2. Introduction to Randomized Algorithms

Artur Andrzejak

2.1 Introduction

A *randomized algorithm* is a randomized Turing machine (see Definition 1.10). For our purposes we can consider such an algorithm as a "black box" which receives input data and a stream of random bits for the purpose of making random choices. It performs some computations depending on the input and the random bits and stops. Even for a fixed input, different runs of a randomized algorithm may give different results. For this reason, a description of the properties of a randomized algorithm will involve probabilistic statements. For example, its running time is a random variable.

Randomized algorithms may be faster, more space-efficient and/or simpler than their deterministic counterparts. But, "in almost all cases it is not at all clear whether randomization is really necessary. As far as we know, it may be possible to convert *any* randomized algorithm to a deterministic one without paying any penalty in time, space, or other resources" [Nis96].

So why does randomization make algorithms in many cases simpler and faster? A partial answer to this question gives us the following list of some paradigms for randomized algorithms, taken from [Kar91, MR95b].

Foiling the adversary. In a game theoretic view, the computational complexity of a problem is the value of the following two-person game. The first player chooses the algorithm and as answer the second player, called the *adversary*, chooses the input data to foil this particular algorithm. A randomized algorithm can be viewed as a probability distribution on a set of deterministic algorithms. While the adversary may be able to construct an input that foils one (or a small fraction) of the deterministic algorithms in the set, it is difficult to devise a single input that would foil a *randomly* chosen algorithm.

Abundance of witnesses. A randomized algorithm may be required to decide whether the input data has a certain property; for example, 'is a boolean formula satisfiable?'. Often, it does so by finding an object with the desired property. Such an object is called a *witness*. It may be hard to find a witness deterministically if it lies in a search space too large to be searched exhaustively. But if it can be shown that the search space contains a large number of witnesses, then it often

suffices to choose an element at random from the space. If the input possesses the required property, then a witness is likely to be found within a few trials. On the other hand, the failure to find a witness in a large number of trials gives strong evidence (but not absolute proof) that the input does not have the required property.

Random sampling. Here the idea is used that a random sample from a population is often representative for the whole population. This paradigm is frequently used in selection algorithms, geometric algorithms, graph algorithms and approximate counting.

2.2 Las Vegas and Monte Carlo Algorithms

The popular (but imprecise) understanding of Las Vegas and Monte Carlo algorithms is the following one. A Las Vegas algorithm is a randomized algorithm which always gives a correct solution but does not always run fast. A Monte Carlo algorithm is a randomized algorithm which runs fast but sometimes gives a wrong solution. Depending on the source, the definitions vary. We allow here that an algorithm stops without giving an answer at all, as defined in [Weg93]. Below we state the exact definitions.

A *Las Vegas algorithm* is an algorithm which either stops and gives a correct solution or stops and does not answer. For each run the running time may be different, even on the same input. The question of interest is the probability distribution of the running time. We say that a Las Vegas algorithm is an *efficient Las Vegas algorithm* if on every input its worst-case running time is bounded by a polynomial function of the input size.

A *Monte Carlo algorithm* is an algorithm that either stops and does not answer or stops and gives a solution which is possibly not correct. We require that it is possible to bound the probability of the event that the algorithm errs. For decision problems there are two kinds of Monte Carlo algorithms. A Monte Carlo algorithm has *two-sided error* if it may err when it outputs either YES or NO. It has a *one-sided error* if it never errs when it outputs NO or if it never errs when it outputs YES. Note that a Monte Carlo algorithm with one-sided error is a special case of a Monte Carlo algorithm with two-sided error.

The running time of a Monte Carlo algorithm is usually deterministic (but sometimes also algorithms in which both the running time and the correctness of the output are random variables are called Monte Carlo algorithms). We say that a Monte Carlo algorithm is an *efficient Monte Carlo algorithm* if on every input its worst-case running time is bounded by a polynomial function of the input size. The nice property of a Monte Carlo algorithm is that running it repeatedly makes the failure probability arbitrarily low (at the expense of running time), see Exercises 2.1 and 2.2.

Note that a Las Vegas algorithm is a Monte Carlo algorithm with error probability 0. Every Monte Carlo algorithm can be turned into a Las Vegas algorithm, see Exercise 2.3. The efficiency of such Las Vegas algorithm depends heavily on the time needed to verify the correctness of a solution to a problem. Thus, for decision problems, a Las Vegas algorithm derived from a Monte Carlo algorithm is in general useless.

We may apply these definitions to the complexity classes introduced in the previous chapter:

- The problems in the class $\mathcal{ZPP}$ have efficient Las Vegas decision algorithms.
- The problems in the class $\mathcal{RP}$ have efficient Monte Carlo decision algorithms with one-sided error (the same applies to the class co-$\mathcal{RP}$).
- The problems in the class $\mathcal{PP}$ have efficient Monte Carlo decision algorithms with two-sided error.

In the remainder of this section we show that the class $\mathcal{ZPP}$ can be defined via the following class $\mathcal{C}$ of algorithms. The class $\mathcal{C}$ contains (randomized) decision algorithms which always output a correct answer YES or NO and have expected running time bounded by a polynomial in the size of the input. The difference between an algorithm in the class $\mathcal{C}$ and an efficient Las Vegas decision algorithm is that the earlier cannot output "don't know", but its expected running time is bounded instead of the worst-case running time.

Lemma 2.1. *(i) If L is a language in $\mathcal{ZPP}$, then there is a decision algorithm $A \in \mathcal{C}$ for L.*

(ii) If a language L has a decision algorithm A in $\mathcal{C}$, then $L \in \mathcal{ZPP}$.

Proof. To (i): Let A_1 be an efficient Las Vegas decision algorithm for L with worst-case running time bounded by a polynomial $p(n)$. We construct A as follows. We run A_1 at most $k_0 = p(n)$ times until it gives us an answer YES or NO. If after k_0 runs we still have no definitive answer, we apply the following deterministic algorithm. We may regard A_1 as a deterministic algorithm, once the random string has been fixed. Now we run such a deterministic version of A_1 for every possible random string which A_1 can use, i.e. for each string (of bits) of length $p(n)$. This will give us an answer YES or NO since A_1 outputs "don't know" for at most the half of the random strings by the definition of $\mathcal{ZPP}$.

Let X_n be the running time of A. To bound $\mathbf{E}[X_n]$ observe that for an integer k the probability that there is no definitive answer after running A_1 for k times is at most $(\frac{1}{2})^k$. For each of the first k_0 runs we need time at most $p(n)$. If all these runs fail, our deterministic algorithm needs time at most $2^{p(n)}p(n)$. Thus, the expected running time $\mathbf{E}[X_n]$ of A is at most

$$p(n) + \sum_{k=1}^{k_0-1} \left(\frac{1}{2}\right)^k p(n) + \left(\frac{1}{2}\right)^{p(n)} 2^{p(n)}.$$

But

$$\sum_{k=1}^{k_0-1} \left(\frac{1}{2}\right)^k < 1$$

and so $\mathbf{E}[X_n] \leqslant 3p(n)$.

To (ii): Assume that the expected running time of A is bounded by a polynomial $p(n)$, where n is the size of the input. First we show that there is an efficient Las Vegas decision algorithm A_1 for L with worst-case running time bounded by $2p(n)$. The algorithm is constructed in the following manner. We start A and allow it to run at most $2p(n)$ steps. If after this time there is no answer of A, we stop A and A_1 outputs "don't know". Clearly A_1 is an efficient Las Vegas decision algorithm.

If X_n is the running time of A, then by Markov's inequality we have

$$\text{Prob}[X_n \geqslant 2p(n)] \leqslant \frac{1}{2}.$$

Therefore, the probability that A_1 outputs "don't know" is at most $\frac{1}{2}$ and so $L \in \mathcal{ZPP}$. ∎

Corollary 2.2. *The class $\mathcal{ZPP}$ is the class of all languages which have an algorithm in $\mathcal{C}$.*

In [MR95b] the above statement is used as a definition for $\mathcal{ZPP}$.

Furthermore, each efficient Las Vegas decision algorithm for a language in $\mathcal{ZPP}$ can be simulated by an algorithm in $\mathcal{C}$ and vice versa. This can be used to yet another definition of an efficient Las Vegas decision algorithm, as done by of Motwani and Raghavan [MR95b].

2.3 Examples of Randomized Algorithms

A good way to learn the principles of randomized algorithms is to study the examples. In this section we learn two randomized approximation algorithms, for the problems MAXEkSAT and MAXLINEQ3-2. Both of them will be derandomized in Chapter 3. The obtained deterministic approximation algorithms are important as their respective performance ratios are best possible, unless $\mathcal{P} \neq \mathcal{NP}$. This in turn will be shown in Chapter 7.

Recall that by Definition 1.31 the infimum of $\text{OPT}(x)/\,\mathbf{E}[v(\tilde{x})]$ over all problem instances is the expected performance ratio of a randomized approximation algorithm for some maximization problem. Here $\mathbf{E}[v(\tilde{x})]$ is the expected value of the objective function. (To be consistent with the definitions in [MR95b, MNR96], the value r of the expected performance ratio for maximization problems has to be interchanged with $1/r$.)

2.3.1 A Randomized Approximation Algorithm for the MAXEkSAT-Problem

We will present a simple randomized algorithm for the MAXEkSAT-problem with expected performance ratio $1/(1-2^{-k})$. Especially, for $k=3$, the expected performance ratio is $8/7$.

Let us state the problem:

MAXEkSAT

Instance: A boolean formula Φ in conjunctive normalform with exactly k different literals in each clause.

Problem: Find an assignment for Φ which satisfies the maximum number of clauses.

Let Φ be a boolean formula (instance of the problem MAXEkSAT) with m clauses over n variables. Assume that no clause contains both a literal and its complement, since then it is always satisfied. We can use then the following randomized algorithm:

RANDOM EkSAT ASSIGNMENT

Set each of the n variables of Φ independently to TRUE or FALSE with probability $1/2$.

Theorem 2.3. *The algorithm* RANDOM EkSAT ASSIGNMENT *has expected performance ratio* $1/(1-2^{-k})$.

Proof. A clause of an input formula is *not* satisfied only if each complemented variable is set to TRUE and each uncomplemented variable is set to FALSE. Since each variable is set independently, the probability of this event is $1-2^{-k}$. By the linearity of expectation the expected number of satisfied clauses is $m(1-2^{-k})$, and so $\text{OPT}(\Phi)/\,\mathbf{E}[v(\Phi)] \leqslant 1/(1-2^{-k})$. On the other hand, for each m there is an input Φ such that $\text{OPT}(\Phi)/\,\mathbf{E}[v(\Phi)] = 1/(1-2^{-k})$, namely a satisfiable boolean formula with m clauses. It follows that the expected performance ratio of the algorithm is $1/(1-2^{-k})$. ∎

2.3.2 A Randomized Approximation Algorithm for the MaxLinEq3-2-Problem

The problem to be solved is the following one:

MaxLinEq3-2

Instance: A set of linear equations modulo 2 with exactly three different variables in each equation.

Problem: Find an assignment of values for the variables which maximize the number of satisfied equations.

The following randomized algorithm has expected performance ratio 2.

Random MaxLinEq3-2 Assignment

Given a set of m MaxLinEq3-2-equations on n variables, set each of the n variables independently to 0 or 1 with probability 1/2.

Theorem 2.4. *The algorithm* Random MaxLinEq3-2 Assignment *has expected performance ratio* 2.

Proof. Whether an equation is satisfied or not depends only on the sum of the variables with coefficients different from 0. This sum is 0 or 1 modulo 2 with probability 1/2 each since all variables are set to 0 or 1 independently. Thus, the probability that an equation is satisfied is 1/2 and so the expected number of satisfied equations is $m/2$. In addition, for every m there is a system of equations such that all equations can be satisfied. It follows that the expected performance ratio of the considered randomized algorithm is 2. ∎

2.4 Randomized Rounding of Linear Programs

In this section we explain a technique of solving optimization problems by randomized rounding of linear programs. The technique can be applied for solving a variety of optimization problems, for example the integer multicommodity flow problem, the set cover problem or the computation of approximate shortest paths in a graph. In this section we illustrate the technique by obtaining an optimization algorithm for the MaxSat problem. A detailed treatment of some of these applications and a list of many others can be found in the paper of Motwani, Naor and Raghavan [MNR96].

The technique works in four steps. First, given an optimization problem, we formulate it as an integer linear program with variables assuming the values in $\{0, 1\}$, if possible. Thereafter the integer program is relaxed by allowing the variables to take the values in $[0, 1]$. (This is necessary as no good algorithms are known for solving large zero-one linear programs). In the next step the relaxed

linear program is solved by one of the common methods mentioned below. Finally, *randomized rounding* is used as follows to obtain a zero-one solution vector $\bar{x}$ for the integer linear program. Let $\hat{x}$ denote the solution vector of the relaxed linear program. Then, for each i, we set $\bar{x}_i$ to 1 with probability $\hat{x}_i \in [0,1]$. Note that the last step can be also interpreted in another way. For each i, we pick a random number y uniformly in $[0,1]$. If $\hat{x}_i > y$, we set $\bar{x}_i$ to 1, otherwise $\bar{x}_i$ is set to 0. This view has advantages for implementing the algorithm, as now we only need to compare $\hat{x}_i$ to a randomly chosen threshold.

The advantage of randomized rounding is that the zero-one solution vector $\bar{x}$ does not violate "too much" the constraints of the linear program, which is explained in the following. Let a be a row of the coefficient matrix of the linear program. Then, by linearity of expectation we have $\mathbf{E}[a\bar{x}] = a\hat{x}$. Therefore, the expected value of the left-hand side of every constraint will not violate the bound given by the right-hand side of this constraint.

Several methods for solving rational (i.e., in our case relaxed) linear programs are known. One of the most widely used algorithms is the simplex method due to Dantzig [Dan63]. It exists in many versions. Although fast in practice, the most versions of the simplex method have exponential worst-case running time. On the other hand, both the ellipsoid method of Khachian [Kha79] and the method of Karmarkar [Kar84] have polynomially bounded running time, but in practice they cannot compete with the simplex method. The reader may find more informations about solving linear programs in [PS82].

We begin with the definition of the optimization problem.

MAXSAT
Instance: A boolean formula Φ in conjunctive normalform.
Problem: Find an assignment for Φ which satisfies maximum number of clauses.

The randomized algorithm for MAXSAT presented below has some pleasant properties:

- it gives us an upper bound on the number of clauses which can be simultaneously satisfied in Φ,
- its expected performance ratio is $1/(1-1/e)$ (or even $1/(1-(1-1/k)^k)$ for MAXEkSAT), as we will see below,
- if the the relaxed linear program is solved by the simplex method, then the above algorithm is efficient in practice,
- once the solution of the relaxed linear program is found, we can repeat the randomized rounding and take the best solution. In this way we can improve the solution without great penalty on running time.

RANDOMIZED ROUNDING FOR MAXSAT

1. For an instance Φ of MAXSAT formulate an integer linear program L_Φ. The integer program L_Φ has following variables. For each variable x_i, $i \in \{1,\ldots,n\}$, of Φ let y_i be an indicator variable which is 1 when x_i assumes the value TRUE and 0 otherwise. Also, for each clause C_j, $j \in 1,\ldots,m$ of Φ we introduce an indicator variable z_j. z_j has the value 1 if C_j is satisfied, otherwise 0. For each clause C_j let C_j^+ be the set of indices of variables that appear uncomplemented in C_j and let C_j^- be the set of indices of variables that appear complemented in C_j. Then MAXSAT can be formulated as follows:

$$\text{maximize} \quad \sum_{j=1}^{m} z_j, \tag{2.1}$$

$$\text{where} \quad y_i, z_j \in \{0,1\} \quad \text{for all } i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\},$$

$$\text{subject to} \quad \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geqslant z_j \quad \text{for all } j \in \{1,\ldots,m\}.$$

   Note that the right-hand side of the jth constraint for L_Φ may take the value 1 only if one of the uncomplemented variables in C_j takes the value TRUE or if one of the complemented variables in C_j takes the value FALSE. In other words, $z_j = 1$ if and only if C_j is satisfied.

2. Relax the program L_Φ, allowing the variables $y_1,\ldots,y_n$ and $z_1,\ldots,z_m$ to take the values in $[0,1]$.

3. Solve this relaxed program. Denote the value of the variable y_i in the solution as $\hat{y}_i$ and the value of z_j in the solution as $\hat{z}_j$. Note that $\sum_{j=1}^{m} \hat{z}_j$ bounds from above the number of clauses which can be simultaneously satisfied in Φ, as the value of the objective function for the relaxed linear program bounds from above the value of the objective function for L_Φ.

4. In this step the actual randomized rounding takes place: we obtain the solution vector $\bar{y}$ for the program L_Φ by setting $\bar{y}_i$ independently to 1 with probability $\hat{y}_i$, for each $i \in \{1,\ldots,n\}$.

For the remainder of this section, we put $\beta_k = 1 - (1 - 1/k)^k$.

What is the expected number of clauses satisfied by the algorithm RANDOMIZED ROUNDING FOR MAXSAT? To calculate this value, we show in Lemma 2.6 that for a clause C_j with k literals, the probability that C_j is satisfied is at least $\beta_k \hat{z}_j$. Furthermore, for every $k \geqslant 1$ we have $1-1/e < \beta_k$. By linearity of expectation the expected number of satisfied clauses is at least $(1 - 1/e)\sum_{j=1}^{m} \hat{z}_j$. It follows that the expected performance ratio of the considered algorithm is at most $1/(1-1/e)$.

Remark 2.5. In the case of the MAXEkSAT problem, each clause C_j is satisfied with probability at least $\beta_k \hat{z}_j$. It follows that for MAXEkSAT the expected performance ratio is $1/\beta_k$.

Lemma 2.6. *The probability that a clause C_j with k literals is satisfied by the algorithm* RANDOMIZED ROUNDING *is at least $\beta_k \hat{z}_j$.*

Proof. Since we consider a clause C_j independently from other clauses, we may assume that it contains only uncomplemented variables and that it is of the form $x_1 \vee \ldots \vee x_k$. Accordingly, the jth constraint of the linear program is

$$y_1 + \ldots + y_k \geqslant z_j.$$

Only if all of the variables $y_1, \ldots, y_n$ are set to zero, the clause C_j remains unsatisfied. By Step 4 of the algorithm, this may occur with probability

$$\prod_{i=1}^{k} (1 - \hat{y}_i).$$

It remains to show that

$$1 - \prod_{i=1}^{k} (1 - \hat{y}_i) \geqslant \beta_k \hat{z}_j.$$

The left-hand side is minimized (under the condition of the equation) when each $\hat{y}_i$ assumes the value $\hat{z}_j/k$. Thus, to complete the proof we have to show that $1 - (1 - z/k)^k \geqslant \beta_k z$ for every positive integer k and each real number $z \in [0, 1]$.

Consider the functions $f(z) = 1 - (1 - z/k)^k$ and $g(z) = \beta_k z$. To show that $f(z) \geqslant g(z)$ for all $z \in [0, 1]$ notice that $f(z)$ is a concave function and $g(z)$ a linear one. Therefore we have to check the inequality $f(z) \geqslant g(z)$ only for the endpoints of the interval $[0, 1]$. Indeed, for $z = 0$ and $z = 1$ the inequality $f(z) \geqslant g(z)$ holds, and so we are done. ∎

Consequently, we obtain the following theorem:

Theorem 2.7. *For every instance of* MAXSAT *the expected number of clauses satisfied by the algorithm* RANDOMIZED ROUNDING FOR MAXSAT *is at least $(1 - 1/e)$ times the maximum number of clauses that can be satisfied. For the* MAXEkSAT*-problem, the factor $(1 - 1/e)$ can be interchanged with β_k.*

2.5 A Randomized Algorithm for MAXSAT with Expected Performance Ratio 4/3

The algorithm RANDOM EkSAT ASSIGNMENT is obviously works for the problem MAXSAT, too. Thus, together with the algorithm RANDOMIZED ROUNDING we

k	$1-2^{-k}$	β_k
1	0.5	1.0
2	0.75	0.75
3	0.875	0.704
4	0.938	0.684
5	0.969	0.672

Table 2.1. A comparison of probabilities of satisfying a clause C_j with k literals

have two algorithms for the MAXSAT-problem. Which one is better? Table 2.1 shows that depending on the length k of a clause C_j, one of the two algorithms may satisfy C_j with higher probability than the other algorithm.

It is sensible to combine both algorithms: for an instance of MAXSAT, we run each algorithm and choose the solution satisfying more clauses. We will show now that this procedure yields a randomized algorithm for MAXSAT with expected performance ratio 4/3. (In Chapter 11 an even better randomized algorithm for MAXSAT is presented with expected performance ratio 1.324. Some other algorithms for this problem are also discussed there.)

We use the same notations as in the previous subsection.

Theorem 2.8. *We obtain an approximation algorithm for* MAXSAT *with expected performance ratio* 4/3 *by taking the better solution of the algorithms* RANDOM MAXEkSAT ASSIGNMENT *and* RANDOMIZED ROUNDING FOR MAXSAT *for an instance of* MAXSAT.

Proof. Let n_1 denote the expected number of clauses that are satisfied by the algorithm RANDOM MAXEkSAT ASSIGNMENT. Let n_2 denote the expected number of clauses satisfied by the algorithm RANDOMIZED ROUNDING FOR MAXSAT. It suffices to show that

$$\max\{n_1, n_2\} \geqslant \frac{3}{4} \sum_{j=1}^{m} \hat{z}_j.$$

To this aim we prove that

$$(n_1 + n_2)/2 \geqslant \frac{3}{4} \sum_{j=1}^{m} \hat{z}_j.$$

Put S^k to be the set of clauses of the problem instance with exactly k literals, for $k \geqslant 1$. By Lemma 2.6 we have

$$n_2 \geqslant \sum_{k \geqslant 1} \sum_{C_j \in S^k} \beta_k \hat{z}_j.$$

On the other hand, obviously

$$n_1 = \sum_{k \geqslant 1} \sum_{C_j \in S^k} (1 - 2^{-k}) \geqslant \sum_{k \geqslant 1} \sum_{C_j \in S^k} (1 - 2^{-k}) \hat{z}_j.$$

Thus,

$$(n_1 + n_2)/2 \geqslant \sum_{k \geqslant 1} \sum_{C_j \in S^k} \frac{(1 - 2^{-k} + \beta_k)}{2} \hat{z}_j.$$

It can be shown that $(1 - 2^{-k}) + \beta_k \geqslant 3/2$ for all k, so that we have

$$(n_1 + n_2)/2 \geqslant \frac{3}{4} \sum_{k \geqslant 1} \sum_{C_j \in S^k} \hat{z}_j = 3/4 \sum_{j=1}^{m} \hat{z}_j.$$

■

Exercises

Exercise 2.1. Let $0 < \varepsilon_2 < \varepsilon_1 < 1$. Consider a Monte Carlo algorithm A that gives the correct solution to a problem with probability at least $1 - \varepsilon_1$, regardless of the input. Assume that it is possible to verify a solution of the algorithm A efficiently. How can we derive an algorithm based on A which gives the correct solution with probability at least $1 - \varepsilon_2$, regardless of the input?

Exercise 2.2. Consider the same problem as in Exercise 2.1, but now $0 < \varepsilon_2 < \varepsilon_1 < 1/2$ and it is not possible to verify a solution of the Monte Carlo algorithm A efficiently (this is the case for decision problems).

Exercise 2.3. [MR95b] Let A be a Monte Carlo algorithm for a problem Π. Assume that for every problem instance of size n the algorithm A produces a correct solution with probability $\gamma(n)$ and has running time at most $T(n)$. Assume further that given a solution to Π, we can verify its correctness in time $t(n)$. Show how to derive a Las Vegas algorithm for Π that runs in expected time at most $(T(n) + t(n))/\gamma(n)$.

Exercise 2.4. (trivial) Which expected performance ratio does the algorithm RANDOM MAXLINEQ3-2 ASSIGNMENT have if the number of variables in an equation is $k \neq 3$? Which expected performance ratio do we obtain if the number of variables may be different in each equation?