unavoidable in the case of NP-complete problems (unless  $P=NP$ ), especially if the problem in question needs to be solved exactly and not approximately. If the constant  $c$  is close to 1 such algorithms have practical importance. Deterministic algorithms of exponential complexity usually involve some kind of backtracking. The analysis of such backtracking algorithms in terms of solving recurrence equations is quite well understood. The purpose of the current paper is to show cases in which the constant  $c$  could be significantly reduced, and to point out that there are some randomized exponential-time algorithms which use randomization in some new ways. Most of our examples refer to the 3-SAT problem, i.e. the problem of determining satisfiability of formulas in conjunctive normal form with at most 3 literals per clause.

## 1 Why Exponential-Time Algorithms?

Unless  $P=NP$ , exponential (or at least non-polynomial) algorithms are unavoidable for NP-complete problems such as SAT or 3-SAT. This is especially so if the problem in question needs to be solved or decided exactly, and when there is no use of any type of approximation algorithm.

Having accepted that we have to deal with exponential-time algorithms only, it makes sense to improve the relative efficiency of our algorithms by reducing the value of the base constant  $c$  in the exponential time bound  $O(c^n)$ . New algorithms which are able to reduce the constant  $c$  can mean an tremendous improvement. Moving from an algorithm with complexity  $O(c^n)$  to another, better one, with complexity  $O(d^n)$  where  $d = \sqrt{c}$  means that, within the same given time limit, the input size that can be solved by the new algorithm, doubles. In concrete terms, improving the brute-force algorithm for 3-SAT of complexity  $O(2^n)$  (where  $n$  is the number of Boolean variables) to another one of complexity  $O(1.324^n)$  (as it was indeed the case within the last years [14, 17, 10]). This improvement allows to increase the number of tractable Boolean variables by a factor of more than 2.4.

Another motivating example are SAT-solvers which are extremely useful general-purpose programs which are operating in worst-case exponential-time.

## 2 Backtracking

Most deterministic exponential-time algorithms use some version of backtracking. For example, one can try out all potential  $2^n$  many assignments for a Boolean formula in  $n$  variables in a backtracking manner. In each level of the recursive backtrack tree one more variable is assigned one of the two Boolean values true or false. This direct approach leads to the recursion  $T(n) \leq 2 \cdot T(n-1)$ , giving the complexity  $T(n) = O(2^n)$ .

More sophisticated approaches can reduce the number of variables in the recursive procedure calls not only from  $n$  to  $n-1$ , but to  $n-2$ ,  $n-3$ , etc. The 3-SAT algorithm of Monien and Speckenmeyer [11], for example, first chooses a shortest clause in the formula which, in the worst case, has 3 literals, say  $x, y, z$ . The first recursive call of the procedure assigns  $x = 1$  and simplifies the formula accordingly. If this recursive call is not successful (no satisfying assignment is found), then the partial assignment ( $x = 0, y = 1$ ) is tried next. If again unsuccessful, the last recursive call is done with respect to the partial assignment ( $x = 0, y = 0, z = 1$ ). This leads to the recursion  $T(n) \leq T(n-1) + T(n-2) + T(n-3)$  with the solution  $T(n) = O(1.84^n)$ .

This algorithm has been trimmed even more in [11]. Using the concept of a-tark partial assignments, the new recursive tree structure leads to the equations  $T(n) \leq T'(n-1) + T'(n-2) + T'(n-3)$ ,  $T'(n) \leq \max\{T(n-1), T'(n-1) + T'(n-2)\}$  having the solution  $T(n) = O(1.619^n)$ .

Consider as another example the problem of 3-coloring a given graph with  $n$  vertices. The very naive approach to solve this problem is to assign to each vertex independently one of the 3 colors and try all cases in brute-force manner. This leads to an  $O(3^n)$  algorithm. The next (still) naive approach is to start in a backtracking way with a first node, assign it color 1, then backtrack through all its neighbors and try out all 2 remaining colors. Continue recursively by considering the neighbor vertices which have not been colored yet and try both remaining possibilities. This leads to a  $O(2^n)$  algorithm. The next option is to notice that one of the 3 colors cannot occur more often than  $\binom{n}{n/3}$  many times. Hence, we systematically assign color 1 to at most  $\binom{n}{n/3}$  vertices in the graph. These are

$$\sum_{i=0}^{n/3} \binom{n}{i} \leq 2^{h(1/3)n} \leq 1.89^n$$

many possibilities (where  $h$  is the entropy function [2].) In all those cases where there is no edge between any two vertices with color 1, the remaining graph needs to be 2-colored. Whether this is possible can be determined in polynomial time. Therefore, this procedure has complexity  $O(1.89^n)$ .

## 3 Local Search

Local Search starts with a given assignment. If this assignment is not yet a solution (i.e. a satisfying assignment in case of the SAT problem), the actual

assignment is modified. These modifications are typically very local, for example, a 1 bit flip is performed (one variable which has the actual value true is set to false, or vice versa). These modifications (or "mutations" in evolutionary algorithms terminology) are either done in a random or some deterministic, systematic fashion. The better algorithms of this kind use in some sense the (mis-)behavior of the actual assignment on the input to get some kind of "hint" what modification might be useful and will probably lead in the right direction. "Leading in the right direction" can be quantified in terms of the Hamming distance between the actual assignment and some fixed solution assignment (i.e. satisfying assignment).

In [16, 7, 8] a deterministic backtracking-like recursive procedure  $search(F, a, d)$  is used for solving 3-SAT. This procedure returns true, if there exists a satisfying assignment  $a^*$  for the Boolean formula  $F$  which is within Hamming distance at most  $d$  from the given initial assignment  $a$ . The Hamming distance between two bit vectors of equal length is the number of bits in which they differ. This procedure  $search$  works as follows. First it checks whether one of the trivial cases occurs. If for example  $a$  already satisfies  $F$  (leading to the returned value true), or else, if  $d = 0$  (leading to the returned value false). If these cases do not occur, then  $a$  does not satisfy  $F$  and  $d > 0$ . Since  $a$  does not satisfy  $F$  there is a clause  $C$  in  $F$  which is not satisfied by  $a$ . All  $\leq 3$  literals in  $C$  are set to false by  $a$ . Under the satisfying assignment  $a^*$  that we are looking for, at least one of these literals must be set to true. Hence, if we flip the value of one of the variables in  $C$ , we make the Hamming distance between  $a$  and  $a^*$  smaller by 1. Therefore, we perform 3 recursive calls, in each recursive call one of the bits in the assignment  $a$  which corresponds to a variable in  $C$  is flipped. Further, the parameter  $d$  is reduced to  $d - 1$ . It is clear that this procedure needs time  $T(d) \leq 3 \cdot T(d - 1)$ , hence  $T(d) = O(3^d)$  (ignoring polynomial factors). Now we have to discuss the question, how the initial assignments  $a$  come about. The easiest case is that we take just two initial assignments  $0^n$  and  $1^n$  and set  $d = n/2$ . This gives us already a very simple 3-SAT algorithm of complexity  $O(3^{n/2}) = O(1.74^n)$ .

Next we push this further, and choose the initial assignments systematically from some precomputed list  $L = \{a_1, a_2, a_3, \dots, a_t\}$  where  $t$  is an exponential function in  $n$ , the number of variables. This list  $L$ , together with some suitable chosen Hamming distance  $d$ , should be a so called covering code [6], i.e. every bit vector of length  $n$  should be within Hamming distance at most  $d$  to at least one  $a_i \in L$ . It turns out that the optimal choice for  $d$  is  $n/4$ , and  $t = 1.14^n$  (together with an appropriate choice of the  $a_i$ ). This gives the overall complexity bound  $t \cdot 3^d = 1.14^n \cdot 3^{n/4} = 1.5^n$ .

In [7, 8] it is shown how the procedure  $search$  can be further modified such that the complexity of searching up to Hamming distance  $d$  can be estimated as  $T(d) \leq 6 \cdot T(d - 2) + 6 \cdot T(d - 3)$  with the solution  $T(d) = O(2.85^d)$ . Using this bound and modifying the choices of  $d$  and  $t$  to  $d = 0.26n$  and  $t = 1.13^n$  gives the overall complexity bound  $t \cdot 2.85^d = 1.13^n \cdot 2.85^{0.26n} = 1.481^n$ . (This was recently improved to  $1.473^n$  [5].)

## 4 Randomization: Reducing to a Polynomial-Time Case

Up to now, we have considered deterministic strategies which are able to reduce the exponential constant  $c$  as compared to the direct brute-force strategy (which usually means that the constant is  $c = 2$ ). Randomization can help in various ways to improve such algorithms. Many randomized algorithms are superior to their deterministic competitors.

Often, NP-complete problems have versions which are efficiently solvable (like 2-SAT or 2-colorability). A random choice can reduce a difficult problem to an easy one if some of the parameters (values of variables) are chosen and assigned at random. Of course, by assigning some variable or problem parameter at random in a wrong way, we might lose the chance of finding any solution afterwards. Such random restrictions have to be repeated a certain number of times. The repetition number is determined by the reciprocal of the success probability of one such random restriction. Of course, this repetition number has to be taken into account within the overall complexity estimation. In other words, if we have some randomized algorithm which has complexity  $t$  and success probability  $p$  (both depending on  $n$ ), then this algorithm needs to be repeated  $c \cdot p^{-1}$  many times to achieve an error probability of at most  $(1 - p)^{c/p} \leq e^{-c}$  so that the overall complexity becomes  $c \cdot t \cdot p^{-1}$ .

As an example, in [4] a method for solving the 3-coloring problem is described. For each of the  $n$  vertices decide at random which one of the 3 colors should *not* be placed on this vertex. Each of these random decision can be wrong with probability  $1/3$ , and all decisions are consistent with a potential 3-coloring with probability  $p = (2/3)^n$ . Given such an assignment of 2 possible colors to each of the  $n$  vertices of a graph, it is possible to decide whether such a coloring exists in polynomial time, e.g. by reducing the problem to a 2-SAT problem which can be solved in polynomial time [1]. That is, in this case  $t$  is a polynomial in  $n$ , and the overall complexity for solving 3-coloring becomes some polynomial times  $(3/2)^n$ .

## 5 Randomization: Permuting the Evaluation Order

In some cases, the complexity of an algorithm can depend very much on the order in which the variables (or the possible values of the variables) are processed. To escape the worst case input situations, it might be a good idea to choose the order in a random way. A nice example of this strategy can be found in [15]: A game tree evaluation can be considered as evaluating a quantified Boolean formula of the form

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots Q_n x_n F(x_1, x_2, \dots, x_n)$$

Each variable  $x_i$  can take the possible values true or false. It is standard to evaluate such expressions in a backtracking manner. The recursion tree can be interpreted as an AND-OR tree. Evaluating an OR (or existential quantifier)

can turn out to be rather quickly done if the first subtree returns the value true. In this case there is no need to evaluate the second subtree; the result will be the truth value true. Obviously, we do not know beforehand which (if any) of the two subtrees might return true. Now let us decide at random which of the two arguments to evaluate first. If one of the subtrees has value true, then with probability at least  $1/2$  we select this subtree first and the evaluation of the OR is fast. Even in the case that both subtrees have the truth value false, there is some advantage, namely, in the AND-level (or universal quantifier) which is immediately following, we know now that the result value is false, which means that at least one argument is false. Again (by dual reasoning) this is a good situation when evaluating an AND. This leads to the following recursion for the expected number of recursive calls:  $T(n) \leq S(n-1) + \frac{1}{2} \cdot T(n-1)$ ,  $S(n) = 2 \cdot T(n-1)$ , giving the complexity bounds  $T(n) = O(1.69^n)$  and  $S(n) = O(1.69^n)$ .

Another such random permuting idea can be found in the 3-SAT algorithm of Paturi et al [13, 14]. Here the variable order is chosen at random. Now, given a particular order of the Boolean variables, one variable after the other (according to that order) is assigned a truth value. In the regular case, this is done at random (with probability  $1/2$  this choice might be wrong and will not lead to a satisfying assignment), or after having substituted the partial assignment so far into the formula (and simplified) it sometimes turns out that the next variable to be assigned according to the order forms a unit clause in the (remaining) formula. In this case the variable is assigned such that this unit clause becomes true. Assuming that the previous random choices were correct, this deterministic setting of the variable is also correct. It turns out, that in this (simple version of the) algorithm, on the average, taken over all  $n!$  permutations of the variables, just  $2n/3$  many random choices are needed. Consequently, the probability for finding a satisfying assignment this way is  $2^{-2n/3}$ , and the overall complexity bound is  $O(2^{2n/3}) = O(1.58^n)$ .

Intuitively, one might say that after substituting those partial assignments into the formula, the (resulting) formula occasionally gives a 1-bit "hint" what the next assignment should be. In the second paper [14] the authors show that it is possible to do some preprocessing with the formula  $F$  which is transformed to  $F'$ , a formula having more clauses, but the same number of variables, and being equivalent to the original formula  $F$ . The advantage of using  $F'$  instead of  $F$  is that  $F'$  is likely to give more such 1-bit hints, on the average, and therefore, more random bits can be saved. The analysis in [14] shows that this is an  $O(1.36^n)$  algorithm for 3-SAT.

## 6 Randomized Local Search

We already discussed a way of performing a deterministic local search, starting from some initial assignment  $a$ , to find out whether there is a satisfying assignment  $a^*$  within Hamming distance at most  $d$  from  $a$ . Instead of flipping each value of the 3 variables systematically in a backtracking fashion as described in Section 3, we now just choose one these variables at random and flip its value.

(This idea appears already in the randomized  $O(n^2)$  2-SAT algorithm of Papadimitriou [12]. Applying and analyzing this principle in the context of 3-SAT was done in [17, 18].) Of course, the probability of choosing the correct variable to flip can be as low as  $1/3$ . Therefore, the probability of finding a satisfying assignment which is by Hamming distance  $d$  away from the initial assignment, is just  $(1/3)^d$ . But now we modify this approach by letting the flipping process last for  $3d$  steps instead of just  $d$  steps. Even if we have  $d$  "mistakes" (within the  $3d$  steps) by flipping variables which have already the correct value, there is still a chance that we correct these mistakes (by another  $d$  steps) and further on do the necessary corrections (by another  $d$  steps) to make the assignment equal to the satisfying assignment  $a^*$ . There are  $\binom{3d}{d}$  possibilities of this situation happening, and each of these has probability  $(2/3)^d \cdot (1/3)^{2d}$ , resulting in a success probability of  $\binom{3d}{d} \cdot (2/3)^d \cdot (1/3)^{2d} = (1/2)^d$  (up to some small polynomial factor). Compare with the  $(1/3)^d$  from above. This means, if we just select a random initial assignment and then perform a random local search (as described) for  $3n$  steps, we get a success probability (probability of finding a satisfying assignment - if one exists) of  $E[(1/2)^D]$ . Here  $D$  is the random variable that indicates the Hamming distance of the initial assignment to some fixed satisfying assignment.

If we produce the initial assignment absolutely at random - without referring to the input formula  $F$  - then this expectation becomes  $E[(1/2)^{X_1+\dots+X_n}] = \prod_{i=1}^n E[(1/2)^{X_i}] = (3/4)^n$ . Here the  $X_i$  are 0-1-valued random variables which indicate whether there is a difference in the  $i$ -th bit. Therefore, we get the complexity bound  $O((4/3)^n)$  (ignoring polynomial factors).

In the paper by Iwama and Tamaki [10] this random walk algorithm is combined with the  $O(1.36^n)$ -algorithm [14] from above. Both algorithms work on the same random initial assignment. This results in the fastest algorithm for 3-SAT known so far with a complexity bound of  $O(1.324^n)$  (where "fastest" refers to the fact that there is a rigorous proof for the upper bound - of course there are numerous other algorithms in the SAT-solver scene using all kind of heuristics and different approaches. Doing performance test on benchmark formulas and random formulas is, of course, serious and important research, but usually leaves us without a proven worst-case upper bound.)

The same principle strategy can be applied to the 3-colorability problem. First, guess an initial coloring at random. Whenever there is an edge with its both vertices of the same color, select one of the vertices at random and change its color to one of the two remaining colors at random. Repeat this "repair" process for  $3n$  steps. The probability of success for one such run turns out to be  $(2/3)^n$ . Therefore the complexity bound is  $O(1.5^n)$  (ignoring polynomial factors), when the basis algorithm is repeated this number of times.

## 7 Randomization with Biased Coins

In the algorithm of the last section, we used the structure of the formula to lead the search process. But on the other hand, when producing the initial assignment we ignored the input formula completely. The question is now whether the input

formula can be used to lead the stochastic choice of the initial assignment. We present some ideas from [3, 9].

The first thing we do is to collect as many mutually variable-disjoint clauses having 3 literals of as possible, by some greedy strategy. Suppose the number  $m$  of clauses collected this way is small (say  $m < 0.1469n$ ). Then we can cycle through all potential  $7^m$  assignments of these variables. After plugging in these partial assignments, the rest of the formula becomes a 2-CNF formula, and its satisfiability can be determined in polynomial time. Hence, in this case, we stay below the total complexity  $O(1.331^n)$ .

If the number of mutually disjoint clauses is larger than  $0.1469n$ , then we assign each of the 3 literals of these clauses at random an initial assignment in a particular biased way: with probability  $3p$  exactly one of the 3 literals is assigned the value true (where each of the 3 literals gets the same chance  $p$ ), With probability  $3q$  we assign exactly two of the 3 literals in the clause the value true. With probability  $1 - 3p - 3q$  assign all three literals the value true. Good values for  $p$  and  $q$  will be determined later.

Supposing that the satisfying assignment  $a^*$  assigns the three literals  $x, y, z$  the values 1, 0, 0, we get that the Hamming distance between the initial assignment  $a$  and  $a^*$ , as a random variable, can have the value 0 (with probability  $p$ ), the value 1 (with probability  $2q$ ), the value 2 (with probability  $1 - p - 3q$ ), and the value 3 (with probability  $q$ ). Hence, the expectation of the random variable  $(1/2)^D$  where  $D$  is the Hamming distance on these 3 variables, is

$$1 \cdot (p) + \frac{1}{2} \cdot (2q) + \frac{1}{4} \cdot (1 - p - 3q) + \frac{1}{8} \cdot q = (1/4 + 3p/4 + 3q/8)$$

For a satisfying assignment which assigns to  $x, y, z$  the values 1, 1, 0 or the values 1, 1, 1 similar calculations can be done. If the number  $m$  of mutually disjoint clauses splits into  $m = m_1 + m_2 + m_3$  where  $m_i$  is the number of clauses which have exactly  $i$  literals true under assignment  $a^*$ , we get for the overall expectation of  $(1/2)^{\text{Hamming-distance}}$ , and hence for the success probability, the term

$$(1/4 + 3p/4 + 3q/8)^{m_1} \cdot (1/2 - 3p/8)^{m_2} \cdot (1 - 9p/4 - 3q/2)^{m_3} \cdot (3/4)^{n-3m}$$

Now, either we determine  $p$  and  $q$  in such a way that all three exponential bases become identical (giving  $p = 4/21$  and  $q = 2/21$ ), or we compute for each of the (polynomially many!) choices for  $m_1, m_2, m_3$  individually a good choice for  $p$  and  $q$  which minimizes the complexity. The latter strategy is somewhat better, both are close to  $O(1.331^n)$ .

## References

1. B. Aspvall, M.F. Plass, and R.E. Tarjan: A linear time algorithm for testing the truth of certain quantified Boolean formulas, *Information Processing Letters* 8(3) (1979) 121–123.
2. R.B. Ash: *Information Theory*. Dover 1965.

3. S. Baumer and R. Schuler: Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs, In *Theory and Applications of Satisfiability Testing*, SAT 2003, Lecture Notes in Computer Science, Vol. 2919, 150–161, 2004.
4. R. Beigel and D. Eppstein: 3-coloring in time  $O(1.3446^n)$ : a no-MIS algorithm. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, IEEE 1995, pages 444–452.
5. T. Brueggemann and W. Kern: *An improved local search algorithm for 3-SAT*. Memorandum No. 1709, Department of Applied Mathematics, University of Twente, 2004.
6. G. Cohen, I. Honkala, and S. Litsyn, A. Lobstein: *Covering Codes*. North-Holland 1997.
7. E. Dantsin, A. Goerdt, E.A. Hirsch, and U. Schöning: Deterministic algorithms for  $k$ -SAT based on covering codes and local search. *Proc. 27th International Colloquium on Automata, Languages and Programming 2000*. Springer Lecture Notes in Computer Science, Vol. 1853, pages 236–247, 2000.
8. E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning: A deterministic  $(2 - \frac{2}{k+1})^n$  algorithm for  $k$ -SAT based on local search. To appear in *Theoretical Computer Science*.
9. T. Hofmeister, U. Schöning, and R. Schuler, O. Watanabe: *A probabilistic 3-SAT algorithm further improved*. In *Proc. of the 19th Sympos. on Theoretical Aspects of Computer Science*. Lecture Notes in Computer Science, Vol. 2285, 193–202, 2002.
10. K. Iwama and S. Tamaki: Improved upper bounds for 3-SAT. *Electronic Colloquium on Computational Complexity*. Report No. 53, 2003. Also: *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete algorithms*, ACM, 2004, pp. 328–328.
11. B. Monien, E. Speckenmeyer: Solving satisfiability in less than  $2^n$  steps. *Discrete Applied Mathematics* 10 (1985) 287–295.
12. C.H. Papadimitriou: On selecting a satisfying truth assignment. *Proceedings of the 32nd Ann. IEEE Symp. on Foundations of Computer Science*, pp. 163–169, 1991.
13. R. Paturi, P. Pudlák, and F. Zane: Satisfiability coding lemma. *Proceedings 38th IEEE Symposium on Foundations of Computer Science 1997*, 566–574.
14. R. Paturi, P. Pudlák, M.E. Saks, and F. Zane: An improved exponential-time algorithm for  $k$ -SAT. *Proceedings 39th IEEE Symposium on Foundations of Computer Science 1998*, pp. 628–637.
15. M. Saks and A. Wigderson: Probabilistic boolean decision trees and the complexity of evaluating game trees. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Science Press, 1986, pp. 29–38.
16. U. Schöning: *On The Complexity of Constraint Satisfaction Problems*. Ulmer Informatik-Berichte, Nr. 99-03. Universität Ulm, Fakultät für Informatik, 1999.
17. U. Schöning: A probabilistic algorithm for  $k$ -SAT and constraint satisfaction problems. *Proceedings 40th IEEE Symposium on Foundations of Computer Science 1999*, 410–414.
18. U. Schöning: A probabilistic algorithm for  $k$ -SAT based on limited local search and restart. *Algorithmica* 32,4 (2002) 615–623.