

CS787: Advanced Algorithms**Scribe:** Yudong Tang and Baris Aydinlioglu**Lecturer:** Shuchi Chawla**Topic:** Randomized Algorithms**Date:** September 17, 2007

In this lecture and the next we cover randomized algorithms. Today we begin by motivating the use of randomized algorithms. Next we recall some related definitions and results from probability theory. We end today's lecture with studying Karger's randomized algorithm for computing a min-cut of a graph.

6.1 Introduction

We define a *randomized algorithm* as an algorithm that can toss coins and act depending on the outcome of those tosses. There are two kinds of randomized algorithms that we will be studying:

- **Las Vegas Algorithms:** These refer to the randomized algorithms that always come up with a/the correct answer. Their “expected” running time is polynomial in the size of their input, which means that the average running time over all possible coin tosses is polynomial. In the worst case, however, a Las Vegas algorithm may take exponentially long. One example of a Las Vegas algorithm is Quicksort: it makes some of its decisions based on coin-tosses, it always produces the correct result, and its expected running and worst-case running times are $n \log n$ and n^2 , respectively.
- **Monte Carlo Algorithms:** These refer to the randomized algorithms that sometimes come up with an incorrect answer. Like Las Vegas algorithms their expected running time is polynomial but may be exponential.

By performing independent runs of a Monte Carlo algorithm, we can decrease the chances of obtaining an incorrect result to a value as small as we like. For example, if a particular Monte Carlo algorithm produces an incorrect result with probability $1/4$, and we have the ability to detect an incorrect result, then the probability of not obtaining a correct answer after t independent runs is $(1/4)^t$. In some contexts, such as in optimization problems, we do not have the ability to detect an incorrect answer, but can compare different answers and pick the best one. The same analysis would also apply in these cases, because if picking the best answer does not give the optimal answer, then it means the algorithm produced an incorrect answer in all of its independent runs.

Sometimes we may not have an algorithm with an error probability of a small constant, but we may have one with error probability a function of the input size. In such a case, we aim for an error probability from which we can obtain desired low values with a small number (i.e., polynomial in the size of the input) of independent runs.

6.2 Motivation

Randomized algorithms have several advantages over deterministic ones. We discuss them here:

- Randomized algorithms tend to bring **simplicity**. For example, recall from Lecture 2 the problem of finding the k th smallest element in an unordered list, which had a rather involved deterministic algorithm. In contrast, the strategy of picking a random element to partition the problem into subproblems and recursing on one of the partitions is much simpler.
- Another advantage randomized algorithms can sometimes provide is **runtime efficiency**. Indeed, there are problems for which the best known deterministic algorithms run in exponential time. A current example is Polynomial Identity Testing, where given a n -variate polynomial f of degree d over a field F , the goal is to determine whether it is identically zero. If f were univariate then merely evaluating f at $d + 1$ points in F would give us a correct algorithm, as f could not have more than d roots in F . With n variables, however, there may be n^d roots, and just picking $n^d + 1$ points in F gives us exponential time. While we don't know of a deterministic polytime algorithm for this problem, there is a simple algorithm that evaluates f at only $2d$ points, based on a theorem by Zippel and Schwarz.

A prevalent example where randomization helps with runtime efficiency used to be Primality testing, where given an n -bit number, the goal is to determine if it is prime. While straightforward randomized algorithms for this problem existed since the 70's, which ran in polynomial time and erred only if the number was prime, it was not known until two years ago whether a deterministic polytime algorithm was possible—in fact, this problem was being viewed as evidence that the class of polytime randomized algorithms with one-sided error is more powerful than the class of polytime deterministic algorithms (the so called RP vs P question). With the discovery of a polytime algorithm for Primality, however, the question still remains open and can go either way.

- There are problems in which randomized algorithms help us where deterministic algorithms can provably not work. A common feature of such problems is **lack of information**. A classical example involves two parties, named A (for Alice) and B (for Bob), each in possession of an n bit number (say x and y resp.) that the other does not know, and each can exchange information with the other through an expensive communication channel. The goal is to find out whether they have the same number while incurring minimal communication cost. Note that the measure of efficiency here is the number of bits exchanged, and not the running time. It can be shown that a deterministic protocol has to exchange n bits in order to achieve guaranteed correctness. On the other hand, the following randomized protocol, based on the idea of *fingerprinting*, works correctly with almost certainty: A picks a prime p in the range $[2n, 4n]$ at random, computes $x \bmod p$, and sends both x and p to B , to which B responds with 1 or 0 indicating whether there was a match. With only $O(\log n)$ bits exchanged, the probability of error in this protocol—i.e., $\Pr[x \neq y \text{ and } x \bmod p = y \bmod p]$ —can be shown to be at most $1/n$. By repeating the protocol, we can reduce the probability of an error to less than that of an asteroid colliding with either of A or B 's residences, which should be a practically sufficient threshold.

We now list several more examples of tasks where randomization is useful:

Load Balancing: In this problem there are m machines and incoming jobs, and the goal is to assign the jobs to the machines such that the machines are equally utilized. One simple way that

turns out very effective here is to simply pick at random, given a job, one of the m machines and assign the job to it. We will study load balancing in a later lecture.

Symmetry Breaking: Randomization comes handy also in designing contention resolution mechanisms for distributed protocols, where a common channel is shared among the communicating entities of a network. The Ethernet protocol is a typical example, where a node that has something to send instantly attempts to send it. If two or more nodes happen to be sending at overlapping windows then they all abort, each pick at random a back-off duration, and reattempt after its picked time elapses. This turns out to be a very effective mechanism with good performance and without any prior communication between sending nodes.

Sampling: Another scenario where a randomized approach works really well is when we want to compute the integral of a multivariate function over a region. Similar to the polynomial identity testing example above, in the case of a univariate polynomial this problem can be solved rather easily, by piecewise linear approximations. But the same method does not work well in the general case as there are exponentially many pieces to compute. The only known efficient method for solving this problem uses randomization, and works as follows. It computes a bounding box for the function, samples points from the box, and determines what portion of the points sampled lies below the function, from which an approximate value for the integral is then computed. We may get to this problem later in the course.

The sampling method applies generally to a number of problems that involve counting. For example, the problem of finding the number of satisfying assignments of a boolean formula involving n variables, which is NP-hard as we will see a later lecture, can be reasonably approximated by sampling points from $\{0, 1\}^n$.

Searching for witnesses: A recurring theme among the above examples is the task of searching for witnesses for verifying whether a certain property holds. Randomization yields effective techniques in these cases if the density of witnesses are high. For example, in Polynomial Identity Testing, it can be shown through algebra that if a polynomial is not identically zero, then the points where it evaluates to non-zero has high density relative to all points where it is defined.

The reader should be convinced by now that flipping coins is a powerful tool in algorithm design.

6.3 Elementary Probability Theory

In the following, let X_i be random variables with x_i their possible values, and let \mathcal{E}_i be events.

We have the following definitions for a random variable X :

- Expectation: $\mathbf{E}[X] = \mu(X) = \sum_x x \cdot \mathbf{Pr}[X = x]$.
- Variance: $\mathit{Var}(X) = \mathbf{E}[(X - \mathbf{E}[X])^2] = \mathbf{E}[X^2] - \mathbf{E}[X]^2$
- Standard Deviation: $\sigma(X) = \sqrt{\mathit{Var}(X)}$

We have the following rules:

- Linearity of Expectation: $\mathbf{E}[\sum_i \alpha_i \cdot X_i] = \sum_i \alpha_i \mathbf{E}[X_i]$, where α_i 's are scalars and no independence assumptions are made on X_i 's. It follows from this rule that the above two definitions of variance are identical.
- Multiplication Rule: $\mathbf{E}[X_1 \cdot X_2] = \mathbf{E}[X_1] \cdot \mathbf{E}[X_2]$ if X_1, X_2 are independent.
- Bayes' Rule: If \mathcal{E}_i are disjoint events and $\mathbf{Pr}[\bigcup_i \mathcal{E}_i] = 1$, then $\mathbf{E}[X] = \sum_i \mathbf{E}[X|\mathcal{E}_i] \cdot \mathbf{Pr}[\mathcal{E}_i]$.
- Union Bound: $\mathbf{Pr}[\bigcup_i \mathcal{E}_i] \leq \sum_i \mathbf{Pr}[\mathcal{E}_i]$.
- Markov's Inequality: If $X \geq 0$ with probability 1, then $\mathbf{Pr}[X > \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}$

We only give a proof of the last statement:

Proof: This is a proof of Markov's inequality.

$$\mathbf{E}[X] = \mathbf{E}[X|0 \leq X \leq \alpha] \cdot \mathbf{Pr}[0 \leq X \leq \alpha] + \mathbf{E}[X|\alpha < X] \cdot \mathbf{Pr}[\alpha < X] \quad (6.3.1)$$

$$\geq \mathbf{E}[X|\alpha < X] \cdot \mathbf{Pr}[\alpha < X] \quad (6.3.2)$$

$$\geq \alpha \cdot \mathbf{Pr}[\alpha < X] \quad (6.3.3)$$

■

6.4 Global Min Cut

6.4.1 Description of the problem

Let $G = (V, E)$ be a graph. For simplicity, we assume G is undirected. Also assume the capacity of each edge is 1, i.e., $C_e = 1, \forall e \in E$. A *cut* is a partition (S, \bar{S}) of V such that $S \neq \emptyset, V$. We define the capacity of (S, \bar{S}) to be the sum of the capacities of the edges between S and \bar{S} . Since $C_e = 1, \forall e \in E$, it is clear that the capacity of (S, \bar{S}) is equal to the number of edges from S to \bar{S} , $|E(S, \bar{S})| = |(S \times \bar{S}) \cap E|$.

Goal: Find a cut of globally minimal capacity in the above setup.

6.4.2 Deterministic method

This problem can be solved by running max-flow computation (Ford-Fulkerson algorithm) $n - 1$ times, $n = |V|$. One fixes a node s to be the source, and let the sink run through all other nodes. In each case, calculate the min cut using the Ford-Fulkerson algorithm, and compare these min cuts. The minimum of the $n - 1$ min cuts will be the global min cut.

To see the above statement, consider the following: Let (S, \bar{S}) be the global min cut. For the node s we fixed, either $s \in S$ or $s \in \bar{S}$. If $s \in S$, then pick a node t in \bar{S} as the sink. The max-flow computation with s as source and t as sink gives a flow f . f must be no greater than the capacity of (S, \bar{S}) since f has to flow from S to \bar{S} . So the min cut when s is source and t is sink must be no

greater than (S, \bar{S}) . Therefore, when $t \in \bar{S}$ is chosen as sink, the min cut has the same capacity as the global min cut. In the case $s \in \bar{S}$, notice that G is undirected, so flow can be reversed and it goes back to the first case.

6.4.3 Karger's algorithm: a randomized method

Description of the algorithm:

1. Set $S(v) = \{v\}, \forall v \in V$.
2. If G has only two nodes v_1, v_2 , output cut $(S(v_1), S(v_2))$.
3. Otherwise, pick a random edge $(u, v) \in E$, merge u and v into a single node uv , and set $S(uv) = S(u) \cup S(v)$. Remove any self-loops from G .

Theorem 6.4.1 *Karger's algorithm returns a global min cut with probability $p \geq \frac{1}{\binom{n}{2}}$.*

Proof: Let (S, \bar{S}) be a global min cut of G , and k be the capacity of (S, \bar{S}) . Let \mathcal{E} be the event that the Karger's algorithm produces the cut (S, \bar{S}) .

First, we show that \mathcal{E} occurs if and only if the algorithm never picks up any edge from S to \bar{S} . The "only if" part is clear, so we show the "if" part: Assume the algorithm doesn't pick up any edge of cut (S, \bar{S}) . Let e be the first edge of cut (S, \bar{S}) which is eliminated by the algorithm, if there is any. The algorithm can only eliminate edges in two ways: Pick it up or remove it as a self loop. Since e can't be picked up by assumption, it must be eliminated as a self-loop. e becomes a self-loop if and only if its two end points u, v are merged, which means an edge e' connects u, v is selected in some iteration. In that iteration, u, v may be the combination of many nodes in original graph G , ie, $u = u_1 u_2 \dots u_i, v = v_1 v_2 \dots v_j$, but since none of the previous picked up edges is in cut (S, \bar{S}) , the nodes merged together must belong to the same set of S or \bar{S} . So if e connects S to \bar{S} , the e' will also connects S to \bar{S} , which contradicts to the assumption that the algorithm doesn't pick up edges of cut (S, \bar{S}) .

Second, notice that (S, \bar{S}) remains to be global min cut in the updated graph G as long as no edge of (S, \bar{S}) has been picked up. This is because any cut of the updated graph is also a cut in the original graph and the edges removed are only internal edges, which don't affect the capacity of the cut.

Now let \mathcal{E}_i be the event that the algorithm doesn't pick any edge of (S, \bar{S}) in i -th iteration. Then,

$$\Pr[\mathcal{E}] = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2|\mathcal{E}_1] \cdot \Pr[\mathcal{E}_3|\mathcal{E}_1 \cap \mathcal{E}_2] \dots \quad (6.4.4)$$

Observe that, in the i -th iteration, assuming $\bigcap_{1 \leq j \leq i-1} \mathcal{E}_j$ has occurred, every node has at least k edges. For otherwise the node with fewer than k edges and the rest of the graph would form a partition with capacity less than k . It follows that there are at least $(n - i + 1)k/2$ edges in i -th

iteration if no edge of (S, \bar{S}) has been selected. Therefore,

$$\Pr \left[\mathcal{E}_i \mid \bigcap_{1 \leq j \leq i-1} \mathcal{E}_j \right] \geq 1 - \frac{k}{(n-i+1)k/2} = \frac{n-i-1}{n-i+1} \quad (6.4.5)$$

$$\Pr[\mathcal{E}] \geq \frac{n-2}{n} \frac{n-3}{n-1} \cdots \frac{2}{4} \frac{1}{3} = \frac{2}{n(n-1)} \quad (6.4.6)$$

■

To decrease the chances of Karger's algorithm returning a non-minimal cut, as mentioned earlier in the lecture we can perform independent runs of the algorithm and take the smallest of the cuts produced. Then if anyone of the runs produces a min-cut, we get a correct solution. If the algorithm is independently run t times, then the probability that none of the runs result in the min-cut would be

$$\Pr[\text{Karger's algorithm is wrong in one run}]^t \leq \left(1 - \frac{2}{n(n-1)}\right)^t \quad (6.4.7)$$

In particular, if $t = k \frac{n(n-1)}{2}$, then roughly, $\left(1 - \frac{2}{n(n-1)}\right)^t \leq e^{-k}$.

One interesting implication of Karger's algorithm is the following.

Theorem 6.4.2 *There are at most $\binom{n}{2}$ global min-cuts in any graph G .*

Proof: For each global min-cut, the algorithm has probability at least $\frac{2}{n(n-1)}$ to produce that min-cut. Let $C_i, 1 \leq i \leq m$ be all the global min-cuts. Run the algorithm once and obtain the cut C . The probability that C is a global min-cut is

$$\Pr[C \text{ is global min cut}] = \sum_i \Pr[C = C_i] \geq m \frac{2}{n(n-1)} \quad (6.4.8)$$

Now since probability is always no greater than 1, we obtain $m \leq \frac{n(n-1)}{2}$. ■