CPSC 536N: Randomized Algorithms 2011-12 Term 2

Lecture 1

Prof. Nick Harvey *University of British Columbia*

1 Course Overview

1.1 What are randomized algorithms?

This course is about **randomized algorithms**. It is important to understand that this is **not** the same as **average case analysis** of algorithms, in which one analyzes the performance of a deterministic algorithm when the inputs are drawn from a certain distribution. Instead, we will look at algorithms that can "flip random coins" and whose behavior depends on the outcomes of those coin flips, and we will do **worst-case analysis**, meaning that we analyze the performance and accuracy of the algorithm on its worst possible input.

Depending on the random coin flips, these algorithms might either run for a very long time, or output the wrong answer. At first, one might feel uncomfortable with algorithms that can exhibit such failures. However, typically it will be possibile to choose the failure probability to be extremely tiny. The following amusing quotation of Christos Papadimitriou justifies ignoring these small failure probabilities:

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least 2^{-100}.

Of course, this quotation is not formally correct and presumably tongue-in-cheek, since we will always view probability distributions as purely formal mathematical objects; we will not touch on the philosophical questions about whether true randomness has any realization in the physical universe. Nevertheless, every algorithm we describe can be usefully implemented on any physical computer with a reasonable pseudo-random generator.

1.2 Why?

So why are we interested in randomized algorithms? What do we gain by allowing small probabilities of failure?

- Randomized algorithms are very often *simpler* than the best known deterministic algorithm.
- Randomized algorithms are often more efficient (faster, or use less space) than the best known deterministic algorithm.
- There are many scenarios in which randomized algorithms can be *provably better* than the best possible deterministic algorithm.
- There are even scenarios in which no deterministic algorithm can do anything non-trivial — every interesting algorithm must necessarily be randomized.
- Sometimes ideas from randomization lead to interesting deterministic algorithms too.

1.3 Objectives

Over the past thirty years, randomization has seen steadily increasing use in theoretical computer science. As a rough estimate, I would guess that 50% of research papers in the "core theory" conferences use randomness in a non-trivial way. In more applied areas (which I do not follow as closely), such as computer networking, databases, and scientific computing, the last 10-15 years have seen several interesting uses of randomized algorithms. So I think it is fair to say that, for several areas of computer science, understanding these ideas is important for cutting-edge research.

In this course, I intend to cover many of the key techniques, and to illustrate each technique with important, often amazing, applications. The two goals are:

- for you to understand these central techniques, so that you can understand them when you encounter them in papers, and perhaps use them in your own papers.
- to introduce you to the areas in which these techniques are useful, so that you will be comfortable if you encounter those areas again in the future, and possibly to entice you to work in these areas.

To a large degree, the techniques that we will encounter are quite general and useful in other areas, such as discrete mathematics and machine learning.

1.4 Techniques

The main techniques we will encounter are:

1. Concentration (i.e., tail bounds)
2. Avoiding zeros of polynomials
3. Random matrix theory
4. Dimensionality reduction
5. Use of pseudorandom objects
6. Pairwise independence
7. Hashing
8. Martingales

Of these, the first will certainly be the most important.

1.5 Strategies

Here is a list of some very high-level strategies for using randomization, as well as some applications (many of which we will see later) in which these strategies were successfully employed. Do not worry if these strategies are too vague to be understood at this point; hopefully seeing the application later will clarify what I mean.

1. *Generating short "fingerprints".* Examples: Hashing, dimensionality reduction, streaming algorithms.

2. *Distilling a problem to its core.* Examples: computing minimum cuts, sparsification.
3. *Finding hay in a haystack.* Examples: polynomial identity testing, perfect matching.
4. *Avoiding pathological inputs.* Examples: quicksort.
5. *Fooling adversaries.* Examples: online algorithms, cryptography.
6. *Rounding (converting continuous objects into integral objects).* Examples: congestion minimization, sparsest cut.

1.6 What won't we do?

Unfortunately there is too much beautiful work in this area and we can't cover everything. I could easily do dozens of lectures on this topic, but we would all run out of stamina before I run out of material. Some areas we definitely won't discuss include computational geometry, parallel computing, number theory and Markov Chain Monte Carlo methods. There are also many beautiful results that feel to me rather isolated, in that they do not seem to involve general purpose techniques, or have not generated much follow-up work. I will try to avoid such results.

2 Examples

2.1 Example 1: Testing Equality

Suppose you download a large movie from an internet server. Before watching it, you'd like to check that your downloaded file has no errors, i.e., the file on your machine is identical to the file on the server. You would like to do this check without much additional communication, so sending the entire file back to the server is not a good solution. Ignoring cryptographic considerations, this is essentially the problem of computing a **checksum** and there are standard ways to do this, e.g., CRCs.

For concreteness, say that the file is n bits long, the server has the bitvector $a = (a_1, \ldots, a_n)$ and you have the bits $b = (b_1, \ldots, b_n)$. For standard checksums, the guarantee is:

- For "most" vectors a and b, the checksum will detect if they are not identical. So the guarantee is with respect to a supposed "distribution" on the vectors a and b.

However, as stated in Section 1, our mindset is **worst-case analysis**. We are interested in algorithms that work well even for the worst possible inputs, so this guarantee is too weak. Instead, we'd like a guarantee of this sort:

- For **every** vectors a and b, our algorithm will flip some random coins, and for most outcomes of the coins, will detect whether or not a and b are identical.

This guarantee differs in that any failures are no longer due to potentially bad inputs (a and b), but only due to potentially bad coin flips.

The main tool we will use is this simple theorem.

Theorem 1 *Let $p(x)$ be a (univariate) polynomial of degree at most d over any field. Then p has at most d roots (i.e., p evaluates to zero on at most d elements of the field).*

The proof follows from the facts that any polynomial can be uniquely factored into irreducibles, irreducibles of degree 1 have exactly one root, and irreducibles of degree greater than 1 have no roots.

Construct the polynomials $p_a(x) = \sum_{i=1}^{n} a_i x^i$ and $p_b(x) = \sum_{i=1}^{n} b_i x_i$. Since all coefficients of p_a and p_b are either 0 or 1 (which are well-defined elements of any field), we can view them as polynomials over whatever field we wish. We will choose a finite field $\mathbb{F}$ whose number of elements $|\mathbb{F}|$ is between $2n$ and $4n$. (A fact known as Bertrand's Postulate implies that such a field exists, and brute force search can find one in poly(n) time.)

Let $q = p_a - p_b$. Note that $a = b$ if and only if q is "identically zero" (meaning, the polynomial with all coefficients equal to zero). On the other hand, if $a \neq b$, then q is a non-zero polynomial of degree at most n, so Theorem 1 implies it has at most n roots. So, if we pick an element $x \in \mathbb{F}$ uniformly at random then its probability of being a root of q is at most $n/|\mathbb{F}| \leq 1/2$.

This suggests the following algorithm. You and the server somehow agree on the field $\mathbb{F}$. The server picks $x \in \mathbb{F}$ uniformly at random. It sends you x and $p_a(x)$, which are both elements of $\mathbb{F}$ and hence each can be represented with at most $\log(4n)$ bits. You compute $q(x) = p_a(x) - p_b(x)$. If $q(x) = 0$ the algorithm announces "a and b are equal". If $q(x) \neq 0$ the algorithm announces "a and b are not equal".

As argued above, this algorithm makes an error only if $a \neq b$ and x is a root of q, so the algorithm fails with probability at most $1/2$. Two points are worth noting:

- This analysis is valid **for all** a and b and the only randomness used in our probabilistic analysis comes from the random choice of x.
- The algorithm only makes a mistake if $a \neq b$ and never makes a mistake if $a = b$. We say that such an algorithm has **one-sided error**.

The number of bits exchanged between you and the server is only $O(\log n)$. Furthermore, it is known that this is optimal: every randomized algorithm for this problem that succeds with constant probability requires $\Omega(\log n)$ bits of communication.

Recall our lists of techniques and strategies from Section 1. This example uses technique 2 (avoiding zeros of polynomials) and strategy 3 (finding hay in a haystack), where the field $\mathbb{F}$ is the haystack and the non-roots of q are the hay.

Note that we only achieved a failure probability of $1/2$. As mentioned earlier, it is usually easy to decrease the failure probability down to any desired level. For example, if we pick the field size to be $\Theta(n^2)$, then the failure probability decreases to $O(1/n)$.

2.2 Example 2: Max Cut

The next problem we consider is the **Max Cut problem**, which is a very important problem in combinatorial optimization and approximation algorithms.

Let $G = (V, E)$ be an undirected graph. For $U \subseteq V$, let

$$\delta(U) = \{uv \in E \,:\, u \in U \text{ and } v \notin U\}.$$

The set $\delta(U)$ is called the **cut** determined by U. The Max Cut problem is to solve

$$\max\{|\delta(U)| \,:\, U \subseteq V\}.$$

This is NP-hard (and in fact was one of the original problems shown to be NP-hard by Karp in his famous 1972 paper), so we cannot hope to solve it exactly. Instead, we will be content to find a cut that is sufficiently large.

More precisely, let OPT denote the size of the maximum cut. We want an algorithm for which there exists a factor $\alpha > 0$ (independent of G) such that the set U output by the algorithm is guaranteed to have $|\delta(U)| \geq \alpha\, OPT$. (If the algorithm is randomized, we want this guarantee to hold with good probability.)

Here is a brief summary of what is known about this problem.

- Folklore: there is an algorithm (in fact many of them) with $\alpha = 1/2$.
- Goemans and Williamson 1995: there is an algorithm with $\alpha = 0.878....$
- Bellare, Goldreich and Sudan 1998: no efficient algorithm has $\alpha > 83/84$, unless $P = NP$.
- Håstad 2001: no efficient algorithm has $\alpha > 16/17$, unless $P = NP$.
- Khot, Kindler, Mossel, O'Donnel and Oleszkiewicz 2004-2005: no efficient algorithm has $\alpha > 0.878...$, assuming the Unique Games Conjecture.

We will give a randomized algorithm achieving $\alpha = 1/2$. In fact, this algorithm appears in an old paper of Erdos. The algorithm couldn't possibly be any simpler: it simply lets U be a uniformly random subset of V. One can check that this is equivalent to independently adding each vertex to U with probability $1/2$. Note that the algorithm does not even look at the edges of G! I would also view this algorithm as employing strategy 3 (finding hay in a haystack), where the cuts are the haystack and the near-maximum cuts are the hay.

The following claim analyzes this algorithm.

Claim 2 *Let U be the set chosen by the algorithm. Then $\mathrm{E}[|\delta(U)|] \geq OPT/2$.*

Proof: For every edge $uv \in E$, let X_{uv} be the indicator random variable which is 1 if $uv \in \delta(U)$. Then

$$\begin{aligned}
\mathrm{E}[|\delta(U)|] &= \mathrm{E}[\sum_{uv \in E} X_{uv}] \\
&= \sum_{uv \in E} \mathrm{E}[X_{uv}] \quad \text{(linearity of expectation)} \\
&= \sum_{uv \in E} \Pr[X_{uv} = 1]
\end{aligned}$$

Now we note that

$$\begin{aligned}
\Pr[X_{uv} = 1] &= \Pr[\,(u \in U \wedge v \notin U) \;\vee\; (u \notin U \wedge v \in U)\,] \\
&= \Pr[u \in U \wedge v \notin U] \;+\; \Pr[u \notin U \wedge v \in U] \quad \text{(these are disjoint events)} \\
&= \Pr[u \in U] \cdot \Pr[v \notin U] \;+\; \Pr[u \notin U] \cdot \Pr[v \in U] \quad \text{(independence)} \\
&= 1/2.
\end{aligned}$$

Thus $\mathrm{E}[|\delta(U)|] = |E|/2$. The claim follows since OPT is certainly no larger than $|E|$. □

Is the analysis really complete? Not quite. We wish to show that $\Pr[|\delta(U)| \geq OPT/2]$ is large, but we have only shown that $\mathrm{E}[|\delta(U)|] \geq OPT/2$. To connect these two statements, we need to show that $|\delta(U)|$ is likely to be close to its mean. This is the topic of the next lecture.