15-859(M): Randomized Algorithms **Lecturer:** Anupam Gupta
Topic: Introduction **Date:** September 13, 2004
Scribe: Kedar Dhamdhere

1.1 Introduction

In this lecture, we shall first discuss the motivation for randomized algorithms. Then we shall briefly talk about the main ideas used in randomized algorithms. Finally, we will give two examples of randomized algorithms.

1.2 Why Randomized Algorithms

Simplicity This is the first and foremost reason for using randomized algorithms. There are numerous examples where an easy randomized algorithm can match (or even surpass) the performance of a deterministic algorithm.

Example: *Sorting algorithms.* The Merge-Sort algorithm is asymptotically best deterministic algorithm. It is not too hard to describe. However, the same asymptotic running time can be achieved by the simple randomized Quick-sort algorithm. The algorithm picks a random element as a pivot and partitions the rest of the elements: those smaller than the pivot and those bigger than the pivot. Recurse on these two partitions. We will give the analysis of the running time later. ■

Speed For some problems, the best known randomized algorithms are faster than the best known deterministic algorithms. This is achieved by requiring that the correct answer be found only with high probability or that the algorithm should run in expected polynomial time. This means that the randomized algorithm may not find a correct answer in some cases or may take very long time.

Example: *Checking if a multi-variate polynomial is the zero polynomial.* There is no known deterministic polynomial-time algorithm that determines if the given multi-variate polynomial is the zero polynomial. On the other hand, there is a very simple and efficient randomized algorithm for this problem: just evaluate the given polynomial at random points.

Note that such a polynomial could be represented implicitly. e.g., as a determinant of a matrix whose entries contain different variables. ■

Remark 1.2.1 Primality testing used to another example for which there is simple randomized algorithm. A deterministic polynomial time algorithm was given by Agarwal, Kayal, and Saxena (2002).

Possibly $P \subsetneq RP$ It is possible that the class RP is strictly bigger than P. (See Lecture 2 for a formal definition of the class RP.)

Online Algorithms Online algorithms are the algorithms which have to take decisions based on partial knowledge of the input, e.g., when future inputs are not known. In such situations, randomness is useful because it allows us to defeat an adversary who does not see the random decisions that we are making.

Communication Complexity Randomization helps in this case too.

Example: Consider the game with two players: Alice and Bob. Each of them has an n-bit string. They want to compare their strings to check if they are equal. However, they are charged for each bit sent. The best deterministic algorithm for this problem requires sending n bits. It is essentially equivalent to Alice sending her whole string over to Bob. On the other hand, with randomization the communication can be reduced by only requiring that the protocol succeed with high probability. ■

1.3 Ideas

Before we discuss the main ideas used in randomized algorithms, it is important to note that the randomized algorithms is not the same thing as probabilistic analysis. In probabilistic analysis, (deterministic) algorithms are analyzed assuming random input, e.g., in sorting one might analyze the deterministic quick-sort (with fixed pivot) on the input which is a random permutation of n elements. On the other hand, randomized algorithms use random coin flips for their execution, which amounts to picking a deterministic algorithm from a suite of algorithms. Moreover, randomized algorithms give guarantees for the worst case input.

Random variables The choice of *right* random variable lies at the heart of the analysis of any randomized algorithm. An idea which is most often used is the linearity of expectations. Let X and Y be random variables. Then we have the following equality.

$$\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y] \tag{1.3.1}$$

It is important to note that this equation does not require independence between X and Y. This idea lets us simplify calculations that would be quite complex otherwise. This is demonstrated quite well by the following example.

Example: Suppose there are 26 students in the classroom and the professor has alphabet cookies with him – each cookie has a different letter of the alphabet on it. Suppose the professor randomly gives one cookie per student. Then what is the expected number of people who get a cookie with the same letter as the first letter of their name?

To answer this question, consider the random variables: $X_1, X_2, \ldots, X_{26}$, where for each i, $1 \le i \le 26$ we define

$$X_i = \begin{cases} 1 & \text{if } i^{\text{th}} \text{ person gets the correct letter} \\ 0 & \text{otherwise} \end{cases}$$

Since the cookies were distributed randomly, the probability that the i^{th} person gets the correct letter is $\frac{1}{26}$. Therefore, we have: $\mathbf{E}[X_i] = \frac{1}{26}$ for all i.

Finally, the expected number of people who get correct letters is given by

$$\mathbf{E}[X_1 + X_2 + \ldots + X_{26}] = \sum_i \mathbf{E}[X_i] = 1$$

Note that there was no assumption whatsoever made about the first letters of peoples' names. ■

Lot of witnesses The reason why randomized algorithms are successful in finding the solution is that often the problem instance has a lot of witnesses for "yes" or "no".

For example, in case of checking whether a multi-variate polynomial is the zero polynomial, when the input polynomial is indeed the zero polynomial, then it evaluates to zero at all the points. When the polynomial is not zero, then there are a lot of points where the polynomial is non-zero. Hence, finding one witness out of the many is achieved easily by random sampling.

Load balancing Randomization is a good way of distributing the load. e.g., consider a case where we have n servers and n jobs. If each of the n jobs was assigned to one of the servers at random, then w.h.p. no machine gets more than $\log n$ jobs. We can put a slight twist in this simple randomized strategy: for each job pick two servers at random and assign the job to the lightly loaded server among those two. It can be shown that this modified scheme has no more than $\log \log n$ jobs for any server w.h.p. Just to get an idea, if there were a billion jobs, then each server will get at most five jobs.

Another example of load balancing is found in hash tables.

Sampling Another useful technique is random sampling: pick a small sample out of the large input problem. Solve the problem on the sample and then take back the solution to the original problem. Min-cut problem has randomized algorithms that use this technique. Another problem is to find out number of satisfying assignments to some formula.

Symmetry breaking For parallel computation, it is important to break the symmetry. For example, in Ethernet when two machines start sending packets at the same time, there is a collision. The easiest way to get out of the situation is by exponential back-off – breaking symmetry through randomization. Another example is the problem of finding a perfect matching in a graph in parallel. It's important that all the processors compute the same matching. Here randomization can be used to break the symmetry between different possible matchings and isolate a single matching.

Derandomization Some simple randomized algorithms can be converted to (not-so-simple) deterministic algorithms via *derandomization*. Many times the best guarantees on performance for deterministic algorithms are achieved via derandomizing simple randomized algorithms.

1.4 Applications

Now we will study two applications of some of the ideas presented above.

1.4.1 Max-cut

Given a graph $G = (V, E)$, the problem is to partition the vertex set into two sets (A, B). An edge is said to be *cut* by this partition, if its endpoints lie on different sides of the partition. The objective of the problem is to find a partition that maximizes the number of edges cut. (See figure 1.4.1). This problem is NP-hard. We will give a simple 0.5-approximation algorithm for this problem

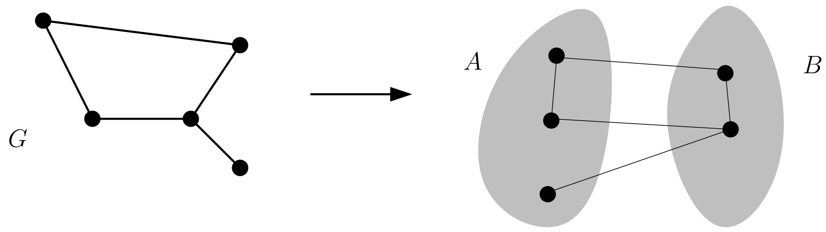


Figure 1.4.1: Max-cut problem

through randomization. This was the best known approximation guarantee for this problem for several years.

Let $|V| = n$ and $|E| = m$.

Claim 1.4.1 *There exists a partition $(A, V \setminus A)$ of the vertex set such that*

$$\textit{Number of edges cut } \geq \frac{|E|}{2} = \frac{m}{2}$$

Proof: We build the set A (i.e. one side of the partition) by including each vertex in A with probability 0.5. Now we ask the question: what is the expected number of edges cut by the partition $(A, V \setminus A)$.

To answer this question, for every edge $(i, j) \in E$ we define

$$X_{ij} = \begin{cases} 1 & \text{if the edge } (i,j) \text{ is cut by the partition} \\ 0 & \text{otherwise} \end{cases} \tag{1.4.2}$$

We can write the following expression for the number of edges cut.

$$\text{Number of edges cut } = X = \sum_{(i,j)\in E} X_{ij}.$$

The question we want to answer is: $\mathbf{E}[X] =$? We use the linearity of expectations. First, convince yourself that $\mathbf{E}[X_{ij}] = \frac{1}{2}$. This holds, since conditioned on i being in the set A or not, there is exactly one choice for j which makes the edge (i, j) cut. Hence we get the following.

$$\mathbf{E}[X] = \sum_{(i,j)\in E} \mathbf{E}[X_{ij}] = \frac{m}{2}$$

To complete the proof, we note that there must be some outcome of the random experiment for which the number of edges cut is at least $\frac{m}{2}$. ■

Note that the number of edges cut by the best partition can be no more than m. And above algorithm produces a cut of size $\frac{m}{2}$. Hence, this is a 0.5 approximation.

Remark 1.4.2 There is a simple way to derandomize this algorithm using a greedy algorithm. Include first vertex in A and then go through rest of the vertices one by one. For each vertex, include it in A if and only if the inclusion cuts more edges to the previously considered vertices, otherwise don't include in A.

1.4.2 Long path problem

Given a graph $G = (V, E)$ and an integer k, the problem is to find a (simple) path of length k in the graph G. Note that for $k = n - 1$, this problem is same as finding a Hamiltonian path in G. Here, we will see a randomized algorithm due to Alon, Yuster and Zwick (1995) for this problem that runs in polynomial time for $k = O(\log n)$. First we study a simpler algorithm which works for $k = O(\frac{\log n}{\log \log n})$ and then show how to improve it.

1.4.2.1 The Simpler Algorithm

The central idea of the algorithm hinges on the fact that for a directed acyclic graph (DAG), there is a polynomial-time algorithm to find the longest path.

Exercise 1.4.3 *Give an algorithm using dynamic programming to find the longest path in a DAG, which runs in time $O(|E|)$.*

Suppose we map the vertices of G to the set $\{1, 2, \ldots, n\}$ randomly (with each permutation equally likely). Furthermore, direct all the edges from left to right. Thus we get a DAG that we denote by $\vec{G}$. (See figure 1.4.2.1).

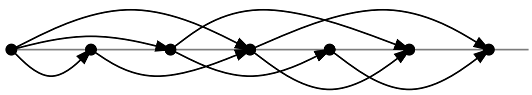
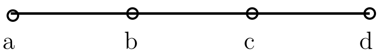


Figure 1.4.2: Creating a DAG

Claim 1.4.4 *If G had a path P of length k, then P will be a directed path in $\vec{G}$ with probability $\alpha = \frac{2}{(k+1)!}$.*

Proof: First consider an example of an undirected path in G of length 3. This will be a directed path in $\vec{G}$ if and only if either $a < b < c < d$ or $a > b > c > d$ in the mapping. In general, the



path P will be a directed path in $\vec{G}$ if and only if the $k + 1$ vertices in P appear in either strictly increasing or strictly decreasing order after the mapping. Since all permutations of these $k + 1$ vertices are equally likely and only two of them give a valid directed path in $\vec{G}$, it follows that P is a directed path in $\vec{G}$ with probability $\alpha = \frac{2}{(k+1)!}$ ■

Corollary 1.4.5 *If G had a path P of length k, then $\vec{G}$ has a directed path of length k with probability* at least α.

Algorithm Repeat $t = \frac{1}{\alpha}$ times

1. Assign directions to the edges of graph G randomly to get $\vec{G}$, as described above.
2. Find the longest path in $\vec{G}$ using dynamic programming. (We "fail" if we do not find a path of length k.)

Theorem 1.4.6 *The above algorithm finds a path of length $k = O(\frac{logn}{\log\log n})$ (if one exists) in randomized polynomial time.*

Proof: Assume that there is a path of length k in graph G. We want to compute the probability that we fail to find a path of length k in $\vec{G}$ in all t trials, which is the same as the probability that $\vec{G}$ does not have a directed path of length k in all the t trials.

$$\begin{aligned} \mathbf{Pr}[\text{fail in all } t \text{ trials}] &\leq (1-\alpha)^t \\ &\leq e^{-\alpha t} \leq 1/e. \end{aligned}$$

The first inequality holds because in each trial, the graph $\vec{G}$ does not have a directed path of length k with probability at most $(1-\alpha)$ (due to Corollary 1), and each of the t trials are independent. The second inequality is the consequence of the fact that $1 + x \leq e^x$ for all x. Thus the algorithm can find a path of length k (if one exists) with probability at least $1 - \frac{1}{e}$.

The running time of this algorithm is

$$\begin{aligned} \text{running time} &= t \times (\text{running time for the DAG}) \\ &= \frac{(k+1)!}{2} \times O(|E|) \end{aligned}$$

Note that $(k+1)! \leq (k+1)^k$, and setting $k = \frac{c\log n}{\log\log n}$ causes the running time to be at most $O(mn^c)$, which is polynomial for any constant c. ■

1.4.2.2 Improved algorithm

To get an improved algorithm, we do the following. Let C denote a set of $k+1$ colors. Color each vertex (randomly) with a color in C. A *colorful* path in G is a path in which no color repeats. Note that the longest colorful path in G can have length k as there are only $k+1$ distinct colors. A colorful path in G can be found in time linear in m (but exponential in k).

Exercise 1.4.7 *Give a dynamic programming based algorithm that finds a colorful path of length k (if one exists) in time $O(2^k mk)$.*

Fact 1.4.8 (Stirling's approximation) $n! \approx (n/e)^n\sqrt{2\pi n}$.

Claim 1.4.9 *If P is a path in G of length k, then*

$$\mathbf{Pr}[P \textit{ is colorful}] \approx \frac{\sqrt{2\pi(k+1)}}{e^{k+1}}$$

Proof: After the random assignment of colors to the vertices, P remains colorful if all the vertices in P get different colors. The probability of such an event can be computed as follows. The number of ways in which the vertices of the path P can be colored with colors in C is $|C|^{k+1} = (k+1)^{k+1}$, and the number of ways in which P can be colorful is $(k+1)!$. Hence

$$\begin{aligned}\mathbf{Pr}[P \text{ is colorful}] &= \frac{(k+1)!}{(k+1)^{k+1}} \\ &\approx \frac{\sqrt{2\pi(k+1)}\left(\frac{k+1}{e}\right)^{k+1}}{(k+1)^{k+1}} \\ &= \frac{\sqrt{2\pi(k+1)}}{e^{k+1}}.\end{aligned}$$

Let us denote this last quantity by β. ■

Now our improved algorithm is as follows: Repeat $t = \frac{1}{\beta}$ times:

1. Color graph G randomly with $k+1$ colors.
2. Find the longest colorful path in G. (We "fail" if we do not find a colorful path of length k.)

Theorem 1.4.10 *The above algorithm finds a path of length $k = O(logn)$ (if one exists) in randomized polynomial time.*

Proof: The probability that the algorithm fails to find a colorful path of length k in all the trials can be computed as follows.

$$\begin{aligned}\mathbf{Pr}[\text{fail in all } t \text{ trials}] &\leq (1-\beta)^t \\ &\leq \mathrm{e}^{-\beta t} \leq 1/\mathrm{e}.\end{aligned}$$

The running time of this algorithm is $O(2^k\, mk \times \mathrm{e}^{k+1})$. For $k = c \log n$, the running time is $O(mn^O(c))$, which is again polynomial for constant c. ■

Recent Developments The above results were recently improved by a result of Gabow (2004) that, if the graph has a path of length L, finds a path of length $\exp(\sqrt{\log L/\log\log L})$.

References

Noga Alon, Raphael Yuster, and Uri Zwick (1995) Color-coding. *J. Assoc. Comput. Mach.*, 42(4):844–856.

Harold N. Gabow (2004) Finding paths and cycles of superpolylogarithmic length. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 407–416. ACM Press.