COS 521: A Graduate Course in Algorithm Design and Analysis

Sanjeev Arora
Department of Computer Science
Princeton University

March 22, 2015

Abstract

These are lecture notes from a graduate course for Computer Science graduate students at Princeton University in Fall 2013 and Fall 2014. (The course also attracts undergrads and non-CS grads and total enrollment in 2014 was 35.) The course assumes prior knowledge of algorithms at the undergraduate level.

This new course arose out of thinking about the right algorithms training for CS grads today, since the environment for algorithms design and use has greatly changed since the 1980s when the canonical grad algorithms courses were designed. The course topics are somewhat nontraditional, and some of the homeworks involve simple programming assignments that encourage students to play with algorithms using simple environments like Matlab and Scipy.

Since this is the last theory course many of my students (grad or undergrad) might take for the rest of their lives, I think of the scope as more than just algorithms; my goal is to make them look at the world anew with a mathematical/algorithmic eye. For instance, I discovered many holes in my students' undergraduate CS education: information/coding theory, economic utility and game theory, decision-making under uncertainty, cryptography (anything beyond the RSA cryptosystem), etc. So I created space for these topics as well, figuring that the value added by this was greater than by, say, presenting detailed analysis of the Ellipsoid algorithm (which I sketch instead).

Programming assignments went out of fashion in most algorithms courses in the past few decades, but I think it is time to bring them back. First, CS students are used to a hands-on learning experience; an algorithm becomes real only once they see it run on real data. Second, the computer science world increasingly relies on off-the-shelf packages and library routines, and this is how algorithms are implemented in industry. One can write a few lines of code in `matlab` and `scipy`, and run it within minutes on datasets of millions or billions of numbers. No JAVA or C++ needed! Algorithms education should give students at least a taste of such powerful tools. Finally, even for theory students it can be very beneficial to play with algorithms and data a bit; this will help them develop a different kind of theory.

The course gives students a choice between taking a 48-hour final, or doing a collaborative term project. Some sample term projects can be found at the course home page. `http://www.cs.princeton.edu/courses/archive/fall14/cos521/`

This course is very much a work in progress, and I welcome your feedback and suggestions.

I thank numerous colleagues for useful suggestions during the design of this course. Above all, I thank my students for motivating me to teach them better; their feedback and questions have helped shape these course notes.

Sanjeev Arora
March 2015

About this course

Algorithms are integral to computer science: every computer scientist (even as an undergrad) has designed some. So has many a physicist, electrical engineer, mathematician etc. This course is meant to be your one-stop shop to learn how to design a variety of algorithms. The operative word is "variety."In other words you will avoid the blinders that one often sees in domain experts. A bayesian needs to see priors on the data before he can begin designing algorithms; an optimization expert needs to cast all problems as convex optimization; a systems designer has never seen any problem that cannot be solved by hashing. (OK, mostly kidding but there is some truth in these stereotypes.) These and more domain-specific ideas make an appearance in our course, but we will learn to not be wedded to any single approach.

The primary skill you will learn in this course is how to *analyse* algorithms: prove their correctness and their running time and any other relevant properties. Learning to analyse a variety of algorithms (designed by others) will let you design better algorithms later in life. I will try to fill the course with beautiful algorithms. Be prepared for frequent rose-smelling stops, in other words.

Difference between undergrad algorithms and this course

Undergrad algorithms is largely about algorithms discovered before 1990; grad algorithms is a lot about algorithms discovered since 1990. What happened in 1990 that caused this change, you may ask? Nothing. I chose this arbitrarily; maybe I could have said 1985 or 1995. There was no single event but just a gradual shift in the emphasis and goals of computer science as it became a more mature field.

In the first few decades of computer science, algorithms research was driven by the goal of designing basic components of a computer: operating systems, compilers, networks, etc. Other motivations were classical problems in discrete mathematics, operations research, graph theory. The algorithmic ideas that came out of these quests form the core of undergraduate course: data structures, graph traversal, string matching, parsing, network flows, matchings, etc. Starting around 1990 theoretical computer science broadened its horizons and started looking at new problems: algorithms for bioinformatics, algorithms and mechanism design for e-commerce, algorithms to understand big data or big networks. This changed algorithms research and the change is ongoing. One big change is that it is often unclear *what the algorithmic problem even is*. Identifying it is part of the challenge. Thus good *modeling* is important. This in turn is shaped by understanding what is *possible* (given our understanding of computational complexity) and what is *reasonable* given the limitations of the type of inputs we are given.

Some examples of this change:

The changing graph. In undergrad algorithms the graph is given and arbitrary (worst-case). In grad algorithms we are willing to look at the domain (social network, computer vision etc.) that the graph came from since the properties of graphs in those domains may be germane to designing a good algorithm. (This is not a radical idea of course but we will see that formulating good graph models is not easy. This is why you see a lot of heuristic work in practice, without any mathematical proofs of correctness.)

Changing data structures: In undergrad algorithms the data structures were simple and often designed to hold data generated by other algorithms. A stack allows you to hold vertices during depth-first search traversal of a graph, or instances of a recursive call to a procedure. A heap is useful for sorting and searching.

But in the newer applications, data often comes from sources we don't control. Thus it may be noisy, or inexact, or both. It may be high dimensional. Thus something like heaps will not work, and we need more advanced data structures.

We will encounter the "curse of dimensionality"which constrains algorithm design for high-dimensional data.

Changing notion of input/output: Algorithms in your undergrad course have a simple input/output model. But increasingly we see a more nuanced interpretation of what the input is: datastreams (useful in analytics involving routers and webservers), online (sequence of requests), social network graphs, etc. And there is a corresponding subtlety in settling on what an appropriate output is, since we have to balance output quality with algorithmic efficiency. In fact, design of a suitable algorithm often goes hand in hand with understanding what kind of output is reasonable to hope for.

Type of analysis: In undergrad algorithms the algorithms were often *exact* and work on *all* (i.e., worst-case) inputs. In grad algorithms we are willing to relax these requirements.

Contents

About this course

Algorithms are integral to computer science: every computer scientist (even as an undergrad) has designed some. So has many a physicist, electrical engineer, mathematician etc. This course is meant to be your one-stop shop to learn how to design a variety of algorithms. The operative word is "variety."In other words you will avoid the blinders that one often sees in domain experts. A bayesian needs to see priors on the data before he can begin designing algorithms; an optimization expert needs to cast all problems as convex optimization; a systems designer has never seen any problem that cannot be solved by hashing. (OK, mostly kidding but there is some truth in these stereotypes.) These and more domain-specific ideas make an appearance in our course, but we will learn to not be wedded to any single approach.

The primary skill you will learn in this course is how to *analyse* algorithms: prove their correctness and their running time and any other relevant properties. Learning to analyse a variety of algorithms (designed by others) will let you design better algorithms later in life. I will try to fill the course with beautiful algorithms. Be prepared for frequent rose-smelling stops, in other words.

Difference between undergrad algorithms and this course

Undergrad algorithms is largely about algorithms discovered before 1990; grad algorithms is a lot about algorithms discovered since 1990. OK, I picked 1990 as an arbitrary cutoff. Maybe it is 1985, or 1995. What happened in 1990 that caused this change, you may ask? Nothing. It was no single event but just a gradual shift in the emphasis and goals of computer science as it became a more mature field.

In the first few decades of computer science, algorithms research was driven by the goal of designing basic components of a computer: operating systems, compilers, networks, etc. Other motivations were classical problems in discrete mathematics, operations research, graph theory. The algorithmic ideas that came out of these quests form the core of undergraduate course: data structures, graph traversal, string matching, parsing, network flows, etc. Starting around 1990 theoretical computer science broadened its horizons and started looking at new problems: algorithms for bioinformatics, algorithms and mechanism design for e-commerce, algorithms to understand big data or big networks. This changed algorithms research and the change is ongoing. One big change is that it is often unclear *what the algorithmic problem even is*— identifying it is part of the challenge. Thus good *modeling* is important. This in turn is shaped by understanding what is *possible* (given our understanding of computational complexity) and what is *reasonable* given the limitations of the type of inputs we are given.

Some examples of this change:

The changing graph. In undergrad algorithms the graph is given and arbitrary (worst-case). In grad algorithms we are willing to look at the domain (social network, computer vision etc.) that the graph came from since the properties of graphs in those domains may be germane to designing a good algorithm. (This is not a radical idea of course but we will see that formulating good graph models is not easy. This is why you see a lot of heuristic work in practice, without any mathematical proofs of correctness.)

Changing data structures: In undergrad algorithms the data structures were simple and often designed to hold data generated by other algorithms (and hence under our control). A stack allows you to hold vertices during depth-first search traversal of a graph, or instances of a recursive call to a procedure. A heap is useful for sorting and searching.

But in the newer applications, data often comes from sources we don't control. Thus it may be noisy, or inexact, or both. It may be high dimensional. Thus something like heaps will not work, and we need more advanced data structures.

We will encounter the "curse of dimensionality"which constrains algorithm design for high-dimensional data.

Changing notion of input/output: Algorithms in your undergrad course have a simple input/output model. But increasingly we see a more nuanced interpretation of what the input is: datastreams (useful in analytics involving routers and webservers), online (sequence of requests), social network graphs, etc. And there is a corresponding subtlety in settling on what an appropriate output is, since we have to balance output quality with algorithmic efficiency. In fact, design of a suitable algorithm often goes hand in hand with understanding what kind of output is reasonable to hope for.

Type of analysis: In undergrad algorithms the algorithms were often *exact* and work on *all* (i.e., worst-case) inputs. In grad algorithms we are willing to relax these requirements.

Chapter 1

Hashing

Today we briefly study hashing, both because it is such a basic data structure, and because it is a good setting to develop some fluency in probability calculations.

1.1 Hashing: Preliminaries

Hashing can be thought of as a way to *rename* an address space. For instance, a router at the internet backbone may wish to have a searchable database of destination IP addresses of packets that are whizing by. An IP address is 128 bits, so the number of possible IP addresses is 2^{128}, which is too large to let us have a table indexed by IP addresses. Hashing allows us to rename each IP address by fewer bits. Furthermore, this renaming is done probabilistically, and the renaming scheme is decided in advance before we have seen the actual addresses. In other words, the scheme is *oblivious* to the actual addresses.

Formally, we want to store a subset S of a large universe U (where $|U| = 2^{128}$ in the above example). And $|S| = m$ is a relatively small subset. For each $x \in U$, we want to support 3 operations:

- $insert(x)$. Insert x into S.
- $delete(x)$. Delete x from S.
- $query(x)$. Check whether $x \in S$.

A hash table can support all these 3 operations. We design a hash function

$$h : U \longrightarrow \{0, 1, \ldots, n-1\} \tag{1.1}$$

such that $x \in U$ is placed in $T[h(x)]$, where T is a table of size n.

Since $|U| \gg n$, multiple elements can be mapped into the same location in T, and we deal with these collisions by constructing a linked list at each location in the table.

One natural question to ask is: how long is the linked list at each location?

This can be analysed under two kinds of assumptions:

1. Assume the input is the random.

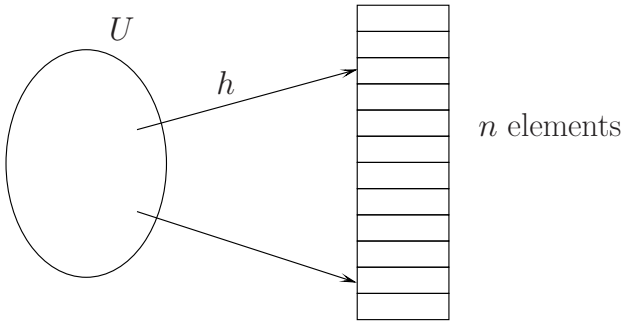

Figure 1.1: Hash table. x is placed in $T[h(x)]$.

2. Assume the input is arbitrary, but the hash function is random.

Assumption 1 may not be valid for many applications.

Hashing is a concrete method towards Assumption 2. We designate a set of hash functions $\mathcal{H}$, and when it is time to hash S, we choose a random function $h \in \mathcal{H}$ and hope that on average we will achieve good performance for S. This is a frequent benefit of a randomized approach: no single hash function works well for every input, but the average hash function may be good enough.

1.2 Hash Functions

Say we have a family of hash functions $\mathcal{H}$, and for each $h \in \mathcal{H}$, $h : U \longrightarrow [n]$[1]. What do mean if we say these functions are random?

For any $x_1, x_2, \ldots, x_m \in S$ ($x_i \neq x_j$ when $i \neq j$), and any $a_1, a_2, \ldots, a_m \in [n]$, ideally a random $\mathcal{H}$ should satisfy:

- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1] = \frac{1}{n}$.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2] = \frac{1}{n^2}$. Pairwise independence.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_k) = a_k] = \frac{1}{n^k}$. k-wise independence.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_m) = a_m] = \frac{1}{n^m}$. Full independence (note that $|U| = m$).

Generally speaking, we encounter a tradeoff. The more random $\mathcal{H}$ is, the greater the number of random bits needed to generate a function h from this class, and the higher the cost of computing h.

For example, if $\mathcal{H}$ is a fully random family, there are n^m possible h, since each of the m elements at S have n possible locations they can hash to. So we need $\log |\mathcal{H}| = m \log n$ bits to represent each hash function. Since m is usually very large, this is not practical.

[1] We use $[n]$ to denote the set $\{0, 1, \ldots, n-1\}$

But the advantage of a random hash function is that it ensures very few collisions with high probability. Let L_x be the length of the linked list containing x; this is just the number of elements with the same hash value as x. Let random variable

$$I_y = \begin{cases} 1 & \text{if } h(y) = h(x), \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

So $L_x = 1 + \sum_{y \in S; y \neq x} I_y$, and

$$E[L_x] = 1 + \sum_{y \in S; y \neq x} E[I_y] = 1 + \frac{m-1}{n} \tag{1.3}$$

Usually we choose $n > m$, so this expected length is less than 2. Later we will analyse this in more detail, asking how likely is L_x to exceed say 100.

The expectation calculation above doesn't need full independence; pairwise independence would actually suffice. This motivates the next idea.

1.3 2-Universal Hash Families

Definition 1 (Carter Wegman 1979) *Family $\mathcal{H}$ of hash functions is 2-universal if for any $x \neq y \in U$,*

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n} \tag{1.4}$$

Note that this property is even weaker than 2 independence.

We can design 2-universal hash families in the following way. Choose a prime $p \in \{|U|, \ldots, 2|U|\}$, and let

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0) \tag{1.5}$$

And let

$$h_{a,b}(x) = f_{a,b}(x) \mod n \tag{1.6}$$

Lemma 1
For any $x_1 \neq x_2$ and $s \neq t$, the following system

$$ax_1 + b = s \mod p \tag{1.7}$$
$$ax_2 + b = t \mod p \tag{1.8}$$

has exactly one solution.

Since $[p]$ constitutes a finite field, we have that $a = (x_1 - x_2)^{-1}(s - t)$ and $b = s - ax_1$. Since we have $p(p-1)$ different hash functions in $\mathcal{H}$ in this case,

$$\Pr_{h \in \mathcal{H}}[h(x_1) = s \wedge h(x_2) = t] = \frac{1}{p(p-1)} \tag{1.9}$$

Claim $\mathcal{H} = \{h_{a,b} : a, b \in [p] \wedge a \neq 0\}$ *is 2-universal.*

PROOF: For any $x_1 \neq x_2$,

$$\begin{aligned}
&\mathbf{Pr}[h_{a,b}(x_1) = h_{a,b}(x_2)] && (1.10)\\
=&\sum_{s,t\in[p],s\neq t} \delta_{(s=t \mod n)}\, \mathbf{Pr}[f_{a,b}(x_1) = s \wedge f_{a,b}(x_2) = t] && (1.11)\\
=&\frac{1}{p(p-1)} \sum_{s,t\in[p],s\neq t} \delta_{(s=t \mod n)} && (1.12)\\
\leq&\frac{1}{p(p-1)} \frac{p(p-1)}{n} && (1.13)\\
=&\frac{1}{n} && (1.14)
\end{aligned}$$

where δ is the Dirac delta function. Equation (1.13) follows because for each $s \in [p]$, we have at most $(p-1)/n$ different t such that $s \neq t$ and $s = t \mod n$. □

Can we design a collision free hash table then? Say we have m elements, and the hash table is of size n. Since for any $x_1 \neq x_2$, $\mathbf{Pr}_h[h(x_1) = h(x_2)] \leq \frac{1}{n}$, the expected number of total collisions is just

$$E[\sum_{x_1\neq x_2} h(x_1) = h(x_2)] = \sum_{x_1\neq x_2} E[h(x_1) = h(x_2)] \leq \binom{m}{2}\frac{1}{n} \tag{1.15}$$

Let's pick $m \geq n^2$, then

$$E[\text{number of collisions}] \leq \frac{1}{2} \tag{1.16}$$

and so

$$\mathbf{Pr}_{h\in H}[\exists \text{ a collision}] \leq \frac{1}{2} \tag{1.17}$$

So if the size the hash table is large enough $m \geq n^2$, we can easily find a collision free hash functions. But in reality, such a large table is often unrealistic. We may use a two-layer hash table to avoid this problem.

Specifically, let s_i denote the number of collisions at location i. If we can construct a second layer table of size s_i^2, we can easily find a collision-free hash table to store all the s_i elements. Thus the total size of the second-layer hash tables is $\sum_{i=0}^{m-1} s_i^2$.

Note that $\sum_{i=0}^{m-1} s_i(s_i - 1)$ is just the number of collisions calculated in Equation (1.15), so

$$E[\sum_i s_i^2] = E[\sum_i s_i(s_i-1)] + E[\sum_i s_i] = \frac{m(m-1)}{n} + m \leq 2m \tag{1.18}$$

1.4 Load Balancing

Now we think a bit about how large the linked lists (ie number of collisions) can get. Let us think for simplicity about hashing n keys in a hash table of size n. This is the famous

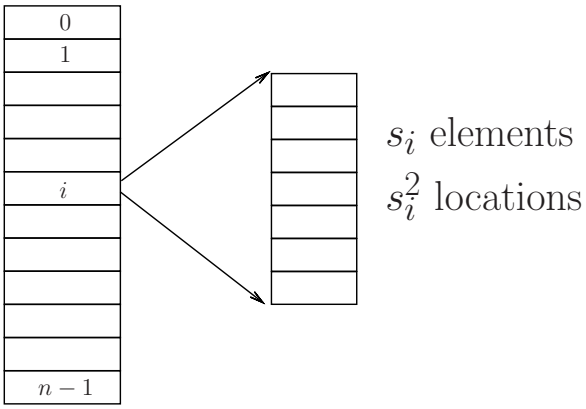

Figure 1.2: Two layer hash tables.

balls-and-bins calculation, also called load balance problem. We have n balls and n bins, and we randomly put the balls into bins. Then for a given i,

$$\mathbf{Pr}[\text{bin}_i \text{ gets more than } k \text{ elements}] \leq \binom{n}{k} \cdot \frac{1}{n^k} \leq \frac{1}{k!} \tag{1.19}$$

By Stirling's formula,

$$k! \sim \sqrt{2nk}(\frac{k}{e})^k \tag{1.20}$$

If we choose $k = O(\frac{\log n}{\log \log n})$, we can let $\frac{1}{k!} \leq \frac{1}{n^2}$. Then

$$\mathbf{Pr}[\exists \text{ a bin} \geq k \text{ balls}] \leq n \cdot \frac{1}{n^2} = \frac{1}{n} \tag{1.21}$$

So with probability larger than $1 - \frac{1}{n}$[2],

$$\max \text{load} \leq O(\frac{\log n}{\log \log n}) \tag{1.22}$$

Aside: The above load balancing is not bad; no more than $O(\frac{\log n}{\log \log n})$ balls in a bin with high probability. Can we modify the method of throwing balls into bins to improve the load balancing? We use an idea that you use at the supermarket checkout: instead of going to a random checkout counter you try to go to the counter with the shortest queue. In the load balancing case this is computationally too expensive: one has to check all n queues. A much simpler version is the following: when the ball comes in, pick 2 random bins, and place the ball in the one that has fewer balls. Turns out this modified rule ensures that the maximal load drops to $O(\log \log n)$, which is a huge improvement. This called the *power of two choices*.

[2]this can be easily improve to $1 - \frac{1}{n^c}$ for any constant c

Chapter 2

Karger's Min Cut Algorithm

Today's topic is simple but gorgeous: Karger's min cut algorithm and its extension. It is a simple randomized algorithm for finding the *minimum cut* in a graph: a subset of vertices S in which the set of edges leaving S, denoted $E(S, \overline{S})$ has minimum size among all subsets. You may have seen a polynomial-time algorithm for this problem in your undergrad class that uses maximum flow. Karger's algorithm is much more elementary and and a great introduction to randomized algorithms.

The algorithm is this: *Pick a random edge, and merge its endpoints into a single "supernode."Repeat until the graph has only two supernodes, which is output as our guess for min-cut. (As you continue, the supernodes may develop parallel edges; these are allowed. Selfloops are ignored.)* See Figure 2.1.

Note that if you pick a random edge, it is more likely to come from parts of the graph that contain more edges in the first place. Thus this algorithm looks like a great heuristic to try on all kinds of real-life graphs, where one wants to *cluster* the nodes into "tightly-knit" portions. For example, social networks may cluster into communities; graphs where edges capture similarity of pixels may cluster to give different portions of the image (sky, grass, road etc.). Thus instead of continuing Karger's algorithm until you have two supernodes left, you could stop it when there are k supernodes and try to understand whether these correspond to a reasonable clustering. (Aside: There are much better clustering algorithms out there.)

Today we will first see that the above version of the algorithm yields the optimum min cut with probability at least $2/n^2$. Thus we can repeat it say $20n^2$ times, and output the smallest cut seen in any iteration. The probability that the optimum cut is not seen in any repetition is at most $(1 - 2/n^2)^{20n^2} < 0.01$. Unfortunately, this simple version has running time about n^4 which is not great. So then we see a better version with a simple tweak that brings the running time down to closer to n^2.

2.1 Analysis of Karger's algorithm

Clearly, the two supernodes at the end correspond to a cut of the original graph, so the algorithm does always return a cut.

Main Claim: *The cut at the end is a minimum cut of the original cut with probability at least* $2/n(n-1)$.

Thus repeating the algorithm K times where $K = n(n-1)/2$ and taking the smallest cut ever discovered in these repetitions will yield a minimum cut with chance at least $1 - (1 - 1/K)^K = 1 - 1/e$. (Aside: the approximation $(1 - 1/K)^K \approx 1/e$ for large K is very useful and will reappear in later lectures.) It is relatively easy using data structures you learnt in undergrad algorithms to implement each repetition of the algorithm in $O(n^2)$ time, so the overall running time is $O(n^4)$.

The analysis is rather simple. First, recall that the sum of node degrees in an undirected graph $G = (V, E)$ is exactly $2|E|$ (since adding the degrees counts each edge twice). Thus if $|V| = n$, there exists a node of degree at most $2|E|/n$. Putting this vertex on one side of the cut and all other vertices on the other side gives a cut of size at most $2|E|/n$. Thus the minimum cut cannot have any more than $2|E|/n$ edges.

Let $(S, \overline{S})$ be any minimum cut. Then the probability that a random edge picked at the first step by Karger's algorithm lies in this particular cut is at most $2|E|/n|E| = 2/n$. If it doesn't lie in the cut, then contracting the edge maintains $(S, \overline{S})$ as a viable cut in the graph.

Since each edge contraction reduces the number of nodes by 1, the total number of edge pickings is $n - 1$ and the probability that $(S, \overline{S})$ survives all of them is

$$\Pr[\text{first edge not in this cut}] \times \Pr[\text{second edge not in this cut} \mid \text{first edge was not in the cut}] \times \cdots,$$

which is at least

$$\begin{aligned}
&(1 - \frac{2}{n})(1 - \frac{2}{n-1})(1 - \frac{2}{n-2}) \times \cdots \times \frac{3}{4} \times \frac{1}{2} \\
&\quad = (\frac{n-2}{2})(\frac{n-3}{n-1})(\frac{n-4}{n-2}) \times \cdots \times \frac{3}{4} \times \frac{1}{2} \quad \text{(Note: telescoping!!)} \\
&\quad = \frac{2}{n(n-1)}
\end{aligned}$$

Aside: We have proven a stronger result than we had needed to: *every* minimum cut remains at the end with probability at least $2/n(n-1)$. This implies in particular that the *number* of minimum cuts in an undirected graph is at most $n(n-1)/2$. (Note that the number of cuts in the graph is the set of all nonempty subsets, which is $2^n - 1$, so this implies only a tiny number of all cuts are minimum cuts.) This upper bound has had great impact in subsequent theory of algorithms, though we will not have occasion to explore that in this course.

2.2 Improvement by Karger-Stein

Karger and Stein improved the algorithm to run in $O(n^2 \log^2 n)$ time.

The idea is roughly that *repetition ensures fault tolerance*. The real-life advice of making two backups of your hard drive is related to this: the probability that both fail is much smaller than one does. In case of Karger's algorithm, the overall probability of success is too low at $2/n(n-1)$. But if run part of the way until the graph has $n/\sqrt{2}$ supernodes, then the same calculation as before shows that the probability that the mincut has survived (i.e.

no edge has been picked in it) is at least $1/2$. So you make two independent runs that go down to $n/\sqrt{2}$ supernodes, and recursively solve both of these with the same Karger-Stein algorithm. Then return the smaller of the two cuts returned by the recursive calls.

The running time for such an algorithm satisfies

$$T(n) = O(n^2) + 2T(n/\sqrt{2}),$$

which the Master theorem of ugrad algorithms[1] shows to yield $T(n) = O(n^2 \log n)$. As you might suspect, this is not the end of the story but improvements beyond this get more hairy. If anybody is interested I can give more pointers.

Claim: *The probability the algorithm returns a minimum cut is at least* $1/\log n$.

Thus repeating the algorithm $O(\log n)$ times gives a success probability at least 0.9 (say) and a running time of $O(n^2 \log^2 n)$.

To prove the claim we note that if $P(n)$ is the probability that the procedure returns a minimum cut, then

$$P(n) \geq 1 - (1 - \frac{1}{2}P(n/\sqrt{2}))^2,$$

where the term $\frac{1}{2}P(n/\sqrt{2})$ represents the probability of the event that a minimum cut survived in the shrinkage to $n/\sqrt{2}$ vertices, *and* the recursive call then recovered this minimum cut.

To see that this solves to $P(n) \geq 1/\log n$ we can do a simple induction, where the inductive step needs to verify that

$$\frac{1}{\log n} \leq 1 - (1 - \frac{1}{2}\frac{1}{\log n - 0.5})^2 = \frac{1}{\log n - 0.5} - \frac{1}{4(\log n - 0.5)^2},$$

which is true using the approximation

$$\frac{1}{\log n - 0.5} \approx \frac{1}{\log n} + \frac{0.5}{\log^2 n}.$$

Bibliography

1) Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. David Karger, Proc. ACM-SIAM SODA 1993.

2) A new approach to the minimum cut problem. David Karger and Cliff Stein, JACM 43(4):601640, 1996.

[1]Hush, hush, don't tell anybody, but most researchers don't use the Master theorem, even though it was stressed a lot in undergrad algorithms. When we need to solve such recurrences, we just unwrap the recurrence a few times and see that there are $O(\log n)$ levels, and each involves $O(n^2)$ work, for a total of $O(n^2 \log n)$.

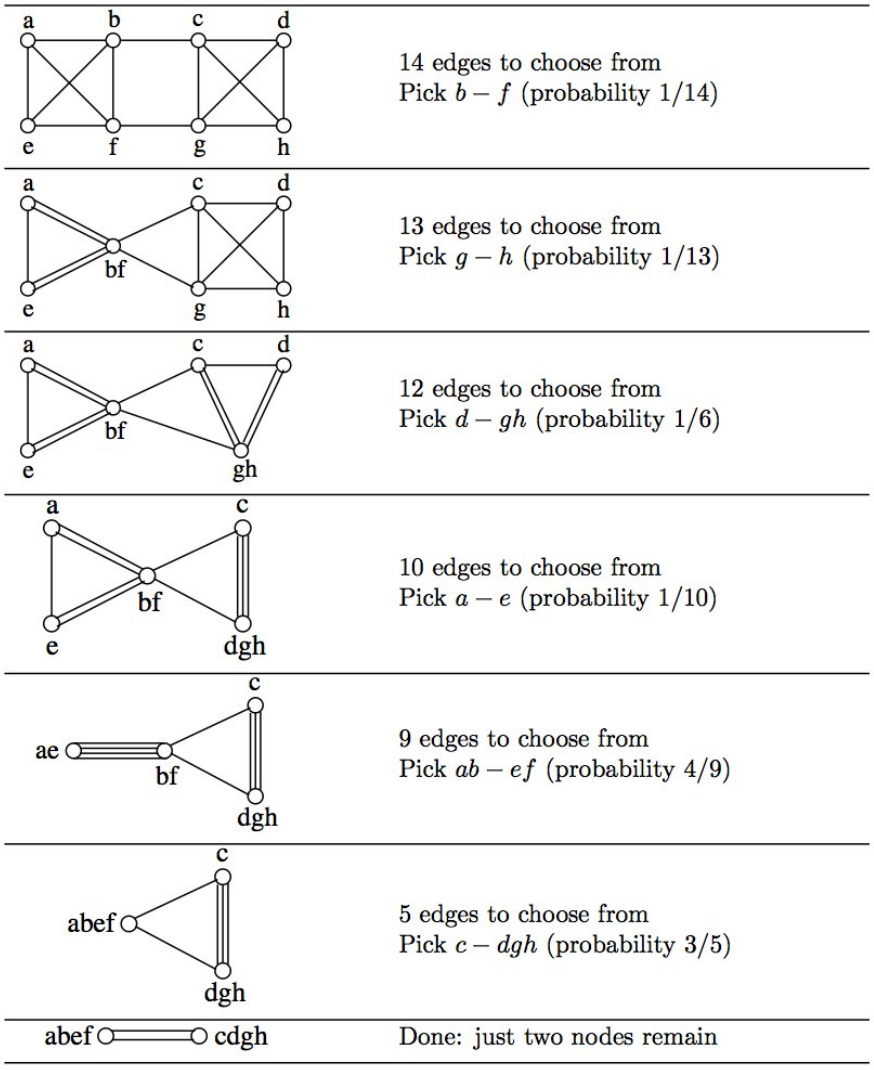

Figure 2.1: Illustration of Karger's Algorithm (borrowed from lecture notes of Sanjoy Dasgupta)

Chapter 3

Large deviations bounds and applications

Today's topic is deviation bounds: what is the probability that a random variable deviates from its mean by a lot? Recall that a random variable X is a mapping from a probability space to $\mathbf{R}$. The *expectation* or *mean* is denoted $\mathbf{E}[X]$ or sometimes as μ.

In many settings we have a set of n random variables $X_1, X_2, X_3, \ldots, X_n$ defined on the same probability space. To give an example, the probability space could be that of all possible outcomes of n tosses of a fair coin, and X_i is the random variable that is 1 if the ith toss is a head, and is 0 otherwise, which means $E[X_i] = 1/2$.

The first observation we make is that of the **Linearity of Expectation**, viz.

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i]$$

It is important to realize that linearity holds *regardless* of the whether or not the random variables are independent.

Can we say something about $\mathbf{E}[X_1 X_2]$? In general, nothing much but if X_1, X_2 are independent events (formally, this means that for all a, b $\mathbf{Pr}[X_1 = a, X_2 = b] = \mathbf{Pr}[X_1 = a]\,\mathbf{Pr}[X_2 = b]$) then $\mathbf{E}[X_1 X_2] = \mathbf{E}[X_1]\,\mathbf{E}[X_2]$.

Note that if the X_i's are pairwise independent (i.e., each pair are mutually independent) then this means that $\mathrm{var}[\sum_i X_i] = \sum_i \mathrm{var}[X_i]$.

3.1 Three progressively stronger tail bounds

Now we give three methods that give progressively stronger bounds.

3.1.1 Markov's Inequality (aka averaging)

The first of a number of inequalities presented today, **Markov's inequality** says that any *non-negative* random variable X satisfies

$$\mathbf{Pr}\left(X \geq k\,\mathbf{E}[X]\right) \leq \frac{1}{k}.$$

Note that this is just another way to write the trivial observation that $\mathbf{E}[X] \geq k \cdot \mathbf{Pr}[X \geq k]$.

Can we give any meaningful upperbound on $\mathbf{Pr}[X < c \cdot \mathbf{E}[X]]$ where $c < 1$, in other words the probability that X is a lot less than its expectation? In general we cannot. However, if we know an upperbound on X then we can. For example, if $X \in [0, 1]$ and $\mathbf{E}[X] = \mu$ then for any $c < 1$ we have (simple exercise)

$$\mathbf{Pr}[X \leq c\mu] \leq \frac{1-\mu}{1-c\mu}.$$

Sometimes this is also called an averaging argument.

EXAMPLE 1 Suppose you took a lot of exams, each scored from 1 to 100. If your average score was 90 then in at least half the exams you scored at least 80.

3.1.2 Chebyshev's Inequality

The *variance of a random variable* X is one measure (there are others too) of how "spread out"it is around its mean. It is defined as $E[(x-\mu)^2] = E[X^2] - \mu^2$.

A more powerful inequality, **Chebyshev's inequality**, says

$$\mathbf{Pr}[|X - \mu| \geq k\sigma] \leq \frac{1}{k^2},$$

where μ and σ^2 are the mean and variance of X. Recall that $\sigma^2 = \mathbf{E}[(X-\mu)^2] = \mathbf{E}[X^2] - \mu^2$. Actually, Chebyshev's inequality is just a special case of Markov's inequality: by definition,

$$\mathbf{E}\left[|X-\mu|^2\right] = \sigma^2,$$

and so,

$$\mathbf{Pr}\left[|X-\mu|^2 \geq k^2\sigma^2\right] \leq \frac{1}{k^2}.$$

Here is simple fact that's used a lot: *If $Y_1, Y_2, \ldots, Y_t$ are iid (which is jargon for independent and identically distributed) then the variance of their average $\frac{1}{k}\sum_i Y_t$ is exactly $1/t$ times the variance of one of them.* Using Chebyshev's inequality, this already implies that the average of iid variables converges sort-of strongly to the mean.

Example: Load balancing

Suppose we toss m balls into n bins. You can think of m jobs being randomly assigned to n processors. Let $X =$ number of balls assigned to the first bin. Then $\mathbf{E}[X] = m/n$. What is the chance that $X > 2m/n$? Markov's inequality says this is less than $1/2$.

To use Chebyshev we need to compute the variance of X. For this let Y_i be the indicator random variable that is 1 iff the ith ball falls in the first bin. Then $X = \sum_i Y_i$. Hence

$$\mathbf{E}[X^2] = \mathbf{E}[\sum_i Y_i^2 + 2\sum_{i<j} Y_i Y_j] = \sum_i \mathbf{E}[Y_i^2] + \sum_{i<j} \mathbf{E}[Y_i Y_j].$$

Now for independent random variables $\mathbf{E}[Y_i Y_j] = \mathbf{E}[Y_i]\,\mathbf{E}[Y_j]$ so $\mathbf{E}[X^2] = \frac{m}{n} + \frac{m(m-1)}{n^2}$. Hence the variance is very close to m/n, and thus Chebyshev implies that the probability that $\Pr[X > 2\frac{m}{n}] < \frac{n}{m}$. When $m > 3n$, say, this is stronger than Markov.

3.1.3 Large deviation bounds

When we toss a coin many times, the expected number of heads is half the number of tosses. How tightly is this distribution concentrated? Should we be very surprised if after 1000 tosses we have 625 heads?

The *Central Limit Theorem* says that the sum of n independent random variables (with bounded mean and variance) converges to the famous Gaussian distribution (popularly known as the *Bell Curve*). This is very useful in algorithm design: we maneuver to design algorithms so that the analysis boils down to estimating the sum of independent (or somewhat independent) random variables.

To do a back of the envelope calculation, if all n coin tosses are fair (Heads has probability $1/2$) then the Gaussian approximation implies that the probability of seeing N heads where $|N - n/2| > a\sqrt{n}$ is at most $e^{-a^2/2}$. The chance of seeing at least 625 heads in 1000 tosses of an unbiased coin is less than 5.3×10^{-7}. These are pretty strong bounds!

This kind of back-of-the-envelope calculations will get most of the credit in homeworks.

Of course, for finite n the sum of n random variables need not be an exact Gaussian and that's where Chernoff bounds come in. (By the way these bounds are also known by other names in different fields since they have been independently discovered.)

First we give an inequality that works for general variables that are real-valued in $[-1, 1]$. (To apply it to more general bounded variables just scale them to $[-1, 1]$ first.)

THEOREM 2 (QUANTITATIVE VERSION OF CLT DUE TO H. CHERNOFF; SLIGHTLY INEXACT VERSION BUT GOO
If $X_1, X_2, \ldots, X_n$ are independent random variables and each $X_i \in [-1, 1]$. Let $\mu_i = E[X_i]$ and $\sigma_i^2 = var[X_i]$. Then $X = \sum_i X_i$ satisfies

$$\Pr[|X - \mu| > k\sigma] \leq 2\exp(-\frac{k^2}{4}),$$

where $\mu = \sum_i \mu_i$ and $\sigma^2 = \sum_i \sigma_i^2$. Also, $k \leq \sigma/2$ (say).

Instead of proving the above we prove a simpler theorem for binary valued variables which showcases the basic idea.

THEOREM 3
Let $X_1, X_2, \ldots, X_n$ be independent 0/1-valued random variables and let $p_i = \mathbf{E}[X_i]$, where $0 < p_i < 1$. Then the sum $X = \sum_{i=1}^n X_i$, which has mean $\mu = \sum_{i=1}^n p_i$, satisfies

$$\mathbf{Pr}[X \geq (1 + \delta)\mu] \leq (c_\delta)^\mu$$

where c_δ is shorthand for $\left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]$.

Remark: There is an analogous inequality that bounds the probability of deviation *below* the mean, whereby δ becomes negative and the $\geq$ in the probability becomes $\leq$ and the c_δ is very similar.

PROOF: Surprisingly, this inequality also is proved using the Markov inequality, albeit applied to a different random variable.

We introduce a positive dummy variable t and observe that

$$\mathbf{E}[\exp(tX)] = \mathbf{E}[\exp(t\sum_i X_i)] = \mathbf{E}[\prod_i \exp(tX_i)] = \prod_i \mathbf{E}[\exp(tX_i)], \tag{3.1}$$

where the last equality holds because the X_i r.v.s are independent. Now,

$$\mathbf{E}[\exp(tX_i)] = (1 - p_i) + p_i e^t,$$

therefore,

$$\begin{aligned}\prod_i \mathbf{E}[\exp(tX_i)] = \prod_i [1 + p_i(e^t - 1)] &\leq \prod_i \exp(p_i(e^t - 1)) \\ &= \exp(\sum_i p_i(e^t - 1)) = \exp(\mu(e^t - 1)),\end{aligned} \tag{3.2}$$

as $1 + x \leq e^x$. Finally, apply Markov's inequality to the random variable $\exp(tX)$, viz.

$$\mathbf{Pr}[X \geq (1+\delta)\mu] = \mathbf{Pr}[\exp(tX) \geq \exp(t(1+\delta)\mu)] \leq \frac{\mathbf{E}[\exp(tX)]}{\exp(t(1+\delta)\mu)} = \frac{\exp((e^t - 1)\mu)}{\exp(t(1+\delta)\mu)},$$

using lines (3.1) and (3.2) and the fact that t is positive. Since t is a dummy variable, we can choose any positive value we like for it. The right hand size is minimized if $t = \ln(1+\delta)$—just differentiate—and this leads to the theorem statement. □

3.2 Application 1: Sampling/Polling

Opinion polls and statistical sampling rely on tail bounds. Suppose there are n arbitrary numbers in $[0, 1]$ If we pick t of them randomly (with replacement!) then the sample mean is within $(1 \pm \epsilon])$ of the true mean with probability at least $1 - \delta$ if $t > \Omega(\frac{1}{\epsilon^2} \log 1/\delta)$. (Verify this calculation!)

In general, Chernoff bounds implies that taking k independent estimates and taking their mean ensures that the value is highly concentrated about their mean; large deviations happen with exponentially small probability.

3.3 Balls and Bins revisited: Load balancing

Suppose we toss m balls into n bins. You can think of m jobs being randomly assigned to n processors. Then the expected number of balls in each bin is m/n. When $m = n$ this expectation is 1 but we saw in Lecture 1 that the most overloaded bin has $\Omega(\log n / \log \log n)$ balls. However, if $m = cn \log n$ then the expected number of balls in each bin is $c \log n$. Thus Chernoff bounds imply that the chance of seeing less than $0.5c \log n$ or more than $1.5c \log n$ is less than $\gamma^{c \log n}$ for some constant γ (which depends on the $0.5, 1.5$ etc.) which can be made less than say $1/n^2$ by choosing c to be a large constant.

Moral: if an office boss is trying to allocate work fairly, he/she should first create more work and then do a random assignment.

3.4 What about the median?

Given n numbers in $[0, 1]$ can we approximate the median via sampling? This will be part of your homework.

Exercise: Show that it is impossible to estimate the *value* of the median within say 1.1 factor with $o(n)$ samples.

But what is possible is to produce a number that is an approximate median: it is greater than at least $n/2 - n/t$ numbers below it and less than at least $n/2 - n/t$ numbers. The idea is to take a random sample of a certain size and take the median of that sample. (Hint: Use balls and bins.)

One can use the approximate median algorithm to describe a version of quicksort with very predictable performance. Say we are given n numbers in an array. Recall that (random) quicksort is the sorting algorithm where you randomly pick one of the n numbers as a *pivot*, then partition the numbers into those that are bigger than and smaller than the pivot (which takes $O(n)$ time). Then you recursively sort the two subsets.

This procedure works in expected $O(n \log n)$ time as you may have learnt in an undergrad course. But its performance is uneven because the pivot may not divide the instance into two exactly equal pieces. For instance the chance that the running time exceeds $10n \log n$ time is quite high.

A better way to run quicksort is to first do a quick estimation of the median and then do a pivot. This algorithm runs in very close to $n \log n$ time, which is optimal.

Chapter 4

Hashing with real numbers and their big-data applications

Using only memory equivalent to 5 lines of printed text, you can estimate with a typical accuracy of 5 per cent and in a single pass the total vocabulary of Shakespeare. This wonderfully simple algorithm has applications in data mining, estimating characteristics of huge data flows in routers, etc. It can be implemented by a novice, can be fully parallelized with optimal speed-up and only need minimal hardware requirements. Theres even a bit of math in the middle!

Opening lines of a paper by Durand and Flajolet, 2003.

As we saw in Lecture 1, hashing can be thought of as a way to *rename* an address space. For instance, a router at the internet backbone may wish to have a searchable database of destination IP addresses of packets that are whizzing by. An IP address is 128 bits, so the number of possible IP addresses is 2^{128}, which is too large to let us have a table indexed by IP addresses. Hashing allows us to rename each IP address by fewer bits. In Lecture 1 this hash was a number in a finite field (integers modulo a prime p). In recent years large data algorithms have used hashing in interesting ways where the hash is viewed as a *real number*. For instance, we may hash IP addresses to real numbers in the unit interval $[0, 1]$.

Example 2 (Dartthrowing method of estimating areas) Suppose gives you a piece of paper of irregular shape and you wish to determine its area. You can do so by pinning it on a piece of graph paper. Say, it lies completely inside the unit square. Then throw a dart n times on the unit square and observe the fraction of times it falls on the irregularly shaped paper. This fraction is an estimator for the area of the paper.

Of course, the digital analog of throwing a dart n times on the unit square is to take a random hash function from $\{1, \ldots, n\}$ to $[0, 1] \times [0, 1]$.

Strictly speaking, one cannot hash to a real number since computers lack infinite precision. Instead, one hashes to *rational* numbers in $[0, 1]$. For instance, hash IP addresses to the set $[p]$ as before, and then think of number "i mod p"as the rational number i/p. This works OK so long as our method doesn't use too many bits of precision in the real-valued hash.

A general note about sampling. As pointed out in Lecture 3 using the random variable "Number of ears," the expectation of a random variable may never be attained at any point in the probability space. But if we draw a random sample, then we know by Chebysev's inequality that the sample has chance at least $1 - 1/k^2$ of taking a value in the interval $[\mu - k\sigma, \mu + k\sigma]$ where μ, σ denote the mean and variance respectively. Thus to get any reasonable idea of μ we need σ to be less than μ. But if we take t independent samples (even pairwise independent will do) then the variance of the mean of these samples is σ^2/t. Hence by increasing t we can get a better estimate of μ.

4.1 Estimating the cardinality of a set that's too large to store

Continuing with the router example, suppose the router wishes to maintain a count of the number of *distinct* IP addresses seen in the past hour. It would be too wasteful to actually store all the IP addresses; an approximate count is fine. This is also the application alluded to in the quote at the start of the lecture.

An idea: Pick k random hash functions $h_1, h_2, \ldots, h_k$ that map a 128-bit address to a random real number in $[0, 1]$. (For now let's suppose that these are actually random functions.) Now maintain k registers, initialized to 0. Whenever a packet whizzes by, and its IP address is x, compute $h_i(x)$ for each i. If $h_i(x)$ is less than the number currently stored in the ith register, then write $h_i(x)$ in the ith register.

Let Y_i be the random variable denoting the contents of the ith register at the end. (It is a random variable because the hash function was chosen randomly. The packet addresses are not random.) Realize that Y_i is nothing but the *lowest value of $h_i(x)$ among all IP addresses seen so far.*

Suppose the number of distinct IP addresses seen is N. This is what we are trying to estimate.

Fact: $\mathbf{E}[Y_i] = \frac{1}{N+1}$ and the variance of Y_i is $1/(N+1)^2$.

The expectation looks intuitively about right: the minimum of N random elements in $[0, 1]$ should be around $1/N$.

Let's do the expectation calculation. The probability that Y_i is z is the probability that one of the IP addresses mapped to z and all the others mapped to numbers greater than z.

$$\mathbf{E}[Y_i] = \int_{z=0}^{1} \Pr[Y_i > z] dz = \int_{z=0}^{1} (1-z)^N dz = \frac{1}{N+1}.$$

(Here's a slick alternative proof of the $1/(N+1)$ calculation. Imagine picking $N+1$ random numbers in $[0, 1]$ and consider the chance that the $N+1$th element is the smallest. By symmetry this chance is $1/(N+1)$. But this chance is exactly the expected value of the minimum of the first N numbers. QED.)

Since we picked k random hash functions, the Y_i's are iid. Let $\overline{Y}$ is be their mean. Then the variance of $\overline{Y}$ is $1/k(N+1)^2$, in other words, k times lower than the variance of each individual Y_i. Thus if $1/k$ is less than ϵ^2 the standard deviation is less than $\epsilon/(N+1)$, whereas the mean is $1/(N+1)$. Thus with constant probability the estimate $1/\overline{Y}$ is within $(1+\epsilon)$ factor of N.

All this assumed that the hash functions are random functions from 128-bit numbers to $[0, 1]$. Let's now show that it suffices to pick hash functions from a pairwise independent family, albeit now yielding an estimate that is only correct up to some constant factor. Specifically, the algorithm will take k pairwise independent hashes and see if the majority of the min values are contained in some interval of the type $[1/3x, 3/x]$. Then x is our estimate for N, the number of elements. This estimate will be correct up to a factor 3 with probability at least $1 - 1/k$.

What is the probability that we hash N different elements using such a hash function and the smallest element is *less* than $1/3N$? For each element x, $\Pr[h(x) < 1/3N]$ is at most $1/3N$, so by the union bound, the probability in question is at most $N \times 1/3N = 1/3$. Similarly, the probability that $\Pr[\exists x \ : h(x) \leq 1/N]$ can be lowerbounded by the inclusion-exclusion bound.

LEMMA 4 (INCLUSION-EXCLUSION BOUND)
$\Pr[A_1 \vee A_2 \ldots \vee A_n]$, *the probability that at least one of the events* $A_1, A_2, \ldots, A_n$ *happens, satisfies*

$$\sum_i \Pr[A_i] - \sum_{i \neq j} \Pr[A_i \wedge A_j] \leq \Pr[A_1 \vee A_2 \ldots \vee A_n] \leq \sum_i \Pr[A_i].$$

Since our events are pairwise independent we obtain

$$\Pr[\exists x \ : h(x) \leq 1/N] \geq N \times \frac{1}{N} - \binom{N}{2} \frac{1}{N^2} \geq \frac{1}{2}.$$

Using a little more work it can be shown that with probability at least 0.6 the minimum hash is in the interval $[1/3N, 3/N]$. (NB: These calculations can be improved if the hash is from a 4-wise independent family.) Thus if we repeat with k hashes, the probability that the majority of min values are not contained in $[1/3N, 3/N]$ drops as $O(1/k)$.

4.2 Estimating document similarity

One of the aspects of the data deluge on the web is that often one finds duplicate copies of the same thing. Sometimes the copies may not be exactly identical: for example mirrored copies of the same page but some are out of date. The same news article or blog post may be reposted many times, sometimes with editorial comments. By detecting duplicates and near-duplicates internet companies can often save on storage by an order of magnitude.

We present a technique called *similarity hashing* that allows this approximately. It is a hashing method such that the hash preserves some "sketch" of the document. Two documents' similarity can be estimate by comparing their hashes. This is an example of a burgeoning research area of hashing while preserving some *semantic* information. In general finding similar items in databases is a big part of data mining (find customers with similar purchasing habits, similar tastes, etc.). Today's simple hash is merely a way to dip our toes in these waters.

So think of a document as a *set*: the set of words appearing in it. The *Jaccard similarity* of documents/sets A, B is defined to be $|A \cap B| \,/\, |A \cup B|$. This is 1 iff $A = B$ and 0 iff the sets are disjoint.

Basic idea: Pick a random hash function mapping the underlying universe of elements to $[0, 1]$. Define the hash of a set A to be the *minimum* of $h(x)$ over all $x \in A$. Then by symmetry, $\Pr[\text{hash}(A) = \text{hash}(B)]$ is exactly the Jaccard similarity. (Note that if two elements x, y are different then $\Pr[h(x) = h(y)]$ is 0 when the hash is real-valued. Thus the only possibility of a collision arises from elements in the intersection of A, B.) Thus one could pick k random hash functions and take the fraction of instances of $\text{hash}(A) = \text{hash}(B)$ as an estimate of the Jaccard similarity. This has the right expectation but we need to repeat with k different hash functions to get a better estimate.

The analysis goes as follows. Suppose we are interested in flagging pairs of documents whose Jaccard-similarity is at least 0.9. Then we compute k hashes and flag the pair if at least $0.9 - \epsilon$ fraction of the hashes collide. Chernoff bounds imply that if $k = \Omega(1/\epsilon^2)$ this flags all document pairs that have similarity at least 0.9 and does not flag any pairs with similarity less than $0.9 - 3\epsilon$.

To make this method more realistic we need to replace the idealized random hash function with a real one and analyse it. That is beyond the scope of this lecture. Indyk showed that it suffices to use a k-wise independent hash function for $k = \Omega(\log(1/\epsilon)$ to let us estimate Jaccard-similarity up to error ϵ. Thorup recently showed how to do the estimation with pairwise independent functions. This analysis seems rather sophisticated; let me know if you happen to figure it out.

Bibliography

1. Broder, Andrei Z. (1997), *On the resemblance and containment of documents*, Compression and Complexity of Sequences: Proceedings, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997.

2. Broder, Andrei Z.; Charikar, Moses; Frieze, Alan M.; Mitzenmacher, Michael (1998), *Min-wise independent permutations*, Proc. 30th ACM Symposium on Theory of Computing (STOC '98).

3. Gurmeet Singh, Manku; Das Sarma, Anish (2007), *Detecting near-duplicates for web crawling*, Proceedings of the 16th international conference on World Wide Web, ACM.

4. Indyk, P (1999). A small approximately min-wise independent family of hash functions. Proc. ACM SIAM SODA.

5. Thorup, M. (2013). `http://arxiv.org/abs/1303.5479`.

Chapter 5

Stable matchings, stable marriages and price of anarchy

Guest lecture by Mark Braverman. Handwritten scribe notes available from course website.
`http://www.cs.princeton.edu/courses/archive/fall14/cos521/`

Chapter 6

Linear Thinking

According to conventional wisdom, *linear thinking* describes thought process that is logical or step-by-step (i.e., each step must be completed before the next one is undertaken). *Nonlinear thinking*, on the other hand, is the opposite of linear: creative, original, capable of leaps of inference, etc.

From a complexity-theoretic viewpoint, conventional wisdom turns out to be startlingly right in this case: linear problems are generally computationally easy, and nonlinear problems are generally not.

EXAMPLE 3 Solving linear systems of equations is easy. Let's show solving quadratic systems of equations is NP-hard. Consider the VERTEX COVER problem, which is NP-hard: Given graph $G = (V, E)$ and an integer k we need to determine if there a subset of vertices S of size k such that for each edge $\{i, j\}$, at least one of i, j is in S.

We can rephrase this as a problem involving solving a system of nonlinear equations, where $x_i = 1$ stands for "*i is in the vertex cover.*"

$$\begin{aligned} (1 - x_i)(1 - x_j) &= 0 \quad \forall \{i, j\} \in E \\ x_i(1 - x_i) &= 0 \quad \forall i \in V. \sum_i x_i = k \end{aligned}$$

□

Not all nonlinear problems are difficult, but the ones that turn out to be easy are generally those that can leverage linear algebra (eigenvalues, singular value decomposition, etc.)

In mathematics too linear algebra is simple, and easy to understand. The goal of much of higher mathematics seems to be to reduce study of complicated (nonlinear!) objects to study of linear algebra.

6.1 Simplest example: Solving systems of linear equations

The following is a simple system of equations.

$$\begin{aligned} 2x_1 - 3x_2 &= 5 \\ 3x_1 + 4x_2 &= 6 \end{aligned}$$

More generally we represent a linear system of m equations in n variables as $Ax = b$ where A is an $m \times n$ coefficient matrix, x is a vector of n variables, and b is a vector of m real numbers. In your linear algebra course you learnt that this system is feasible iff b is in the span of the column vectors of A, namely, the rank of $A|b$ (i.e., the matrix where b is tacked on as a new column of A) has rank exactly the same as A. The solution is computed via matrix inversion. One subtlety not addressed in most linear algebra courses is whether this procedure is polynomial time. You may protest that actually they point out that the system can be solved in $O(n^3)$ operations. Yes, but this misses a crucial point which we will address before the end of the lecture.

6.2 Systems of linear inequalities and linear programming

If we replace some or all of the $=$ signs with $\geq$ or $\leq$ in a system of linear equations we obtain a system of linear inequalities.

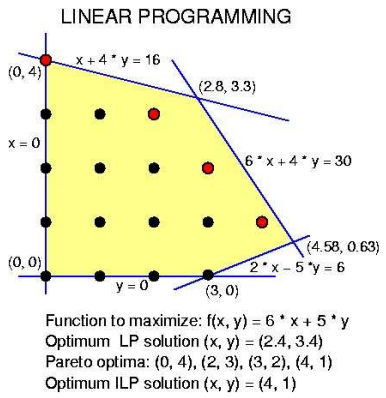

Figure 6.1: A system of linear inequalities and its feasible region

The feasible region has sharp corners; it is a convex region and is called a *polytope*. In general, a region of space is called *convex* if for every pair of points x, y in it, the line segment joining x, y, namely, $\{\lambda \cdot x + (1-\lambda) \cdot y : \lambda \in [0,1]\}$, lies in the region.

In *Linear Programming* one is trying to optimize (i.e., maximize or minimize) a linear function over the set of feasible values. The general form of an LP is

$$\min c^T x \tag{6.1}$$

$$Ax \geq b \tag{6.2}$$

Here $\geq$ denotes componentwise "greater than."

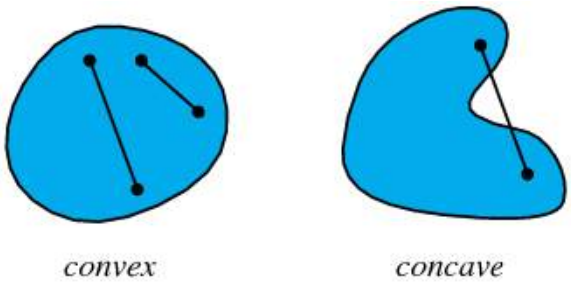

Figure 6.2: Convex and nonConvex regions

This form is very flexible. To express *maximization* instead of minimization, just replace c by $-c$. To include an inequality of the form $a \cdot x \leq b_i$ just write it as $-a \cdot x \geq -b_i$. To include an equation $a \cdot x = b_i$ as a constraint just replace with two inequalities $a \cdot x \geq b_i, a \cdot x \leq b_i$.

Solving LPs: In Figure 6.1 we see the convex feasible region of an LP. The objective function is linear, so it is clear that the optimum of the linear program is attained at some vertex of the feasible region. Thus a trivial algorithm to find the optimum is to enumerate all vertices of the feasible region and take the one with the lowest value of the objective. This method (sometimes taught in high schools) of graphing the inequalities and their feasible region does not scale well with n, m. The number of vertices of this feasible region grows roughly as $m^{n/2}$ in general. Thus the algorithm is exponential time. The famous *simplex* method is a clever method to enumerate these vertices one by one, ensuring that the objective keeps decreasing at each step. It works well in practice. The first polynomial-time method to determine feasibility of linear inequalities was only discovered in 1979 by Khachiyan, a Soviet mathematician. We will discuss the core ideas of this method later in the course. For now, we just assume polynomial-time solvability and see how to use LP as a tool.

Example 4 (Assignment Problem) Suppose n jobs have to be assigned to n factories. Each job has its attendant requirements and raw materials. Suppose all of these are captured by a single number: c_{ij} is the cost of assigning job i to factory j. Let x_{ij} be a variable that corresponds to assigning job i to factory j. We hope this variable is either 0 or 1 but that is not expressible in the LP so we *relax* this to the constraint

$$x_{ij} \geq 0 \quad \text{and} x_{ij} \leq 1 \quad \text{for each } i, j.$$

Each job must be assigned to exactly one factory so we have the constraint $\sum_j x_{ij} = 1$ for each job i. Then we must ensure each factory obtains one job, so we include the constraint $\sum_i x_{ij} = 1$ for each factory j. Finally, we want to minimize overall cost so the objective is

$$\min \sum_{ij} c_{ij} x_{ij}.$$

Fact: the solution to this LP has the property that all x_{ij} variables are either 0 or 1. (Maybe this will be a future homework.) Thus solving the LP actually solves the assignment problem.

In general one doesn't get so lucky: solutions to LPs end up being nonintegral no matter how hard we pray for the opposite outcome. Next lecture we will discuss what to do if that happens.□

In fact linear programming was invented in 1939 by Kontorovich, a Russian mathematician, to enable efficient organization of industrial production and other societal processes (such as the assignment problem). The premise of communist economic system in the 1940s and 1950s was that centralized planning —using linear programming!— would enable optimum use of a society's resources and help avoid the messy "inefficiencies"of the market system! The early developers of linear programming were awarded the Nobel prize in economics! Alas, linear programming has not proved sufficient to ensure a perfect economic system. Nevertheless it is extremely useful and popular in optimizing flight schedules, trucking operations, traffic control, manufacturing methods, etc. At one point it was estimated that 50% of all computation in the world was devoted to LP solving. Then youtube was invented...

6.3 Linear modeling

At the heart of mathematical modeling is the notion of a *system* of variables: some variables are mathematically expressed in terms of others. In general this mathematical expression may not be succinct or even finite (think of the infinite processes captured in the quantum theory of elementary particles). A *linear* model is a simple way to express interrelationships that are linear.

$$y = 0.1x_1 + 9.1x_2 - 3.2x_3 + 7.$$

Example 5 (Diet) You wish to balance meat, sugar, veggies, and grains in your diet. You have a certain dollar budget and a certain calorie goal. You don't like these foodstuffs equally; you can give them a score between 1 and 10 according to how much you like them. Let l_m, l_s, l_v, l_g denote your score for meat, sugar, veggies and grains respectively. Assuming your overall happiness is given by

$$m \times l_m + g \times l_g + v \times l_v + s \times l_s,$$

where m, g, v, s denote your consumption of meat, grain, veggies and sugar respectively (note: this is a modeling assumption about you) then the problem of maximizing your happiness subject to a dollar and calorie budget is a linear program. □

Example 6 (ℓ_1 regression) This example is from Bob Vanderbei's book on linear programming. You are given data containing grades in different courses for various students; say G_{ij} is the grade of student i in course j. (Of course, G_{ij} is not defined for all i, j since each student has only taken a few courses.) You can try to come up with a model for explaining these scores. You hypothesize that a student's grade in a course is determined

by the student's innate aptitude, and the difficulty of the course. One could try various functional forms for how the grade is determined by these factors, but the simplest form to try is linear. Of course, such a simple relationship will not completely explain the data so you must allow for some error. This linear model hypothesizes that

$$G_{ij} = \text{aptitude}_i + \text{easiness}_j + \epsilon_{ij}, \tag{6.3}$$

where ϵ_{ij} is an error term.

Clearly, the error could be positive or negative. A good model is one that has a low value of $\sum_{ij} |\epsilon_{ij}|$. Thus the best model is one that minimizes this quantity.

We can solve this model for the aptitude and easiness scores using an LP. We have the constraints in (6.3) for each student i and course j. Then for each i, j we have the constraints

$$s_{ij} \geq 0 \quad \text{and} - s_{ij} \leq \epsilon_{ij} \leq s_{ij}.$$

Finally, the objective is $\min \sum_{ij} s_{ij}$.

This method of minimizing the sum of absolute values is called ℓ_1-*regression* because the ℓ_1 norm of a vector x is $\sum_i |x_i|$. □

Just as LP is the tool of choice to squeeze out inefficiencies of production and planning, linear modeling is the bedrock of data analysis in science and even social science.

Example 7 (Econometric modeling) Econometrics is the branch of economics dealing with analysis of empirical data and understanding the interrelationships of the underlying economic variables —also useful in sociology, political science etc.. It often relies upon modeling dependencies among variables using linear expressions. Usually the variables have a time dependency. For instance it may a posit a relationship of the form

$$\text{Growth}(T+1) = \alpha \cdot \text{Interest rate}(T) + \beta \cdot \text{Deficit}(T-1) + \epsilon(T),$$

where Interest rate(T) denotes say the interest rate at time T, etc. Here α, β may not be constant and may be *probabilistic variables* (e.g., a random variable uniformly distributed in $[0.5, 0.8]$) since future growth may not be a deterministic function of the current variables.

Often these models are solved (i.e., for α, β in this case) by regression methods related to the previous example, or more complicated probabilistic inference methods that we may study later in the course.□

Example 8 (Perceptrons and Support Vector Machines in machine learning) Suppose you have a bunch of images labeled by whether or not they contain a car. These are data points of the form (x, y) where x is n-dimensional (n= number of pixels in the image)and $y_i \in \{0, 1\}$ where 1 denotes that it contains a car. You are trying to train an algorithm to recognize cars in other unlabeled images. There is a general method called SVM's that allows you to find some kind of a linear model. (Aside: such simple linear models don't work for finding cars in images; this is an example.) This involves hypothesizing that there is an unknown set of coefficients $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n$ such that

$$\sum_i \alpha_i x_i \geq \alpha_0 + \text{error}_x \quad \text{if } x \text{ is an image containing a car,}$$

$$\sum_i \alpha_i x_i \leq 0.5\alpha_0 + \text{error}_x \quad \text{if } x \text{ does not contain a car,}$$

where error_x is required to be nonpositive for each x. Then finding such α_i's while minimizing the sum of the absolute values of the error terms is a linear program. After finding these α_i's, given a new image the program tries to predict whether it has a car by just checking whether $\sum_i \alpha_i x_i \geq \alpha_0$ or $\leq 0.5\alpha_0$. (There is nothing magical about the 0.5 gap here; one usually stipulates a gap or *margin* between the yes and no cases.)

This technique is related to the so-called support vector machines in machine learning (and an older model called perceptrons), though we're dropping a few technical details (ℓ_2-regression, regularization etc.). Also, in practice it could be that the *linear* explanation is a good fit only after you first apply a nonlinear transformation on the x's. This is the idea in kernel SVMs. For instance let z be the vector where the ith coordinate $z_i = \phi(x_i) = \exp(-\frac{x_i^2}{2})$. You then find a linear predictor using the z's. (How to choose such nonlinear transformations is an art.) □

One reason for the popularity of linear models is that the mathematics is simple, elegant, and most importantly, efficient. Thus if the number of variables is large, a linear model is easiest to solve.

A theoretical justification for linear modeling is *Taylor expansion*, according to which every "well-behaved"function is expressible as an infinite series of terms involving the derivatives. Here is the taylor series for an m-variate function f:

$$f(x_1, x_2, \ldots, x_m) = f(0, 0, .., 0) + \sum_i x_i \frac{\partial f}{\partial x_i}(0) + \sum_{i_1 i_2} x_{i_1} x_{i_2} \frac{\partial f}{\partial x_{i_1} \partial x_{i_2}}(0) + \cdots .$$

If we assume the higher order terms are negligible, we obtain a linear expression.

Whenever you see an article in the newspaper describing certain quantitative relationships —eg, the effect of more policing on crime, or the effect of certain economic policy on interest rates—chances are it has probably been obtained via a linear model and ℓ_1 regression (or the related ℓ_2 regression). So don't put blind faith in those numbers; they are necessarily rough approximations to the complex behavior of a complex world.

6.4 Meaning of polynomial-time

Of course, the goal in this course is designing polynomial-time algorithms. When a problem definition involves numbers, the correct definition of polynomial-time is "polynomial in the number of bits needed to represent the input. "

Thus the input size of an $m \times n$ system $Ax = b$ is not mn but the number of bits used to represent A, b, which is at most mnL where L denotes the number of bits used to represent each entry of A, b. We assume that the numbers in A, b are rational, and in fact by clearing denominators we may assume wlog they are integer.

Let's return to the question we raised earlier: *is Gaussian elimination a polynomial-time procedure?* The answer is yes. The reason this is nontrivial is that conceivably during Gaussian elimination we may produce a number that is too large to represent. We have to show it runs in poly(m, n, L) time.

Towards this end, first note that standard arithmetic operations $+, -, \times$ run in time polynomial in the input size (e.g., multiplying two k-bit integers takes time at most $O(k^2)$ even using the gradeschool algorithm).

Next, note that by Cramer's rule for solving linear systems, the numbers produced during the algorithm are related to the determinant of $n \times n$ submatrices of A. For example if A is invertible then the solution to $Ax = b$ is $x = A^{-1}b$, and the i, j entry of A^{-1} is $C_{ij}/\det(A)$, where C_{ij} is a cofactor, i.e. an $(n-1) \times (n-1)$ submatrix of A. The determinant of an $n \times n$ matrix whose entries are L bit integers is at most $n!2^{Ln}$. This follows from the formula for determinant of an $n \times n$ matrix, which is

$$\det(A) = \sum_{\sigma} \operatorname{sgn}(\sigma) \prod_{i} A_{i\sigma(i)},$$

where σ ranges over all permutations of n elements.

The number of bits used to represent determinant is the log of this, which is $n \log n + Ln$, which is indeed polynomial. Thus doing arithmetic operations on these numbers is also polynomial-time.

The above calculation has some consequence for linear programming as well. Recall that the optimum of a linear program is attained at a vertex of the polytope. The vertex is defined as the solution of all the equations obtained from the inequalities that are tight there. We conclude that each vertex of the polytope can be represented by $n \log n + Ln$ bits. This at least shows that the solution can be *written down* in polynomial time (a necessary precondition for being able to compute it in polynomial time!).

Chapter 7

Provable Approximation via Linear Programming

One of the running themes in this course is the notion of *approximate solutions*. Of course, this notion is tossed around a lot in applied work: whenever the exact solution seems hard to achieve, you do your best and call the resulting solution an approximation. In theoretical work, approximation has a more precise meaning whereby you *prove* that the computed solution is close to the exact or optimum solution in some precise metric. We saw some earlier examples of approximation in sampling-based algorithms; for instance our hashing-based estimator for set size. It produces an answer that is whp within $(1+\epsilon)$ of the true answer. Today we will see many other examples that rely upon linear programming (LP).

Recall that most NP-hard optimization problems involve finding 0/1 solutions. Using LP one can find *fractional* solutions, where the relevant variables are constrained to take real values in $[0, 1]$.

Recall the example of the assignment problem from last time, which is also a 0/1 problem (a job is either assigned to a particular factory or it is not) but the LP relaxation magically produces a 0/1 solution (although we didn't prove this in class). Whenever the LP produces a solution in which all variables are 0/1, then this must be the optimum 0/1 solution as well since it is the best *fractional* solution, and the class of fractional solutions contains every 0/1 solution. Thus the assignment problem is solvable in polynomial time.

Needless to say, we don't expect this magic to repeat for NP-hard problems. So the LP relaxation yields a fractional solution in general. Then we give a way to *round* the fractional solutions to 0/1 solutions. This is accompanied by a mathematical proof that the new solution is provably approximate.

7.1 Deterministic Rounding (Weighted Vertex Cover)

First we give an example of the most trivial rounding of fractional solutions to 0/1 solutions: round variables $< 1/2$ to 0 and $\geq 1/2$ to 1. Surprisingly, this is good enough in some settings.

In the *weighted vertex cover* problem, which is NP-hard, we are given a graph $G = (V, E)$ and a weight for each node; the nonnegative weight of node i is w_i. The goal is to find a *vertex cover*, which is a subset S of vertices such that every edge contains at least one vertex

of S. Furthermore, we wish to find such a subset of minimum total weight. Let $VC_{\min}$ be this minimum weight. The following is the LP relaxation:

$$\begin{array}{lll} \min & \sum_i w_i x_i & \\ & 0 \leq x_i \leq 1 & \forall i \\ & x_i + x_j \geq 1 & \forall \{i,j\} \in E. \end{array}$$

Let OPT_f be the optimum value of this LP. It is no more than $VC_{\min}$ since every 0/1 solution (including in particular the 0/1 solution of minimum cost) is also an acceptable fractional solution.

Applying *deterministic* rounding, we can produce a new set S: every node i with $x_i \geq 1/2$ is placed in S and every other i is left out of S.

Claim 1: S is a vertex cover.

Reason: For every edge $\{i,j\}$ we know $x_i + x_j \geq 1$, and thus at least one of the x_i's is at least $1/2$. Hence at least one of i, j must be in S.

Claim 2: The weight of S is at most $2OPT_f$.

Reason: $OPT_f = \sum_i w_i x_i$, and we are only picking those i's for which $x_i \geq 1/2$. $\square$.

Thus we have constructed a vertex cover whose cost is within a factor 2 of the optimum cost *even though we don't know the optimum cost per se.*

Exercise: Show that for the complete graph the above method indeed computes a set of size no better than 2 times OPT_f.

Remark: This 2-approximation was discovered a long time ago, and despite myriad attempts we still don't know if it can be improved. Using the so-called PCP Theorems Dinur and Safra showed (improving a long line of work) that 1.36-approximation is NP-hard. Khot and Regev showed that computing a $(2-\epsilon)$-approximation is UG-hard, which is a new form of hardness popularized in recent years. The bibliography mentions a popular article on UG-hardness.

7.2 Simple randomized rounding: MAX-2SAT

Simple randomized rounding is as follows: if a variable x_i is a fraction then toss a coin which comes up heads with probability x_i. (In Homework 1 you figured out how to do this given a binary representation of x_i.) If the coin comes up heads, make the variable 1 and otherwise let it be 0. The expectation of this new variable is exactly x_i. Furthermore, linearity of expectations implies that if the fractional solution satisfied some linear constraint $c^T x = d$ then the new variable vector satisfies the same constraint *in the expectation.* But in the analysis that follows we will in fact do something more.

A 2CNF formula consists of n boolean variables $x_1, x_2, \ldots, x_n$ and *clauses* of the type $y \vee z$ where each of y, z is a *literal,* i.e., either a variable or its negation. The goal in MAX2SAT is to find an assignment that *maximises* the number of satisfied clauses. (Aside: If we wish to satisfy all the clauses, then in polynomial time we can check if such an assignment exists. Surprisingly, the maximization version is NP-hard.) The following is the LP relaxation where J is the set of clauses and y_{j1}, y_{j2} are the two literals in clause j. We have a variable z_j for each clause j, where the intended meaning is that it is 1 if the assignment decides to satisfy that clause and 0 otherwise. (Of course the LP can choose to give z_j a fractional value.)

$$\begin{aligned} \min \quad & \textstyle\sum_{j \in J} z_j \\ & 1 \geq x_i \geq 0 \quad \forall i \\ & y_{j1} + y_{j2} \geq z_j \end{aligned}$$

Where y_{j1} is shorthand for x_i if the first literal in the jth clause is the ith variable, and shorthand for $1 - x_i$ if the literal is the negation of the i variable. (Similarly for y_{j2}.)

If MAX-2SAT denotes the number of clauses satisfied by the best assignment, then it is no more than OPT_f, the value of the above LP. Let us apply randomized rounding to the fractional solution to get a 0/1 assignment. How good is it?

Claim: **E**[number of clauses satisfied] $\geq \frac{3}{4} \times OPT_f$.

We show that the probability that the jth clause is satisfied is at least $3z_j/4$ and then the claim follows by linear of expectation.

If the clause is of size 1, say x_r, then the probability it gets satisfied is x_r, which is at least z_j. Since the LP contains the constraint $x_r \geq z_j$, the probability is certainly at least $3z_j/4$.

Suppose the clauses is $x_r \vee x_s$. Then $z_j \leq x_r + x_s$ and in fact it is easy to see that $z_j = \min\{1, x_r + x_s\}$ at the optimum solution: after all, why would the LP not make z_j as large as allowed; its goal is to maximize $\sum_j z_j$. The probability that randomized rounding satisfies this clause is exactly $1 - (1 - x_r)(1 - x_s) = x_r + x_s - x_r x_s$.

But $x_r x_s \leq \frac{1}{4}(x_r + x_s)^2$ (prove this!) so we conclude that the probability that clause j is satisfied is at least $z_j - z_j^2/4 \geq 3z_j/4$. $\square$.

Remark: This algorithm is due to Goemans-Williamson, but the original 3/4-approximation is due to Yannakakis. The 3/4 factor has been improved by other methods to 0.94.

7.3 Dependent randomized rounding: Virtual circuit routing

Often a simple randomized rounding produces a solution that makes no sense. Then one must resort to a more dependent form of rounding whereby chunks of variables may be rounded up or down in a correlated way. Now we see an example of this from a classic paper of Raghavan and Tompson.

In networks that break up messages into packets, a *virtual circuit* is sometimes used to provide *quality of service* guarantees between endpoints. A fixed path is identified and reserved between the two desired endpoints, and all messages are sped over that fixed path with minimum processing overhead.

Given the capacity of all network edges, and a set of endpoint pairs $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$ it is NP-hard to determine if there is a set of paths which provide a unit capacity link between each of these pairs and which together fit into the capacity constraints of the network.

Now we give an approximation algorithm where we assume that (a) a unit-capacity path is desired between each given endpoint pair (b) the total capacity c_{uv} of each edge is at least $d \log n$, where d is a sufficiently large constant.

We give a somewhat funny approximation. Assuming there exists an integral solution that connects all k endpoint pairs and which uses at most 0.9 fraction of each edge's capacity, we give an integral solution that connects at least $(1 - 1/e)$ fraction of the endpoints pairs and does not exceed any edge's capacity.

The idea is to write an LP. For each endpoint pair i, j that have to be connected and each edge $e = (u, v)$ we have a variable $x_{uv}^{i,j}$ that is supposed to be 1 if the path from i to j passes through (u, v), and 0 otherwise. (Note that edges are directed.) Then for each edge (u, v) we can add a capacity constraint

$$\sum_{i,j:\text{endpoints}} x_{uv}^{i,j} \leq c_{uv}.$$

But since we can't require variables to be 0/1 in an LP, we relax to $0 \leq x_{uv}^{i,j} \leq 1$. This allows a path to be split over many paths (this will remind you of network flow if you have seen it in undergrad courses). Of course, this seems all wrong since avoiding such splitting was the whole point in the problem! Be patient just a bit more.

Furthermore we need the so-called *flow conservation* constraints. These say that the fractional amount of paths leaving i and arriving at j is 1, and that paths never get stranded in between.

$$\begin{array}{ll} \sum_v x_{uv}^{ij} = \sum_v x_{vu}^{ij} & \forall u \neq i, j \\ \sum_v x_{uv}^{ij} - \sum_v x_{vu}^{ij} = 1 & u = i \\ \sum_v x_{vu}^{ij} - \sum_v x_{uv}^{ij} = 1 & u = j \end{array}$$

Under our hypothesis about the problem, this LP is feasible and we get a fractional solution $\left\{x_{uv}^{i,j}\right\}$. These values can be seen as bits and pieces of paths lying strewn about the network.

Let us first see that neither deterministic rounding nor simple randomized rounding is a good idea. Consider a node u where $x_u^{ij} v$ is 1/3 on three incoming edges and 1/2 on two outgoing edges. Then deterministic rounding would round the incoming edges to 1 and outgoing edges to 1, creating a bad situation where the path never enters u but leaves it on two edges! Simple randomized rounding will also create a similar bad situation with $\Omega(1)$ (i.e., constant) probability. Clearly, it would be much better to round along entire paths instead of piecemeal.

Flow decomposition: For each endpoint pair i, j we create a finite set of paths $p_1, p_2, \ldots,$ from i to j as well as associated weights $w_{p_1}, w_{p_2}, \ldots,$ that lie in $[0, 1]$ and sum up to 1. Furthermore, for each edge (u, v): $x_{u,v}^{i,j}$ = sum of weights of all paths among these that contain u, v.

Flow decomposition is easily accomplished via *depth first search*. Just repeatedly find a path from i to j in the weighted graph defined by the x_{uv}^{ij}'s: the flow conservation constraints imply that this path can leave every vertex it arrives at except possibly at j. After you find such a path from i to j subtract from all edges on it the minimum x_{uv}^{ij} value along this path. This ensures that at least one x_{uv}^{ij} gets zeroed out at every step, so the process is finite.

Randomized rounding: For each endpoint pair i, j pick a path from the above decomposition randomly by picking it with probability proportional to its weight.

Part 1: We show that this satisfies the edge capacities approximately.

This follows from Chernoff bounds. The expected number of paths that use an edge $\{u, v\}$ is

$$\sum_{i,j:\text{endpoints}} x_{u,v}^{i,j}.$$

The LP constraint says this is at most c_{uv}, and since $c_{uv} > d \log n$ this is a sum of at least $d \log n$ random variables. Chernoff bounds (see our earlier lecture) imply that this is at most $(1+\epsilon)$ times its expectation for all edges with high probability. Chernoff bounds similarly imply that the overall number of paths is pretty close to k.)

Part 2: We show that in the expectation, $(1 - 1/e)$ fraction of endpoints get connected by paths. Consider any endpoint pair. Suppose they are connected by t fractional paths $p_1, p_2, ..$ with weights $w_1, w_2..$ etc. Then $\sum_i w_i = 1$ since the endpoints were fractionally connected. The probability that the randomized rounding will round all these paths down to 0 is

$$\begin{aligned} \prod_i (1 - w_i) &\leq (\tfrac{\sum_i (1-w_i)}{t})^t \qquad \text{(Geometric mean} \leq \text{Arithmetic mean)} \\ &\leq (1 - 1/t)^t \quad \leq 1/e. \end{aligned}$$

The downside of this rounding is that some of the endpoint pairs may end up with zero paths, whereas others may end up with 2 or more. We can of course discard extra paths. (There are better variations of this approximation but covering them is beyond the scope of this lecture.)

Remark: We have only computed the expectation here, but one can check using Markov's inequality that the algorithm gets arbitrarily close to this expectation with probability at least $1/n$ (say).

Bibliography

1. *New* 3/4-*approximation to MAX-SAT* by M. X. Goemans and D. P. Williamson, SIAM J. Discrete Math 656-666, 1994.

2. *Randomized rounding: A technique for provably good algorithms and algorithmic proofs* by P. Raghavan and C. T. Tompson, Combinatorica pp 365-374 1987.

3. *On the hardness of approximating minimum vertex cover* by I. Dinur and S. Safra, Annals of Math, pp 439485, 2005.

4. *Approximately hard: the Unique Games Conjecture.* by E. Klarreich. Popular article on `https://www.simonsfoundation.org/`

Chapter 8

Decision-making under uncertainty: Part 1

This lecture is an introduction to *decision theory*, which gives tools for making *rational* choices in face of *uncertainty*. It is useful in all kinds of disciplines from electrical engineering to economics. In computer science, a compelling setting to consider is an autonomous vehicle or robot navigating in a new environment. It may have some prior notions about the environment but inevitably it encounters many different situations and must respond to them. The actions it chooses (drive over the object on the road or drive around it?) *changes* the set of future events it will see, and thus its choice of the immediate action must necessarily take into account the continuing effects of that choice *far into the future*. You can immediately see that the same issues arise in any kind of decision-making in real life: save your money in stocks or bonds; go to grad school or get a job; marry the person you are dating now, or wait a few more years?

Of course, italicized terms in the previous paragraph are all very loaded. What is a rational choice? What is "uncertainty"? In everyday life uncertainty can be interpreted in many ways: risk, ignorance, probability, etc.

Decision theory suggests some answers —perhaps simplistic, but a good start. The first element of this theory is its *probabilistic* interpretation of uncertainty: there is a probability distribution on future events that the decision maker is assumed to know. The second element is quantifying "rational choice." It is assumeed that each outcome has some *utility* to the decisionmaker, which is a number. The decision-making is said to be *rational* if it maximises the *expected* utility.

EXAMPLE 9 Say your utility involves job satisfaction quantified in some way. If you decide to go for a PhD the distribution of your utility is given by random variable X_0. If you decide to take a job instead, your return is a random variable X_1. Decision theory assumes that you (i.e.,the decision-maker) know and understand these two random variables. You choose to get a PhD if $\mathbf{E}[X_0] > \mathbf{E}[X_1]$.

EXAMPLE 10 17th century mathematician Blaise Pascal's famous *wager* is an early example of an argument recognizable as modern decision theory. He tried to argue that it is the rational choice for humans to believe in God (he meant Christian god, of course). If you

choose to be a disbeliever and sin all your life, you may have infinite loss if God exists (eternal damnation). If you choose to believe and live your life in virtue, and God doesn't exist it is all for naught. Therefore if you think that the probability that God exists is nonzero, you must choose to live as a believer to avoid an infinite expected loss. (Aside: how convincing is this argument to you?) □

We will not go into a precise definition of utility (wikipedia moment) but illustrate it with an example. You can think of it as a quantification of "satisfaction ". In computer science we also use *payoff*, *reward* etc.

EXAMPLE 11 (Meaning of utility) You have bought a cake. On any single day if you eat x percent of the cake your utility is $\sqrt{x}$. (This happiness is sublinear because the 5th bite of the cake brings less happiness than the first.) The cake reaches its expiration date in 5 days and if any is still left at that point you might as well finish it (since there is no payoff from throwing away cake).

What schedule of cake eating will maximise your total utility over 5 days? Your optimal choice is to eat 20% of the cake each day, since it yields a payoff of $5 \times \sqrt{20}$, which is a lot more than any of the alternatives. For instance, eating it all on day 1 would produce a much lower payoff $\sqrt{5 \times 20}$.

This example is related to Modigliani's *Life cycle hypothesis,* which suggests that consumers consume wealth in a way that evens out consumption over their lifetime. (For instance, it is rational to take a loan early in life to get an education or buy a house, because it lets you enjoy a certain quality of life, and pay for it later in life when your earnings are higher.)

In our class discussion some of you were unconvinced about the axiom about maximising expected utility. (And the existence of lotteries in real life suggests you are on to something.) Others objected that one doesn't truly know —at least very precisely—the distribution of outcomes, as in the PhD vs job example. Very true. (The financial crash of 2008 relates to some of this, but that's a story for another day.) It is important to understand the limitations of this powerful theory.

8.1 Decision-making as dynamic programming

Often you can think of decision-making under uncertainty as playing a game against a random opponent, and the optimum policy can be computed via dynamic programming.

EXAMPLE 12 (Cake eating revisited) Let's now complicate the cake-eating problem. In addition to the expiration date, your decision must contend with actions of your housemates, who tend to eat small amounts of cake when you are not looking. On each day with probability 1/2 they eat 10% of the cake.

Assume that each day the amount you eat as a percentage of the original is a multiple of 10. You have to compute the cake eating schedule that maximises your expected utility.

Now you can draw a tree of depth 5 that describes all possible outcomes. (For instance the first level consists of a 11-way choice between eating $0\%, 10\%, \ldots, 100\%$.) Computing your optimum cake-eating schedule is a simple dynamic programming over this tree. □

The above cake-eating examples can be seen as a metaphor for all kinds of decision-making in life: e.g., how should you spend/save throughout your life to maximize overall happiness[1]?

Decision choice theory says that all such decisions can be made by an appropriate dynamic programming over some tree. Say you think of time as discrete and you have a finite choice of actions at each step: say, two actions labeled 0 and 1. In response the environment responds with a coin toss. (In cake-eating if the coin comes up heads, 10% of the cake disappears.) Then you receive some payoff/utility, which is a real number, and depends upon the sequence of T moves made so far. If this goes on for T steps, we can represent this entire game as a tree of depth T.

Then the best decision at each step involves a simple dynamic programming where the operation at each action node is *max* and the operation at each probabilistic node is *average*. If the node is a leaf it just returns its value. Note that this takes time *exponential* [2] in T. Interestingly, dynamic programming was invented by R. Bellman in this decision-theory context. (If you ever wondered what the "dynamic"in dynamic programming refers to, well now you know. Check out wikipedia for the full story.) The dynamic programming is also related to the game-theoretic notion of *backwards induction*.

The cake example had a finite horizon of 5 days and often such a finite horizon is *imposed* on the problem to make it tractable.

But one can consider a process that goes on for ever and still make it tractable using *discounted* payoffs. The payoff is being accumulated at every step, but the decision-maker discounts the value of payoffs at time t as γ^t where γ is the discount factor. This notion is based upon the observation that most people, given a choice between getting 10 dollars now versus 11 a year from now, will choose the former. This means that they discount payoffs made a year from now by 10/11 at least.

Since $\gamma^t \to 0$ as t gets large, discounting ensures that payoffs obtained a large time from now are perceived as almost zero. Thus it is a "soft "way to impose a finite horizon.

Aside: Children tend to be fairly shortsighted in their decisions, and don't understand the importance of postponement of gratification. Is growing up a process of adjusting your γ to a higher value? There is evidence that people are born with different values of γ, and this is known to correlate with material success later in life. (See the wikipedia page on the Stanford marshmallow experiment.)

8.2 Markov Decision Processes (MDPs)

This is the version of decision-making most popular in AI and robotics, and is used in autonomous vehicles, drones etc. (Of course, the difficult "engineering"part is figuring out the correct MDP description.) The literature on this topic is also vast.

The MDP framework is a way to succinctly represent the decision-maker's interaction with the environment. The decision-maker has a finite number of *states* and a finite number

[1]Several Nobel prizes were awarded for figuring out the implications of this theory for explaining economic behavior, and even phenomena like marriage/divorce.

[2]In fact in a reasonable model where each node of the tree can be computed in time polynomial in the description of the node, Papadimitriou showed that the problem of computing the optimum policy is PSPACE-complete, and hence exp(T) time is unavoidable.

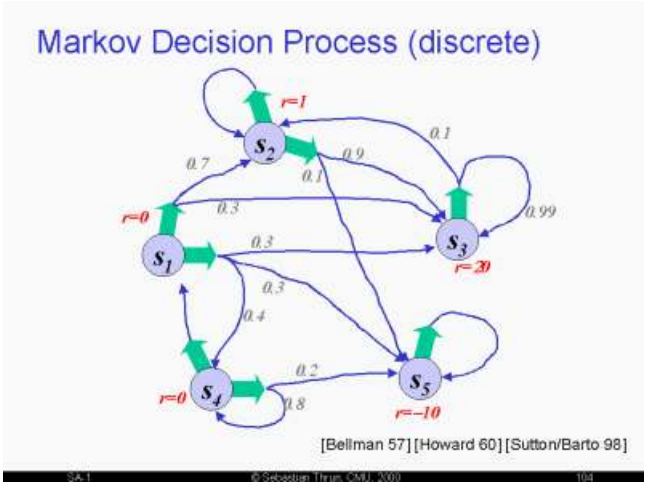

Figure 8.1: An MDP (from S. Thrun's notes)

of actions it is allowed to take in each state. (For example, a state for an autonomous vehicle could be defined using a finite set of variables: its speed, what lane it is in, whether or not there is a vehicle in front/back/left/right, whether or not one of them is getting closer at a fast rate.) Upon taking an action the decision-maker gets a *reward* and then "nature"or "chance"transitions him probabilistically to another state. The optimal policy is defined as one that maximises the total reward (or discounted reward).

For simplicity assume the set of states is labeled by integers $1, \ldots, n$, the possible actions in each state are 0/1. For each action b there is a probability $p(i, b, j)$ of transitioning to state j if this action is taken in that state. Such a transition brings an immediate reward of $R(i, b, j)$. Note that this process goes forever; the decision-maker keeps taking actions, which affect the sequence of states it passes through and the rewards it gets.
The name Markov: This refers to the *memoryless* aspect of the above setup: the reward and transition probabilities do not depend upon the past history.

EXAMPLE 13 If the decision-maker always takes action 0 and $s_1, s_2, \ldots,$ are the random variables denoting the states it passes through, then its total reward is

$$\sum_{t=1}^{\infty} R(s_t, 0, s_{t+1}).$$

Furthermore, the distribution of s_t is completely determined (as described above) given s_{t-1} (i.e., we don't need to know the earlier sequence of states that were visited).

This sum of rewards is typically going to be infinite, so if we use a discount factor γ then the discounted reward of the above sequence is

$$\sum_{t=1}^{\infty} \gamma^t R(s_t, 0, s_{t+1}).$$

□

8.3 Optimal MDP policies via LP

A *policy* is a strategy for the decision-maker to choose its actions in the MDP. You can think of it as the *driver* of the hardware whose workings are described by the MDP. One idea — based upon the discussion above—is to let the policy be dictated by a dynamic programming that is limited to lookahead T steps ahead. But this is computationally difficult for even moderate T. Ideally we would want a simple precomputed answer.

The problem with precomputed answers is that in general the optimal action in a state at a particular time could depend upon the precise sequence of states traversed in the past. Dynamic programming allows this possibility.

We are interested in *history-independent* policies: each time the decision-maker enters the state it takes the same action. This is computationally trivial to implement in real-time. The above example contained a very simple history-independent policy: always take the action 0. In general such a policy is a mapping $\pi : \{1, \ldots, n\} \to \{0, 1\}$. So there are 2^n possible policies. Are they any good?

For each fixed policy the MDP turns into a simple (but infinite) random walk on states, where the probability of transitioning from i to j is $p(i, \pi(i), j)$. To talk sensibly about an optimum policy one has to make the total reward finite, so we assume a discount factor $\gamma < 1$. Then the expression for reward is

$$\sum_{i=1}^{\infty} \gamma^t (\text{reward at time } t).$$

Clearly this converges. Under some technical condition it can be shown that the optimum policy is history-independent[3]

To compute the rewards from the optimum policy one ignores *transient* effects as the random walk settles down, and look at the final steady state. This computation can be done via linear programming.

Let V_i be the expected reward of following the optimum policy if one starts in state i. In the first step the policy takes action $\pi(i) \in \{0, 1\}$, and transitiitions to another state j. Then the subpolicy that kicks in after this transition must also be optimal too, though its contribution is attenuated by γ. So V_i must satisfy

$$V_i = \sum_{j=1}^{n} p(i, \pi(i), j)(R(i, \pi(i), j) + \gamma V_j). \tag{8.1}$$

Thus if the allowed actions are $0, 1$ the optimum policy must satisfy:

$$V_i \geq \sum_{j=1}^{n} p(i, 0, j)(R(i, 0, j) + \gamma V_j),$$

and

$$V_i \geq \sum_{j=1}^{n} p(i, 1, j)(R(i, 1, j) + \gamma V_j).$$

[3]This condition has to do with the *Ergodicity* of the MDP. For each fixing of the policy the MDP turns into a simple random walk on the state space. One needs this to converge to a stationary distribution whereby each state i appears during the walk some p_i fraction of times.

The objective is to minimize $\sum_i V_i$ subject to the above constraints. (Note that the constraints for other states will have V_i on the other side of the inequality, which will constrain it also.) So the LP is really solving for

$$V_i = \max_{b \in \{0,1\}} \sum_{j=1}^{n} p(i,b,j)(R(i,b,j) + \gamma V_j).$$

After solving the LP one has to look at which of the above two inequalities involving V_i is tight to figure out whether the optimum action $\pi(i)$ is 0 or 1.

In practice solving via LP is considered too slow (since the number of states could be 100,000 or more) and iterative methods are used instead. We'll see some iterative methods later in the course in other contexts.

Bibliography

1. C. Papadimitriou, *Games against Nature.* JCSS 31, 288-301 (1985)

Chapter 9

Decision-making under total uncertainty: the multiplicative weight algorithm

(*Today's notes below are largely lifted with minor modifications from a survey by Arora, Hazan, Kale in Theory of Computing journal, Volume 8 (2012), pp. 121-164.*)

Today we study decision-making under total uncertainty: there is no a priori distribution on the set of possible outcomes. (This line will cause heads to explode among devout Bayesians, but it makes sense in many computer science settings. One reason is computational complexity or general lack of resources: the decision-maker usually lacks the computational power to construct the tree of all $\exp(T)$ outcomes possible in the next T steps, and the resources to do enough samples/polls/surveys to figure out their distribution. Or the algorithm designer may not be a Bayesian.)

Such decision-making (usually done with efficient algorithms) is studied in the field of *online computation*, which takes the view that the algorithm is responding to a sequence of requests that arrive one by one. The algorithm must take an action as each request arrives, and it may discover later, after seeing more requests, that its past actions were suboptimal. But past actions cannot be unchanged.

See the book by Borodin and El-Yaniv for a fuller introduction to online algorithms. This lecture and the next covers one such success story: an online optimization tool called the multiplicative weight update method. The power of the method arises from the very minimalistic assumptions, which allow it to be plugged into various settings (as we will do in next lecture).

9.1 Motivating example: weighted majority algorithm

Now we briefly illustrate the general idea in a simple and concrete setting. This is known as the *Prediction from Expert Advice* problem.

Imagine the process of picking good times to invest in a stock. For simplicity, assume that there is a single stock of interest, and its daily price movement is modeled as a sequence

of binary events: up/down. (Below, this will be generalized to allow non-binary events.) Each morning we try to predict whether the price will go up or down that day; if our prediction happens to be wrong we lose a dollar that day, and if it's correct, we lose nothing.

The stock movements can be *arbitrary* and even *adversarial*[1]. To balance out this pessimistic assumption, we assume that while making our predictions, we are allowed to watch the predictions of n "experts". These experts could be arbitrarily correlated, and they may or may not know what they are talking about. The algorithm's goal is to limit its cumulative losses (i.e., bad predictions) to roughly the same as the *best* of these experts. At first sight this seems an impossible goal, since it is not known until the end of the sequence who the best expert was, whereas the algorithm is required to make predictions all along.

For example, the first algorithm one thinks of is to compute each day's up/down prediction by going with the majority opinion among the experts that day. But this algorithm doesn't work because a majority of experts may be consistently wrong on every single day, while some single expert in this crowd happens to be right every time.

The *weighted majority algorithm* corrects the trivial algorithm. It maintains a *weighting* of the experts. Initially all have equal weight. As time goes on, some experts are seen as making better predictions than others, and the algorithm increases their weight proportionately. The algorithm's prediction of up/down for each day is computed by going with the opinion of the weighted majority of the experts for that day.

Weighted majority algorithm

Initialization: Fix an $\eta \leq \frac{1}{2}$. For each expert i, associate the weight ${w_i}^{(1)} := 1$.
For $t = 1, 2, \ldots, T$:

1. Make the prediction that is the weighted majority of the experts' predictions based on the weights ${w_1}^{(t)}, \ldots, {w_n}^{(t)}$. That is, predict "up" or "down" depending on which prediction has a higher total weight of experts advising it (breaking ties arbitrarily).

2. For every expert i who predicts wrongly, decrease his weight for the next round by multiplying it by a factor of $(1 - \eta)$:

$$ {w_i}^{(t+1)} = (1 - \eta){w_i}^{(t)} \qquad \text{(update rule)}. \tag{9.1} $$

THEOREM 5
After T steps, let ${m_i}^{(T)}$ be the number of mistakes of expert i and $M^{(T)}$ be the number of mistakes our algorithm has made. Then we have the following bound for every i:

$$ M^{(T)} \leq \; 2(1 + \eta){m_i}^{(T)} + \frac{2 \ln n}{\eta}. $$

In particular, this holds for i which is the best expert, i.e. having the least ${m_i}^{(T)}$.

[1]Note that finance experts have studied stock movements for over a century and there are all kinds of stochastic models fitted to them. But we are doing computer science here, and we will see that this adversarial view will help us apply the same idea to a variety of other settings.

PROOF: A simple induction shows that $w_i^{(t+1)} = (1-\eta)^{m_i^{(t)}}$. Let $\Phi^{(t)} = \sum_i w_i^{(t)}$ ("the potential function"). Thus $\Phi^{(1)} = n$. Each time we make a mistake, the weighted majority of experts also made a mistake, so at least half the total weight decreases by a factor $1-\eta$. Thus, the potential function decreases by a factor of at least $(1-\eta/2)$:

$$\Phi^{(t+1)} \;\leq\; \Phi^{(t)}\left(\frac{1}{2} + \frac{1}{2}(1-\eta)\right) \;=\; \Phi^{(t)}(1-\eta/2).$$

Thus simple induction gives $\Phi^{(T+1)} \leq n(1-\eta/2)^{M^{(T)}}$. Finally, since $\Phi^{(T+1)} \geq w_i^{(T+1)}$ for all i, the claimed bound follows by comparing the above two expressions and using the fact that $-\ln(1-\eta) \leq \eta + \eta^2$ since $\eta < \frac{1}{2}$. □

The beauty of this analysis is that it makes no assumption about the sequence of events: they could be arbitrarily correlated and could even depend upon our current weighting of the experts. In this sense, the algorithm delivers more than initially promised, and this lies at the root of why (after obvious generalization) it can give rise to the diverse algorithms mentioned earlier. In particular, the scenario where the events are chosen adversarially resembles a zero-sum game, which we will study in a future lecture.

9.1.1 Randomized version

The above algorithm is deterministic. When $m_i^{(T)} \gg \frac{2\ln n}{\eta}$ we see from the statement of Theorem 5 that the number of mistakes made by the algorithm is bounded from above by roughly $2(1+\eta)m_i^{(T)}$, i.e., approximately twice the number of mistakes made by the best expert. This is tight for any deterministic algorithm (Exercise: prove this!). However, the factor of 2 can be removed by substituting the above deterministic algorithm by a randomized algorithm that predicts according to the majority opinion with probability proportional to its weight. (In other words, if the total weight of the experts saying "up" is 3/4 then the algorithm predicts "up" with probability 3/4 and "down" with probability 1/4.) Then the number of mistakes after T steps is a random variable and the claimed upper bound holds for its *expectation*. Now we give this calculation.

First note that the randomized algorithm can be restated as picking an expert i with probability proportional to its weight and using that expert's prediction. Note that the probability of picking the expert is

$$p_i^{(t)} \stackrel{\text{def}}{=} \frac{w_i^{(t)}}{\sum_j w_j^{(t)}} = \frac{w_i^{(t)}}{\Phi^{(t)}}.$$

Now let's slightly change notation: $m_i^{(t)}$ be 1 if expert i makes a wrong prediction at time t and 0 else. (Thus $m_i^{(t)}$ is the *cost* incurred by this expert at that time.) Then the probability the algorithm makes a mistake at time t is simply $\sum_i p_i^{(t)} m_i^{(t)}$, which we will write as the inner product of the m and p vectors: $\mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}$. Thus the expected number of mistakes by our algorithm at the end is

$$\sum_{t=0}^{T-1} \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}.$$

Now lets compute the change in potential $\Phi^{(t)} = \sum_i {w_i}^{(t)}$:

$$\begin{aligned}
\Phi^{(t+1)} &= \sum_i {w_i}^{(t+1)} \\
&= \sum_i {w_i}^{(t)}(1-\eta {m_i}^{(t)}) \\
&= \Phi^{(t)} - \eta\Phi^{(t)} \sum_i {m_i}^{(t)} {p_i}^{(t)} \\
&= \Phi^{(t)}(1-\eta \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}) \\
&\leq \Phi^{(t)} \exp(-\eta \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}).
\end{aligned}$$

Note that this potential drop is not a random variable; it is a deterministic quantity that depends only on the loss vector $\mathbf{m}^{(t)}$ and the current expert weights (which in turn are determined by the loss vectors of the previous steps).

We conclude by induction that the final potential is at most

$$\prod_{t=0}^{T} \Phi^{(0)} \exp(-\eta \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}) = \Phi^{(0)} \exp(-\eta \sum_t \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}).$$

For each i this final potential is at least the final weight of the ith expert, which is

$$\prod_t (1-\eta {m_i}^{(t)}) \geq (1-\eta)^{\sum_t {m_i}^{(t)}}.$$

Thus taking logs and that $-\log(1-\eta) \leq \eta(1+\eta)$ we conclude that $\sum_{t=0}^{T-1} \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}$ (which is also the expected number of mistakes by our algorithm) is at most $(1+\eta)$ times the number of mistakes by expert i, plus the same old additive factor $2\log n/\eta$.

9.2 The Multiplicative Weights algorithm

(*Now we give a more general result that was not done in class but is completely analogous. We will use the statement in the next class; you can find the proof in the AHK survey if you like.*)

In the general setting, we have a choice of n *decisions* in each round, from which we are required to select one. (The precise details of the decision are not important here: think of them as just indexed from 1 to n.) In each round, each decision incurs a certain *cost*, determined by nature or an adversary. All the costs are revealed *after* we choose our decision, and we incur the cost of the decision we chose. For example, in the prediction from expert advice problem, each decision corresponds to a choice of an expert, and the cost of an expert is 1 if the expert makes a mistake, and 0 otherwise.

To motivate the Multiplicative Weights (MW) algorithm, consider the naïve strategy that, in each iteration, simply picks a decision at random. The expected penalty will be that of the "average" decision. Suppose now that a few decisions are clearly better in the long run. This is easy to spot as the costs are revealed over time, and so it is sensible to

reward them by increasing their probability of being picked in the next round (hence the multiplicative weight update rule).

Intuitively, being in complete ignorance about the decisions at the outset, we select them uniformly at random. This maximum entropy starting rule reflects our ignorance. As we learn which ones are the good decisions and which ones are bad, we lower the entropy to reflect our increased knowledge. The multiplicative weight update is our means of skewing the distribution.

We now set up some notation. Let $t = 1, 2, \ldots, T$ denote the current round, and let i be a generic decision. In each round t, we select a distribution $\mathbf{p}^{(t)}$ over the set of decisions, and select a decision i randomly from it. At this point, the costs of all the decisions are revealed by nature in the form of the vector $\mathbf{m}^{(t)}$ such that decision i incurs cost $m_i{}^{(t)}$. We assume that the costs lie in the range $[-1, 1]$. This is the only assumption we make on the costs; nature is completely free to choose the cost vector as long as these bounds are respected, even with full knowledge of the distribution that we choose our decision from.

The expected cost to the algorithm for sampling a decision i from the distribution $\mathbf{p}^{(t)}$ is

$$\mathop{\mathbf{E}}_{i \in \mathbf{p}^{(t)}}[m_i{}^{(t)}] \;=\; \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}.$$

The total expected cost over all rounds is therefore $\sum_{t=1}^{T} \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)}$. Just as before, our goal is to design an algorithm which achieves a total expected cost not too much more than the cost of the best decision in hindsight, viz. $\min_i \sum_{t=1}^{T} m_i{}^{(t)}$. Consider the following algorithm, which we call the Multiplicative Weights Algorithm. This algorithm has been studied before as the `prod` algorithm of Cesa-Bianchi, Mansour, and Stoltz.

Multiplicative Weights algorithm

Initialization: Fix an $\eta \leq \frac{1}{2}$. For each decision i, associate the weight $w_i{}^{(t)} := 1$.
For $t = 1, 2, \ldots, T$:

1. Choose decision i with probability proportional to its weight $w_i{}^{(t)}$. I.e., use the distribution over decisions $\mathbf{p}^{(t)} = \{w_1{}^{(t)}/\Phi^{(t)}, \ldots, w_n{}^{(t)}/\Phi^{(t)}\}$ where $\Phi^{(t)} = \sum_i w_i{}^{(t)}$.

2. Observe the costs of the decisions $\mathbf{m}^{(t)}$.

3. Penalize the costly decisions by updating their weights as follows: for every decision i, set

$$w_i{}^{(t+1)} \;=\; w_i{}^{(t)}(1 - \eta m_i{}^{(t)}) \tag{9.2}$$

Figure 9.1: The Multiplicative Weights algorithm.

The following theorem —completely analogous to Theorem 5— bounds the total expected cost of the Multiplicative Weights algorithm (given in Figure 9.1) in terms of the total cost of the best decision:

THEOREM 6
Assume that all costs $m_i{}^{(t)} \in [-1, 1]$ *and* $\eta \leq 1/2$. *Then the Multiplicative Weights algo-*

rithm guarantees that after T rounds, for any decision i, we have

$$\sum_{t=1}^{T} \mathbf{m}^{(t)} \cdot \mathbf{p}^{(t)} \;\leq\; \sum_{t=1}^{T} {m_i}^{(t)} + \eta \sum_{t=1}^{T} |{m_i}^{(t)}| + \frac{\ln n}{\eta}.$$

Note that we have not addressed the optimal choice of η thus far. Firstly, it should be small enough that all calculations in the analysis hold, say $\left|\eta \cdot {\mathbf{m}_i}^{(t)}\right| \leq 1/2$ for all i, t. Typically this is done by rescaling the payoffs to lie in $[-1, 1]$, which means that $\sum_{t=1}^{T} |{m_i}^{(t)}| \leq T$. Then setting $\eta \approx \sqrt{\ln n / T}$ gives the tightest upperbound on the right hand side in Theorem 6, by reducing the additive error to about $\sqrt{T \ln n}$. Of course, this is a safe choice; in practice the best η depends upon the actual sequence of events, but of course those are not known in advance.

BIBLIOGRAPHY

S. Arora, E. Hazan, S. Kale. *The multiplicative weights update method: A meta algorithm and its applications.* Theory of Computing, Volume 8 (2012), pp. 121164.

A. Borodin and R. El Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

Chapter 10

Applications of multiplicative weight updates: LP solving, Portfolio Management

Today we see how to use the multiplicative weight update method to solve other problems. In many settings there is a natural way to make local improvements that "make sense."The multiplicative weight updates analysis from last time (via a simple potential function) allows us to understand and analyse the net effect of such sensible improvements. (Formally, what we are doing in many settings is analysing an algorithm called *gradient descent* which we'll encounter more formally later in the course.)

10.1 Solving systems of linear inequalities

We encountered systems of linear inequalities in Lecture 6. Today we study a version that seems slightly more restricted but is nevertheless as powerful as general linear programming. (Exercise!)

SYSTEM 1

$$
\begin{aligned}
a_1 \cdot x &\geq b_1 \\
a_2 \cdot x &\geq b_2 \\
&\vdots \\
a_m \cdot x &\geq b_m \\
x_i &\geq 0 \qquad \forall i = 1, 2, \ldots, n \\
\sum_i x_i &= 1.
\end{aligned}
$$

In your high school you learnt the "graphical"method to solve linear inequalities, and as we discussed in Lecture 6, those can take $m^{n/2}$ time. Here we design an algorithm that,

given an error parameter $\varepsilon > 0$, runs in $O(mL/\varepsilon)$ time and either tells us that the original system is infeasible, or gives us a solution x satisfying the last two lines of the above system, and

$$a_j \cdot x \geq b_j - \varepsilon \qquad \forall j = 1, \ldots, m.$$

(Note that this allows the possibility that the system is infeasible *per se* and nevertheless the algorithm returns such an approximate solution. In that case we have to be happy with the approximate solution.) Here L is an instance-specific parameter that will be clarified below; roughly speaking it is the maximum absolute value of any coefficient. (Recall that the dependence would need to be poly($\log L$) to be considered polynomial time. We will study such a method later on in the course.)

What is a way to certify to somebody that the system is infeasible? The following is *sufficient*: Come up with a system of *nonnegative* weights $w_1, w_2, \ldots, w_m$, one per inequality, such that the following linear program has a negative value:

SYSTEM 2

$$\max \sum_j w_j (a_j \cdot x - b_j)$$
$$x_i \geq 0 \qquad \forall i = 1, 2, \ldots, n$$
$$\sum_i x_i = 1.$$

Note: the w_j's are fixed constants. So this linear program has only two nontrivial constraints (not counting the constraints $x_i \geq 0$) so it is trivial to find a solution quickly, as we saw in class.

EXAMPLE 14 The system of inequalities $x_1 + x_2 \geq 1, x_1 - 5x_2 \geq 5$ is infeasible when combined with the constraints $x_1 + x_2 = 1, x_1 \geq 0, x_2 \geq 0$ since we can multiply the first inequality by 5 and the second by 1 and add to obtain $6x_1 \geq 10$. Note that $6x_1 - 10$ cannot take a positive value when $x_1 \leq 1$.

This method of certifying infeasibility is eminently sensible and the weighting of inequalities is highly reminiscent of the weighting of experts in the last lecture. So we can try to leverage it into a precise algorithm. It will have the following guarantee: (a) Either it finds a set of nonnegative weights certifying infeasibility or (b) It finds a solution $x^{(f)}$ that approximately satisfies the system, in that $a_j \cdot x - b_j \geq -\varepsilon$. Note that conditions (a) and (b) are not disjoint; if a system satisfies both conditions, the algorithm can do either (a) or (b).

We use the meta theorem on MW (Theorem 2) from Lecture 8, where experts have positive or negative costs (where negative costs can be seen as payoffs) and the algorithm seeks to minimize costs by adaptively decreasing the weights of experts with larger cost. The meta theorem says that the algorithm's payoff over many steps tracks —within $(1+\varepsilon)$ multiplicative factor—the cost incurred by the best player, plus an additive term $O(\log n/\varepsilon)$.

We identify m "experts," one per inequality. We maintain a weighting of experts, with ${w_1}^{(t)}, {w_2}^{(t)}, \ldots, {w_m}^{(t)}$ denoting the weights at step t. (At $t = 0$ all weights are 1.) Solve

SYSTEM 2 using these weights. If it turns out to have a negative value, we have proved the infeasibility of SYSTEM 1 and can HALT right away. Otherwise take any solution, say $x^{(t)}$, and think of it as imposing a "cost "of $m_j{}^{(t)} = a_i \cdot x^{(t)} - b_i$ on the jth expert. (In particular, the first line of SYSTEM 2 is merely —up to scaling by the sum of weights— the expected cost for our MW algorithm, and it is positive.) Thus the MW update rule will update the experts' weights as:

$$w_j{}^{(t+1)} \leftarrow w_j{}^{(t)}(1 - \eta\ m_j{}^{(t)}).$$

We continue thus for some number T of steps and if we never found a certificate of the infeasibility of SYSTEM 1 we output the solution $x^{(f)} = \frac{1}{T}(x^{(1)} + x^{(2)} + \cdot + x^{(T)})$, which is the *average* of all the solution vectors found at various steps. Now let L denote the maximum possible absolute value of any $a_i \cdot x - b$ subject to the final two lines of SYSTEM 2.

CLAIM: *If $T > L^2 \log n/\varepsilon^2$ then $x^{(f)}$ satisfies $a_j \cdot x^{(f)} - b_j \geq -\varepsilon$ for all j.*

The proof involves the MW meta theorem which requires us to rescale (multiplying by $1/L$) so all costs lie in $[-1, 1]$ and setting $\varepsilon = \sqrt{\log n/T}$.

We wish to make T large enough so that the per-step additive error $\sqrt{\log n/T} < \varepsilon/L$, which implies $T > L^2 \log n/\varepsilon^2$.

Then we can reason as follows: (a) The expected per-step cost of the MW algorithm was positive (in fact it was positive in each step). (b) The quantity $a_j \cdot x^{(f)} - b_j$ is simply the average cost for expert j per step. (c) The total number of steps is large enough that our MW theorem says that (a) cannot be ε more than (b).

Here is another intuitive explanation that suggests why this algorithm makes sense independent of the experts idea. Vectors $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$ represent simplistic attempts to find a solution to SYSTEM 1. If $a_i \cdot x^{(t)} - b_i$ is positive (resp., negative) this means that the jth constraint was satisfied (resp., unsatisfied) and thus designating it as a cost (resp., reward) ensures that the constraint is given less (resp., more) weight in the next round. Thus the multiplicative update rule is a reasonable way to search for a weighting of constraints that gives us the best shot at proving infeasibility.

Remarks: See the AHK survey on multiplicative weights for the history of this algorithm, which is actually a quantitative version of an older algorithm called *Lagrangian relaxation*.

10.1.1 Duality Theorem

The duality theorem for linear programming says that our method of showing infeasibility of SYSTEM 1 —namely, show for some weighting that SYSTEM 2 has negative value–is not just sufficient but also *necessary*.

This follows by imagining letting ε go to 0. If the system is infeasible, then there is some ε_0 (depending upon the number of constraints and the coefficient values) such that there is no ε-close solution with the claimed properties of $x^{(f)}$ for $\varepsilon < \varepsilon_0$. Hence at one of the steps we must have failed to find a positive solution for SYSTEM 2.

We'll further discuss LP duality in a later lecture.

10.2 Portfolio Management

Now we return to a a more realistic version of the stock-picking problem that motivated our MW algorithm. (You will study this further in a future homework.) There is a set of n

stocks (e.g., the 500 stocks in S& P 500) and you wish to manage an investment portfolio using them. You wish to do at least as well as the best stock in hindsight, and also better than *index* funds, which keep a fixed proportion of wealth in each stock. Let $c_i^{(t)}$ be the price of stock i at the end of day t.

If you have $P_i^{(t)}$ fraction of your wealth invested in stock i then on the tth day your portfolio will rise in value by a multiplicative factor $\sum_i P_i^{(t)} c_i^{(t)}/c_i^{(t-1)}$. Looks familiar?

Let $r_i^{(t)}$ be shorthand for $c_i^{(t)}/c_i^{(t-1)}$.

If you invested all your money in stock i on day 0 then the rise in wealth at the end is

$$\frac{c_i^{(T)}}{c_i^{(0)}} = \prod_{t=0}^{T-1} r_i^{(t)}.$$

Since $\log ab = \log a + \log b$ this gives us the idea to set up the MW algorithm as follows. We run it by looking at n imagined experts, each corresponding to one of the stocks. The payoff for expert i on day t is $\log r_i^{(t)}$. Then as noted above, the total payoff for expert i over all days is $\sum_t \log r_i^{(t)} = \log(c_i^{(T)}/c_i^{(0)})$. This is simply the log of the *multiplicative factor* by which our wealth would increase in T days if we had just invested all of it in stock i on the first day. (This is the jackpot we are shooting for: imagine the money we could have made if we'd put all our savings in Google stock on the day of its IPO.)

Our algorithm plays the canonical MW strategy from last lecture with a suitably small η and with the probability distribution $P^{(t)}$ on experts at time t being interpreted as follows: $P_i^{(t)}$ is the fraction of wealth invested in stock i at the start of day t. Thus we are no longer thinking of picking one expert to follow at each time step; the distribution on experts is the way of splitting our money into the n stocks. In particular on day t our portfolio increases in value by a factor $\sum_i P_i^{(t)} \cdot r^{(t)}$.

Note that we are playing the MW strategy that involves maximising payoffs, not minimizing costs. (That is, increase the weight of experts if they get positive payoff; and reduce weight in case of negative payoff.) The MW theorem says that the total payoff of the MW strategy, namely,
$\sum_t \sum_i P_i^{(t)} \cdot \log r_i^{(t)}$, is at least $(1-\varepsilon)$ times the payoff of the best expert provided T is large enough.

It only remains to make sense of the total payoff for the MW strategy, namely, $\sum_t \sum_i P_i^{(t)} \cdot \log r_i^{(t)}$, since thus far it is just an abstract quantity in a mental game that doesn't make sense *per se* in terms of actual money made.

Since the logarithm is a concave function (i.e. $\frac{1}{2}(\log x + \log y) \leq \log \frac{x+y}{2}$) and $\sum_i P_i^{(t)} = 1$, simple calculus shows that

$$\sum_i P_i^{(t)} \cdot \log r^{(t)} \leq \log(\sum_i P_i^{(t)} \cdot r^{(t)}).$$

The right hand side is exactly the logarithm of the rise in value of the portfolio of the MW strategy on day t. Thus we conclude that the total payoff over all days lower bounds the sum of the logarithms of these rises, which of course is the log of the ratio (final value of the portfolio)/(initial value).

All of this requires that the number of steps T should be large enough. Specifically, if $\left|\log r_i^{(t)}\right| \leq 1$ (i.e., no stock changes value by more than a factor 2 on a single day) then

the total difference between the desired payoff and the actual payoff is $\sqrt{\log n/T}$ times $\max_i \sum_t \left|\log r_i^{(t)}\right|$, as noted in Lecture 8. This performance can be improved by other variations of the method (see the paper by Hazan and Kale). In practice this method doesn't work very well; we'll later explore a better algorithm.

REMARK: One limitation of this strategy is that we have ignored trading costs (ie dealer's commisions). As you can imagine, researchers have also incorporated trading costs in this framework (see Blum and Kalai). Perhaps the bigger limitation of the MW strategy is that it assumes *nothing* about price movements whereas there is a lot known about the (random-like) behavior of the stock market. Traditional portfolio management theory assumes such stochastic models, and is more akin to the decision theory we studied two lectures ago. But stochastic models of the stock market fail sometimes (even catastrophically) and so ideally one wants to combine the stochastic models with the more pessimistic viewpoint taken in the MW method. See the paper by Hazan and Kale. See also a recent interesting paper by Abernathy et al. that suggests that the standard stochastic model arises from optimal actions of market players.

Thomas Cover was the originator of the notion of managing a portfolio against an adversarial market. His strategy is called *universal portfolio.*

BIBLIOGRAPHY

1. A. Blum and A. Kalai. *Efficient Algorithms for Universal Portfolios.* J. Machine Learning Research, 2002.

2. E. Hazan and S. Kale. *On Stochastic and Worst-case Models for Investing.* Proc. NIPS 2009.

3. J. Abernethy, R. Frongillo, A. Wibisono. *Minimax Option Pricing Meets Black-Scholes in the Limit.* Proc. ACM STOC 2012.

Chapter 11

High Dimensional Geometry, Curse of Dimensionality, Dimension Reduction

High-dimensional vectors are ubiquitous in applications (gene expression data, set of movies watched by Netflix customer, etc.) and this lecture seeks to introduce some common properties of these vectors. We encounter the so-called *curse of dimensionality* which refers to the fact that algorithms are simply harder to design in high dimensions and often have a running time exponential in the dimension. We also encounter the *blessings of dimensionality*, which allows us to reason about higher dimensional geometry using tools like Chernoff bounds. We also show the possibility of *dimension reduction* — it is sometimes possible to reduce the dimension of a dataset, for some purposes.

Notation: For a vector $x \in \Re^d$ its *ℓ_2-norm* is $|x|_2 = (\sum_i x_i^2)^{1/2}$ and the ℓ_1-norm is $|x|_1 = \sum_i |x_i|$. For any two vectors x, y their Euclidean distance refers to $|x-y|_2$ and Manhattan distance refers to $|x-y|_1$.

High dimensional geometry is inherently different from low-dimensional geometry.

Example 15 Consider how many *almost orthogonal* unit vectors we can have in space, such that all pairwise angles lie between 88 degrees and 92 degrees.

In $\Re^2$ the answer is 2. In $\Re^3$ it is 3. (Prove these to yourself.)

In $\Re^d$ the answer is $\exp(cd)$ where $c > 0$ is some constant.

Example 16 Another example is the ratio of the the volume of the unit sphere to its circumscribing cube (i.e. cube of side 2). In $\Re^2$ it is $\pi/4$ or about 0.78. In $\Re^3$ it is $\pi/6$ or about 0.52. In d dimensions it is $\exp(-c \cdot d \log d)$.

Let's start with useful generalizations of some geometric objects to higher dimensional geometry:

- The *n-cube* in $\Re^n$: $\{(x_1 ... x_n) : 0 \leq x_i \leq 1\}$. To visualize this in $\Re^4$, think of yourself as looking at one of the faces, say $x_1 = 1$. This is a cube in $\Re^3$ and if you were

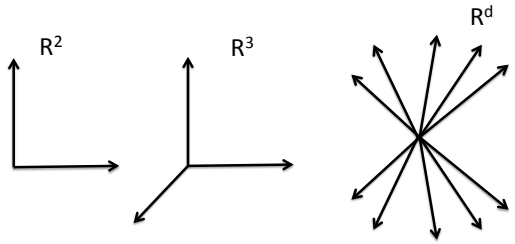

Figure 11.1: Number of almost-orthogonal vectors in $\Re^2, \Re^3, \Re^d$

able to look in the fourth dimension you would see a parallel cube at $x_1 = 0$. The visualization in $\Re^n$ is similar.

The volume of the n-cube is 1.

- The unit *n-ball* in $\Re^d$: $B_d := \{(x_1...x_d) : \sum x_i^2 \leq 1\}$. Again, to visualize the ball in $\Re^4$, imagine you have sliced through it with a hyperplane, say $x_1 = 1/2$. This slice is a ball in $\Re^3$ of radius $\sqrt{1 - 1/2^2} = \sqrt{3}/2$. Every parallel slice also gives a ball.

  The volume of B_d is $\frac{\pi^{d/2}}{(d/2)!}$ (assume d is even if the previous expression bothers you), which is $\frac{1}{d^{\Theta(d)}}$.

- In $\Re^2$, if we slice the unit ball (i.e., disk) with a line at distance $1/2$ from the center then a significant fraction of the ball's volume lies on each side. In $\Re^d$ if we do the same with a hyperplane, then the radius of the $d-1$ dimensional ball is $\sqrt{3}/2$, and so the volume on the other side is negligible. In fact a constant fraction of the volume lies within a slice at distance $1/\sqrt{d}$ from the center, and for any $c > 1$, a $(1-1/c)$-fraction of the volume of the d-ball lies in a strip of width $O(\sqrt{\frac{\log c}{d}})$ around the center.

- A good approximation to picking a random point on the surface of B_n is by choosing random $x_i \in \{-1, 1\}$ independently for $i = 1..n$ and normalizing to get $\frac{1}{\sqrt{n}}(x_1, ..., x_n)$.

  An exact way to pick a random point on the surface of B^n is to choose x_i from the standard normal distribution for $i = 1..n$, and to normalize: $\frac{1}{l}(x_1, ..., x_n)$, where $l = (\sum_i x_i^2)^{1/2}$.

11.1 Number of almost-orthogonal vectors

Now we show there are $\exp(d)$ vectors in $\Re^d$ that are almost-orthogonal. Recall that the angle between two vectors x, y is given by $\cos(\theta) = \langle x, y \rangle / |x|_2 |y|_2$.

Lemma 7
Suppose a is a unit vector in $\Re^n$. Let $x = (x_1, ..., x_n) \in R^n$ be chosen from the surface of B_n by choosing each coordinate at random from $\{1, -1\}$ and normalizing by factor $\frac{1}{\sqrt{n}}$.

Denote by X the random variable $a \cdot x = \sum a_i x_i$. Then:

$$Pr(|X| > t) < e^{-nt^2}$$

PROOF: We have:

$$\mu = E(X) = E(\sum a_i x_i) = 0$$

$$\sigma^2 = E[(\sum a_i x_i)^2] = E[\sum a_i a_j x_i x_j] = \sum a_i a_j E[x_i x_j] = \sum \frac{a_i^2}{n} = \frac{1}{n}.$$

Using the Chernoff bound, we see that,

$$Pr(|X| > t) < e^{-(\frac{t}{\sigma})^2} = e^{-nt^2}.$$

□

COROLLARY 8
If x, y are chosen at random from $\{-1, 1\}^n$, and the angle between them is $\theta_{x,y}$ then

$$Pr\left[|cos(\theta_{x,y})| > \sqrt{\frac{\log c}{n}}\right] < \frac{1}{c}.$$

Hence by if we pick say $\sqrt{c}/2$ random vectors in $\{-1, 1\}^n$, the union bound says that the chance that they all make a pairwise angle with cosine less than $\sqrt{\frac{\log c}{n}}$ is less than $1/2$. Hence we can make $c = \exp(0.01n)$ and still have the vectors be almost-orthogonal (i.e. cosine is a very small constant).

11.2 Curse of dimensionality

Suppose we have a set of vectors in d dimensions and given another vector we wish to determine its closest neighbor (in ℓ_2 norm) to it. Designing fast algorithms for this in the plane (i.e., $\Re^2$) uses the fact that in the plane there are only $O(1)$ distinct points whose pairwise distance is about $1 \pm \varepsilon$. In $\Re^d$ there can be $\exp(d)$ such points.

Thus most algorithms —nearest neighbor, minimum spanning tree, point location etc.— have a running time depending upon $\exp(d)$. This is the *curse of dimensionality* in algorithms. (The term was invented by R. Bellman, who as we saw earlier had a knack for giving memorable names.)

In machine learning and statistics sometimes the term refers to the fact that available data is too sparse in high dimensions; it takes $\exp(d)$ amount of data (say, points on the sphere)to ensure that each new sample is close to an existing sample. This is a different take on the same underlying phenomenon.

I hereby coin a new term: *Blessing of dimensionality.* This refers to the fact that many phenomena become much clearer and easier to think about in high dimensions because one can use simple rules of thumb (e.g., Chernoff bounds, measure concentration) which don't hold in low dimensions.

11.3 Dimension Reduction

Now we describe a central result of high-dimensional geometry (at least when distances are measured in the ℓ_2 norm). Problem: Given n points $z^1, z^2, ..., z^n$ in $\Re^n$, we would like to find n points $u^1, u^2, ..., u^n$ in $\Re^m$ where m is of low dimension (compared to n) and the metric restricted to the points is almost preserved, namely:

$$\|z^i - z^j\|_2 \leq \|u^i - u^j\|_2 \leq (1+\varepsilon)\|z^j - z^j\|_2 \ \forall i, j. \tag{11.1}$$

The following main result is by Johnson & Lindenstrauss :

THEOREM 9
In order to ensure (11.1), $m = O(\frac{\log n}{\varepsilon^2})$ suffices, and in fact the mapping can be a linear mapping.

The following ideas do not work to prove this theorem (as we discussed in class): (a) take a random sample of m coordinates out of n. (b) Partition the n coordinates into m subsets of size about n/m and *add* up the values in each subset to get a new coordinate.

PROOF: Choose m vectors $x^1, ..., x^m \in \Re^n$ at random by choosing each coordinate randomly from $\{\sqrt{\frac{1+\varepsilon}{m}}, -\sqrt{\frac{1+\varepsilon}{m}}\}$. Then consider the mapping from $\Re^n$ to $\Re^m$ given by

$$z \longrightarrow (z \cdot x^1, z \cdot x^2, \ldots, z \cdot x^m).$$

In other words $u^i = (z^i \cdot x^1, z^i \cdot x^2, ..., z^i \cdot x^m)$ for $i = 1, \ldots, k$. We want to show that with positive probability, $u^1, ..., u^k$ has the desired properties. This would mean that there exists at least one choice of $u^1, ..., u^k$ satisfying inequality 11.1. To show this, first we write the expression $\|u^i - u^j\|$ explicitly:

$$\|u^i - u^j\|^2 = \sum_{k=1}^{m} \left(\sum_{l=1}^{n} (z_l^i - z_l^j) x_l^k \right)^2.$$

Denote by z the vector $z^i - z^j$, and by u the vector $u^i - u^j$. So we get:

$$\|u\|^2 = \|u^i - u^j\|^2 = \sum_{k=1}^{m} \left(\sum_{l=1}^{n} z_l x_l^k \right)^2.$$

Let X_k be the random variable $(\sum_{l=1}^{n} z_l x_l^k)^2$. Its expectation is $\mu = \frac{1+\varepsilon}{m}\|z\|^2$ (can be seen similarly to the proof of lemma 7). Therefore, the expectation of $\|u\|^2$ is $(1+\varepsilon)\|z\|^2$. If we show that $\|u\|^2$ is concentrated enough around its mean, then it would prove the theorem. More formally, this is done in the following Chernoff bound lemma. □

LEMMA 10
There exist constants $c_1 > 0$ and $c_2 > 0$ such that:

1. $Pr[\|u\|^2 > (1+\beta)\mu] < e^{-c_1\beta^2 m}$
2. $Pr[\|u\|^2 < (1-\beta)\mu] < e^{-c_2\beta^2 m}$

Therefore there is a constant c such that the probability of a "bad" case is bounded by:

$$Pr[(\|u\|^2 > (1+\beta)\mu) \vee (\|u\|^2 < (1-\beta)\mu)] < e^{-c\beta^2 m}$$

Now, we have $\binom{n}{2}$ random variables of the type $\|u_i - u_j\|^2$. Choose $\beta = \frac{\varepsilon}{2}$. Using the union bound, we get that the probability that any of these random variables is not within $(1 \pm \frac{\varepsilon}{2})$ of their expected value is bounded by

$$\binom{n}{2} e^{-c\frac{\varepsilon^2}{4}m}.$$

So if we choose $m > \frac{8(\log n + \log c)}{\varepsilon^2}$, we get that with positive probability, all the variables are close to their expectation within factor $(1 \pm \frac{\varepsilon}{2})$. This means that for all i,j:

$$(1-\frac{\varepsilon}{2})(1+\varepsilon)\|z^i - z^j\|^2 \leq \|u^i - u^j\|^2 \leq (1+\frac{\varepsilon}{2})(1+\varepsilon)\|z^i - z^j\|^2$$

Therefore,

$$\|z_i - z_j\|^2 \leq \|u^i - u^j\|^2 \leq (1+\varepsilon)^2\|z^i - z^j\|^2,$$

and taking square root:

$$\|z^i - z^j\| \leq \|u^i - u^j\| \leq (1+\varepsilon)\|z^i - z^j\|,$$

as required.

Question: The above dimension reduction preserves (approximately) ℓ_2-distances. Can we do dimension reduction that preserves ℓ_1 distance? This was an open problem for many years until Brinkman and Charikar showed in 2004 that no such dimension reduction is possible.

Question: Is the theorem tight, or can we reduce the dimension even further below $O(\log n/\varepsilon^2)$? Alon has shown that this is essentially tight.

Finally, we note that there is a now-extensive literature on more efficient techniques for JL-style dimension reduction, with a major role played by a 2006 paper of Ailon and Chazelle. Do a google search for "Fast Johnson Lindenstrauss Transforms."

11.3.1 Locality preserving hashing

Suppose we wish to hash high-dimensional vectors so that nearby vectors tend to hash into the same bucket. To do this we can do a random projection into say the cube in 5 dimensions. We discretise the cube into smaller cubes of size ε. Then there are $1/\varepsilon^5$ smaller cubes; these can be the buckets.

This is simplistic; more complicated schemes have been constructed. Things get even more interesting when we are interested in ℓ_1-distance.

11.3.2 Dimension reduction for efficiently learning a linear classifier

Suppose we are given a set of m data points in $\Re^d$, each labeled with 0 or 1. For example the data points may represent emails (represented by the vector giving frequencies of various

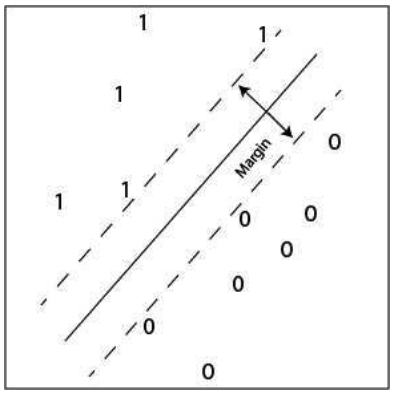

Figure 11.2: Margin of a linear classifier with respect to some labeled points

words in them) and the label indicates whether or not the user labeled them as spam. We are trying to learn the rule (or "classifier") that separates the 1's from 0's.

The simplest classifier is a halfspace. Finding whether there exists a halfspace $\sum_i a_i x_i \geq b$ that separates the 0's from 1's is solvable via Linear Programming. This LP has $n+1$ variables and m constraints.

However, there is no guarantee in general that the halfspace that separates the training data will generalize to new examples? ML theory suggests conditions under which the classifier does generalize, and the simplest is *margin*. Suppose the data points are unit vectors. We say the halfspace has *margin* ε if every datapoint has distance at least ε to the halfspace.

In the next homework you will show that if such a margin exists then dimension reduction to $O(\log n/\varepsilon^2)$ dimensions at most halves the margin. Hence the LP to find it only has $O(\log n/\varepsilon^2)$ variables instead of $n+1$.

Bibliography:

1. W. Brinkman and M. Charikar. On the impossibility of Dimension Reduction in ℓ_1. IEEE FOCS 2003.

Chapter 12

Random walks, Markov chains, and how to analyse them

Today we study random walks on graphs. When the graph is allowed to be directed and weighted, such a walk is also called a *Markov Chain*. These are ubiquitous in modeling many real-life settings.

Example 17 (Drunkard's walk) There is a sequence of $2n + 1$ pubs on a street. A drunkard starts at the middle house. At every time step, if he is at pub number i, then with probability $1/2$ he goes to pub number $i - 1$ and with probability $1/2$ to pub $i + 1$. How many time steps does it take him to reach either the first or the last pub?

Thinking a bit, we quickly realize that the first m steps correspond to m coin tosses, and the distance from the starting point is simply the difference between the number of heads and the number of tails. We need this difference to be n. Recall that the number of heads is distributed like a normal distribution with mean $m/2$ and standard deviation $\sqrt{m}/2$. Thus m needs to be of the order of n^2 before there is a good chance of this random variable taking the value $m + n/2$.

Thus being drunk slows down the poor guy by a quadratic factor.

Example 18 (Exercise) Suppose the drunkard does his random walk in a city that's designed like a grid. At each step he goes North/South/East/West by one block with probability $1/4$. How many steps does it take him to get to his intended address, which is n blocks north and n blocks east away?

Random walks in space are sometimes called *Brownian motion*, after botanist Robert Brown, who in 1826 peered at a drop of water using a microscope and observed tiny particles (such as pollen grains and other impurities) in it performing strange random-looking movements. He probably saw motion similar to the one in the above figure. Explaining this movement was a big open problem. In 1905, during his "miraculous year" (when he solved 3 famous open problems in physics) Einstein explained Brownian motion as a random walk in space caused by the little momentum being imparted to the pollen in random directions by the (invisible) molecules of water. This theoretical prediction was soon experimentally confirmed and seen as a "proof"of the existence of molecules. Today random

walks and brownian motion are used to model the movements of many systems, including stock prices.

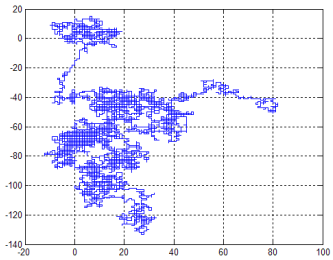

Figure 12.1: A 2D Random Walk

Example 19 (Random walks on graph) We can consider a random walk on a d-regular graph $G = (V, E)$ instead of in physical space. The particle starts at some vertex v_0 and at each step, if it is at a vertex u, it picks a random edge of u with probability $1/d$ and then moves to the other vertex in that edge. There is also a *lazy* version of this walk where he stays at u with probability $1/2$ and moves to a random neighbor with probability $1/2d$.

Thus the drunkard's walk can be viewed as a random walk on a line graph.

One can similarly consider random walks on directed graph (randomly pick an outgoing edge out of u to leave from) and walks on weighted graph (pick an edge with probability proportional to its weight). Walks on directed weighted graphs are called *Markov Chains*.

In a random walk, the next step does not depend upon the previous history of steps, only on the current position/state of the moving particle. In general, the term *markovian* refers to systems with a "memoryless"property. In an earlier lecture we encountered Markov Decision Processes, which also had this memoryless property.

Example 20 (Bigram and trigram models in speech recognition) Language recognition systems work by constantly predicting what's coming next. Having heard the first i words they try to generate a prediction of the $i + 1$th word[1]. This is a very complicated piece of software, and one underlying idea is to model language generation as a markov chain. (This is not an exact model; language is known to not be markovian, at least in the simple way described below.)

The simplest idea would be to model this as a markov chain on the words of a dictionary. Recall that everyday English has about $5,000$ words. A simple markovian model consists of thinking of a piece of text as a random walk on a space with 5000 states (= words). A state corresponds to the last word that was just seen. For each word pair w_1, w_2 there is a probability p_{w_1,w_2} of going from w_1 to w_2. According to this Markovian model, the probability of generating a sentence with the words w_1, w_2, w_3, w_4 is $q_{w_1} p_{w_1 w_2} p_{w_2 w_3} p_{w_3 w_4}$ where q_{w_1} is the probability that the first word is w_1.

[1]You can see this in the typing box on smartphones, which always display their guesses of the next word you are going to type. This lets you save time by clicking the correct guess.

To actually fit such a model to real-life text data, we have to estimate $5,000$ probabilities q_{w_1} for all words and $(5,000)^2$ probabilities $p_{w_1 w_2}$ for all word pairs. Here

$$p_{w_1 w_2} = \Pr[w_2 \mid w_1] = \frac{\Pr[w_2 w_1]}{\Pr[w_1]},$$

namely, the probability that word w_2 is the next word given that the last word was w_1.

One can derive empirical values of these probabilities using a sufficiently large text corpus. (Realize that we have to estimate 25 million numbers, which requires either a very large text corpus or using some shortcuts.)

An even better model in practice is a *trigram model* which uses the previous two words to predict the next word. This involves a markov chain containing one state for every *pair* of words. Thus the model is specified by $(5,000)^3$ numbers of the form $\Pr[w_3 \mid w_2 w_1]$. Fitting such a model is beyond the reach of current computers but we won't discuss the shortcuts that need to be taken.

12.1 Recasting a random walk as linear algebra

A *Markov chain* is a discrete-time stochastic process on n states defined in terms of a transition probability matrix (M) with rows i and columns j.

$$\mathbf{M} = \left(M_{ij}\right)$$

where M_{ij} corresponds to the probability that the state at time step $t+1$ will be j, given that the state at time t is i. This process is *memoryless* in the sense that this transition probability does not depend upon the history of previous transitions.

Therefore, each row in the matrix $\mathbf{M}$ is a distribution, implying $M_{ij} \geq 0 \forall i, j \in S$ and $\sum_j M_{ij} = 1$. The bigram or trigram models are examples of Markov chains.

Using a slight twist in the viewpoint we can use linear algebra to analyse random walks. Instead of thinking of the drunkard as being at a specific point in the state space, we think of the vector that specifies his probability of being at point $i \in S$. Then the randomness goes away and this vector evolves according to deterministic rules. Let us understand this evolution.

Let the initial distribution be given by the row vector $\mathbf{x} \in \Re^n$, $x_i \geq 0$ and $\sum_i x_i = 1$. After one step, the probability of being at space i is $\sum_j x_j M_{ji}$, which corresponds to a new distribution $\mathbf{xM}$. It is easy to see that $\mathbf{xM}$ is again a distribution.

Sometimes it is useful to think of x as describing the amount of *probability fluid* sitting at each node, such that the sum of the amounts is 1. After one step, the fluid sitting at node i distributes to its neighbors, such that M_{ij} fraction goes to j.

Suppose we take two steps in this Markov chain. The memoryless property implies that the probability of going from i to j is $\sum_k M_{ik} M_{kj}$, which is just the (i,j)th entry of the matrix M^2. In general taking t steps in the Markov chain corresponds to the matrix M^t, and the state at the end is $\mathbf{x}M^t$. Thus the

DEFINITION 2 *A distribution π for the Markov chain $\mathbf{M}$ is a* stationary distribution *if $\pi\mathbf{M} = \pi$.*

EXAMPLE 21 (DRUNKARD'S WALK ON n-CYCLE) Consider a Markov chain defined by the following random walk on the nodes of an n-cycle. At each step, stay at the same node with probability 1/2. Go left with probability 1/4 and right with probability 1/4.

The uniform distribution, which assigns probability $1/n$ to each node, is a stationary distribution for this chain, since it is unchanged after applying one step of the chain.

DEFINITION 3 *A Markov chain* $\mathbf{M}$ *is* ergodic *if there exists a unique stationary distribution* π *and for every (initial) distribution* $\mathbf{x}$ *the limit* $\lim_{t\to\infty} \mathbf{x}\mathbf{M}^t = \pi$.

In other words, no matter what initial distribution you choose, if you let it evolve long enough the distribution converges to the stationary distribution. Some basic questions are when stationary distributions exist, whether or not they are unique, and how fast the Markov chain converges to the stationary distribution.

Does Definition 2 remind you of something? Almost all of you know about eigenvalues, and you can see that the definition requires π to be an eigenvector which has all nonnegative coordinates and whose corresponding eigenvalue is 1.

In today's lecture we will be interested in Markov chains corresponding to undirected d-regular graphs, where the math is easier because the underlying matrix is *symmetric*: $M_{ij} = M_{ji}$.

Eigenvalues. Recall that if $M \in \Re^{n\times n}$ is a square symmetric matrix of n rows and columns then an **eigenvalue** of M is a scalar $\lambda \in \Re$ such that exists a vector $x \in \Re^n$ for which $M \cdot x = \lambda \cdot x$. The vector x is called the **eigenvector** corresponding to the eigenvalue λ. M has n real eigenvalues denoted $\lambda_1 \leq ... \leq \lambda_n$. (The multiset of eigenvalues is called the *spectrum*.) The eigenvectors associated with these eigenvalues form an orthogonal basis for the vector space $\Re^n$ (for any two such vectors the inner product is zero and all vectors are linear independent). The word *eigenvector* comes from German, and it means "one's own vector. "The eigenvectors are n prefered directions $u_1, u_2, \ldots, u_n$ for the matrix, such that applying the matrix on these directions amounts to simple scaling by the corresponding eigenvalue. Furthermore these eigenvectors span $\Re^n$ so every vector x can be written as a linear combination of these.

EXAMPLE 22 We show that every eigenvalue λ of M is at most 1. Suppose $\vec{e}$ is the corresponding eigenvector. Say the largest coordinate is i. Then $\lambda e_i = \sum_{j:\{i,j\}\in E} \frac{1}{d} e_j$ by definition. If $\lambda > 1$ then at least one of the neighbors must have $e_j > e_i$, which is a contradiction. By similar argument we conclude that every eigenvalue of M is at most -1 in absolute value.

12.1.1 Mixing Time

Informally, the *mixing time* of a Markov chain is the time it takes to reach "nearly stationary" distribution from any arbitrary starting distribution.

DEFINITION 4 *The mixing time of an ergodic Markov chain* M *is* t *if for every starting distribution* x*, the distribution* xM^t *satisfies* $\left|xM^t - \pi\right|_1 \leq 1/4$*. (Here* $|\cdot|_1$ *denotes the* ℓ_1 *norm and the constant "*1/4*" is arbitrary.)*

Example 23 (Mixing time of drunkard's walk on a cycle) Let us consider the mixing time of the walk in Example 21. Suppose the initial distribution concentrates all probability at state 0. Then $2t$ steps correspond to about t random steps (= coin tosses) since with probability $1/2$ the drunk does not move. Thus the location of the drunk is

$$(\#(\text{Heads}) - \#(\text{Tails})) \pmod{n}.$$

As argued earlier, it takes $\Omega(n^2)$ steps for the walk to reach the other half of the circle with any reasonable probability, which implies that the mixing time is $\Omega(n^2)$. We will soon see that this lowerbound is fairly tight; the walk takes about $O(n^2 \log n)$ steps to mix well.

12.2 Upper bounding the mixing time (undirected d-regular graphs)

For simplicity we restrict attention to random walks on regular graphs. Let M be a Markov chain on a d-regular undirected graph with an adjacency matrix A. Then, clearly $M = \frac{1}{d}A$.

Clearly, $\frac{1}{n}\vec{1}$ is a stationary distribution, which means it is an eigenvector of M. What is the mixing time? In other words if we start in the initial distribution $\mathbf{x}$ then how fast does $\mathbf{x}M^t$ converge to $\vec{1}$?

First, let's identify two hurdles that would prevent such convergence, and in fact prevent the graph from having a unique stationary distribution. (a) *Being disconnected:* if the walk starts in a vertex in one connected component, it never visits another component, and vice versa. So two walks starting in the two components cannot converge to the same distribution, no longer how long we run them. (b) *Being bipartite:* This means the graph consists of two sets A, B such that there are no edges within A and within B; all edges go between A and B. Then the walk starting in A will bounce back and forth between A and B and thus not converge.

Example 24 (Exercise:) Show that if the graph is connected, then every eigenvalue of M apart from the first one is strictly less than 1. However, the value -1 is still possible. Show that if -1 is an eigenvalue then the graph is bipartite.

Note that if $\mathbf{x}$ is a distribution, $\mathbf{x}$ can be written as

$$\mathbf{x} = \vec{1}\frac{1}{n} + \sum_{i=2}^{n} \alpha_i \mathbf{e_i}$$

where $\mathbf{e_i}$ are the eigenvectors of M which form an orthogonal basis and $\mathbf{1}$ is the first eigenvector with eigenvalue 1. (Clearly, $\mathbf{x}$ can be written as a combination of the eigenvectors; the observation here is that the coefficient in front of the first eigenvector $\vec{1}$ is $\vec{1} \cdot x / \left|\vec{1}\right|_2^2$ which is $\frac{1}{n}\sum_i x_i = \frac{1}{n}$.)

$$\begin{aligned} M^t\mathbf{x} &= M^{t-1}(M\mathbf{x}) \\ &= M^{t-1}(\frac{1}{n}\vec{1} + \sum_{i=2}^{n} \alpha_i \lambda_i \mathbf{e_i}) \\ &= M^{t-2}(M(\frac{1}{n}\vec{1} + \sum_{i=2}^{n} \alpha_i \lambda_i \mathbf{e_i})) \\ &\cdots \\ &= \frac{1}{n}\vec{1} + \sum_{i=2}^{n} \alpha_i \lambda_i^t \mathbf{e_i} \end{aligned}$$

Also

$$\| \sum_{i=2}^{n} \alpha_i \lambda_i^t \mathbf{e_i} \|_2 \leq \lambda_{max}^t$$

where λ_{max} is the second largest eigenvalue of M in absolute value. (Note that we are using the fact that the total ℓ_2 norm of any distribution is $\sum_i x_i^2 \leq \sum_i x_i = 1$.)

Thus we have proved $\left|M^t\mathbf{x} - \frac{1}{n}\mathbf{1}\right|_2 \leq \lambda_{max}^t$. Mixing times were defined using ℓ_1 distance, but Cauchy Schwartz inequality relates the ℓ_2 and ℓ_1 distances: $|p|_1 \leq \sqrt{n}\,|p|_2$. So we have proved:

THEOREM 11
The mixing time is at most $O(\frac{\log n}{\lambda_{\max}})$.

Note also that if we let the Markov chain run for $O(k \log n/\lambda_{\max})$ steps then the distance to uniform distribution drops to $\exp(-k)$. This is why we were not very fussy about the constant $1/4$ in the definition of the mixing time earlier.

Remark: What if $\lambda_{\max}$ is 1 (i.e., -1 is an eigenvalue)? This breaks the proof and in fact the walk may not be ergodic. However, we can get around this problem by modifying the random walk to be *lazy*, by adding a self-loop at each node that ensures that the walk stays at a node with probability $1/2$. Then the matrix describing the new walk is $\frac{1}{2}(I + M)$, and its eigenvalues are $\frac{1}{2}(1 + \lambda_i)$. Now all eigenvalues are less than 1 in absolute value. This is a general technique for making walks *ergodic*.

EXAMPLE 25 (EXERCISE) Compute the eigenvalues of the drunkard's walk on the n-cycle and show that its mixing time is $O(n^2 \log n)$.

12.3 Analysis of Mixing Time for General Markov Chains

We did not do this in class; this is extra reading for those who are interested.

In the class we only analysed random walks on d-regular graphs and showed that they converge exponentially fast with rate given by the second largest eigenvalue of the transition matrix. Here, we prove the same fact for general ergodic Markov chains.

THEOREM 12
The following are necessary and sufficient conditions for ergodicity:

1. *connectivity:* $\forall i, j : \mathbf{M}^t(i,j) > 0$ *for some* t.
2. *aperiodicity:* $\forall i : gcd\{t : \mathbf{M}^t(i,j) > 0\} = 1$.

REMARK 1 Clearly, these conditions are necessary. If the Markov chain is disconnected it cannot have a unique stationary distribution —there is a different stationary distribution for each connected component. Similarly, a bipartite graph does not have a unique distribution: if the initial distribution places all probability on one side of the bipartite graph, then the distribution at time t oscillates between the two sides depending on whether t is odd or even. Note that in a bipartite graph $gcd\{t : \mathbf{M}^t(i,j) > 0\} \geq 2$. The sufficiency of these conditions is proved using eigenvalue techniques (for inspiration see the analysis of mixing time later on).

Both conditions are easily satisfied in practice. In particular, any Markov chain can be made aperiodic by adding self-loops assigned probability 1/2.

DEFINITION 5 *An ergodic Markov chain is* reversible *if the stationary distribution* π *satisfies for all* i, j, $\pi_i \mathbf{P}_{ij} = \pi_j \mathbf{P}_{ji}$.

We need a lemma first.

LEMMA 13
Let M *be the transition matrix of an ergodic Markov chain with stationary distribution* π *and eigenvalues* $\lambda_1(=1) \geq \lambda_2 \geq \ldots \geq \lambda_n$, *corresponding to eigenvectors* $v_1(=\pi), v_2, \ldots v_n$. *Then for any* $k \geq 2$,

$$v_k \vec{1} = 0.$$

PROOF: We have $v_k M = \lambda_k v_k$. Mulitplying by $\vec{1}$ and noting that $M\vec{1} = \vec{1}$, we get

$$v_k \vec{1} = \lambda_k v_k \vec{1}.$$

Since the Markov chain is ergodic, $\lambda_k \neq 1$, so $v_k \vec{1} = 0$ as required. □

We are now ready to prove the main result concerning the exponentially fast convergence of a general ergodic Markov chain:

THEOREM 14
In the setup of the lemma above, let $\lambda = max\{|\lambda_2|, |\lambda_n|\}$. *Then for any initial distribution* x, *we have*

$$||xM^t - \pi||_2 \leq \lambda^t ||x||_2.$$

PROOF: Write x in terms of $v_1, v_2, \ldots, v_n$ as

$$x = \alpha_1 \pi + \sum_{i=2}^{n} \alpha_i v_i.$$

Multiplying the above equation by $\vec{1}$, we get $\alpha_1 = 1$ (since $x\vec{1} = \pi\vec{1} = 1$). Therefore $xM^t = \pi + \sum_{i=2}^{n} \alpha_i \lambda_i^t v_i$, and hence

$$\begin{aligned} ||xM^t - \pi||_2 &\leq ||\sum_{i=2}^{n} \alpha_i \lambda_i^t v_i||_2 && (12.1) \\ &\leq \lambda^t \sqrt{\alpha_2^2 + \cdots + \alpha_n^2} && (12.2) \\ &\leq \lambda^t ||x||_2, && (12.3) \end{aligned}$$

as needed. □

Chapter 13

Intrinsic dimensionality of data and low-rank approximations: SVD

Today's topic is a technique called *singular value decomposition* or SVD. We'll take two views of it, and then encounter a surprising algorithm for it, which in turn leads to a third interesting view.

13.1 View 1: Inherent dimensionality of a dataset

In many settings we have a set of m vectors $v_1, v_2, \ldots, v_m$ in $\Re^n$. Think of n as large, and maybe m also. We would like to represent v_i's using fewer number of dimensions, say k. We saw one technique in an earlier lecture, namely, Johnson-Lindenstrauss dimension reduction, which achieves $k = O(\log n/\varepsilon^2)$. As explored in HW 3, JL-dimension reduction is relevant only where we only care about preserving all pairwise ℓ_2 distances among the vectors. Its advantage is that it works for *all* datasets. But to many practitioners, that is also a huge disadvantage: since it is oblivious to the dataset, it cannot be tweaked to leverage properties of the data at hand.

Today we are interested in datasets where the v_i's do have a special structure: they are well-approximated by some low-dimensional set of vectors. By this we mean that for some small k, there are vectors $u_1, u_2, \ldots, u_k \in \Re^n$ such that every v_i is close to the *span* of $u_1, u_2, \ldots, u_k$. In many applications k is fairly small, even 3 or 4, and JL dimension reduction is of no use.

Let's attempt to formalize the problem at hand. We are looking for k-dimensional vectors $u_1, u_2, \ldots, u_k$ and mk coefficients $\alpha_{i1}, \ldots, \alpha_{ik} \in \Re$ such that $\left|v_i - \sum_j \alpha_{ij} u_j\right|_2^2 \approx$ small. But of course any real-life data set has *outliers,* for which this may not hold. But if most vectors fit the conjectured structure, then we expect

$$\sum_i \left|v_i - \sum_j \alpha_{ij} u_j\right|_2^2 \approx \text{small} \tag{13.1}$$

This problem is nonlinear and nonconvex as stated. Today we will try to understand it

more and learn how to solve it. We will find that it is actually easy (which I find one of the miracles of math: one of few natural nonlinear problems that are solvable in polynomial time).

But first some examples of why this problem arises in practice.

EXAMPLE 26 (UNDERSTANDING SHOPPING DATA) Suppose a marketer is trying to assess shopping habits. He observes the shopping behaviour of m shoppers with respect to n goods: how much of each good did they buy? This gives m vectors in $\Re^n$.

The simplest model for this would be: every shopper starts with a budget, and allocates it equally among all m items. Then if B_i is the budget of shopper i and p_j is the price for item j, the ith vector is $\frac{1}{n}(\frac{B_i}{p_1}, \frac{B_i}{p_2}, \ldots, \frac{B_i}{p_n})$. Denoting by $\vec{u}$ the vector of price inverses, namely, $(1/p_1, 1/p_2, \ldots, 1/p_n)$ this is just $\frac{B_i}{n}\vec{u}$. We conclude that the data is 1-dimensional: just scalar multiples of $\vec{u}$.

But maybe the above model is too unrealistic and doesn't fit the data well. Then one could try another model. We assume that the goods partition into k categories like produce, canned goods, etc. $S_1, S_2, \ldots, S_k$. These categories are unknown to us. Assume furthermore that the ith shopper designates a budget B_{it} for the tth category, and then divides this budget equally among goods in that category. Let $u_t \in \Re^n$ denotes the vector inf $\Re^n$ whose coordinate is 0 for goods not in S_t and the inverse price for goods in S_t. Then the quantities of each good purchased by shopper i are given by the vector $\sum_{t=1}^{k} \frac{B_{it}}{|S_t|} u_t$. In other words, this model predicts that the dataset is k-dimensional.

Of course, no model is exact so the data set will only be approximately k-dimensional, and thus the problem in (13.1) is a possible formulation.

One can consider alternative probabilistic models of data generation where the shopper picks items randomly from each category. You'll analyse that in the next homework.

EXAMPLE 27 (UNDERSTANDING MICROARRAY DATA IN BIOLOGY) The number of genes in your cell is rather large, and their activity levels —which depend both upon your genetic code and environmental factors— determine your body's functioning. *Microarrays* are tiny "chips" of chemicals sites that can screen the activity levels —aka *gene expression* levels—of a large number of genes in one go, say $n = 10,000$ genes. Typically these genes would have been chosen because they are suspected to be related to the phenomenon being studied, say a particular disease, immune reaction etc. After testing m individuals, one obtains m vectors in $\Re^n$.

In practice it is found that this gene expression data is low-dimensional in the sense of (13.1). This means that there are say 4 directions u_1, u_2, u_3, u_4 such that most of the vectors are close to their span. These new axis directions usually have biological meaning; eg they help identify genes whose expression (up or down) is controlled by common regulatory mechanisms.

13.2 View 2: Low rank matrix approximations

We have an $m \times n$ matrix M. We suspect it is actually a noisy version of a rank-k matrix, say $\tilde{M}$. We would like to find out $\tilde{M}$. One natural idea is to solve the following optimization

problem

$$\min \sum_{ij} \left| M_{ij} - \tilde{M}_{ij} \right|^2 \qquad \text{s.t.} \quad \tilde{M} \text{ is a rank-}k \text{ matrix} \tag{13.2}$$

Again, seems like a hopeless nonlinear optimization problem. Peer a little harder and you realize that, first, a rank-k matrix is just one whose rows are linear combinations of k independent vectors, and second, if you let M_i denote the ith column of M then you are trying to solve nothing but problem (13.1)!

Example 28 (Planted bisection/Hidden Bisection) Graph bisection is the problem where we are given a graph $G = (V, E)$ and wish to partition V into two equal sets $S, \overline{S}$ such that we minimize the number of edges between $S, \overline{S}$. It is NP-complete. Let's consider the following average case version.

Nature creates a random graph on n nodes as follows. It partitions nodes into S_1, S_2. Within S_1, S_2 it puts each edge with prob. p, and between S_1, S_2 put each edge with prob. q where $q < p$. Now this graph is given to the algorithm. Note that the algorithm doesn't know S_1, S_2. It has to find the optimum bisection.

It is possible to show using Chernoff bounds that if $q = \Omega(\frac{\log n}{n})$ then with high probability the optimum bisection in the graph is the planted one, namely, S_1, S_2. How can the algorithm recover this partition?

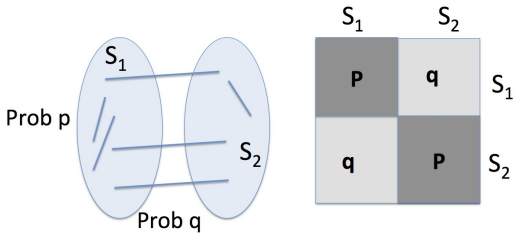

Figure 13.1: Planted Bisection problem: Edge probability is p within S_1, S_2 and q between S_1, S_2 where $q < p$. On the right hand side is the adjacency matrix. If we somehow knew S_1, S_2 and grouped the corresponding rows and columns together, and squint at the matrix from afar, we'd see more density of edges within S_1, S_2 and less density between S_1, S_2. Thus from a distance the adjacency matrix looks like a rank 2 matrix.

The observation in Figure 14.1 suggests that the adjacency matrix is close to a rank 2 matrix shown there: the block within S_1, S_2 have value p in each entry; the blocks between S_1, S_2 have q in each entry.

Maybe if we can solve (13.2) with $k = 2$ we are done? This turns out to be correct as we will see in next lecture.

One can study planted versions of many other NP-hard problems as well.

Many practical problems involve graph partitioning. For instance, image recognition involves first partitioning the image into its component pieces (sky, ground, tree, etc.); a

process called *image segmentation* in computer vision. This is done by graph partitioning on a graph defined on pixels where edges denote *pixel-pixel similarity*. Perhaps planted graphs are a better model for such real-life settings than worst-case graphs.

13.3 Singular Value Decomposition

Now we describe the tool that lets us solve the above problems.

For simplicity let's start with a symmetric matrix M. Suppose its eigenvalues are $\lambda_1, \ldots, \lambda_n$ in decreasing order by absolute value, and the corresponding eigenvectors (scaled to be unit vectors) are $e_1, e_2, \ldots, e_n$. (These are column vectors.) Then M has the following alternative representation.

THEOREM 15 (SPECTRAL DECOMPOSITION)
$M = \sum_i \lambda_i e_i e_i^T$.

PROOF: At first sight, the equality does not even seem to pass a "typecheck"; a matrix on the left and vectors on the right. But then we realize that $e_i e_i^T$ is actually an $n \times n$ matrix (it has rank 1 since every column is a multiple of e_i). So the right hand side is indeed a matrix. Let us call it B.

Any matrix can be specified completely by describing how it acts on an orthonormal basis. By definition, M is the matrix that acts as follows on the orthonormal set $\{e_1, e_2, \ldots, e_n\}$: $Me_j = \lambda_j e_j$. How does B act on this orthonormal set? We have

$$\begin{aligned} Be_j &= (\sum_i \lambda_i e_i e_i^T) e_j \\ &= \sum_i \lambda_i e_i (e_i^T e_j) \qquad \text{(distributivity and associativity of matrix multiplication)} \\ &= \lambda_j e_j \end{aligned}$$

since $e_i^T e_j =< e_i, e_j >$ is 1 if $i = j$ and 0 else. We conclude that $B = M$. □

THEOREM 16 (BEST RANK k APPROXIMATION)
The solution $\tilde{M}$ to (13.2) is simply the sum of the first k terms in the previous Theorem.

The proof of this theorem uses the following, which is not too hard to prove from the spectral decomposition using definitions.

THEOREM 17 (COURANT-FISHER)
If $e_1, e_2, \ldots, e_n$ are the eigenvectors as above then:

1. *e_1 is the unit vector that maximizes $|Mx|_2^2$.*
2. *e_{i+1} is the unit vector that is orthogonal to $e_1, e_2, \ldots, e_i$ and maximizes $|Mx|_2^2$.*

Let's prove Theorem 16 for $k = 1$ by verifying that the first term of the spectral decomposition gives the best rank 1 approximation to M. A rank 1 matrix is one whose each row is a multiple of some unit vector x; in other words is on the line defined by x. Denote

the rows of M as $M_1, M_2, \ldots, M_n$. Then the multiple of x that closest to M_i is simply its projection, namely $< M_i, x > x$. Thus the matrix approximation consists of finding a unit vector x so as to minimize

$$\sum_i |M_i - < M_i, x > x|^2 = \sum_i |M_i|^2 - \sum_i |< M_i, x >|^2 .$$

This minimization is tantamount to maximising

$$\sum_i |< M_i, x >|^2 = |Mx|^2 , \tag{13.3}$$

which by the Courant-Fisher theorem happens for $x = e_1$. Thus the best rank 1 approximation to M is the matrix whose ith row is $< M_i, e_1 > e_1^T$, which of course is $\lambda_1 e_{1i} e_1^T$. Thus the rank 1 matrix approximation is $\lambda_1 e_1^T e_1$, which proves the theorem for $k = 1$. The proof of Theorem 16 for general k follows similarly by induction and is left as exercise.

13.3.1 General matrices: Singular values

Now we look at general matrices that are not symmetric. The notion of eigenvalues and eigenvectors have to be modified. The following theorem is proved similarly as in the symmetric case but with a bit more tedium.

THEOREM 18 (SINGULAR VALUE DECOMPOSITION AND BEST RANK-k-APPROXIMATION) *Every $m \times n$ real matrix has $t \leq \min\{m, n\}$ nonnegative real numbers $\sigma_1, \sigma_2, \ldots, \sigma_t$ (called singular values) and two sets of unit vectors $U = \{u_1, u_2, \ldots, u_t\}$ which are in $\Re^m$ and $V = v_1, v_2, \ldots, v_t \in \Re^n$ (all vectors are column vectors) where U, V are orthogonormal sets and*

$$u_i^T M = \sigma_i v_i \qquad \text{and} \qquad M v_i = \sigma_i u_i^T \tag{13.4}$$

Furthermore, M can be represented as

$$M = \sum_i \sigma_i u_i v_i^T . \tag{13.5}$$

The best rank k approximation to M consists of taking the first k terms of (14.2) and discarding the rest.

This solves problems (13.1) and (13.2). Next time we'll go into some detail of the algorithm for computing them. In practice you can just use matlab or another package.

13.4 View 3: Directions of Maximum Variance

The above proof of Theorem 16, especially the subcase $k = 1$ we proved, also shows yet another view of SVD which is sometimes useful in data analysis. Let us again see this in the case of symmetric matrices. Suppose we shift the given points $M_1, M_2, \ldots, M_n$ so that their mean $\frac{1}{n}\sum_i M_i$ is the origin. Then the rank-1 SVD corresponds to the direction x where the projections of the given data points —a sequence of n real numbers— have maximum *variance*. Since the mean is 0, this variance is exactly the quantity in (13.3). The second

SVD direction corresponds to directions with maximum variance after we have removed the component along the first direction, and so on.

BIBLIOGRAPHY

1. O. Alter, P. Brown, and D. Botstein. Singular value decomposition for genome-wide expression data processing and modeling. *PNAS* August 29, 2000 vol. 97 no. 18

2. Relevant chapter of Hopcroft-Kannan book on data science. (link on course website)

Chapter 14

SVD, Power method, and Planted Graph problems (+ eigenvalues of random matrices)

Today we continue the topic of low-dimensional approximation to datasets and matrices. Last time we saw the singular value decomposition of matrices.

14.1 SVD computation

Recall this theorem from last time.

THEOREM 19 (SINGULAR VALUE DECOMPOSITION AND BEST RANK-k-APPROXIMATION) *An $m \times n$ real matrix has $t \leq \min\{m, n\}$ nonnegative real numbers $\sigma_1, \sigma_2, \ldots, \sigma_t$ (called singular values) and two sets of unit vectors $U = \{u_1, u_2, \ldots, u_t\}$ which are in $\Re^m$ and $V = v_1, v_2, \ldots, v_t \in \Re^n$ (all vectors are column vectors) where U, V are orthogonormal sets and*

$$u_i^T M = \sigma_i v_i \qquad \text{and} \qquad M v_i = \sigma_i u_i^T. \tag{14.1}$$

(When M is symmetric, each $u_i = v_i$ and the σ_i's are eigenvalues and can be negative.) Furthermore, M can be represented as

$$M = \sum_i \sigma_i u_i v_i^T. \tag{14.2}$$

The best rank k approximation to M consists of taking the first k terms of (14.2) and discarding the rest (where $\sigma_1 \geq \sigma_2 \cdots \geq \sigma_r$).

Taking the best rank k approximation is also called *Principal Component Analysis* or PCA.

You probably have seen eigenvalue and eigenvector computations in your linear algebra course, so you know how to compute the PCA for symmetric matrices. The nonsymmetric

case reduces to the symmetric one by using the following observation. If M is the matrix in (14.2) then

$$MM^T = (\sum_i \sigma_i u_i v_i^T)(\sum_i \sigma_i v_i u_i^T) \quad = \sum_i \sigma_i^2 u_i u_i^T \quad \text{since } v_i^T v_j = 1 \text{ iff } i = j \text{ and 0 else.}$$

Thus we can recover the u_i's and σ_i's by computing the eigenvalues and eigenvectors of MM^T, and then recover v_i by using (14.1).

Another application of singular vectors is the *Pagerank* algorithm for ranking webpages.

14.1.1 The power method

The eigenvalue computation you saw in your linear algebra course takes at least n^3 time. Often we are only interested in the top few eigenvectors, in which case there's a method that can work much faster (especially when the matrix is *sparse*, i.e., has few nonzero entries).

As usual, we first look at the subcase of symmetric matrices. To compute the largest eigenvector of matrix M we do the following. Pick a random unit vector x. Then repeat the following a few times: replace x by Mx. We show this works under the following *Gap assumption*: There is a *gap* of γ between the the top two eigenvalues: $|\lambda_1| - |\lambda_2| = \gamma$.

The analysis is the same calculation as the one we used to analyse Markov chains. We can write x as $\sum_i \alpha_i e_i$ where e_i's are the eigenvectors and λ_i's are numbered in decreasing order by absolute value. Then t iterations produces $M^t x = \sum_i \alpha_i \lambda_i^t e_i$. Since x is a unit vector, $\sum_i \alpha_i^2 = 1$.

Since $|\lambda_i| \leq |\lambda_1| - \gamma$ for $i \geq 2$, we have

$$\sum_{i \geq 2} |\alpha_i| \left|\lambda_i^t\right| \leq n\alpha_{\max}(|\lambda_1| - \gamma)^t \quad = n \left|\lambda_1\right|^t (1 - \gamma / \left|\lambda_1\right|)^t,$$

where $\alpha_{\max}$ is the largest coefficient in magnitude.

Furthermore, since x was a random unit vector (and recalling that its projection α_1 on the fixed vector e_1 is normally distributed), the probability is at least 0.99 that $\alpha_1 > 1/(10n)$. Thus setting $t = O(\log n \left|\lambda_1\right| / \gamma)$ the components for $i \geq 2$ become miniscule and $x \approx \alpha_1 \left|\lambda_1\right|^t e_1$. Thus rescaling to make it a unit vector, we get e_1 up to some error. Then we can project all vectors to the subspace perpendicular to e_1 and continue with the process to find the remaining eigenvectors and eigenvalues.

This process works under the above gap assumption. What if the gap assumption does not hold? Say, the first 3 eigenvalues are all close together, and separated by a gap from the fourth. Then the above process ends up with some random vector in the subspace spanned by the top three eigenvectors. For real-life matrices the gap assumption often holds.

14.2 Recovering planted bisections

Now we return to the planted bisection problem, also introduced last time.

The observation in Figure 14.1 suggests that the adjacency matrix is close to a rank 2 matrix shown there: the block within S_1, S_2 have value p in each entry; the blocks between S_1, S_2 have q in each entry. This is rank 2 since it has only two distinct column vectors.

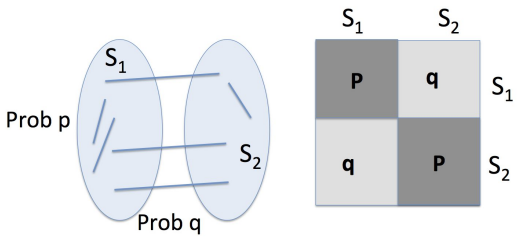

Figure 14.1: Planted Bisection problem: Edge probability is p within S_1, S_2 and q between S_1, S_2 where $q < p$. On the right hand side is the adjacency matrix. If we somehow knew S_1, S_2 and grouped the corresponding rows and columns together, and squint at the matrix from afar, we'd see more density of edges within S_1, S_2 and less density between S_1, S_2. Thus from a distance the adjacency matrix looks like a rank 2 matrix.

Now we sketch why the best rank-2 approximation to the adjacency matrix will more or less recover the planted bisection. Specifically, the idea is to find the rank 2 approximation; with very high probability its columns can be cleanly clustered into 2 clusters. This gives a grouping of the vertices into 2 groups as well, which turns out to be the planted bisection.

Why this works has to do with the properties of rank k approximations. First we define two norms of a matrix.

DEFINITION 6 (FROBENIUS AND SPECTRAL NORM) *If M is an $n \times n$ matrix then its* Frobenius norm $|M|_F$ *is* $\sqrt{\sum_{ij} M_{ij}^2}$ *and its* spectral norm $|M|_2$ *is the maximum value of $|Mx|_2$ over all unit vectors $x \in \Re^n$. (By Courant-Fisher, the spectral norm is also the highest eigenvalue.) For matrices that are not symmetric the definition of Frobenius norm is analogous and the spectral norm is the highest singular value.*

Last time we defined the *best rank k approximation to M* as the matrix $\tilde{M}$ that is rank k and minimizes $\left|M - \tilde{M}\right|_F^2$. The following theorem shows that we could have defined it equivalently using spectral norm.

LEMMA 20
Matrix $\tilde{M}$ as defined above also satisfies that $\left|M - \tilde{M}\right|_2 \leq |M - B|_2$ for all B that have rank k.

THEOREM 21
If $\tilde{M}$ is the best rank-k approximation to M, then for every rank k matrix C:

$$\left|\tilde{M} - C\right|_F^2 \leq 5k\,|M - C|_2^2\,.$$

PROOF: Follows by Spectral decomposition and Courant-Fisher theorem, and the fact that the column vectors in $\tilde{M}$ and C together span a space of dimension at most $2k$. Thus

$\left|\tilde{M} - C\right|_F^2$ involves a matrix of rank at most $2k$. Rest of the details are cut and pasted from Hopcroft-Kannan in Figure 14.2.

□

Returning to planted graph bisection, let M be the adjacency matrix of the graph with planted bisection. Let C be the rank-2 matrix that we *think* is a good approximation to M, namely, the one in Figure 14.1. Let $\tilde{M}$ be the true rank 2 approximation found via SVD. In general $\tilde{M}$ is not the same as C. But Theorem 21 implies that we can upper bound the average coordinate-wise squared difference of $\tilde{M}$ and C by the quantity on the right hand side, which is the spectral norm (i.e., largest eigenvalue) of $M - C$.

Notice, $M - C$ is a *random matrix* whose each coordinate is one of four values $1 - p, -p, 1 - q, -q$. More importantly, the expectation of each coordinate is 0 (since the entry of M is a coin toss whose expected value is the corresponding entry of C). The study of eigenvalues of such random matrices is a famous subfield of science with unexpected connections to number theory (including the famous Riemann hypothesis), quantum physics (quantum gravity, quantum chaos), etc. We show below that $|M - C|_2^2$ is at most $O(np)$. We conclude that the average column vector in $\tilde{M}$ and C (whose square norm is about np) are apart by $O(p)$. Thus intuitively, clustering the columns of C into two will find us the bipartition. Actually showing this requires more work which we will not do.

Here is a generic clustering algorithm into two clusters: Pick a random column of $\tilde{M}$ and put into one cluster all columns whose distance from it is at most $10p$. Put all other columns in the other cluster.

14.2.1 Eigenvalues of random matrices

We sketch a proof of the following classic theorem to give a taste of this beautiful area.

THEOREM 22
Let R be a random matrix such that R_{ij}'s are independent random variables in $[-1, 1]$ of expectation 0 and variance at most σ^2. Then with probability $1 - \exp(-n)$ the largest eigenvalue of R is at most $O(\sigma\sqrt{n})$.

For simplicity we prove this for $\sigma = 1$.
PROOF: Recalling that the largest eigenvalue is $\max_x \left|x^T R x\right|$, we break the proof as follows.

Idea 1) *For any fixed unit vector $x \in \Re^n$, $\left|x^T R x\right| \leq O(\sqrt{n})$ with probability $1-\exp(-Cn)$ where C is an arbitrarily large constant.* This follows from Chernoff-type bounds. Note that $x^T R x = \sum_{ij} R_{ij} x_i x_j$. By Chernoff bounds (Hoeffding's inequality) the probability that this exceeds t is at most

$$\exp(-\frac{t^2}{\sum_i x_i^2 x_j^2}) \quad \leq \exp(-\Omega(t^2)),$$

since $(\sum_{ij} x_i^2 x_j^2)^{1/2} \leq \sum_i x_i^2 = 1$.

Idea 2) *There is a set of $\exp(n)$ special directions $x_{(1)}, x_{(2)}, \ldots,$ that approximately "cover"the set of unit vectors in $\Re^n$.* Namely, for every unit vector v, there is at least one $x_{(i)}$ such that $< v, x_{(i)} > > 0.9$.

First, note that $< v, x^{(i)} > > 0.9$ iff

$$\left|v - x_{(i)}\right|^2 = |v|^2 + \left|x_{(i)}\right|^2 - 2 < v, x_{(i)} > \quad \leq 0.2.$$

In other words we are trying to cover the unit sphere with spheres of radius 0.2.

Try to pick this set greedily. Pick $x_{(1)}$ arbitrarily, and throw out the unit sphere of radius 0.2 around it. Then pick $x_{(2)}$ arbitrarily out of the remaining sphere, and throw out the unit sphere of radius 0.2 around it. And so on.

How many points did we end up with? By construction, each point that was picked has distance at least 0.2 from every other point that was picked, so the spheres of radius 0.1 around the picked points are mutually disjoint. Thus the maximum number of points we could have picked is the number of disjoint spheres of radius 0.1 in a ball of radius at most 1.1. Denoting by $B(r)$ denote the volume of spheres of volume r, this is at most $B(1.1)/B(0.1) = \exp(n)$.

Idea 3) Combining Ideas 1 and 2, and the union bound, we have with high probability, $\left|x_{(i)}^T R x_{(i)}\right| \leq O(\sqrt{n})$ for all the special directions.

Idea 4): *If v is the eigenvector corresponding to the largest eigenvalue satisfies then there is some special direction satisfying* $\left|x_{(i)}^T R x_{(i)}\right| > 0.4 v^T R v$.

By the covering property, there is some special direction $x_{(i)}$ that is close to v. Represent it as $\alpha v + \beta u$ where $u \perp v$ and u is a unit vector. So $\alpha \geq 0.9$ and $\beta \leq \sqrt{0.19} \leq 0.5$. Then $\left|x_{(i)}^T R x_{(i)}\right| = \alpha v^T R v + \beta u^T R u$. But v is the largest eigenvalue so $\left|u^T R u\right| \leq v^T R v$. We conclude $\left|x_{(i)}^T R x_{(i)}\right| \geq (0.9 - 0.5) v^T R v$, as claimed.

The theorem now follows from Idea 3 and 4. □

BIBLIOGRAPHY

1. F. McSherry. *Spectral partitioning of random graphs.* In IEEE FOCS 2001 Proceedings.

Lemma 8.7 *Suppose A is an $n \times d$ matrix and suppose C is an $n \times d$ rank k matrix. Let $\bar{A}$ be the best rank k approximation to A found by SVD. Then, $||\bar{A} - C||_F^2 \leq 5k||A - C||_2^2$.*

Proof: Let $\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k}$ be the top k singular vectors of A. Extend the set of the top k singular vectors to an orthonormal basis $\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_p}$ of the vector space spanned by the rows of $\bar{A}$ and C. Note that $p \leq 2k$ since $\bar{A}$ is spanned by $\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k}$ and C is of rank at most k. Then,

$$||\bar{A} - C||_F^2 = \sum_{i=1}^{k} |(\bar{A} - C)\mathbf{u_i}|^2 + \sum_{i=k+1}^{p} |(\bar{A} - C)\mathbf{u_i}|^2.$$

Since $\{\mathbf{u_i} | \, 1 \leq i \leq k\}$ are the top k singular vectors of A and since $\bar{A}$ is the rank k approximation to A, for $1 \leq i \leq k$, $A\mathbf{u_i} = \bar{A}\mathbf{u_i}$ and thus $\left|\left(\bar{A} - C\right)\mathbf{u_i}\right|^2 = |(A - C)\,\mathbf{u_i}|^2$. For $i > k$, $\bar{A}\mathbf{u_i} = 0$, thus $\left|\left(\bar{A} - C\right)\mathbf{u_i}\right|^2 = |C\mathbf{u_i}|^2$. From this it follows that

$$\begin{aligned} \left\|\bar{A} - C\right\|_F^2 &= \sum_{i=1}^{k} |(A - C)\mathbf{u_i}|^2 + \sum_{i=k+1}^{p} |C\mathbf{u_i}|^2 \\ &\leq k||A - C||_2^2 + \sum_{i=k+1}^{p} |A\mathbf{u_i} + (C - A)\mathbf{u_i}|^2 \end{aligned}$$

Using $|a + b|^2 \leq 2|a|^2 + 2|b|^2$

$$\begin{aligned} \left\|\bar{A} - C\right\|_F^2 &\leq k||A - C||_2^2 + 2\sum_{i=k+1}^{p} |A\mathbf{u_i}|^2 + 2\sum_{i=k+1}^{p} |(C - A)\mathbf{u_i}|^2 \\ &\leq k||A - C||_2^2 + 2\,(p - k - 1)\,\sigma_{k+1}^2(A) + 2\,(p - k - 1)\,||A - C||_2^2 \end{aligned}$$

Using $p \leq 2k$ implies $k > p - k - 1$

$$\left\|\bar{A} - C\right\|_F^2 \leq k||A - C||_2^2 + 2k\sigma_{k+1}^2(A) + 2k||A - C||_2^2. \tag{8.1}$$

As we saw in Chapter 4, for any rank k matrix B, $||A - B||_2 \geq \sigma_{k+1}(A)$ and so $\sigma_{k+1}(A) \leq ||A - C||_2$ and plugging this in, we get the Lemma. ∎

Figure 14.2: Proof of Theorem 21 from Hopcroft-Kannan book

Chapter 15

Semidefinite Programs (SDPs) and Approximation Algorithms

Recall that a set of points K is *convex* if for every two $x, y \in K$ the line joining x, y, i.e., $\{\lambda x + (1-\lambda)y : \lambda \in [0,1]\}$ lies entirely inside K. A function $f : \Re^n \to \Re$ is *convex* if $f(\frac{x+y}{2}) \leq \frac{1}{2}(f(x)+f(y))$. It is called *concave* if the previous inequality goes theother way. A linear function is both convex and concave. A *convex program* consists of a convex function f and a convex body K and the goal is to minimize $f(x)$ subject to $x \in K$. Is is a vast generalization of linear programming and like LP, can be solved in polynomial time under fairly general conditions on f, K. Today's lecture is about a special type of convex program called *semidefinite programs.*

Recall that a symmetric $n \times n$ matrix M is *positive semidefinite* (PSD for short) iff it can be written as $M = AA^T$ for some real-valued matrix A (need not be square). It is a simple exercise that this happens iff every eigenvalue is nonnegative. Another equivalent characterization is that there are n vectors $u_1, u_2, \ldots, u_n$ such that $M_{ij} = \langle u_i, u_j \rangle$. Given a PSD matrix M one can compute such n vectors in polynomial time using a procedure called *Cholesky decomposition.*

Lemma 23
The set of all $n \times n$ PSD matrices is a convex set in $\Re^{n^2}$.

Proof: It is easily checked that if M_1 and M_2 are PSD then so is $M_1 + M_2$ and hence so is $\frac{1}{2}(M_1 + M_2)$. □

Now we are ready to define semidefinite programs. These are very useful in a variety of optimization settings as well as control theory. We will use them for combinatorial optimization, specifically to compute approximations to some NP-hard problems. In this respect SDPs are more powerful than LPs.

View 1: A linear program in n^2 real valued variables Y_{ij} where $1 \leq i, j \leq n$, with the additional constraint "*Y is a PSD matrix.*"

View 2: A *vector program* where we are seeking n vectors $u_1, u_2, \ldots, u_n \in \Re^n$ such that their inner products $\langle u_i, u_j \rangle$ satisfy some set of linear constraints.

Clearly, these views are equivalent.

Exercise: Show that every LP can be rewritten as a (slightly larger) SDP. The idea is that a diagonal matrix, i.e., a matrix whose offdiagonal entries are 0, is PSD iff the entries are nonnegative.

Question: Can the vectors $u_1, \ldots, u_n$ in View 2 be required to be in $\Re^d$ for $d < n$? Answer: This is not known and imposing such a constraint makes the program nonconvex. (The reason is that the sum of two matrices of rank d can have rank higher than d.)

15.1 Max Cut

Given an n-vertex graph $G = (V, E)$ find a cut $(S, \overline{S})$ such that you maximise $E(S, \overline{S})$.

The exact characterization of this problem is to find $x_1, x_2, \ldots, x_n \in \{-1, 1\}$ (which thus represent a cut) so as to maximise

$$\sum_{\{i,j\} \in E} \frac{1}{4} |x_i - x_j|^2.$$

This works since an edge contributes 1 to the objective iff the endpoints have opposite signs.

The SDP relaxation is to find vectors $u_1, u_2, \ldots, u_n$ such that $|u_i|_2^2 = 1$ for all i and so as to maximise

$$\sum_{\{i,j\} \in E} \frac{1}{4} |v_i - v_j|^2.$$

This is a relaxation since every ± 1 solution to the problem is also a vector solution where every u_i is $\pm v_0$ for some fixed unite vector v_0.

Thus when we solve this SDP we get n vectors, then the value of the objective OPT_{SDP} is at least as large as the capacity of the max cut. How do we get a cut out of these vectors? The following is the simplest rounding one can think of. Pick a random vector z. If $\langle u_i, z \rangle$ is positive, put it in S and otherwise in $\overline{S}$. Note that this is the same as picking a random hyperplane passing through the origin and partitioning the vertices according to which side of the hyperplane they lie on.

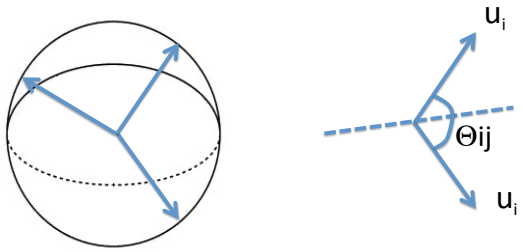

Figure 15.1: SDP solutions are unit vectors and they are rounded to ± 1 by using a random hyperplane through the origin. The probability that i, j end up on opposite sides of the cut is proportional to Θ_{ij}, the angle between them.

THEOREM 24 (GOEMANS-WILLIAMSON'94)
The expected number of edges in the cut produced by this rounding is at least 0.878.. *times* OPT_{SDP}.

PROOF: The rounding is essentially picking a random hyperplane through the origin and vertices i, j fall on opposite sides of the cut iff u_i, u_j lie on opposite sides of the hyperplane. Let's estimate the probability they end up on opposite sides. This may seem a difficult n-dimensional calculation, until we realize that there is a 2-dimensional subspace defined by u_i, u_j, and all that matters is the intercept of the random hyperplane with this 2-dimensional subspace, which is a random line in this subspace. Specifically θ_{ij} be the angle between u_i and u_j. Then the probability that they fall on opposite sides of this random line is θ_{ij}/π. Thus by linearity of expectations,

$$\mathbf{E}[\text{Number of edges in cut}] = \sum_{\{i,j\}\in E} \frac{\theta_{ij}}{\pi}. \tag{15.1}$$

How do we relate this to OPT_{SDP}? We use the fact that $\langle u_i, u_j\rangle = \cos\theta_{ij}$ to rewrite the objective as

$$\sum_{\{i,j\}\in E} \frac{1}{4}|v_i - v_j|^2 = \sum_{\{i,j\}\in E} \frac{1}{4}(|v_i|^2 + |v_j|^2 - 2\langle v_i, v_j\rangle) = \sum_{\{i,j\}\in E} \frac{1}{2}(1-\cos\theta_{ij}). \tag{15.2}$$

This seems hopeless to analyse for us mortals: we know almost nothing about the graph or the set of vectors. Luckily Goemans and Williamson had the presence of mind to verify the following in Matlab: each term of (15.1) is at least 0.878.. times the corresponding term of (15.2)! Specifically, Matlab shows that for all

$$\frac{2\theta}{\pi(1-\cos\theta)} \geq 0.878 \qquad \forall\theta \in [0,\pi]. \tag{15.3}$$

QED □

The saga of 0.878... The GW paper came on the heels of the PCP Theorem (1992) which established that there is a constant

15.2 0.878-approximation for MAX-2SAT

We earlier designed approximation algorithms for MAX-2SAT using LP. The SDP relaxation gives much tighter approximation than the 3/4 we achieved back then. Given a 2CNF formula on n variables with m clauses, we can express MAX-2SAT as a quadratic optimization problem. We want $x_i^2 = 1$ for all i (hence x_i is ± 1; where $+1$ corresponds to setting the variable y_i to true) and we can write a quadratic expression for each clause expressing that it is satisfied. For instance if the clause is $y_i \vee y_j$ then the expression is $1 - \frac{1}{4}(1-x_i)(1-x_j)$. It is 1 if either of x_i, x_j is 1 and 0 else.

Representing this expression directly as we did for MAX-CUT is tricky because of the "1" appearing in it. Instead we are going to look for $n+1$ vectors $u_0, u_1, \ldots, u_n$. The first

vector u_0 is a dummy vector that stands for "1". If $u_i = u_0$ then we think of this variable being set to True and if $u_i = -u_0$ we think of the variable being set to False. Of course, in general $\langle u_i, u_0 \rangle$ need not be ± 1 in the optimum solution.

So the SDP is to find these vectors satisfying $|u_i|^2 = 1$ for all i so as to maximize $\sum_{clausel} v_l$ where v_l is the expression for lth clause. For instance if the clause is $y_i \vee y_j$ then the expression is

$$1 - \frac{1}{4}(u_0 - u_i)(u_0 - u_j) = \frac{1}{4}(1 + u_0 \cdot u_j) + \frac{1}{4}(1 + u_0 \cdot u_i) + \frac{1}{4}(1 - u_i \cdot u_j).$$

This is a very Goemans-Williamson like expression, except we have expressions like $1 + u_0 \cdot u_i$ whereas in MAX-CUT we have $1 - u_i \cdot u_j$. Now we do Goemans-Williamson rounding. The key insight is that since we round to ± 1 each term $1 + u_i \cdot u_j$ becomes 2 with probability $1 - \frac{\theta_{ij}}{\pi} = \frac{\pi - \theta_{ij}}{\pi}$ and is 0 otherwise. Similarly, $1 - u_i \cdot u_j$ becomes 2 with probability θ_{ij}/π and 0 else.

Now the term-by-term analysis used for MAX-CUT works again once we realize that (15.3) also implies (by substituting $\pi - \theta$ for θ in the expression) that $\frac{2(\pi - \theta)}{\pi(1 + \cos\theta)} \geq 0.878$ for $\theta \in [0, \pi]$. We conclude that the expected number of satisfied clauses is at least 0.878 times OPT_{SDP}.

Chapter 16

Going with the slope: offline, online, and randomly

This lecture is about *gradient descent*, a popular method for continuous optimization (especially nonlinear optimization).

We start by recalling that allowing nonlinear constraints in optimization leads to NP-hard problems in general. For instance the following single constraint can be used to force all variables to be 0/1.

$$\sum_i x_i^2(1-x_i)^2 = 0.$$

Notice, this constraint is nonconvex. We saw in earlier lectures that the Ellipsoid method can solve *convex* optimization problems efficiently under fairly general conditions. But it is slow in practice.

Gradient descent is a popular alternative because it is simple and it gives some kind of meaningful result for both *convex* and *nonconvex* optimization. It tries to improve the function value by moving in a direction related to the *gradient* (i.e., the first derivative). For convex optimization it gives the global optimum under fairly general conditions. For nonconvex optimization it arrives at a local optimum.

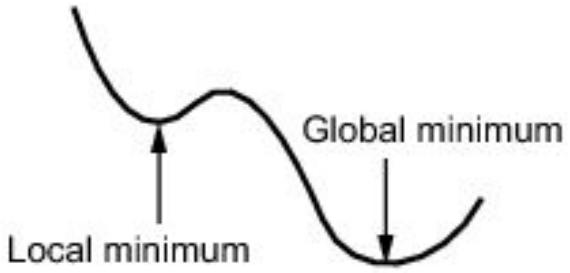

Figure 16.1: For nonconvex functions, a local optimum may be different from the global optimum

We will first study unconstrained gradient descent where we are simply optimizing a function $f(\cdot)$. Recall that the function is *convex* if $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ for all x, y and $\lambda \in [0,1]$.

16.1 Gradient descent for convex functions: univariate case

The gradient for a univariate function f is simply the derivative: $f'(x)$. If this is negative, the value decreases if we increase x a little, and increases if we decrease f. Gradient descent consists of evaluating the derivative and moving a small amount to the right (i.e., increase x) if $f'(x) < 0$ and to move to the left otherwise. Thus the basic iteration is $x \leftarrow x - \eta f'(x)$ for a tiny η called *step size.*

The function is *convex* if between every two points x, y the graph of the function lies *below* the line joining $(x, f(x))$ and $(y, f(y))$. It need not be differentiable everywhere but when all derivatives exist we can do the *Taylor expansion*:

$$f(x+\eta) = f(x) + \eta f'(x) + \frac{\eta^2}{2} f''(x) + \frac{\eta^3}{3!} f'''(x) \cdots . \tag{16.1}$$

If $f''(x) \geq 0$ for all x then the the function is *convex.* This is because $f'(x)$ is an *increasing* function of x. The minimum is attained for x where $f'(x) = 0$ since $f'(x)$ is +ve to the right of it and −ve to the left. Thus moving both left and right of this point increases f and it never drops. The function is *concave* if $f''(x) \leq 0$ for all x; such functions have a unique maximum.

Examples of convex functions: $ax + b$ for any $a, b \in \Re$; $\exp(ax)$ for any $a \in \Re$; x^α for $x \geq 0$, $\alpha \geq 1$ or $\alpha \leq 0$. Another interesting example is the negative entropy: $x \log x$ for $x \geq 0$.

Examples of concave functions: $ax + b$ for any $a, b \in \Re$; x^α for $\alpha \in [0,1]$ and $x \geq 0$; $\log x$ for $x \geq 0$.

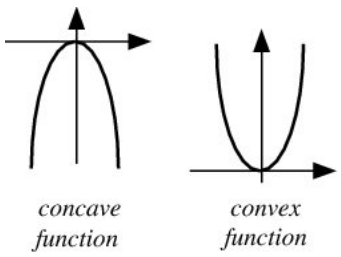

Figure 16.2: Concave and Convex Function

To minimize a convex function by gradient descent we start at some x_0 and at step i update x_i to $x_{i+1} = x_i + \eta f'(x)$ for some small $\eta < 0$. In other words, move in the direction where f *decreases.* If we ignore terms that involve η^3 or higher, then

$$f(x_{i+1}) = f(x_i) + \eta f'(x_i) + \frac{\eta^2}{2} f''(x_i).$$

and the best value for η (which gives the most reduction in one step) is $\eta = -f'(x)/2f''(x)$, which gives

$$f(x_{i+1}) = f(x_i) - \frac{(f'(x_i))^2}{2f''(x_i)}.$$

Thus the algorithm makes progress so long as $f''(x_i) > 0$. Convex functions that satisfy $f''(x) > 0$ for all x are called *strongly convex*.

The above calculation is the main idea in *Newton's method*, which you may have seen in calculus. Proving convergence requires further assumptions.

16.2 Convex multivariate functions

A convex function on $\Re^n$, if it is differentiable, satisfies the following basic inequality, which says that the function lies "above "the tangent plane at any point.

$$f(x+z) \geq f(x) + \nabla f(x) \cdot z \quad \forall x, y. \tag{16.2}$$

Here $\nabla f(x)$ is the vector of first order derivatives where the ith coordinate is $\partial f/\partial x_i$ and called the *gradient*. Sometimes we restate it equivalently as

$$f(x) - f(y) \leq \nabla f(x) \cdot (x - y) \quad \forall x, z \tag{16.3}$$

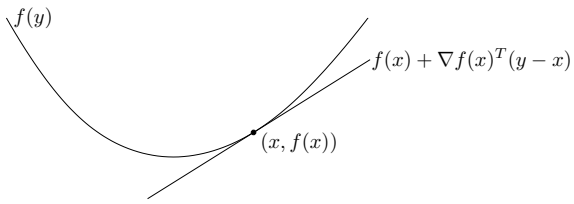

Figure 16.3: A differentiable convex function lies above the tangent plane $f(x) + \nabla f(x) \cdot (y - x)$

If higher derivatives also exist, the multivariate Taylor expansion for an n-variate function f is

$$f(x+y) = f(x) + \nabla f(x) \cdot y + y^T \nabla^2 f(x) y + \cdots . \tag{16.4}$$

Here $\nabla^2 f(x)$ denotes the $n \times n$ matrix whose i, j entry is $\partial^2 f/\partial x_i \partial x_j$ and it is called the *Hessian*. It can be checked that f is *convex* if the Hessian is positive semidefinite; this means $y^T \nabla^2 f y \geq 0$ for all y.

EXAMPLE 29 The following are some examples of convex functions.

- *Norms.* Every ℓ_p norm is convex on $\Re^n$. The reason is that a norm satisfies triangle inequality: $|x + y| \leq |x| + |y|$ $\forall x, y$.

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Figure 16.4: The Hessian

- $f(x) = \log(e^{x_1} + e^{x_2} + \cdots + e^{x_n})$ is convex on $\Re^n$. This fact is used in practice as an analytic approximation of the max function since

  $$\max\{x_1, \ldots, x_n\} \leq f(x) + \leq \max\{x_1, \ldots, x_n\} + \log n.$$

  Turns out this fact is at the root of the multiplicative weight update method; the algorithm for approximately solving LPs that we saw in Lecture 10 can be seen as doing a gradient descent on this function, where the x_i's are the *slacks* of the linear constraints. (For a linear constraint $a^T z \geq b$ the slack is $a^T z - b$.)

- $f(x) = x^T A x = \sum_{ij} A_{ij} x_i x_j$ where A is positive semidefinite. Its Hessian is A.

Some important examples of concave functions are: *geometric mean* $(\prod_{i=1}^n x_i)^{1/n}$ and *log-determinant* (defined for $X \in \Re^{n^2}$ as $\log\det(X)$ where X is interpreted as an $n \times n$ matrix).

Many famous inequalities in mathematics (such as Cauchy-Schwartz) are derived using convex functions. □

Example 30 (Linear equations with PSD constraint matrix) In linear algebra you learnt that the method of choice to solve systems of equations $Ax = b$ is Gaussian elimination. In many practical settings its $O(n^3)$ running time may be too high. Instead one does gradient descent on the function $\frac{1}{2}x^T A x - b^T x$, whose local optimum satisfies $Ax = b$. If A is positive semidefinite this function is also convex since the Hessian is A, and gradient descent will actually find the solution. (Actually in real life these are optimized using more advanced methods such as *conjugate gradient*.) Also, if A is *diagonal dominant*, a stronger constraint than PSD, then Spielman and Teng (2003) have shown how to solve this problem in time that is *near linear* in the number of nonzero entries. This has had surprising applications to basic algorithmic problems like max-flow.

Example 31 (Least squares) In some settings we are given a set of points $a_1, a_2, \ldots, a_m \in \Re^n$ and some *data values* $b_1, b_2, \ldots, b_m$ taken at these points by some function of interest. We suspect that the unknown function is a *line*, except the data values have a little error in them. One standard technique is to find a *least squares* fit: a line that minimizes the sum of squares of the distance to the datapoints to the line. The objective function is $\min |Ax - b|_2^2$ where $A \in \Re^{m \times n}$ is the matrix whose rows are the a_i's. (We saw in an earlier lecture that the solution is also the first singular vector.) This objective is just $x^T A^T A x - 2(Ax)^T b + b^T b$, which is convex.

In the univariate case, gradient descent has a choice of only two directions to move in: *right* or *left*. In n dimensions, it can move in any direction in $\Re^n$. The most direct analog of the univariate method is to move diametrically opposite from the *gradient*.

The most direct analogue of our univariate analysis would be to assume a *lowerbound* of $y^T \nabla^2 f y$ for all y (in other words, a lowerbound on the eigenvalues of $\nabla^2 f$). This will be explored in the homework. In the rest of lecture we will only assume (16.2).

16.3 Gradient Descent for Constrained Optimization

As studied in previous lectures, constrained optimization consists of solving the following where $\mathcal{K}$ is a convex set and $f(\cdot)$ is a convex function.

$$\min f(x) \quad \text{s.t.} \quad x \in \mathcal{K}.$$

Example 32 (Spam classification via SVMs) This example will run through the entire lecture. Support Vector Machine is the name in machine learning for a *linear classifier*; we saw these before in Lecture 6 (Linear Thinking). Suppose we wish to train the classifier to classify emails as spam/nonspam. Each email is represented using a vector in $\Re^n$ that gives the frequencies of various words in it ("bag of words"model). Say $a_1, a_2, \ldots, a_N$ are the emails, and for each there is a corresponding bit $b_i \in \{-1, 1\}$ where $b_i = 1$ means X_i is spam. SVMs use a *linear classifier* to separate spam from nonspam. If spam were perfectly identifiable by a linear classfier, there would be a function $W \cdot x$ such that $W \cdot a_i \geq 1$ if a_i is spam, and $W \cdot a_i \leq -1$ if not. In other words,

$$1 - b_i W \cdot a_i \leq 0 \quad \forall i \tag{16.5}$$

Of course, in practice a linear classifier makes errors, so we have to allow for the possibility that (16.5) is violated by some a_i's. The obvious thing to try is to find a W that satisfies as many of the constraints as possible, but that leads to a nonconvex NP-hard problem. (Even approximating this weakly is NP-hard.) Thus a more robust version of this problem is

$$\min \sum_i \text{Loss}(1 - W \cdot (b_i a_i)) \tag{16.6}$$
$$|W|_2^2 \leq n \qquad \text{(scaling constraint)}$$

where Loss$(\cdot)$ is a function that penalizes unsatisfied constraints according to the amount by which they are unsatisfied. (Note that W is the vector of variables, and the scaling constraint gives meaning to the separation of "1 "in (16.5) by saying that W is a vector in the sphere of radius n, which is a convex constraint.) The most obvious loss function would be to count the *number of unsatisfied constraints* but that is nonconvex. For this lecture we focus on convex loss functions; the simplest is the *hinge loss*: Loss$(t) = \max\{0, t\}$. Applying it to $1 - W \cdot (b_i a_i)$ insures that correctly classified emails contribute 0 to the loss, and incorrectly classified emails contribute as much to the loss as the amount by which they fail the inequality. The function in (16.6) is convex because the function inside Loss() is linear and thus convex, and Loss() preserves convexity since it can only lift the value of the linear function even further.

If $x \in \mathcal{K}$ is the current point and we use the gradient to step to $x - \eta \triangle x$ then in general this new point will not be in $\mathcal{K}$. Thus one needs to do a *projection*.

DEFINITION 7 *The* projection *of a point y on $\mathcal{K}$ is $x \in \mathcal{K}$ that minimizes $|y - x|_2$. (It is also possible to use other norms than ℓ_2 to define projections.)*

A projection oracle *for the convex body a black box that, for every point y, returns its projection on $\mathcal{K}$.*

Often convex sets used in applications are simple to project to.

EXAMPLE 33 If $\mathcal{K}$ = unit sphere, then the projection of y is $y/|y|_2$.

Here is a simple algorithm for solving the constrained optimization problem. The algorithm only needs to access f via a *gradient oracle* and $\mathcal{K}$ via a *projection oracle*.

DEFINITION 8 (GRADIENT ORACLE) *A* gradient oracle *for a function f is a black box that, for every point z, returns $\nabla f(z)$ the gradient valuated at point z. (Notice, this is a linear function of the form gx where g is the vector of partial derivatives evaluated at z.)*

The same value of η will be used throughout.

GRADIENT DESCENT FOR CONSTRAINED OPTIMIZATION
Let $\eta == \frac{D}{G\sqrt{T}}$.
Repeat for $i = 0$ **to** T
$y^{(i+1)} \leftarrow x^{(i)} - \eta \nabla f(x^{(i)})$
$x^{(i+1)} \leftarrow$ Projection of $y^{(i+1)}$ on $\mathcal{K}$.
At the end output $z = \frac{1}{T}\sum_i x^{(i)}$.

Let us analyse this algorithm as follows. Let x^* be the point where the optimum is attained. Let G denote an upperbound on $|\nabla f(x)|_2$ for any $x \in \mathcal{K}$, and let $D = max_{x,y \in \mathcal{K}} |x - y|_2$ be the so-called *diameter* of $\mathcal{K}$. To ensure that the output z satisfies $f(z) \leq f(x^*) + \varepsilon$ we will use $T = \frac{4D^2G^2}{\varepsilon^2}$.

Since $x^{(i)}$ is a projection of $y^{(i)}$ on $\mathcal{K}$ we have

$$\begin{aligned}\left|x^{(i+1)} - x^*\right|^2 &\leq \left|y^{(i+1)} - x^*\right|^2 \\ &= \left|x^{(i)} - x^* - \eta \nabla f(x^{(i)})\right|^2 \\ &= \left|x^{(i)} - x^*\right|^2 + \eta^2 \left|\nabla(f)(x^{(i)})\right|^2 - 2\eta \nabla f(x^{(i)}) \cdot (x^{(i)} - x^*)\end{aligned}$$

Reorganizing and using definition of G we obtain:

$$\nabla f(x^{(i)}) \cdot (x^* - x^{(i)}) \leq \frac{1}{2\eta}\left(\left|x^{(i)} - x^*\right|^2 - \left|x^{(i+1)} - x^*\right|^2\right) + \frac{\eta}{2}G^2$$

Using (16.3), we can lowerbound the left hand side by $f(x^{(i)}) - f(x^*)$. We conclude that

$$f(x^{(i)}) - f(x^*) \leq \frac{1}{2\eta}\left(\left|x^{(i)} - x^*\right|^2 - \left|x^{(i+1)} - x^*\right|^2\right) + \frac{\eta}{2}G^2. \tag{16.7}$$

Now sum the previous inequality over $i = 1, 2, \ldots, T$ and use the telescoping cancellations to obtain

$$\sum_{i=1}^{T}(f(x^{(i)}) - f(x^*)) \leq \frac{1}{2\eta}(\left|x^{(0)} - x^*\right|^2 - \left|x^{(T)} - x^*\right|^2) + \frac{T\eta}{2}\left|G\right|^2.$$

Finally, by convexity $f(\frac{1}{T}\sum_i x^{(i)}) \leq \frac{1}{T}\sum_i f(x^{(i)})$ so we conclude that the point $z = \frac{1}{T}\sum_i x^{(i)}$ satisfies

$$f(z) - f(z^*) \leq \frac{D^2}{2\eta T} + \frac{\eta}{2}G^2.$$

Now set $\eta = \frac{D}{G\sqrt{T}}$ to get an upperbound on the right hand side of $2\frac{DG}{\sqrt{T}}$. Since $T = \frac{4D^2G^2}{\varepsilon^2}$ we see that $f(z) \leq f(x^*) + \varepsilon$.

16.4 Online Gradient Descent

In online gradient descent we deal with the following scenario. There is a convex set $\mathcal{K}$ given via a projection oracle. For $i = 1, 2, \ldots, T$ we are presented at step i a convex function f_i. At step i we have to put forth our *guess* solution $x^{(i)} \in \mathcal{K}$ but the catch is that we do not know the functions that will be presented in future. So our online decisions have to be made such that if x^* is the point w that minimizes $\sum_i f_i(w)$ (i.e. the point that we would have chosen in *hindsight* after all the functions were revealed) then the following quantity (called *regret*) should stay small:

$$\sum_i f_i(x^{(i)}) - f_i(x^*).$$

This notion should remind you of multiplicative weights, except here we may have general convex functions as "payoffs."

Example 34 (Spam classification against adaptive adversaries) We return to the spam classification problem of Example 32, with the new twist that this classifier *changes* over time, as spammers learn to evade the current classifier. Thus there is no *fixed* distribution of spam emails and it is fruitless to train the classifier at one go. It is better to have it improve and adapt itself as new emails arrive. At step t the optimum classifier f_t may not be known and is presented using a gradient oracle. This function just corresponds to the term in (16.6) corresponding to the latest email that was classfied as spam/nonspam. The goal is to do as well as the best single classfier we would want to use in hindsight.

Zinkevich noticed that the analysis of gradient descent applies to this much more general scenario. Specifically, modify the above gradient descent algorithm to this problem by replacing $\nabla f(x^{(i)})$ by $\nabla f_i(x^{(i)})$. This algorithm is called *Online Gradient Descent*. The earlier analysis works essentially unchanged, once we realize that the left hand side of (16.7) has the regret for trial i. Summing over i gives the total regret on the left side, and the right hand side is analysed and upperbounded as before. Thus we have shown:

THEOREM 25 (ZINKEVICH 2003)
If D is the diameter of K and G is an upperbound on the norm of the gradient of any of the presented functions, and η is set to $\frac{D}{G\sqrt{T}}$ then the regret per step after T steps is at most $\frac{2DG}{\sqrt{T}}$.

16.5 Stochastic Gradient Descent

Stochastic gradient descent is a variant of the algorithm in Section 16.3 that works with convex functions presented using an even weaker notion: an *expected gradient* oracle. Given a point z, this oracle returns a linear function $gx + f$ that is drawn from a probability distribution $\mathcal{D}_z$ such that the expectation $E_{g,f\in\mathcal{D}_z}[gx + f]$ is exactly the gradient of f at z.

EXAMPLE 35 (SPAM CLASSIFICATION USING SGD) Returning to the spam classification problem of Example 32, we see that the function in (16.6) is a sum of many similar terms. If we randomly pick a single term and compute just its gradient (which is very quick to do!) then by linearity of expectations, the expectation of this gradient is just the true gradient. Thus the expected gradient oracle may be a much faster computation than the gradient oracle (a million times faster if the number of email examples is a million!). In fact this setting is not atypical; often the convex function of interest is a sum of many similar terms.

Stochastic gradient descent can be analysed using *Online Gradient Descent* (OGD). Let $g_i \cdot x$ be the gradient at step i. Then we use this function —which is a linear function and hence convex— as f_i in the ith step of OGD. Let $z = \frac{1}{T}\sum_{i=1}^{T} x^{(i)}$. Let x^* be the point in $\mathcal{K}$ where f attains its minimum value.

THEOREM 26
$\mathbf{E}[f(z)] \leq f(x^*) + \frac{2DG}{\sqrt{T}}$, *where D is the diameter as before and G is an upperbound of the norm of any gradient vector ever output by the oracle.*

PROOF:

$$\begin{aligned}
\mathbf{E}[f(z) - f(x^*)] &\leq \frac{1}{T}\mathbf{E}[\sum_i (f(x^{(i)}) - f(x^*))] && \text{by convexity of } f \\
&\leq \frac{1}{T}\sum_i \mathbf{E}[\nabla f(x^{(i)}) \cdot (x^{(i)} - x^*)] && \text{using (16.2)} \\
&= \frac{1}{T}\sum_i \mathbf{E}[g_i \cdot (x^{(i)} - x^*)] && \text{Since expected gradient is the true gradient} \\
&= \frac{1}{T}\sum_i \mathbf{E}[f_i(x^{(i)}) - f_i(x^*)] && \text{Defn. of } f_i \\
&= \frac{1}{T}\mathbf{E}[\sum_i (f_i(x^{(i)}) - f_i(x^*)]
\end{aligned}$$

and the theorem now follows since the expression in the $\mathbf{E}[\cdot]$ is just the regret, which is *always* upperbounded by the quantity given in Zinkevich's theorem, so the same upperbound holds also for the expectation. □

16.6 Portfolio Management via Online gradient descent

(*This was actually covered at the start of Lecture 17*)

Let's return to the portfolio management problem discussed in context of multiplicative weights. We are trying to invest in a set of n stocks and maximise our wealth. For $t = 1, 2, \ldots,$ let $r^{(t)}$ be the vector of relative price increase on day t, in other words

$$r_i^{(t)} = \frac{\text{Price of stock } i \text{ on day } t}{\text{Price of stock } i \text{ on day } t-1}.$$

Some thought shows (confirming conventional wisdom) that it can be very suboptimal to put all money in a single stock. A strategy that works better in practice is *Constant Rebalanced Portfolio* (CRB): decide upon a *fixed* proportion of money to put into each stock, and buy/sell individual stocks each day to maintain this proportion.

EXAMPLE 36 Say there are only two assets, *stocks* and *bonds*. One CRB strategy is to put split money equally between these two. Notice what this implies: if an asset's price falls, you tend to *buy more of it*, and if the price rises, you tend to *sell it*. Thus this strategy roughly implements the age-old advice to "buy low, sell high."Concretely, suppose the prices each day fluctuate as follows.

	Stock $r^{(t)}$	Bond $r^{(t)}$
Day 1	4/3	3/4
Day 2	3/4	4/3
Day 3	4/3	3/4
Day 4	3/4	4/3
...	...	...

Note that the prices go up and down by the same ratio on alternate days, so money parked fully in stocks or fully in bonds earns nothing in the long run. (Aside: This kind of fluctuation is not unusual; it is generally observed that bonds and stocks move in opposite directions.) And what happens if you split your money equally between these two assets? Each day it increases by a factor $0.5 \times (4/3 + 3/4) = 0.5 \times 25/12 \approx 1.04$. Thus your money grows exponentially!

Exercise: Modify the price increases in the above example so that keeping all money in stocks or bonds alone will cause it to drop exponentially, but the 50-50 CRB increases money at an exponential rate.

CRB uses a fixed split among n assets, but what is this split? Wouldn't it be great to have an angel whisper in our ears on day 1 what this magic split is? Online optimization is precisely such an angel. Suppose the algorithm uses the vector $x^{(t)}$ at time t; the ith coordinate gives the proportion of money in stock i at the start of the tth day. Then the

algorithm's wealth increases on t by a factor $r^{(t)} \cdot x^{(t)}$. Thus the goal is to find $x^{(t)}$'s to maximize the final wealth, which is

$$\prod_t r^{(t)} \cdot x^{(t)}.$$

Taking logs, this becomes

$$\sum_t \log(r^{(t)} \cdot x^{(t)}) \tag{16.8}$$

For any fixed $r^{(1)}, r^{(2)}, \ldots$ this function happens to be concave, but that is fine since we are interested in maximization. Now we can try to run online gradient descent on this objective. By Zinkevich's theorem, the quantity in (16.8) converges to

$$\sum_t \log(r^{(t)} \cdot x^*) \tag{16.9}$$

where x^* is the best money allocation in hindsight.

This analysis needs to assume very little about the $r^{(t)}$'s, except a bound on the norm of the gradient at each step, which translates into a weak condition on price movements. In the next homework you will apply this simple algorithm on real stock data.

16.7 Hints of more advanced ideas

Gradient descent algorithms come in dozens of flavors. (The Boyd-Vandenberghe book is a good resource. and Nesterov's lecture notes are terser but still have a lot of intuition.)

We know the optimal running time (i.e., number of iterations) of gradient descent in the oracle model; see the books by Hazan and Bubeck.

Surprisingly, just going along the gradient (more precisely, diametrically opposite direction from gradient) is not always the best strategy. *Steepest descent* direction is defined by quantifying the best decrease in the objective function obtainable via a step of unit length. The catch is that different norms can be used to define "unit length."For example, if distance is measured using ℓ_1 norm, then the best reduction happens by picking the largest coordinate of the gradient vector and reducing the corresponding coordinate in x (*coordinate descent*). The classical *Newton method* is a subcase where distance is measured using the *ellipsoidal norm* defined using the *Hessian*.

Gradient descent ideas underlie recent advances in algorithms for problems like Spielman-Teng style solver for Laplacian systems, near-linear time approximation algorithms for maximum flow in undirected graphs, and Madry's faster algorithm for maximum weight matching.

Bibliography

1. *Convex Optimization*, S. Boyd and L. Vandenberghe. Cambridge University Press. (pdf available online.)

2. *Introductory Lectures on Convex Optimization: A Basic Course.* Y. Nesterov. Springer 2004.

3. *Online Convex Programming and Generalized Infinitesimal Gradient Ascent.* M. Zinkevich, ICML 2003.

4. Book draft *Online convex optimization.* Elad Hazan.

5. Lecture notes on online optimization. S. Bubeck.

Chapter 17

Oracles, Ellipsoid method and their uses in convex optimization

> **Oracle:** *A person or agency considered to give wise counsel or prophetic predictions or precognition of the future, inspired by the gods.*

Recall that Linear Programming is the following problem:

$$\begin{aligned} \text{maximize } & c^T x \\ & Ax \leq b \\ & x \geq 0 \end{aligned}$$

where A is a $m \times n$ real constraint matrix and $x, c \in \mathbf{R}^n$. Recall that if the number of bits to represent the input is L, a polynomial time solution to the problem is allowed to have a running time of $poly(n, m, L)$.

The Ellipsoid algorithm for linear programming is a specific application of the ellipsoid method developed by Soviet mathematicians Shor(1970), Yudin and Nemirovskii(1975). Khachiyan(1979) applied the ellipsoid method to derive the first polynomial time algorithm for linear programming. Although the algorithm is theoretically better than the Simplex algorithm, which has an exponential running time in the worst case, it is very slow practically and not competitive with Simplex. Nevertheless, it is a very important theoretical tool for developing polynomial time algorithms for a large class of convex optimization problems, which are much more general than linear programming.

In fact we can use it to solve convex optimization problems that are even too large to write down.

17.1 Linear programs too big to write down

Often we want to solve linear programs that are too large to even write down in polynomial time.

EXAMPLE 37 Semidefinite programming (SDP) uses the convex set of PSD matrices in $\Re^n$. This set is defined by the following infinite set of constraints: $a^T X a \geq 0 \quad \forall a \in Re^n$. This is really a linear constraint on the X_{ij}'s:

$$\sum_{ij} X_{ij} a_i a_j \geq 0.$$

Thus this set is defined by *infinitely many* linear constraints.

EXAMPLE 38 (HELD-KARP RELAXATION FOR TSP) In the traveling salesman problem (TSP) we are given n points and *distances* d_{ij} between every pair. We have to find a salesman tour, which is a sequence of hops among the points such that each point is visited exactly once and the total distance covered is minimized.

An *integer programming* formulation of this problem is:

$$\begin{array}{c} \min \sum_{ij} d_{ij} X_{ij} \\ X_{ij} \in \{0,1\} \quad \forall i,j \\ \sum_{i \in S, j \in \overline{S}} X_{ij} \geq 2 \qquad \forall S \subseteq V, \quad S \neq \emptyset, V \qquad \text{(subtour elimination)} \end{array}$$

The last constraint is needed because without it the solution could be a disjoint union of subtours, and hence these constraints are called *subtour elimination constraints*. The Held-Karp relaxation relaxes the first constraint to $0 \leq X_{ij} \leq 1$. Now this is a linear program, but it has $2^n + n^2$ constraints! We cannot afford to write them down (for then we might as well use the trivial exponential time algorithm for TSP).

Clearly, we would like to solve such large (or infinite) programs, but we need a different paradigm than the usual one that examines the entire input.

17.2 A general formulation of convex programming

A convex set $\mathcal{K}$ in $\Re^n$ is a subset such that for every $x, y \in \mathcal{K}$ and $\lambda \in [0,1]$ the point $\lambda x + (1-\lambda)y$ is in $\mathcal{K}$. (In other words, the line joining x, y lies in $\mathcal{K}$.) If it is compact and bounded we call it a *convex body*. It follows that if $\mathcal{K}_1, \mathcal{K}_2$ are both convex bodies then so is $\mathcal{K}_1 \cap \mathcal{K}_2$.

A general formulation of convex programming is

$$\begin{array}{c} \min c^T x \\ x \in \mathcal{K} \end{array}$$

where $\mathcal{K}$ is a convex body.

EXAMPLE 39 Linear programming is exactly this problem where $\mathcal{K}$ is simply the polytope defined by the constraints.

EXAMPLE 40 Some lectures ago we were interested in semidefinite programming, where $\mathcal{K}$ = set of PSD matrices. This is convex since if X, Y are psd matrices then so is $(X+Y)/2$. The set of PSD matrices is a convex set but extends to ∞. In the examples last time it

was finite since we had a constraint like $X_{ii} = 1$ for all i, which implies that $|X_{ij}| \leq 1$ for all i, j. Usually in most settings of interest we can place some *a priori* upper bound on the desired solution that ensures $\mathcal{K}$ is a finite body.

In fact, since we can use binary search to reduce optimization to decision problem, we can replace the objective by a constraint $c^T x \geq c_0$. Then we are looking for a point in the convex body $\mathcal{K} \cap \{x : c^T x \geq c_0\}$, which is another convex body $\mathcal{K}'$. We conclude that convex programming boils down to finding a *single point* in a convex body (where we may repeat this basic operation multiple times with different convex bodies).

Here are other examples of convex sets and bodies.

1. The whole space $\mathbf{R}^n$ is trivially an infinite convex set.
2. Hypercube length l is the set of all x such that $0 \leq x_i \leq l, 1 \leq i \leq n$.
3. Ball of radius r around the origin is the set of all x such that $\sum_{i=1}^{n} x_i^2 \leq r^2$.

17.2.1 Presenting a convex body: separation oracles

We need a way to work with a convex body $\mathcal{K}$ without knowing its full description. The simplest way to present a body to the algorithm is via a *membership oracle*: a blackbox program that, given a point x, tells us if $x \in \mathcal{K}$. We will work with a stronger version of the oracle, which relies upon the following fact.

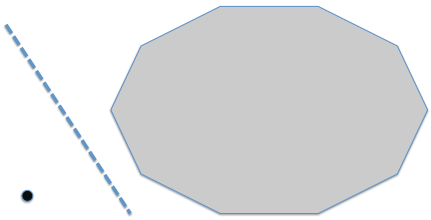

Figure 17.1: Farkas's Lemma: Between every convex body and a point outside it, there's a hyperplane

Farkas's Lemma: If $\mathcal{K} \subseteq \mathbf{R}^n$ is a convex set and $p \in \mathbf{R}^n$ is a point, then one of the following holds
(i) $p \in \mathcal{K}$
(ii) there is a hyperplane that separates p from $\mathcal{K}$. (Recall that a hyperplane is the set of points satisfying a linear equation of the form $ax = b$ where $a, x, b \in \mathbf{R}^n$.)

This Lemma is intuitively clear but the proof takes a little formal math and is omitted.

This prompts the following definition of a polynomial time Separating Oracle.

DEFINITION 9 *A polynomial time* **Separation Oracle** *for a convex set $\mathcal{K}$ is a procedure which given p, either tells that $p \in K$ or returns a hyperplane that separates p and all of K. The procedure runs in polynomial time.*

EXAMPLE 41 Consider the polytope defined by the Held-Karp relaxation. We are given a candidate solution $P = (P_{ij})$. Suppose $P_{12} = 1.1$. Then it violates the constraint $X_{12} \leq 1$, and thus the hyperplane $X_{12} = 1$ separates the polytope from P.

Thus to check that it lies in the polytope defined by all the constraints, we first check that $\sum_j P_{ij} = 2$ for all i. This can be done in polynomial time. If the equality is violated for any i then that is a separating hyperplane.

If all the other constraints are satisfied, we finally turn to the subtour elimination constraints. We construct the weighted graph on n nodes where the weight of edge $\{i, j\}$ is P_{ij}. We compute the minimum cut in this weighted graph. The subtour elimination constraints are all satisfied iff the minimum cut $S, \overline{S}$ has capacity ≥ 2. If the mincut $S, \overline{S}$ has capacity less than 2 then the hyperplane

$$\sum_{i \in S, j \in \overline{S}} X_{ij} = 2,$$

has P on the < 2 side and the Held-Karp polytope on the ≥ 2 side.

Thus you can think of a separation oracle as providing a "letter of rejection"to the point outside it explaining why it is not in the body K.

EXAMPLE 42 For the set of PSD matrices, the separation oracle is given a matrix P. It computes eigenvalues and eigenvectors to check if P only has nonnegative eigenvalues. If not, then it takes an eigenvector a corresponding to a negative eigenvalue and returns the hyperplane $\sum_{ij} X_{ij} a_i a_j = 0$. (Note that a_i's are constants here.) Then the PSD matrices are on the ≥ 0 side and P is on the < 0 side.

17.3 Ellipsoid Method

The Ellipsoid algorithm solves the basic problem of finding a point in a convex body $\mathcal{K}$. The basic idea is *divide and conquer*. At each step the algorithm asks the separation oracle about a particular point p. If p is in $\mathcal{K}$ then the algorithm can declare success. Otherwise the algorithm is able to divide the space into two (using the hyperplane provided by the separation oracle) and recurse on the correct side. (To quote the classic GLS text: *How do you catch a lion in the Sahara? Fence the Sahara down the middle. Gaze on one side and see if you spot the lion on the left. If so, continue on the left side, else continue on the right.*)

The only problem is to make sure that the algorithm makes progress at every step. After all, space is infinite and the body could be anywhere it. Cutting down an infinite set into two still leaves infinite sets. For this we use the notion of the *containing Ellipsoid* of a convex body.

An *axis aligned ellipsoid* is the set of all x such that

$$\sum_{i=1}^{n} \frac{x_i^2}{\lambda_i^2} \leq 1,$$

where λ_i's are nonzero reals. in $3D$ this is an egg-like object where a_1, a_2, a_3 are the radii along the three axes (see Figure 17.2). A *general ellipsoid* in $\mathbf{R}^n$ can be represented as

$$(x-a)^T B(x-a) \leq 1,$$

where B is a positive semidefinite matrix. (Being positive semidefinite means B can be written as $B = AA^T$ for some $n \times n$ real matrix A. This is equivalent to saying $B = Q^{-1}DQ$, where Q is a unitary and D is a diagonal matrix with all positive entries.)

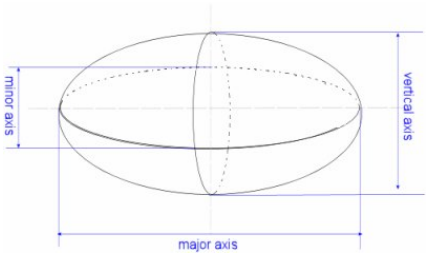

Figure 17.2: 3D-Ellipsoid and its axes

The convex body $\mathcal{K}$ is presented by a membership oracle, and we are told that the body lies somewhere inside some ellipsoid E_0 whose description is given to us. At the ith iteration algorithm maintains the invariant that the body is inside some ellipsoid E_i. The iteration is very simple.

Let p = central point of E_i. Ask the oracle if $p \in \mathcal{K}$. If it says "Yes," declare succes. Else the oracle returns some halfspace $a^T x \geq b$ p that contains $\mathcal{K}$ whereas p lies on the other side. Let E_{i+1} = minimum containing ellipsoid of the convex body $E_i \cap \left\{x : a^T x \geq b\right\}$.

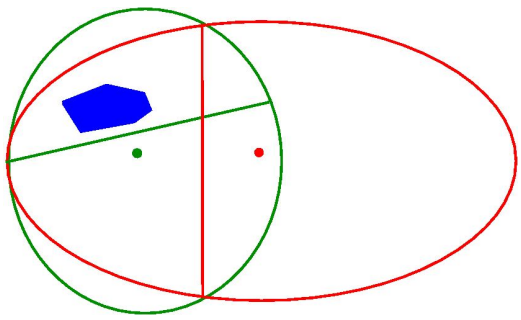

Figure 17.3: Couple of runs of the Ellipsoid method showing the tiny convex set in blue and the containing ellipsoids. The separating hyperplanes do not pass through the centers of the ellipsoids in this figure.

The running time of each iteration depends on the running time of the separation oracle and the time required to find E_{i+1}. For linear programming, the separation oracle runs in

$O(mn)$ time as all we need to do is check whether p satisfies all the constraints, and return a violating constraint as the halfspace (if it exists). The time needed to find E_{i+1} is also polynomial by the following non-trivial lemma from convex geometry.

LEMMA 27
The minimum volume ellipsoid surrounding a half ellipsoid (i.e. $E_i \bigcap H^+$ where H^+ is a halfspace as above) can be calculated in polynomial time and

$$Vol(E_{i+1}) \leq \left(1 - \frac{1}{2n}\right) Vol(E_i)$$

Thus after t steps the volume of the enclosing ellipsoid has dropped by $(1 - 1/2n)^t \leq \exp(-t/2n)$.

Technically speaking, there are many fine points one has to address. (i) The Ellipsoid method can never say unequivocally that the convex body was empty; it can only say after T steps that the volume is less than $\exp(-T/2n)$. In many settings we know a priori that the volume of $\mathcal{K}$ if nonempty is at least $\exp(-n^2)$ or some such number, so this is good enough. (ii) The convex body may be low-dimensional. Then its n-dimensional volume is 0 and the containing ellipsoid continues to shrink forever. At some point the algorithm has to take notice of this, and identify the lower dimensional subspace that the convex body lies in, and continue in that subspace.

As for linear programming can be shown that for a linear program which requires L bits to represent the input, it suffices to have volume of $E_0 = 2^{c_2 nL}$ (since the solution can be written in $c_2 nL$ bits, it fits inside an ellipsoid of about this size) and to finish when volume of $E_t = 2^{-c_1 nL}$ for some constants c_1, c_2, which implies $t = O(n^2 L)$. Therefore, the after $O(n^2 L)$ iterations, the containing ellipsoid is so small that the algorithm can easily "round" it to some vertex of the polytope. (This number of iterations can be improved to $O(nL)$ with some work.) Thus the overall running time is $poly(n, m, L)$. For a detailed proof of the above lemma and other derivations, please refer to Santosh Vempala's notes linked from the webpage. The classic [GLS] text is a very readable yet authoritative account of everything related (and there's a lot) to the Ellipsoid method and its variants.

BIBLIOGRAPHY

[GLS] M. Groetschel, L. Lovasz, A. Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer 1993.

Chapter 18

Duality and MinMax Theorem

We are used to the concept of duality in life: yin and yang, Mars and Venus, etc. In mathematics duality refers to the phenomenon whereby two objects that look very different are actually the same in a technical sense.

Today we first see LP duality, which will then be explored a bit more in the homeworks. Duality has several equivalent statements.

1. If K is a polytope and p is a point outside it, then there is a hyperplane separating p from K.

2. The following system of inequalities

$$\begin{array}{rcl} \mathbf{a}_1 \cdot \mathbf{X} & \geq & b_1 \\ \mathbf{a}_2 \cdot \mathbf{X} & \geq & b_2 \\ & \vdots & \\ \mathbf{a}_m \cdot \mathbf{X} & \geq & b_m \\ \mathbf{X} & \geq & 0 \end{array} \tag{18.1}$$

   is infeasible iff using positive linear combinations of the inequalities it is possible to derive $-1 \geq 0$, i.e. there exist $\lambda_1, \lambda_2, \ldots \lambda_m \geq 0$ such that

$$\sum_{i=1}^{m} \lambda_i \mathbf{a}_i < 0 \quad \text{and} \quad \sum_{i=1}^{m} \lambda_i b_i > 0.$$

   This statement is called *Farkas's Lemma*.

18.1 Linear Programming and Farkas' Lemma

In courses and texts duality is taught in context of LPs. Say the LP looks as follows:

GIVEN: vectors $\mathbf{c}, \mathbf{a}_1, \mathbf{a}_2, \ldots \mathbf{a}_m \in \mathbf{R}^n$, and real numbers $b_1, b_2, \ldots b_m$.
OBJECTIVE: find $\mathbf{X} \in \mathbf{R}^n$ to minimize $\mathbf{c} \cdot \mathbf{X}$, subject to:

$$\begin{array}{rcl} \mathbf{a}_1 \cdot \mathbf{X} & \geq & b_1 \\ \mathbf{a}_2 \cdot \mathbf{X} & \geq & b_2 \\ & \vdots & \\ \mathbf{a}_m \cdot \mathbf{X} & \geq & b_m \\ \mathbf{X} & \geq & 0 \end{array} \tag{18.2}$$

The notation $\mathbf{X} > \mathbf{Y}$ simply means that $\mathbf{X}$ is componentwise larger than $\mathbf{Y}$. Now we represent the system in (18.2) more compactly using matrix notation. Let

$$A = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Then the Linear Program (LP for short) can be rewritten as:

$$\begin{array}{rl} \min & \mathbf{c}^T\mathbf{X} : \\ A\mathbf{X} & \geq \mathbf{b} \\ \mathbf{X} & \geq 0 \end{array} \tag{18.3}$$

This form is general enough to represent any possible linear program. For instance, if the linear program involves a linear equality $\mathbf{a} \cdot \mathbf{X} = b$ then we can replace it by two inequalities

$$\mathbf{a} \cdot \mathbf{X} \geq b \quad \text{and} \quad -\mathbf{a} \cdot \mathbf{X} \geq -b.$$

If the variable X_i is unconstrained, then we can replace each occurence by $X_i^+ - X_i^-$ where X_i^+, X_i^- are two new non-negative variables.

18.2 LP Duality Theorem

With every LP we can associate another LP called its *dual*. The original LP is called the *primal*. If the primal has n variables and m constraints, then the dual has m variables and n constraints. Thus there is a primal variable corresponding to each dual constraint, and a dual variable for each primal constraint.

Primal		Dual	
min	$\mathbf{c}^T\mathbf{X}$:	max	$\mathbf{Y}^T\mathbf{b}$:
$A\mathbf{X}$	$\geq \mathbf{b}$	$\mathbf{Y}^T A$	$\leq \mathbf{c^T}$
$\mathbf{X}$	≥ 0	$\mathbf{Y}$	≥ 0

(18.4)

(Aside: if the primal contains an equality constraint instead of inequality then the corresponding dual variable is unconstrained.)

It is an easy exercise that the dual of the dual is just the primal.

THEOREM 28
The Duality Theorem. *If both the Primal and the Dual of an LP are feasible, then the two optima coincide.*

PROOF: The proof involves two parts:

1. Primal optimum $\geq$ Dual optimum.
   This is the easy part. Suppose $\mathbf{X}^*, \mathbf{Y}^*$ are the respective optima. This implies that

   $$A\mathbf{X}^* \geq \mathbf{b}.$$

   Now, since $\mathbf{Y}^* \geq 0$, the product $\mathbf{Y}^* A\mathbf{X}^*$ is a non-negative linear combination of the rows of $A\mathbf{X}^*$, so the inequality

   $$\mathbf{Y}^{*T} A\mathbf{X}^* \geq \mathbf{Y}^{*T}\mathbf{b}$$

   holds. Again, since $\mathbf{X}^* \geq 0$ and $\mathbf{c}^T \geq \mathbf{Y}^{*T} A$, we obtain the inequality

   $$\mathbf{c}^T\mathbf{X}^* \geq (\mathbf{Y}^{*T} A)\mathbf{X}^*.$$

   Examining the previous two lines we conclude $\mathbf{c}^T\mathbf{X}^* \geq \mathbf{Y}^{*T}\mathbf{b}$, which completes the proof of this part.

2. Dual optimum $\geq$ Primal optimum.
   Let k be the optimum value of the primal. Since the primal is a minimization problem, the following set of linear inequalities is infeasible for any $\varepsilon > 0$:

   $$\begin{array}{rl} -\mathbf{c}^T\mathbf{X} & \geq -(k-\varepsilon) \\ A\mathbf{X} & \geq \mathbf{b} \\ X & \geq 0 \end{array} \tag{18.5}$$

   Here, ε is a small positive quantity. Therefore, by Farkas' Lemma, there exist $\lambda_0, \lambda_1, \ldots \lambda_m \geq 0$ such that

   $$-\lambda_0\mathbf{c} + \sum_{i=1}^{m} \lambda_i \mathbf{a}_i < 0 \tag{18.6}$$

   $$-\lambda_0(k-\varepsilon) + \sum_{i=1}^{m} \lambda_i b_i > 0. \tag{18.7}$$

   Note that $\lambda_0 > 0$ omitting the first inequality in (18.5) leaves a feasible system by assumption about the primal. Thus, consider the nonnegative vector

   $$\Lambda = (\frac{\lambda_1}{\lambda_0}, \ldots \frac{\lambda_m}{\lambda_0})^T.$$

   The inequality (18.6) implies that $\Lambda^T A \leq \mathbf{c}^T$. So Λ is a feasible solution to the Dual. The inequality (18.7) implies that $\Lambda^T\mathbf{b} > (k-\varepsilon)$, and since the Dual is a maximization problem, this implies that the Dual optimal is bigger than $k-\varepsilon$. Since this holds for every $\varepsilon > 0$, by compactness we conclude that there is a Dual feasible solution of value k. Thus, this part is proved, too. Hence the Duality Theorem is proved.

□

My thoughts on this business:
(1) Usually textbooks bundle the case of infeasible systems into the statement of the Duality theorem. This muddies the issue for the student. Usually all applications of LPs fall into two cases: (a) We either know (for trivial reasons) that the system is feasible, and are only interested in the value of the optimum or (b) We do not know if the system is feasible and that is precisely what we want to determine. Then it is best to just use Farkas' Lemma.
(2) The proof of the Duality theorem is interesting. The first part shows that for any dual feasible solution $\mathbf{Y}$ the various Y_i's can be used to obtain a *weighted* sum of primal inequalities, and thus obtain a lowerbound on the primal. The second part shows that this method of taking weighted sums of inequalities is *sufficient* to obtain the best possible lowerbound on the primal: there is no need to do anything fancier (e.g., taking products of inequalities or some such thing).

18.3 Example: Max Flow Min Cut theorem in graphs

The input is a directed graph $G(V, E)$ with one source s and one sink t. Each edge e has a capacity c_e. The flow on any edge must be less than its capacity, and at any node apart from s and t, flow must be conserved: total incoming flow must equal total outgoing flow. We wish to maximize the flow we can send from s to t. The maximum flow problem can be formulated as a Linear Program as follows:

Let $\mathcal{P}$ denote the set of all (directed) paths from s to t. Then the max flow problem becomes:

$$\max \sum_{P \in \mathcal{P}} f_P : \tag{18.8}$$

$$\forall P \in \mathcal{P} : f_P \geq 0 \tag{18.9}$$

$$\forall e \in E : \sum_{P : e \in P} f_P \leq c_e \tag{18.10}$$

Since P could contain exponentially many paths, this is an LP with exponentially many variables. Luckily duality tells us how to solve it using the Ellipsoid method.

Going over to the dual, we get:

$$\min \sum_{e \in E} c_e y_e : \tag{18.11}$$

$$\forall e \in E : y_e \geq 0 \tag{18.12}$$

$$\forall P \in \mathcal{P} : \sum_{e \in P} y_e \geq 1 \tag{18.13}$$

Notice that the dual in fact represents the fractional min $s - t$ cut problem: think of each edge e being picked up to a fraction y_e. The constraints say that a total weight of 1 must be picked on each path. Thus the usual s-t min cut problem simply involves $0 - 1$ solutions to the y_e's in the dual.

EXERCISE 1 Prove that the optimum solution does have $y_e \in \{0, 1\}$, and thus the solution to the dual is the best s-t min cut.

Thus, LP duality implies max-st-flow = (capacity of) min-cut.

Polynomial-time algorithms? The primal has exponentially many variables! (Aside: turns out it is equivalent to a more succinct LP but lets' proceed with this one.) Nevertheless we can use the Ellipsoid method by applying it to the dual, which has m variables and exponentially many constraints. As we saw last time, we only need to show a polynomial-time separation oracle for the dual. Namely, for each candidate vector (y_e) we need to check if it satisfies all the dual constraints. This can be done by creating a weighted version of the graph where the weight on edge e is y_e. Then compute the shortest path from s to t in this weighted graph. If the shortest path has length < 1 then we have found a violated constraint.

Of course, for Max Flow we know of much faster algorithms than the Ellipsoid method (e.g., the algorithms you saw in your undergrad course), but there are other LPs with exponentially many variables for which the only known polynomial time algorithms go via the Ellipsoid method.

18.4 Game theory and the minmax theorem

In the 1930s, polymath John von Neumann (professor at IAS, now buried in the cemetery close to downtown) was interested in applying mathematical reasoning to understand strategic interactions among people —or for that matter, nations, corporations, political parties, etc. He was a founder of *game theory*, which models rational choice in these interactions as maximization of some payoff function.

A starting point of this theory is the *zero-sum* game. There are two players, 1 and 2, where 1 has a choice of m possible moves, and 2 has a choice of n possible moves. When player 1 plays his ith move and player 2 plays her jth move, the outcome is that player 1 pays A_{ij} to player 2. Thus the game is completely described by an $m \times n$ *payoff* matrix.

-	**scissors**	**paper**	**rock**
rock	1	−1	0
paper	−1	0	1
scissors	0	1	−1

Figure 18.1: Payoff matrix for Rock/Paper/Scissor

This setting is called *zero sum* because what one player wins, the other loses. By contrast, war (say) is a setting where both parties may lose material and men. Thus their combined worth at the end may be lower than at the start. (Aside: An important stimulus

for development of game theory in the 1950s was the US government's desire to behave "strategically "in matters of national defence, e.g. the appropriate tit-for-tat policy for waging war —whether nuclear or conventional or cold.)

von Neumann was interested in a notion of equilibrium. In physics, chemistry etc. an equilibrium is a stable state for the system that results in no further change. In game theory it is a pair of strategies g_1, g_2 for the two players such that each is the optimum response to the other.

Let's examine this for zero sum games. If player 1 announces he will play the ith move, then the *rational* move for player 2 is the move j that maximises A_{ij}. Conversely, if player 2 announces she will play the jth move, player 1 will respond with move i' that minimizes $A_{i'j}$. In general, there may be no *equilibrium* in such announcements: the response of player 1 to player 2's response to his announced move i will not be i in general:

$$\min_i \max_j A_{ij} \neq \max_j \min_i A_{ij}.$$

In fact there is no such equilibrium in Rock/paper/scissors either, as every child knows.

von Neumann realized that this lack of equilibrium disappears if one allows players' announced strategy to be a *distribution* on moves, a so-called *mixed* strategy. Player 1's distribution is $x \in \Re^m$ satisfying $x_i \geq 0$ and $\sum_i x_i = 1$; Player 2's distribution is $y \in \Re^n$ satisfying $y_j \geq 0$ and $\sum_j y_j = 1$. Clearly, the expected payoff from Player 1 to Player 2 then is $\sum_{ij} x_i A_{ij} y_j = x^T A y$.

But has this fixed the problem about nonexistence of equilibrium? If Player 1 announces first the payoff is $\min_x \max_y x^T A y$ whereas if Player 2 announces first it is $\max_y \min_x x^T A y$. The next theorem says that it doesn't matter who announces first; neither player has an incentive to change strategies after seeing the other's announcement.

Theorem 29 (Famous Min-Max Theorem of Von Neumann)
$\min_x \max_y x^T A y = \max_y \min_x x^T A y$.

Turns out this result is a simple consequence of LP duality and is equivalent to it. You will explore it further in the homework.

What if the game is not zero sum? Defining an equilibrium for it was an open problem until John Nash at Princeton managed to define it in the early 1950s; this solution is called a Nash equilibrium. We'll return to it in a future lecture. BTW, you can still sometimes catch a glimpse of Nash around campus.

Chapter 19

Equilibria and algorithms

Economic and game-theoretic reasoning —specifically, how agents respond to economic incentives as well as to each other's actions– has become increasingly important in algorithm design. Examples: (a) Protocols for networking have to allow for sharing of network resources among users, companies etc., who may be mutually cooperating or competing. (b) Algorithm design at Google, Facebook, Netflix etc.—what ads to show, which things to recommend to users, etc.—not only has to be done using objective functions related to economics, but also with an eye to how users and customers *change* their behavior in response to the algorithms and to each other.

Algorithm design mindful of economic incentives and strategic behavior is studied in a new field called *Algorithmic Game Theory.* (See the book by Nisan et al., or many excellent lecture notes on the web.)

Last lecture we encountered zero sum games, a simple setting. Today we consider more general games.

19.1 Nonzero sum games and Nash equilibria

Recall that a 2-player game is *zero sum* if the amount won by one player is the same as the amount lost by the other. Today we relax this. Thus if player 1 has n possible actions and player 2 has m, then specifying the game requires two a $n \times m$ matrices A, B such that when they play actions i, j respectively then the first player wins A_{ij} and the second wins B_{ij}. (For zero sum games, $A_{ij} = -B_{ij}$.)

A Nash equilibrium is defined similarly to the equilibrium we discussed for zero sum games: a pair of strategies, one for each player, such that each is the optimal response to the other. In other words, if they both announce their strategies, neither has an incentive to deviate from his/her announced strategy. The equilibrium is *pure* if the strategy consists of deterministically playing a single action.

Example 43 (Prisoners' Dilemma) This is a classic example that people in myriad disciplines have discussed for over six decades. Two people suspected of having committed a crime have been picked up by the police. In line with usual practice, they have been placed in separate cells and offered the standard deal: help with the investigation, and you'll be

treated with leniency. How should each prisoner respond: Cooperate (i.e., stick to the story he and his accomplice decided upon in advance), or Defect (rat on his accomplice and get a reduced term)?

Let's describe their incentives as a 2×2 matrix, where the first entry describes payoff for the player whose actions determine the row. If they both cooperate, the police can't

	Cooperate	Defect
Cooperate	3, 3	0, 4
Defect	4, 0	1, 1

prove much and they get off with fairly light sentences after which they can enjoy their loot (payoff of 3). If one defects and the other cooperates, then the defector goes scot free and has a high payoff of 4 whereas the other one has a payoff of 0 (long prison term, plus anger at his accomplice).

The only pure Nash equilibrium is (Defect, Defect), with both receiving payoff 1. In every other scenario, the player who's cooperating can improve his payoff by switching to Defect. This is much worse for both of them than if they play (Cooperate, Cooperate), which is also the social optimum —where the sum of their payoffs is highest at 6—is to cooperate. Thus in particular the social optimum solution is not a Nash equilibrium. ((OK, we are talking about criminals here so maybe social optimum is (Defect, Defect) after all. But read on.)

One can imagine other games with similar payoff structure. For instance, two companies in a small town deciding whether to be polluters or to go green. Going green requires investment of money and effort. If one does it and the other doesn't, then the one who is doing it has incentive to also become a polluter. Or, consider two people sharing an office. Being organized and neat takes effort, and if both do it, then the office is neat and both are fairly happy. If one is a slob and the other is neat, then the neat person has an incentive to become a slob (saves a lot of effort, and the end result is not much worse).

Such games are actually ubiquitous if you think about it, and it is a miracle that humans (and animals) cooperate as much as they do. Social scientists have long pondered how to cope with this paradox. For instance, how can one change the game definition (e.g. a wise governing body changes the payoff structure via fines or incentives) so that cooperating with each other —the socially optimal solution—becomes a Nash equilibrium? The game can also be studied via the *repeated game* interpretation, whereby people realize that they participate in repeated games through their lives, and playing nice may well be a Nash equilibrium in that setting. As you can imagine, many books have been written. □

Example 44 (Chicken) This dangerous game was supposedly popular among bored teenagers in American towns in the 1950s (as per some classic movies). Two kids would drive their cars at high speed towards each other on a collision course. The one who swerved away first to avoid a collision was the "chicken."How should we assign payoffs in this game? Each player has two possible actions, *Chicken* or *Dare*. If both play Dare, they wreck their cars and risk injury or death. Lets call this a payoff of 0 to each. If both go Chicken, they both live and have not lost face, so let's call it a payoff of 5 for each. But if one goes Chicken and the other goes Dare, then the one who went Dare looks like the tough one (and presumably

attracts more dates), whereas the Chicken is better of being alive than dead but lives in shame. So we get the payoff table:

	Chicken	Dare
Chicken	5, 5	1, 6
Dare	6, 1	0, 0

This has two pure Nash equilibria: (Dare, Chicken) and (Dare, Chicken). We may think of this as representing two types of behavior: the reckless type may play Dare and the careful type may play Chicken.

Note that the socially optimal solution —both players play chicken, which maximises their total payoff—is not a Nash equilibrium.

Many games do not have any pure Nash equilibrium. Nash's great insight during his grad school years in Princeton was to consider what happens if we allow players to play a *mixed* strategy, which is a probability distribution over actions. An equilibrium now is a pair of mixed strategies x, y such that each strategy is the optimum response (in terms of maximising expected payoff) to the other.

THEOREM 30 (NASH 1950)
For every pair of payoff matrices A, B there is an odd number (hence nonzero) of mixed equilibria.

Unfortunately, Nash's proof doesn't yield an efficient algorithm for computing an equilibrium: when the number of possible actions is n, computation may require $\exp(n)$ time. Recent work has shown that this may be inherent: computing Nash equilibria is PPAD-complete (Chen and Deng'06).

The Chicken game has a mixed equilibrium: play each of Chicken and Dare with probability 1/2. This has expected payoff $\frac{1}{4}(5+1+6+0) = 3$ for each, and a simple calculation shows that neither can improve his payoff against the other by changing to a different strategy.

19.2 Multiplayer games and Bandwidth Sharing

One can define multiplayer games and equilibria analogously to single player games. One can also define games where each player's set of moves comes from a continuous set like the interval $[0, 1]$. Now we do this in a simple setting: multiple users sharing a single link of fixed bandwidth, say 1 unit. They have different utilities for internet speed, and different budgets. Hence the owner of the link can try to allocate bandwidth using a game-theoretic view, which we study using a game introduced by Frank Kelly.

1. There are n users. If user i gets x units of bandwidth by paying w dollars, his/her utility is $U_i(x) - w$, where the *utility* function U_i is nonnegative, increasing, *concave*[1]

[1]Concavity implies that the going from 0 units to 1 brings more happiness than going from 1 to 2, which in turn brings more happiness than going from 2 to 3. For twice differentiable functions, concavity means the second derivative is negative.

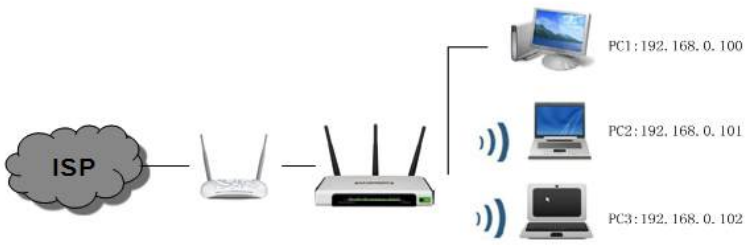

Figure 19.1: Sharing a fixed bandwidth link among many users

and differentiable. If a unit of bandwidth is priced at p, this utility describes the amount of bandwidth desired by a utility-maximizing user: the ith user demands x_i that maximises $U_i(x_i) - px_i$. This maximum can be computed by calculus.

2. The game is as follows: user i offers to pay a sum of w_i. The link owner allocates $w_i / \sum_j w_j$ portion of the bandwidth to user i. Thus the entire bandwidth is used up and the effective price for the entire bandwidth is $\sum_j w_j$.

What n-tuple of strategies $w_1, w_2, \ldots, w_n$ is a Nash equilibrium? Note that this n-tuple implies a per unit price p of $\sum_j w_j$, and for each i his received amount is optimal at this price if $x_i = w_i / \sum_j w_j$ is the solution to $\max U_i(x_i) - w$, which requires (by chain rule of differentiation):

$$\begin{aligned} U_i'(x_i)(\frac{1}{p} - \frac{w_i}{p^2}) &= 1 \\ \Rightarrow U_i'(x_i)(1 - x_i) &= p. \end{aligned}$$

This implicitly defines x_i in terms of p. Furthermore, the left hand side is easily checked to be a decreasing function of x_i. (Specifically, its derivative is $(1 - x_i)U_i"(x_i) - U'(x_i)$, whose first term is negative by concavity and the second because $U_i'(x_i) \geq 0$ by our assumption that U_i is an increasing function.) Thus $\sum_i x_i$ is a decreasing function of p. When $p = +\infty$, the x_i's that maximise utility are all 0, whereas for $p = 0$ the x_i's are all 1, which violates the constraint $\sum_i x_i = 1$. By the mean value theorem, there must exceed a choice of p between 0 and $+\infty$ where $\sum_i x_i = 1$, and the corresponding values of w_i's then constitute a Nash equilibrium.

Is this equilibrium socially optimal? Let p^* be the socially optimal price. At this price the ith user desires a bandwidth x_i that maximises $U_i(x_i) - p^* x_i$, which is the unique x_i that satisfies $U_i'(x_i) = p^*$. Furthermore these x_i's must sum to 1.

By contrast, the Nash equilibrium price p_N corresponds to solving $U_i'(x_i)(1 - x_i) = p_N$. If the number of users is large (and the utility functions not "too different"so that the x_i's are not too different) then each x_i is small and $1 - x_i \approx 1$. Thus the Nash equilibrium price is close to but not the same as the socially optimal choice.

Price of Anarchy

One of the notions highlighted by algorithmic game theory is *price of anarchy,* which is the ratio between the cost of the Nash equilibrium and the social optimum. The idea behind this name is that Nash equilibrium is what would be achieved in a free market, whereas social optimum is what could be achieved by a planner who knows everybody's utilities. One identifies a family of games, such as bandwidth sharing, and looks at the *maximum* of this ratio over all choices of the players' utilities. The price of anarchy for the bandwidth sharing game happens to be 4/3. Please see the chapter on inefficiency of equilibria in the AGT book.

19.3 Correlated equilibria

In HW 3 you were asked to simulate two strategies that repeatedly play Rock-Paper-Scissors while minimizing regret. The Payoffs were as follows:

	Rock	Paper	Scissor
Rock	0,0	0, 1	1, 0
Paper	1, 0	0, 0	0, 1
Scissor	0, 1	1, 0	0, 0

Possibly you originally guessed that they would converge to playing Rock, Paper, Scissor randomly. However, this is not regret minimizing since it leads to payoff 0 every third round in the expectation. What you probably saw in your simulation was that the players converged to a *correlated* strategy that guarantees one of them a payoff every other round. Thus they learnt to game the system together and maximise their profits.

This is a subcase of a more general phenomenon, whereby playing low-regret strategies in general leads to a different type of equilibrium, called *correlated equilibrium.*

Example 45 In the game of Chicken, the following is a correlated equilibrium: each of the three pairs of moves other than (Dare, Dare) with probability 1/3. This is a correlated strategy: there is a global random string (or higher agency) that tells the players what to do. Neither player knows what the other has chosen.

Suppose we think of the game being played between two cars approaching a traffic intersection from two directions. Then the correlated equilibrium of the previous paragraph has a nice interpretation: a traffic light! Actually, it is what a traffic light would look like if there were no traffic police to enforce the laws. The traffic light would be programmed to repeatedly pick one of three states with equal probability: (Red, Red), (Green, Red), and (Red, Green). (By contrast, real-life lights cycle between (Red, Green), and (Green, Red); where we are ignoring Yellow for now.) If a motorist arriving at the intersection sees Green, he knows that the other motorist sees Red and so can go through without hesitation. If he sees Red on the other hand, he only knows that there is equal chance that the other motorist sees Red or Green. So acting rationally he will come to a stop since otherwise he has probability 1/2 of getting into an accident. Note that this means that when the light is (Red, Red) then the traffic would be sitting at a halt in both directions.

The previous example illustrates the notion of correlated equilibrium, and we won't define it more precisely. The main point is that it can be arrived at using a simple algorithm, namely, multiplicative weights (this statement also has caveats; see the relevant chapter in the AGT book). Unfortunately, correlated equilibria are also not guaranteed to maximise social welfare.

Bibliography

1. Algorithmic Game Theory. Nisan, Roughgarden, Tardos, Vazirani (eds.), Cambridge University Press 2007.

2. The mathematics of traffic in networks. Frank Kelly. In *Princeton Companion to Mathematics* (T. Gowers, Ed.). PU Press 2008.

3. Settling the Complexity of 2-Player Nash Equilibrium. X. Chen and X. Deng. IEEE FOCS 2006.

Chapter 20

Protecting against information loss: coding theory

Computer and information systems are prone to data loss—lost packets, crashed or corrupted hard drives, noisy transmissions, etc.—and it is important to prevent actual loss of important information when this happens. Today's lecture concerns *error correcting codes*, a stepping point to many other ideas, including a big research area (usually based in EE departments) called *information theory*. This area started with a landmark paper by Claude Shannon in 1948, whose key insight was that data transmission is possible despite noise and errors if the data is *encoded* in some redundant way.

Example 46 (Elementary ways of introducing redundancy) The simplest way to introduce *redundancy* is to repeat each bit, say 5 times. The cons are (a) large inefficiency (b) no resistance to *bursty* error, which may wipe out all 5 copies.

Another simple method is *checksums*. For instance suppose we transmit 3 bits b_1, b_2, b_3 as $b_1, b_2, b_3, b_1 \oplus b_2 \oplus b_3$ where the last bit is the *parity* of the first three. Then if one of the bits gets flipped, the parity will be incorrect. However, if two bits get corrupted, the parity becomes correct again! Thus this method can detect when a single bit has been corrupted. It is useful in settings where errors are rare: if an error in the checksum is detected, the entire information/packet can be retransmitted.

A cleverer checksum method used by some cloud services is to store three bits b_1, b_2, b_3 as 7 bits on 7 servers: $b_1, b_2, b_3, b_1 \oplus b_2, b_1 \oplus b_3, b_2 \oplus b_3, b_1 \oplus b_2 \oplus b_3$. It is easily checked that: if up to three servers fail, each bit is still recoverable, and in fact by querying at most 2 servers. A cleverer design of such *data storage codes* recently saved Microsoft 13% space on its cloud servers.

Example 47 (Generalized Checksums) A trivial extension of the checksum idea is to encode k bits using 2^k checksums: take the parity of all possible subsets. This works to protect the data even if close to half the bits get flipped (though we won't prove it; requires some Fourier analysis).

Another form of checksums is to designate some random subsets of $\{1, 2, \ldots, k\}$, say $S_1, S_2, \ldots, S_m$. Then encode any k bit vector using the m checksums corresponding to

these subsets. This works against $\Omega(m)$ errors but we don't know of an efficient decoding algorithm. (Decoding in $\exp(k)$ time is no problem.)

20.1 Shannon's Theorem

Shannon considered the following problem: a message $x \in \{0,1\}^n$ has to be sent over a channel which flips every bit with probability p. How can we ensure that the message is recovered correctly at the other end? A couple of years later Hamming introduced a related notion whereby the channel flips up to p *fraction* of bits —and can adversarially decide which subset of bits to flip. He was concerned that real channels exhibit burstiness: make a lot of errors in one go and then no errors for long periods. By Chernoff bounds, Shannon's channel is a subcase (whp) of the Hamming channel since the chance of flipping more than $p+\varepsilon$ fraction of bits in total is $\exp(-\Theta(n))$. Both kinds of channels have been studied since then and we will actually use Hamming's notion today.

Shannon suggested that the message be encoded using a function $E : \{0,1\}^n \to \{0,1\}^m$ and at the other end it should be *decoded* using a function $D : \{0,1\}^m \to \{0,1\}^n$ with the property that $D(E(x) \oplus \eta) = x$ for any noise vector $\eta \in \{0,1\}^m$ that is 1 in at most pm indices and 0 in the rest. (Here $\oplus$ of two bit vectors denotes bitwise parity.)

Clearly, such a decoding is possible if for every two messages x, x' their encodings differ in more than $2pm$ bits: then $E(x) \oplus \eta_1$ will not be confused for $E(x') \oplus \eta_2$ for any two noise vectors η_1, η_2 that only are nonzero in pm bits. We say such a code has *minimum distance* at least $2pm$.

The famous entropy function appearing in the following theorem is graphed below. (The notion of Entropy used in the 2nd law of thermodynamics is closely related.)

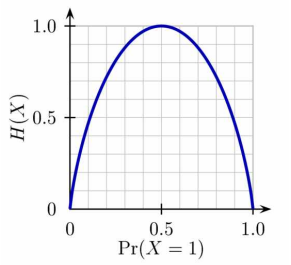

Figure 20.1: The graph of $H(X)$ as a function of X.

THEOREM 31
Such E, D do not exist if $m < \frac{n}{1-H(p)}$, and do exist for $p \leq 1/4$ if $m > \frac{n}{1-H(2p)}$. Here $H(p) = p\log_2 \frac{1}{p} + (1-p)\log_2 \frac{1}{1-p}$ is the so-called entropy *function.*

PROOF: We only prove *existence*; the method does not give efficient algorithms to encode/decode. For any string $y \in \{0,1\}^m$ let $\text{Ball}(y)$ denote the set of strings that differ

from y in at most

$$\binom{m}{0} + \binom{m}{1} + \cdots \binom{m}{2pm},$$

which is at most $2^{H(2p)m}$ by Stirling's approximation.

Define the encoding function E using the following greedy procedure. Number the strings in $\{0,1\}^n$ from 1 to 2^n and one by one assign to each string x its encoding $E(x)$ as follows. The first string is assigned an arbitary string in $\{0,1\}^m$. At step i the ith string is assigned an arbitary string that lies outside Ball$(E(x)$ for all $x \leq i-1$.

By design, such an encoding function satisfies that $E(x)$ and $E(x')$ differ in at least $2pm$ fraction. Thus we only need to show that the greedy procedure succeeds in assigning an encoding to each string. To do this it suffices to note that if $2^m > 2^n 2^{H(2p)m}$ then the greedy procedure never runs out of strings to assign as encodings.

The nonexistence is proved in a similar way. Now for $y' \in \{0,1\}^m$ let Ball$'(y)$ be the set of strings that differ from y in at most pm indices. By a similar calculation as above, this has cardinality about $2^{H(p)m}$. If an encoding function exists, then Ball$'(E(x))$ and Ball$'(E(x'))$ must be disjoint for all $x \neq x'$ (since otherwise any string in the intersection would not have an unambiguous encoding). Hence $2^n \times 2^{H(p)m} < 2^m$, which implies that $m > \frac{n}{1-H(p)}$. □

20.2 Finite fields and polynomials

Below we will design error correcting codes using polynomials over finite fields. Here *finite field* will refer to Z_q, the integers modulo a prime q. Recall that one can define $+, \times, \div$ over these numbers, and that $x \times y = 0$ iff at least one of x, y is 0. A degree d polynomial $p(x)$ has the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d.$$

It can be seen as a function that maps $x \in Z_q$ to $p(x)$.

Lemma 32 (Polynomial Interpolation)
For any set of $n+1$ pairs $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ where the x_i's are distinct elements of Z_q, there is a unique degree n polynomial $g(x)$ satisfying $g(x_i) = y_i$ for each i.

Proof: Let $a_0, a_1, \ldots, a_n$ be the coefficients of the desired polynomial. Then the constraint $g(x_i) = y_i$ corresponds to the following linear system.

This system has a unique solution iff the matrix on the left is invertible, i.e., has nonzero determinant. This is nothing but the famous *Vandermonde* matrix, whose determinant is $\prod_{i \leq n} \prod_{j \neq i} (x_i - x_j)$. This is nonzero since the x_i's are distinct. Thus the system has a solution. Actually the solution has a nice description via the Lagrange interpolation formula:

$$g(x) = \sum_{i=0}^{n} y_i \prod_{j \neq i} \frac{(x - x_j)}{x_i - x_j}.$$

□

Corollary 33
If a degree d has more than d roots (i.e., points where it takes zero value) then it is the zero polynomial.

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Figure 20.2: Linear system corresponding to polynomial interpolation; matrix on left side is *Vandermonde*.

20.3 Reed Solomon codes and their decoding

The Reed Solomon code from 1960 is ubiquitous, having been used in a host of settings including data transmission by NASA vehicles and the storage standard for music CDs. It is simple and inspired by Lemma 32. The idea is to break up a message into chunks of $\lfloor \log q \rfloor$ bits, where each chunk is interpreted as an element of the field Z_q. If the message has $(d+1)\lfloor \log q \rfloor$ bits then it can be interpreted as coefficients of a degree d polynomial $p(x)$. The encoding consists of evaluating this polynomial at n points $u_1, u_2, \dots, v_n \in Z_q$ and defining the encoding to be $p(u_1), p(u_2), \dots, p(u_n)$.

Suppose the channel corrupts k of these values, where $n - k \geq d + 1$. Let $v_1, v_2, \dots, v_n$ denote the received values. If we knew which values are uncorrupted, the decoder could use polynomial interpolation to recover p. Trouble is, the decoder has no idea which received value has been corrupted. We show how to recover p if $k < \frac{n-d}{2} - 1$.

LEMMA 34
There exists a nonzero degree k polynomial $e(x)$ and a polynomial $c(x)$ of degree at most $d + k$ such that

$$c(u_i) = e(u_i)v_i \qquad \text{for } i = 1, 2, \dots, n. \tag{20.1}$$

PROOF: Let $I \subseteq \{1, 2, \dots, n\}$, with $|I| = k$ be the subset of indices i such that v_i has been corrupted. Then (20.1) is satisfied by $e(x) = \prod_{i \in I}(x - u_i)$ and $c(x) = e(x)p(x)$ since $e(u_i) = 0$ for each $i \in I$ and nonzero outside I. □

The polynomial e in the previous proof is called the *error locator polynomial*. Now note that if we let the coefficients of c, e be unknowns, then (20.1) is a system of n equations in $d+2k+2$ unknowns. This system is *overdetermined* since the number of constraints exceeds the number of variables. But Lemma 34 guarantees this system is feasible, and thus can be solved in polynomial time by Gaussian elimination.

We will need the notion of a polynomial *dividing* another. For instance $x^2 + 2$ divides $x^3 + x^2 + 2x + 2$ since $x^3 + x^2 + 2x + 2 = (x^2 + 2)(x + 1)$. The algorithm to divide one polynomial by another is the obvious analog of integer division.

LEMMA 35
If $n > d + 2k + 1$ then any solution $c(x), e(x)$ to the system of Lemma 34 satisfies (i) $e(x)$ divides $c(x)$ as a polynomial (ii) $c(x)/e(x)$ is $p(x)$.

PROOF: The polynomial $c(x) - e(x)p(x)$ has a root at u_i whenever v_i is uncorrupted since $p(u_i) = v_i$. Thus this polynomial, which has degree $d + k$, has $n - k$ roots. Thus if $n - k > d + k + 1$ this polynomial is identically 0. □

20.4 Code concatenation

Technically speaking, the Reed-Solomon code only works if the error rate of the channel is less than $1/\log_2 q$, since otherwise the channel could corrupt one bit in *every* value of the polynomial.

To allow error rate $\Omega(1)$ one uses *code concatenation*. This means that we encode each value of p —which is a string of $t = \lceil \log_2 q \rceil$ bits—with another code that maps t bits to $O(t)$ bits and has minimum distance $\Omega(t)$. Wait a minute: you might say. If we had such a code all along then why go to the trouble of defining the Reed-Solomon code?

The reason is that we do have such a code by Shannon's construction (or by trivial checksums; see Example 47): but since we are only applying it on strings of size t it can be encoded and decoded in $\exp(t)$ time, which is only q. Thus if q is polynomial in the message size, we still get encoding/decoding in polynomial time.

This technique is called code concatenation. One can also use any other error correcting code instead of Shannon's trivial code.

Chapter 21

Counting and Sampling Problems

Today's topic of counting and sampling problems is motivated by computational problems involving multivariate statistics and estimation, which arise in many fields. For instance, we may have a probability density function $\phi(x)$ where $x \in \Re^n$. Then we may want to compute *moments* or other parameters of the distribution, e.g. $\int x^3 \phi(x) dx$. Or, we may have a model for how links develop faults in a network, and we seek to compute the probability that two nodes i, j stay connected under this model. This is a complicated probability calculation.

In general, such problems can be intractable (eg, NP-hard). The simple-looking problem of integrating a multivariate function is NP-hard in the worst case, even when we have an explicit expression for the function $f(x_1, x_2, \ldots, x_n)$ that allows f to be computed in polynomial (in n) time.

$$\int_{x_1=0}^{1} \int_{x_2=0}^{1} \cdots \int_{x_n=0}^{1} f(x_1, x_2, \ldots, x_n) dx_1 dx_2 \ldots dx_n.$$

In fact even approximating such integrals can be NP-hard, as shown by Koutis (2003).

Valiant (1979) showed that the computational heart of such problems is combinatorial *counting problems*. The goal in such problems is to compute the size of a set S where we can test *membership* in S in polynomial time. The class of such problems is called $\sharp P$.

EXAMPLE 48 $\sharp$SAT is the problem where, given a boolean formula φ, we have to compute the *number* of satisfying assignments to φ. Clearly it is NP-hard since if we can solve it, we can in particular solve the *decision* problem: *decide* if the number of satisfying assignments at least 1.

$\sharp$CYCLE is the problem where, given a graph $G = (V, E)$, we have to compute the number of *cycles* in G. Here the decision problem ("is G acyclic?") is easily solvable using breadth first search. Nevertheless, the counting problem turns out to be NP-hard.

$\sharp$SPANNINGTREE is the problem where, given a graph $G = (V, E)$, we have to compute the number of *spanning trees* in G. This is known to be solvable using a simple determinant computation (Kirchoff's matrix-tree theorem) since the 19th century.

Valiant's class $\sharp$P captures most interesting counting problems. Many of these are NP-hard, but not all. You can learn more about them in *COS 522: Computational Complexity*, usually taught in the spring semester. □

It is easy to see that the above integration problem can be reduced to a counting problem with some loss of precision. First, recall that integration basically involves summation: we appropriately discretize the space and then take the sum of the integrand values (assuming in each cell of space the integrand doesn't vary much). Thus the integration reduces to some sum of the form

$$\sum_{x_1\in[N],x_2\in[N],\ldots,x_n\in[N]} g(x_1,x_2,\ldots,x_n),$$

where $[N]$ denotes the set of integers in $0,1,\ldots,N$. Now assuming $g(\cdot)\geq 0$ this is easily estimated using sizes of of the following sets:

$$\{(x,c) : x\in[N]^n;\ c\leq g(x)\leq c+\varepsilon\}.$$

Note if g is computable in polynomial time then we can test membership in this set in polynomial time given (x,c,ε) so we've shown that integration is a $\sharp P$ problem.

We will also be interested in *sampling* a random element of a set S. In fact, this will turn out to be intimately related to the problem of counting.

21.1 Counting vs Sampling

We say that an algorithm is an *approximation scheme* for a counting problem if for every $\varepsilon>0$ it can output an estimate of the size of the set that is correct within a multiplicative factor $(1+\varepsilon)$. We say it is a *randomized fully polynomial approximation scheme* (FPRAS) if it is randomized and it runs in $\text{poly}(n,1/\varepsilon,\log 1/\delta)$ time and has probability at least $(1-\delta)$ of outputting such an answer. We will assume $\delta<1/poly(n)$ so we can ignore the probability of outputting an incorrect answer.

An *fully polynomial-time approximate sampler* for S is one that runs in $\text{poly}(n,1/\varepsilon,\log 1/\delta)$ and outputs a sample $u\in S$ such that $\sum_{u\in S}\left|\Pr[u \text{ is output}]-\frac{1}{|S|}\right|\leq\varepsilon$.

Theorem 36 (Jerrum, Valiant, Vazirani 1986)
For "nicely behaved "counting problems (the technical term is "downward self-reducible") sampling in the above sense is equivalent to counting (i.e., a algorithm for one task can be converted into one for the other).

Proof: For concreteness, let's prove this for the problem of counting the number of satisfying assignments to a boolean formula. Let $\sharp\varphi$ denote the number of satisfying assignments to formula φ.

Sampling $\Rightarrow$ Approximate counting: Suppose we have an algorithm that is an approximate sampler for the set of satisfying assignments for any formula. For now assume it is an exact sampler instead of approximate. Take m samples from it and let p_0 be the fraction that have a 0 in the first bit x_i, and p_1 be the fraction that have a 1. Assume $p_0\geq 1/2$. Then the estimate of p_0 is correct up to factor $(1+1/\sqrt{m})$ by Chernoff bounds. But denoting by $\varphi|_{x_1=0}$ the formula obtained from φ by fixing x_1 to 0, we have

$$p_0=\frac{\sharp\varphi|_{x_1=0}}{\sharp\varphi}.$$

Since we have a good estimate of p_0, to get a good estimate of $\sharp\varphi$ it suffices to have a good estimate of $\sharp\varphi|_{x_1=0}$. So produce the formula $\varphi|_{x_1=0}$ obtained from φ by fixing x_1 to 0, then use the same algorithm recursively on this smaller formula to estimate N_0, the value of $\sharp\varphi|_{x_1=0}$. Then output N_0/p_0 as your estimate of $\sharp\varphi$. (Base case $n = 1$ can be solved exactly of course.)

Thus if Err_n is the error in the estimate for formulae with n variables, this satisfies

$$\mathrm{Err}_n \leq (1 + 1/\sqrt{m})\mathrm{Err}_{n-1},$$

which solves to $\mathrm{Err}_n \leq (1 + 1/\sqrt{m})^n$. By picking $m >> n^2/\varepsilon^2$ this error can be made less than $1 + \varepsilon$. It is easily checked that if the sampler is not exact but only approximate, the algorithm works essentially unchanged, except the sampling error also enters the expression for the error in estimating p_0.

Approximate counting $\Rightarrow$ Sampling: This involves reversing the above reasoning. Given an approximate counting algorithm we are trying to generate a random satisfying assignment. First use the counting algorithm to approximate $\sharp\varphi|_{x_1=0}$ and $\sharp\varphi$ and take the ratio to get a good estimate of p_0, the fraction of assignments that have 0 in the first bit. (If p_0 is too small, then we have a good estimate of $p_1 = 1 - p_0$.) Now toss a coin with $\Pr[\text{heads}] = p_0$. If it comes up heads, output 0 as the first bit of the assignment and then recursively use the same algorithm on $\varphi|_{x_1=0}$ to generate the remaining $n - 1$ bits. If it comes up tails, output 1 as the first bit of the assignment and then recursively use the same algorithm on $\varphi|_{x_1=1}$ to generate the remaining $n - 1$ bits.

Note that the quality ε of the approximation suffers a bit in going between counting and sampling. □

21.1.1 Monte Carlo method

The classical method to do counting via sampling is the *Monte Carlo* method. A simple example is the ancient method to estimate the area of a circle of unit radius. Draw the circle in a square of side 2. Now throw darts at the square and measure the fraction that fall in the circle. Multiply that fraction by 4 to get the area of the circle.

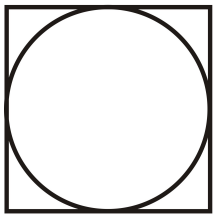

Figure 21.1: Monte Carlo (dart throwing) method to estimate the area of a circle. The fraction of darts that fall inside the disk is $\pi/4$.

Now replace "circle" with any set S and "square" with any set Ω that contains S and can be sampled in polynomial time. Then just take many samples from Ω and just observe the

fraction that are in S. This is an estimate for $|S|$. The problem with this method is that usually the obvious Ω is much bigger than S, and we need $|\Omega| / |S|$ samples to get any that lie in S. (For instance the obvious Ω for computing $\sharp\varphi$ is the set of all possible assignments, which may be exponentially bigger.)

21.2 Dyer's algorithm for counting solutions to KNAPSACK

The Knapsack problem models the problem faced by a kid who is given a knapsack and told to buy any number of toys that fit in the knapsack. The problem is that not all toys give him the same happiness, so he has to trade off the happiness received from each toy with its size; toys with high happiness/size ratio are prefered. Turns out this problem is NP-hard if the numbers are given in binary. We are interested in a counting version of the problem that uses just the sizes.

DEFINITION 10 *Given n weights $w_1, w_2, \ldots, w_n$ and a target weight W, a* feasible solution *to the knapsack problem is a subset T such that $\sum_{i \in T} w_i \leq W$.*

We wish to approximately count the number of feasible solutions. This had been the subject of some very technical papers, until M. Dyer gave a very elementary solution in 2003.

First, we note that the counting problem can be solved *exactly* in $O(nW)$ time, though of course this is not polynomial since W is given to us in binary, i.e. using $\log W$ bits. The idea is dynamic programming. Let $\text{Count}(i, U)$ denote the number of feasible solutions involving only the first i numbers, and whose total weight is at most U. The dynamic programming follows by observing that there are two types of solutions: those that involve the ith element, and those that don't. Thus

$$\text{Count}(i, U) = \begin{cases} \text{Count}(i-1, U-w_i) + \text{Count}(i-1, U) \\ 1 \qquad \text{if } i = 1 \text{ and } w_1 \leq U \\ 0 \qquad \text{if } i = 1 \text{ and } w_1 > U \end{cases}$$

Denoting by S the set of feasible solutions, $|S| = \text{Count}(n, W)$. But as observed, computing this exactly is computationally expensive and not polynomial-time. Dyer's next idea is to find a set Ω containing S but at most n times bigger. This set Ω can be exactly counted as well as sampled from. So then by the Monte Carlo method we can estimate the size of S in polynomial time by drawing samples from Ω.

Ω is simply the set of solutions to a Knapsack instance in which the weights have been *rounded* to lie in $[0, n^2]$. Specifically, let $w'_i = \lfloor \frac{w_i n^2}{W} \rfloor$ and $W' = n^2$. Then Ω is the set of feasible solutions to this modified knapsack problem.

CLAIM 1: $S \subseteq \Omega$. (Consequently, $|S| \leq |\Omega|$.)
This follows since if $T \in S$ is a feasible solution for the original problem, then $\sum_i w'_i \leq \sum_i w_i n^2 / W \leq n^2$, and so T is a feasible solution for the rounded problem.

CLAIM 2: $|\Omega| \leq n\,|S|$.
To prove this we give a mapping g from Ω to S that is at most n-to-1.

$$g(T') = \begin{cases} = T' & \text{if } T' \in S \\ = T' \setminus \{j\} & \text{(else)} \end{cases} \qquad \text{where } j = \text{index of element in } T' \text{ with highest value of } w'_j$$

In the second case note that this element j satisfies $w_j > W/n$ which implies $w'_j \geq n$.

Clearly, g is at most n-to-1 since a set T in S can have at most n pre-images under g. Now let's verify that $T = g(T')$ lies in S.

$$\begin{aligned}\sum_{i\in T} w_i &\leq \sum_{i\in T} \frac{W}{n^2}(w'_i + 1) \\ &\leq \frac{W}{n^2} \times (W' - w'_j + n - 1) \\ &\leq W \qquad (\text{since } W' = n^2 \text{ and } w'_j \geq n\ )\end{aligned}$$

which implies $T \in S$. □

Sampling algorithm for Ω To sample from Ω, use our earlier equivalence of approximate counting and sampling. That algorithm needs an approximate count not only for $|\Omega|$ but also for the subset of Ω that contain the first element. This is another knapsack problem and can thus be solved by Dyer's dynamic programming. And same is true for instances obtained in the recursion.

Bibliography

1. On the Hardness of Approximate Multivariate Integration. I. Koutis, *Proc. Approx-Random 2013*. Springer Verlag.

2. The complexity of enumeration and reliability problems. L. Valiant. *SIAM J. Computing*, 8:3 (1979), pp.410-421.

Chapter 22

Taste of cryptography: Secret sharing and secure multiparty computation

Cryptography is the ancient art/science of sending messages so they cannot be deciphered by somebody who intercepts them. This field was radically transformed in the 1970s using ideas from computational complexity. Encryption schemes were designed whose decryption by an eavesdropper requires solving computational problems (such as integer factoring) that're believed to be intractable. You may have seen the famous RSA cryptosystem at some point. It is a system for giving everybody a pair of keys (currently each is a 1024-bit integer) called a *public key* and a *private key*. The public key is published on a public website; the private key is known only to its owner. Person x can look up person y's public-key and encrypt a message using it. Only y has the private key necessary to decode it; everybody else will gain no information from seeing the encrypted message.

Since the 1980s though, the purview of cryptography greatly expanded. In inventions that anticipated threats that wouldn't materialize for another couple of decades, cryptographers designed solutions such as private multiparty computation, proofs that yield nothing but their validity, digital signatures, digital cash, etc. Today's lecture is about one such invention due to Ben-or, Goldwasser and Wigderson (1988), secure multiparty computation, which builds upon the Reed Solomon codes studied last time.

The model is the following. There are n players, each holding a private number (say, their salary, or their *vote* in an election). The ith player holds s_i. They wish to compute a joint function of their inputs $f(s_1, s_2, \ldots, s_n)$ such that nobody learns anything about anybody else's secret input (except of course what can be inferred from the value of f). The function f is known to everybody in advance (e.g., $s_1^2 + s_2^2 + \cdots + s_n^2$).

Admittedly, this sounds impossible when you first hear it.

22.1 Shamir's secret sharing

We first consider a *static* version of the problem that introduces some of the ideas.

Say we want to distribute a secret among n, say a_0. (For example, a_0 could be the secret key to decrypt an important message.) We want the following property: every subset of $t+1$ people should be able to pool their information and recover the secret, but no subset of t people should not be able to pool their information to recover any information at all about the secret.

For simplicity interpret a_0 as a number in a finite field Z_q. Then pick t random numbers $a_1, a_2, \ldots, a_t$ in Z_q and constructing the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_t x^t$ and evaluate it at n points $x_1, x_2, \ldots, x_n$ that are known to all of them. Then give $p(x_i)$ to person i.

Notice, the set of shares are t-wise independent random variables. (Each subset of t shares is distributed like a random t-tuple over Z_q.) This follows from polynomial interpolation (which we explained last time using the Vandermode determinant): for every t-tuple of people and every t-tuple of values $y_1, y_2, \ldots, y_t \in Z_q$, there is a unique polynomial whose constant term is a_0 and which takes these values for those people. Thus every t-tuple of values is equally likely, *irrespective of* a_0, and gives no information about a_0.

Furthermore, since p has degree t, each subset of $t+1$ shares can be used to reconstruct $p(x)$ and hence also the secret a_0.

22.2 Multiparty computation: the model

Multiparty computation vastly generalizes Shamir's idea, allowing the players to do arbitrary algebraic computation on the secret input using their "shares."

Player i holds secret s_i and the goal is for everybody to know $f(s_1, s_2, \ldots, s_n)$ at the end, where f is a publicly known function (everybody has the code). No subset of t players can pool their information to get any information about anybody else's input that is not implicit in the output $f(s_1, s_2, \ldots, s_n)$. (Note that if $f()$ just outputs its first coordinate, then there is no way for the first player's secret s_1 to not become public at the end.)

We are given a *secret* channel between each pair of players, which cannot be eavesdropped upon by anybody else. Such a secret channel can be ensured using, for example, a public-key infrastructure. If everybody's public keys are published, player i can look up player j's public-key and encrypt a message using it. Only player j has the private key necessary to decode it; everybody else will gain no information from seeing the encrypted message.

The result only applies to algebraic computations.

Definition 11 (Algebraic programs) *A size m algebraic straight line program with inputs $x_1, x_2, \ldots, x_n \in Z_q$ is a sequence of m lines of the form*

$$y_i \leftarrow y_{i_1} \, op \, y_{i_2},$$

where $i_1, i_2 < i$; $op =$ "$+$"or "$\times$,"or "$-$"and $y_i = x_i$ for $i = 1, 2, \ldots, n$. The output of this straight line program is defined to be y_m.

A simple induction shows that a straight line program with inputs $x_1, x_2, \ldots x_n$ computes a multivariate polynomial in these variables. The degree can be rather high, about 2^m. So this is a powerful model.

(Aside: Straight line programs are sometimes called *algebraic circuits*. If you replace the arithmetic operations with boolean operations $\vee, \neg, \wedge$ you get a model that can do any computation at all.)

22.3 Easy protocol: linear combinations of inputs

First we describe a simple protocol that allows the players to compute $f(s_1, s_2, \ldots, s_n) = \sum_i c_i s_i$ for any coefficients $c_1, c_2, \ldots, c_n \in Z_q$ known to all of them.

Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be n distinct nonzero values in Z_q known to all.

Each player does a version of Shamir's secret sharing. Player i picks t random numbers $a_{i1}, a_{i2}, \ldots, a_{it} \in Z_q$ and evaluates the polynomial $p_i(x) = s_i + a_{i1}x + a_{i2}x^2 + \cdots + a_{it}x^t$ at $\alpha_1, \alpha_2, \ldots, \alpha_n$, and sends those values to the respective n players (keeping the value at α_i for himself) using the secret channels. Let γ_{ij} be the secret sent by player i to player j.

After all these shares have been sent around, the players get down to computing f, i.e., $\sum_i c_i s_i$. This is easy. Player k computes $\sum_i c_i \gamma_{ik}$. In other words, he treats the shares he received from the others as *proxies* for their input.

OBSERVATION: *The numbers computed by the kth player correspond to value of the following polynomial at $x = \alpha_k$:*

$$\sum_i c_i s_i + \sum_r (\sum_i a_{ir}) x^r.$$

Thus the first t players can now send their computed numbers to everybody else. Then everybody has $t+1$ values of this polynomial, allowing them to reconstruct it and thus also reconstruct the constant term, which is the desired output.

22.4 General protocol: + and × suffice

The above protocol for + seems rather trivial. But our definition of Algebraic programs shows that if we can design a protocol that allows *multiplying* of secret values, then that is good enough to implement any algebraic computation. Let the variables in the algebraic program be $y_1, y_2, \ldots, y_m$.

DEFINITION 12 ((t,n)- SECRETSHARING) *If $a_0 \in Z_q$ then its* (t,n)- secretsharing *is a sequence of n numbers $\beta_1, \beta_2, \ldots, \beta_n$ obtained as in Section 22.1 by using a polynomial of the form $a_0 + \sum_{i=1}^{t} a_i x^i$, where $a_1, a_2, \ldots, a_n$ are random numbers in Z_q.*

The general invariant maintained by the protocol is the following: *At the end of step i, the n players hold the n values in some (t,n)-secretsharing of the value of y_i.*

Clearly, at the start of the protocol such a secretsharing for the values of the n input variables $x_1, x_2, \ldots, x_n$ has been divided among the players. So the invariant is true for $i \leq n$. Assuming it is true for i we show how to maintain it for $i+1$. If y_{i+1} is the + of two earlier variables, then the simple protocol of Section 22.3 allows the invariant to be maintained.

So assume y_{i+1} is the × of two earlier variables. If these two earlier variables were secretshared using polynomials $g(x) = \sum_{r=0}^{t} g_r x^r$ and $h(x) \sum_{r=0}^{t} h_r x^r$ then the values being secretshared are g_0, h_0 and the obvious polynomial to secretshare their product is $\pi(x) =$

$g(x)h(x) = \sum_{r=0}^{2t} x^r \sum_{j \le r} g_j h_{r-j}$. The constant term in this polynomial is $g_0 h_0$ which is indeed the desired product. Secretsharing this polynomial means everybody takes their share of g and h respectively and multiplies them. Nothing more to do.

Unfortunately, this polynomial π has two problems: the degree is $2t$ instead of t and, more seriously, its coefficients are not random numbers in Z_q. Thus it is not a (t, n)-secretsharing of $g_0 h_0$.

The degree problem is easy to solve: just drop the higher degree terms and stay with the first t terms. Dropping terms is a linear operation and can be done using the simple protocol of Section 22.3. We won't go into details.

To solve the problem about the coefficients not being random numbers, each of the players does the following. The kth player picks a random degree $2t$ polynomial $r_k(x)$ whose constant term is 0. Then he secret shares this polynomial among all the other players. Now the players can compute their secretshares of the polynomial

$$\pi(x) + \sum_{k=1}^{n} r_k(x),$$

and the constant term in this polynomial is still $g_0 h_0$. Then they apply truncation to this procedure to drop the higher order terms. Thus at the end the players have a (t, n)-secretsharing of the value y_{i+1}, thus maintaining the invariant.

Subtleties The above description assumes that the malicious players follow the protocol. In general the t malicious players may not follow the protocol in an attempt to learn things they otherwise can't. Modifying the protocol to handle this —and proving it works—is more nontrivial.

BIBLIOGRAPHY

1. M. BenOr, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault Tolerant Distributed Computation Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88), Chicago, Illinois, pages 1-10, May 1988.

2. A. Shamir. How to share a secret", Communications of the ACM 22 (11): 612613, 1979.

Chapter 23

Real-life environments for big-data computations (MapReduce etc.)

First 2/3rd based upon the guest lecture of Kai Li, and the other 1/3rd upon Sanjeev's lecture

23.1 Parallel Processing

These days many algorithms need to be run on huge inputs; gigabytes still can fit in RAM on a single computer, but terabytes or more invariably require a multiprocessor architecture. This state of affairs seems here to stay since (a) Moore's law has slowed a lot, and there is no solution in sight. So processing speeds and RAM sizes are no longer growing as fast as in the past. (b) Data sets are growing very fast.

Multiprocessors (aka parallel computers) involve multiple CPUs operating on some form of distributed memory. This often means that processors may compete in writing to the same memory location, and the system design has to take this into account. Parallel systems have been around for many decades, and continue to evolve with changing needs and technology.

There are three major types of systems based upon their design:

Shared memory multiprocessor. Multiple processors operate on the same memory; there are explicit mechanisms for handling conflicting updates. The underlying memory architecture may be complicated but the abstraction presented to the programmer is that of a single memory. The programming abstraction often involves *threads* that use *synchronization* primitives to handle conflicting updates to a memory location. *Pros:* The programming is relatively easy; data structures are familiar. *Cons:* Hard to scale to very large sizes. In particular, cannot handle hardware failure (all hell can break lose otherwise).

Message passing models: Different processors control their own memory; data movement is via message passing (this provides implicit synchronization). *Pros:* Such systems are easier to scale. Can use checkpoint/recovery to deal with node failures. *Cons:* No clean data structures.

Commodity clusters: Large number of off-the-shelf computers (with their own memories) linked together with a LAN. There is no shared memory or storage. *Pros:* Easy to scale; can easily handle the petabyte-size or larger data sets. *Cons:* Programming model has to deal explicitly with failures.

Tech companies and data centers have gravitated towards commodity clusters with tens of thousands or more processors. The power consumption may approach that of a small town. In such massive systems failures —processor, power supplies, hard drives etc.—are inevitable. The software must be designed to provide reliability on top of such frequent failures. Some techniques: (a) replicate data on multiple disks/machines (b) replicate computation by splitting into smaller subtasks (c) use good data placement to avoid long latency.

Google pioneered many such systems for their data centers and released some of these for general use. MapReduce is a notable example. The open source community then came up with its own versions of such systems, such as Hadoop. SPARK is another programming environment developed for ML applications.

23.2 MapReduce

MapReduce is Google's programming interface for commodity computing. It is evolved from older ideas in functional programming and databases. It is easy to pick up, but achieving high performance requires mastery of the system.

It abstracts away issues of data replication, processor failure/retry etc. from the programmer. One consequence is that there is no guarantee on running time.

The programming abstraction is rather simple: the data resides in an unsorted clump of (*key, value*) pairs. We call this a *database* to ease exposition. (The programmer has to write a MAPPER function that produces this database from the data.) Starting with such a database, the system applies a SORT that moves all pairs with the same *key* to the same physical location. Then it applies a REDUCE operation –provided by the programmer—that takes a bunch of pairs with the same *key* and applies some combiner function to produce a new single pair with that key and whose *value* is some specified combination of the old values.

EXAMPLE 49 (Word Count) The analog to the usual *Hello World* program in the MapReduce world is the program to count the number of repetitions of each word. The programmer provides the following.

MAPPER Input: a text corpus. Output: for each word w, produce the pair $(w, 1)$. This gives a database.

REDUCE: Given a bunch of pairs of type $(w, count)$ produces a pair of type (w, C) where C is the sum of all the counts.

EXAMPLE 50 (Matrix Vector Multiplication)

MAPPER: Input is an $n \times n$ matrix M, and a $n \times 1$ vector V. Output: Pairs $(i, m_{ij} \cdot v_j)$ for all $\{i, j\}$ for which $m_{ij} \neq 0$.

REDUCER: Again, just adds up all pairs with the same *key* and sum up their values.

One can similarly do other linear algebra operations.

Some other examples of mapreduce programs appear in Jelani Nelson's notes
`http://people.seas.harvard.edu/ minilek/cs229r/lec/lec24.pdf`

The MapReduce paradigm was introduced in the following paper:

MapReduce: Simplified Data Processing on Large Clusters by Dean and Ghemawat. OSDI 2004.

While it has been very influential, it is not suited for all applications. A criticial appraisal appears in a blog post *MapReduce: a major step backwards,* by DeWitt and Stonebraker.

Chapter 24

Heuristics: Algorithms we don't know how to analyze

Any smart teenager who knows how to program can come up with a new algorithm. Analysing algorithms, by contrast, is not easy and usually beyond the teenager's skillset. In fact, if the algorithm is complicated enough, proving things about it (i.e., whether or not it works) becomes very difficult for even the best experts. Thus not all algorithms that have been designed have been analyzed. The algorithms we study today are called *heuristics*: for most of them we know that they do *not* work on worst-case instances, but there is good evidence that they work very well on many instances of practical interest. Explaining this discrepancy theoretically is an interesting and challenging open problem.

Though the heuristics apply to many problems, for pedagogical reasons, throughout the lecture we use the same problem as an example: 3SAT. Recall that the input to this problem consists of *clauses* which are $\vee$ (i.e., logical OR) of three literals, where a literal is one of n variables $x_1, x_2, \ldots, x_n$, or its negation. For example: $(x_1 \vee \neg x_4 \vee x_7) \wedge (x_2 \vee x_3 \vee \neg x_4)$. The goal is to find an assignment to the variables that makes all clauses evaluate to true.

This is the canonical NP-complete problem: every other NP problem can be reduced to 3SAT (Cook-Levin Theorem, early 1970s). More importantly, problems in a host of areas are actually solved this way: convert the instance to an instance of 3SAT, and use an algorithm for 3SAT. In AI this is done for problems such as constraint satisfaction and motion planning. In hardware and software verification, the job of *verifying* some property of a piece of code or circuit is also reduced to 3SAT.

Let's get the simplest algorithm for 3SAT out of the way: try all assignments. This has the disadvantage that it takes 2^n time on instances that have few (or none) satisfying assignments. But there are more clever algorithms, which run very fast and often solve 3SAT instances arising in practice, even on hundreds of thousand variables. The codes for these are publicly available, and whenever faced with a difficult problem you should try to represent it as 3SAT and use these solvers.

24.1 Davis-Putnam procedure

The Davis-Putnam procedure from the 1950s is very simple. It involves assigning values to variables one by one, and *simplifying* the formula at each step. For instance, if it contains a clause $x_3 \vee \neg x_5$ and we have just assigned x_5 to T (i.e., true) then the clause becomes true and can be removed. Conversely, if we assign it F the then the only way the remaining variables can satisfy the formula is if $x_3 = T$. Thus $x_5 = F$ *forces* $x_3 = T$. We call these effects the *simplification* of the formula.

Say the input is φ. Pick a variable, say x_i. Substitute $x_i = T$ in φ and simplify it. Recursively check the simplified formula for satisfiability. If it turns out to be unsatisfiable, then substitute $x_i = F$ in φ, simplify it, and recursively check that formula for satisfiability. If that also turns out unsatisfiable, then declare φ unsatisfiable.

When implementing this algorithm schema one has various choices. For instance, which variable to pick? Random, or one which appears in the most clauses, etc. Similarly, whether to try the value T first or F? What data structure to use to keep track of the variables and clauses? Many such variants have been studied and surprisingly, they do very well in practice. Hardware and software verification today relies upon the ability to solve instances with hundreds of thousands of variables.

CLAUSE LEARNING. The most successful variants of this algorithm involves learning from experience. Suppose the formula had clauses $(x_1 \vee x_7 \vee x_9)$ and $(x_1 \vee \neg x_9 \vee \neg x_6)$ and along some branch the algorithm tried $x_1 = F, x_7 = F, x_6 = T$, which led to a contradiction since x_9 is being forced to both T and F. Then the algorithm has learnt that this combination is forbidden, not only at this point but on every other branch it will explore in future. This knowledge can be added in the form of a new clause $x_1 \vee x_7 \vee \neg x_6$, since every satisfying assignment has to satisfy it. As can be imagined, clause learning comes in myriad variants, depending upon what rule is used to infer and add new clauses.

One of you asked why adding clauses (ie more constraints) simplifies the problem instead of making it harder. The answer is that the clauses can be seen as guidance towards a satisfying assignment (if one exists). The clauses can be used in making the crucial decision in DPLL procedures about which variable to set, and how to set it (T or F). The wrong decision may cause you to potentially incur huge cost. So anything that lowers the probability of wrong decision by even a bit could drastically change your running time.

24.2 Local search

The above procedures set variables one by one. There is a different family of algorithms that does this in a different way. A typical is Papadimitriou's **Walksat** algorithm: *Start with a random assignment. At each step, pick a random variable and switch its value. If this increases the number of satisfied clauses, make this the new assignment. Continue this way until the number of satisfied clauses cannot be increased.* Papadimitriou showed that this algorithm solves $2SAT$ with high probability.

Such algorithms fit in a more paradigm called *local search*, which can be described as follows.

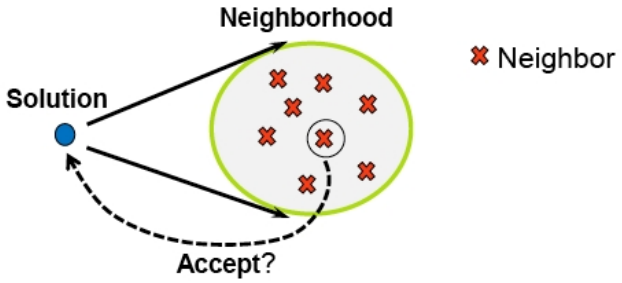

Figure 24.1: Local search algorithms try to improve the solution by looking for small changes that improve it.

> *Maintain a solution at each step. If the current solution is x, look for a solution y in a neighborhood $Ball(x, r)$ of radius r around x (that is, all solutions that differ from x up to some amount small r). If you find such a y that improves over x (in terms of the objective being optimized) then replace x by y. Stop if no such y was found.*

Clearly, when the algorithm stops, the current solution is optimal in its neighborhood (i.e., locally optimal). One can think of this as a discrete analog of *gradient descent*. An example of nonlocal change is any of the global optimization algorithms like Ellipsoid method.

Thus local search is a formalization of improvement strategies that we come up with intuitively, e.g., change ourselves by making small continuous changes. The Japanese have a name for it: *kaizen*[1]

EXAMPLE 51 Local search is a popular and effective heuristic for many other problems including traveling salesman and graph partitioning. For instance, one local search strategy (which even students in my freshman seminar were able to quickly invent) is to start with a tour, and at each step try to improve it by changing up to two edges (2-OPT) or k edges (k-OPT). We can find the best local improvement in polynomial time (there are only $\binom{n}{2}$ ways to choose 2 edges in a tour) but the number of local improvement steps may be exponential in n. So the overall running time may be exponential.

These procedures often do well in practice, though theoretical results are few and far between. One definitive study is
The traveling salesman problem: A case study in local optimization, by D. Johnson and C. McGeoch. 1997

EXAMPLE 52 *Evolution* a la Darwin can be seen as a local search procedure. Mutations occur spontaneously and can be seen as exploring a small neighborhood of the organism's genome. The environment gives feedback over the quality of mutations. If the mutation is good, the descendents thrive and the mutation becomes more common in the gene pool. (Thus the mutated genome becomes the new solution y in the local search). If the mutation is harmful the descendents die out and the mutation is thus removed from the gene pool.

[1] I would like to know if Japanese magazines have cover stories on new kaizen ideas just as cover stories in US magazines promote radical makeovers.

24.3 Difficult instances of 3SAT

We do know of hard instances for 3SAT for such heuristics. A simple family of examples uses the fact that there are small logical circuits (i.e., acyclic digraphs using nodes labeled with the gates $\vee, \wedge, \neg$)for integer multiplication. The circuit for multiplying two n-bit numbers has size about $O(n \log^2 n)$. So take a circuit C that multiplies two 1000 bit numbers. Input two random prime numbers p, q in it and evaluate it to get a result r. Now construct a boolean formula with $2n + O(|C|)$ variables corresponding to the input bits and the internal gates of C, and where the clauses capture the computation of each gate that results in the output r. (Note that the bits of r are "hardcoded "into the formula, but the bits of p, q as well as the values of all the internal gates correspond to variables.) Thus finding a satisfying assignment for this formula would also give the factors of r. (Recall that factoring a product of two random primes is the hard problem underlying public-key cryptosystems.) The above SAT solvers have difficulty with such instances.

Other families of difficult formulae correspond to simple math theorems. A simple one is: *Every partial order on a finite set has a maximal element.* A *partial order* on n elements is a relation $\prec$ satisfying: (a) $x_i \not\prec x_i \quad \forall i$. (b) $x_i \prec x_j$ and $x_j \prec x_k$ implies $x_i \prec x_k$ (transitivity) (c) $x_i \prec x_j$ implies $x_j \not\prec x_i$. (Anti-symmetry).

For example, the relationship "is a divisor of"is a partial order among integers. We can represent a partial order by a directed acyclic graph.

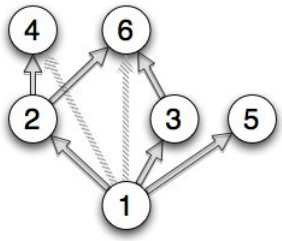

Figure 24.2: The relation "is a divisor of"is a partial order among integers.

Clearly, for every partial order on a finite set, there is a *maximal* element i such that $i \not\prec j$ for all j (namely, any leaf of the directed acyclic graph.) This simple mathematical statement can be represented as an unsatisfiable formula. However, the above heuristics seem to have difficulty detecting that it is unsatisfiable.

This formula has a variables x_{ij} for every pair of elements i, j. There is a family of clauses representing the properties of a partial order.

$$\neg x_{ii} \quad \forall i$$
$$\neg x_{ij} \vee \neg x_{jk} \vee x_{ik} \quad \forall i, j, k$$
$$\neg x_{ij} \vee \neg x_{ji} \quad \forall i, j$$

Finally, there is a family of clauses saying that no i is a maximal element. These clauses

don't have size 3 but can be rewritten as clauses of size 3 using new variables.

$$x_{i1} \vee x_{i2} \vee \cdots \vee x_{in} \quad \forall i$$

24.4 Random SAT

One popular test-bed for 3SAT algorithms are *random* instances. A random formula with m clauses is picked by picking each clauses independently as follows: pick three variables randomly, and then toss a coin for each to decide whether it appears negated or unnegated.

Turns out if $m < 3.9n$ or so, then Davis-Putnal type procedures usually find a satisfying assignment. If $m > 4.3n$ these procedures usually fail. There is a different algorithm called *Survey propagation* that finds algorithms up to m close to $4.3n$. It is conjectured that there is a *phase transition* around $m = 4.3n$ whereby the formula goes from being satisfiable with probability close to 1 to being unsatisfable with probability close to 1. But this conjecture is unproven, as is the conjecture that survey propagation works up to this threshold.

Now we show that if $m > 5.2m$ then the formula is unsatisfiable with high probability. This follows since the expected number of satisfying assignments in such a formula is $2^n(\frac{7}{8})^m$ (this follows by linearity of expectation since there are 2^n possible assignments, and any fixed assignment satisfies all the m independently chosen clauses with probability $(\frac{7}{8})^m$). For $m > 5.2n$ this number is very tiny, so by Markov's inequality the probability it is ≥ 1 is tiny.

Note that we do not know how to prove in polynomial time, given such a formula with $m > 5.2n$, that it is unsatisfiable. In fact it is known that known that for $m > Cn$ for some large constant C, the simple DP-style algorithms take exponential time.

24.5 Metropolis-Hastings and Computational statistics

Now we turn to counting problems and statistical estimation, discussed earlier in Lecture 21. Recall the Monte Carlo method for estimating the area of a region: through darts and see what fraction land in the region.

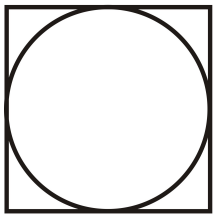

Figure 24.3: Monte Carlo (dart throwing) method to estimate the area of a circle. The fraction of darts that fall inside the disk is $\pi/4$.

Now suppose we are trying to integrate a nonnegative valued function f over the region. Then we should throw a dart which lands at x with probability $f(x)$. We'll examine how to throw such a dart.

First note that this is an example of sampling from a probability distribution for which only the *density* function is known. Say the distribution is defined on $\{0,1\}^n$ and we have a *goodness* function $f(x)$ that is nonnegative and computable in polynomial time given $x \in \{0,1\}^n$. Then we wish to sample from the distribution where probability of getting x is *proportional* to $f(x)$. Since probabilities must sum to 1, we conclude that this probability is $f(x)/N$ where $N = \sum_{x\in\{0,1\}^n} f(x)$ is the so-called *partition function*. The main problem problem here is that N is in general hard to compute; it is complete for the class $\sharp P$ mentioned in the earlier lecture.

EXAMPLE 53 The dart throwing/integration problem arises in machine learning (more generally, statistical procedures). For instance if there is a density $p(x,y)$ and we wish to compute $p(x|y)$ using Bayes' rule then we need $p(xy)/p(y)$, and

$$p(y) = \int f(x,y)dx.$$

Lets note that if one could do such dart throwing in general, then 3SAT becomes easy. Suppose the formula has n variables and m clauses. For any assignment x define $f(x) = 2^{2nf_x}$ where f_x = number of clauses satisfied by x. Then if the formula has a satisfiable assignment, then $N > 2^{2nm}$ whereas if the formula is unsatisfiable then $N < 2^n \times 2^{2n(m-1)} < 2^{2nm}$. In particular, the mass $f(x)$ of a satisfying assignment exceeds the mass of all unsatisfying assignments. So the ability to sample from the distribution would yield a satisfying assignment with high probability.

The Metropolis-Hastings algorithm (named after its inventors) is a heuristic for sampling from such a distribution. Define the following random walk on $\{0,1\}^n$. *At every step the walk is at some $x \in \{0,1\}^n$. (At the beginning use an arbitrary x.) At every step, toss a coin. If it comes up heads, stay at x. (In other words, there is a self-loop of probability at least 1/2.) If the coin came up tails, then randomly pick a neighbor x' of x. Move to x' with probability* $\min 1, \frac{f(x')}{f(x)}$. *(In other words, if $f(x') \geq f(x)$, definitely move. Otherwise move with probability given by their ratio.)*

CLAIM: *If all $f(x) > 0$ then the stationary distribution of this Markov chain is exactly $p(x)/N$, the desired distribution*

PROOF: The markov chain defined by this random walk is ergodic since $f(x) > 0$ implies it is connected, and the self-loops imply it mixes. Thus it suffices to show that the (unique) stationary distribution has the form $f(x)/K$ for some scale factor K, and then it follows that K is the partition function N. To do so it suffices to verify that such a distribution is stationary, i.e., in one step the probability flowing out of a vertex equals its inflow. For any x, lets L be the neighbors with a *lower* f value and H be the neighbors with value at least as high. Then the outflow of probability per step is

$$\frac{f(x)}{2K}(\sum_{x'\in L} \frac{f(x')}{f(x)} + \sum_{x'\in H} 1),$$

whereas the inflow is

$$\frac{1}{2}(\sum_{x'\in L} \frac{f(x')}{K} \cdot 1 + \sum_{x'\in H} \frac{f(x')}{K}\frac{f(x)}{f(x')}),$$

and the two are the same. □

Note: The advantage of the random walk method is that it can in principle explore a space of exponential size while using only space for storing the current x. In this sense it is like local search. In fact it is like a probabilistic version of local search on the objective $f(x)$. In local search one would move from x to x' if that improves f, whereas here the move is made with some probability depending upon $f(x)/f(x')$ and every possible move has a nonzero probability.

Simulated Annealing. If we use the suggested goodness function for 3SAT $f(x) = 2^{2nf_x}$ then this Markov chain can be shown to have poor mixing. So a variant is to use a markov chain that updates itself. The goodness function is initialized to say $2^{\gamma f_x}$ for $\gamma = 1$, then allowed to mix. This stationary distribution may put too little weight on the satisfying assignments. So then slowly increase γ from 1 to $2n$, allowing the chain to mix for a while at each step. This family of algorithms is called *simulated annealing*, named after the physical process of annealing.

For further information see this survey and its list of references.

Satisfiability Solvers, by C.P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. *Handbook of Knowledge Representation*, Elsevier 2008.

PRINCETON UNIVERSITY FALL '14 COS 521:ADVANCED ALGORITHMS

Homework 1

Out: *Sep 25* Due: *Oct 2*

You can collaborate with your classmates, but be sure to list your collaborators with your answer. If you get help from a published source (book, paper etc.), cite that. The answer must be written by you and you should not be looking at any other source while writing it. Also, limit your answers to one page, preferably less —you just need to give enough detail to convince the grader.

Typeset your answer in latex (if you don't know latex, you can write by hand but scan in your answers into pdf form before submitting). You can scanners in the mailroom and also using most smartphones.

§1 The simplest model for a *random graph* consists of n vertices, and tossing a fair coin for each pair $\{i, j\}$ to decide whether this edge should be present in the graph. Call this $G(n, 1/2)$. A triangle is a set of 3 vertices with an edge between each pair. What is the expected number of triangles? What is the variance? Use the Chebyshev inequality to show that the number is concentrated around the expectation and give an expression for the exact decay in probability. Is it possible to use Chernoff bounds in this setting?

§2 (Part 1): You are given a fair coin, and a program that generates the binary expansion of p upto any desired accuracy. Formally describe the procedure to simulate a biased coin that comes up with head with probability p. (This was sketched in class.) (Part 2) Now, show how to do the reverse: generate a fair coin toss using a biased coin but where the bias is unknown.

§3 A cut is said to be a *B-approximate min cut* if the number of edges in it is at most B times that of the minimum cut. Show that a graph has at most $(2n)^{2B}$ cuts that are B-approximate. (Hint: Run Karger's algorithm until it has $2B + 1$ supernodes. What is the chance that a particular B-approximate cut is still available? How many possible cuts does this collapsed graph have?)

§4 Show that given n numbers in $[0, 1]$ it is impossible to estimate the *value* of the median within say 1.1 factor with $o(n)$ samples. (Hint: to show an impossibility result you show two different sets of n numbers that have very different medians but which generate —whp—identical samples of size $o(n)$.)

Now calculate the sample size needed (as a function of t) so that the following is true: with high probability, the median of the sample has at least $n/2 - t$ numbers less than it and at least $n/2 - t$ numbers more than it.

§5 Consider the following process for matching n jobs to n processors. In each step, every job picks a processor at random. The jobs that have no contention on the processors they picked get executed, and all the other jobs *back off* and then try again. Jobs only

take one round of time to execute, so in every round all the processors are available. Show that all the jobs finish executing whp after $O(\log \log n)$ steps.

§6 In class we saw a hash to estimate the size of a set. Change it to estimate frequencies. Thus there is a stream of packets each containing a *key* and you wish to maintain a data structure which allows us to give an estimate at the end of the *number of times* each key appeared in the stream. The size of the data structure should not depend upon the number of distinct keys in the stream but can depend upon the success probability, approximation error etc. Just shoot for the following kind of approximation: if a_k is the true number of times that key k appeared in the stream then your estimate should be $a_k \pm \varepsilon(\sum_k a_k)$. In other words, the estimate is going to be accurate only for keys that appear frequently ("heavy hitters") in the stream. (This is useful in detecting anomalies or malicious attacks.) Hint: Think in terms of maintaining $m_1 \times m_2$ counts using as many independent hash functions, where each key updates m_2 of them.

§7 In Matlab or another suitable programming environment implement a pairwise independent hash function and use it to map $\{100, 200, 300, ..., 100n\}$ to a set of size around n. (Use $n = 10^5$ for starters.) Report the largest bucket size you noticed. Then make up a hash function of your own design (could involve crazy stuff like taking XOR of bits, etc.) and repeat the experiment with it and report the largest bucket size. Include your code with your answer and brief description of any design decisions.

PRINCETON UNIVERSITY FALL '14 COS 521:ADVANCED ALGORITHMS

Homework 2

Out: *Oct 7* Due: *Oct 16*

You can collaborate with your classmates, but be sure to list your collaborators with your answer. If you get help from a published source (book, paper etc.), cite that. The answer must be written by you and you should not be looking at any other source while writing it. Also, limit your answers to one page, preferably less —you just need to give enough detail to convince the grader.

Typeset your answer in latex (if you don't know latex, scan your handwritten work into pdf form before submitting). To make things easier to grade, submit answers in the numbered order listed below, and also make sure your name appears on every page.

§1 Draw the full tree of possibilities for the cake-eating problem discussed in class, and compute the optimum cake-eating schedule. To keep the tree size manageable, draw it with the following slight changes: The amount you eat each day has to be an integer multiple of 1/3, and on each day your roommates will with probability 1/2 eat 1/3 of the cake.) (If multiple branches or sub-branches are identical, you may label the branch with a variable and use the variable in lieu of re-drawing the branch. You may also omit branches where you eat 0 cake.)

§2 (*Stable Matchings with Real-Valued Utilities*) We saw stable matchings in a guest lecture. Another formulation of the bipartite stable matching problem has each agent i submit a real number $u_i(j)$ for each element j in the opposite partition, representing the utility of being matched with that element. We then define a perfect matching M to be *stable* if there does not exist a pair $(v, w) \notin M$ such that both $u_v(w) > u_v(v')$ and $u_w(v) > u_w(w')$ where both (v, v') and (w, w') are in M.

(a) Prove that if the two partitions of the graph are of equal size, $u_v(w) = u_w(v)$ for all pairs (v, w), and $u_v(w) \neq u_{v'}(w')$ for all $\{v, w\} \neq \{v', w'\}$ then there exists a unique stable matching among the agents.

(b) Show by example that if we remove the final condition (that utilities are unique between different pairs of agents) from part (a), then the instance can contain multiple stable matchings.

§3 In ℓ_2 regression you are given datapoints $x_1, x_2, \ldots, x_n \in \Re^k$ and some values $y_1, y_2, \ldots, y_n \in \Re$ and wish to find the "best"linear function that fits this dataset. A frequent choice for best fit is the one with *least squared error*, i.e. find $a \in \Re^k$ that minimizes

$$\sum_{i=1}^{h} |y - a \cdot x_i|^2.$$

Show how to solve this problem in polynomial time (hint: reduce to solving linear equations; at some point you may need a certain matrix to be invertible, which you can assume.).

§4 (Firehouse location) Suppose we model a city as an m-point finite metric space with $d(x, y)$ denoting the distance between points x, y. These $\binom{m}{2}$ distances (which satisfy triangle inequality) are given as part of the input. The city has n houses located at points $v_1, v_2, \ldots, v_n$ in this metric space. The city wishes to build k firehouses and asks you to help find the best locations $c_1, c_2, \ldots, c_k$ for them, which can be located at any of the m points in the city. The *happiness* of a town resident with the final locations depends upon his distance from the closest firehouse. So you decide to minimize the cost function $\sum_{i=1}^{n} d(v_i, u_i)$ where $u_i \in \{c_1, c_2, \ldots, c_k\}$ is the firehouse closest to v_i. Describe an LP-based algorithm that runs in $poly(m)$ time and solves this problem approximately. If OPT is the optimum cost of a solution with k firehouses, your solution is allowed to use $O(k \log m)$ firehouses and have cost at most $(1 + \varepsilon)$OPT.

§5 In class we designed a 3/4-approximation for MAX-2SAT using LP rounding. Extend it to a 3/4-approximation for MAX-SAT (i.e., where clauses can have 1 or more variables). Hint: you may also need the following idea: if a clause has size k and we randomly assign values to the variables (i.e., 0/1 with equal probability) then the probability we satisfy it is $1 - 1/2^k$.

§6 You are given data containing grades in different courses for 5 students. As discussed in Lecture 5, we are trying to "explain" the grades as a linear function of the student's aptitude, the easiness of the course and some error term. Denoting by Grade_{ij} the grade of student i in course j this linear model hypothesizes that

$$\text{Grade}_{ij} = \text{aptitude}_i + \text{easiness}_j + \varepsilon_{ij},$$

where ε_{ij} is an error term.

As we saw in class, the problem of finding the best model that minimizes the sum of the $|\varepsilon_{ij}|$'s can be solved by an LP. Your goal is to use any standard package for linear programming (Matlab/CVX, Freemat, Sci-Python, Excel etc.; we recommend CVX on matlab) to fit the best model to this data. Include a printout of your code, and the calculated easiness values of all the courses and the aptitudes of all the students.

	MAT	CHE	ANT	REL	POL	ECO	COS
Alex			C+	A	B+	A-	C+
Billy	B+	A-			A-	B	B
Chris	B	B+			A	A-	B+
David	A		B-	A		A-	
Elise		B-	C	B+	B	B	C

Assume $A = 10, B = 8$ and so on. Let $B+ = 9$ and $A- = 9.5$. (If you use a different numerical conversion please state it clearly.)

§7 (Optimal life partners via MDP) Your friend is trying to find a life partner by going on dates with n people selected for her by an online dating servie. After each date she has two choices: select the latest person she dated and stop the process, or reject this person and continue to date. She has asked you to suggest the optimum *stopping rule*. You can assume that the n persons are all linearly orderable (i.e. given a choice

between any two, she is not indifferent and prefers one over the other). The dating service presents the n chosen people in a random order, and her goal is to maximise the chance of ending up with the person that she will like the most among these n. (Thus ending up even with her second favorite person out of the n counts as failure; she's a perfectionist.) Represent her actions as an MDP, compute the optimum strategy for her and the expected probability of success by following this strategy.

(Hint: The Optimal rule is of the form: *Date γn people and decide beforehand to pass on them. After that select the first person who is preferable to all people seen so far.* You may also need that $\sum_{k=t_1}^{t_2} \frac{1}{k} \approx \ln \frac{t_2}{t_1}$.)

§8 (extra credit) In question 4 try to design an algorithm that uses k firehouses but has cost $O(\mathrm{OPT})$. (Needs a complicated dependent rounding; you can also try other ideas.) Partial credit available for partial progress.

PRINCETON UNIVERSITY FALL '14 COS 521:ADVANCED ALGORITHMS

Homework 3

Out: *Oct 23* Due: *Nov 10*

1. Compute the mixing time (both upper and lower bounds) of a graph on $2n$ nodes that consists of two complete graphs on n nodes joined by a single edge. (Hint: Use elementary probability calculations and reasoning about "probability fluid"; no need for eigenvalues.)

2. Let M be the Markov chain of a 5-regular undirected graph that is connected. Each node has self-loops with probability 1/2. We saw in class that 1 is an eigenvalue with eigenvector $\vec{1}$. Show that every other eigenvalue has magnitude at most $1 - 1/10n^2$. (Hint: First show that a connected graph cannot have 2 eigenvalues that are 1.) What does this imply about the mixing time for a random walk on this graph from an arbitrary starting point?

3. (Game-playing equilibria) Recall the game of Rock, Paper, Scissors. Let's make it quantitative it by saying that the winning player wins \$ 1 whereas the loser gets \$ 0. (In other words, the game is not zero sum.) A draw results in both getting 0. Suppose we make two copies of the multiplicative weight update algorithm to play each other over many iterations. Both start using the uniformly random strategy (i.e., play each of Rock/paper/scissors with probability 1/3) and learn from experience using the MW rule. One imagines that repeated play causes them to converge to some kind of *equilibrium*. (a) Predict by just calculation/introspection what this equilibrium is. (Be honest; it's Ok to be wrong!). (b) Run this experiment on Matlab or any other programming environment and report what you discovered and briefly explain it. (We'll discuss the result in class.)

4. This question will study how mixing can be much slower on directed graphs. Describe an n-node directed graph (with max indegree and outdegree at most 5) that is fully connected but where the random walk takes $\exp(\Omega(n))$ time to mix (and the walk ultimately does mix). Argue carefully.

5. Describe an example (i.e., an appropriate set of n points in $\Re^n$) that shows that the Johnson-Lindenstrauss dimension reduction method —precisely the transformation described in Lecture—the does *not* preserve ℓ_1 distances within even factor 2. (Extra credit: Show that no *linear transformation* suffices, let alone J-L.)

6. (Dimension reduction for SVM's with margin) Suppose we are given two sets P, N of unit vectors in $\Re^n$ with the guarantee that there exists a hyperplane $a \cdot x = 0$ such that every point in P is on one side and every point in N is on the other. Furthermore, the ℓ_2 distance of each point in P and N to this hyperplane is at least ε. Then show using the Johnson Lindenstrauss lemma (hint: you can use it as a black box) that a random linear mapping to $O(\log n/\varepsilon^2)$ dimensions and such that the points are still separable by a hyperplane with margin $\varepsilon/2$.

7. Suppose you are trying to convince your friend that there is no perfect randomness in his head. One way to do it would be to show that if you ask him to write down 100 random bits (say) then his last 20 are fairly predictable after you see the first 80.

   Describe the design of such a predictor using a Markovian model, carefully describing any assumptions. Implement the predictor in any suitable environment and submit the code with your answer. Report the results from a couple of experiments of the following form. Ask a couple of friends to input 100 bits quickly (or 200 if he is patient), and see how well the model predicts the last 20 (or 50) bits. The metric for the model's success in prediction is
   Number of correct guesses − Number of incorrect guesses.
   In order to do better than random guessing this number should be fairly positive.

8. (Extra credit) Calculate the eigenvectors and eigenvalues of the n-dimensional boolean hypercube, which is the graph with vertex set $\{-1, 1\}^n$ and x, y are connected by an edge iff they differ in exactly one of the n locations. (Hint: Use symmetry extensively.)

PRINCETON UNIVERSITY FALL '14 COS 521:ADVANCED ALGORITHMS

Homework 4

Out: *Nov 13* Due: *Dec 2*

1. Implement the portfolio management appearing in the notes for Lecture 16 in any programming environment and check its performance on S& P stock data (download from http://ocobook.cs.princeton.edu/links.htm). Include your code as well as the final performance (i.e., the percentage gain achieved by your strategy).

2. Consider a set of n objects (images, songs etc.) and suppose somebody has designed a *distance* function $d(\cdot)$ among them where $d(i,j)$ is the distance between objects i and j. We are trying to find a geometric realization of these distances. Of course, exact realization may be impossible and we are willing to tolerate a factor 2 approximation. We want n vectors $u_1, u_2, \ldots, u_n$ such that $d(i,j) \leq |u_i - u_j|_2 \leq 2d(i,j)$ for all pairs i, j. Describe a polynomial-time algorithm that determines whether such u_i's exist.

3. The course webpage links to a grayscale photo. Interpret it as an $n \times m$ matrix and run SVD on it. What is the value of k such that a rank k approximation gives a reasonable approximation (visually) to the image? What value of k gives an approximation that looks high quality to your eyes? Attach both pictures and your code. (In matlab you need mat2gray function.) Extra credit: Try to explain from first principles why SVD works for image compression at all.

4. Suppose we have a set of n images and for some multiset E of image pairs we have been told whether they are *similar* (denoted +edges in E) or *dissimilar* (denoted −edges). These ratings were generated by different users and may not be mutually consistent (in fact the same pair may be rated as + as well as −). We wish to *partition* them into clusters $S_1, S_2, S_3, \ldots$ so as to maximise:

   (# of +edges that lie within clusters) + (# of −edges that lie between clusters).

   Show that the following SDP is an upperbound on this, where $w^+(ij)$ and $w^-(ij)$ are the number of times pair i, j has been rated + and − respectively.

$$\begin{aligned} \max \sum_{(i,j)\in E} & w^+(ij)(x_i \cdot x_j) + w^-(ij)(1 - x_i \cdot x_j) \\ & |x_i|_2^2 = 1 \qquad \forall i \\ & x_i \cdot x_j \geq 0 \qquad \forall i \neq j. \end{aligned}$$

5. For the problem in the previous question describe a clustering into 4 clusters that achieves an objective value 0.75 times the SDP value. (Hint: Use Goemans-Williamson style rounding but with two random hyperplanes instead of one. You may need a quick matlab calculation just like GW.)

6. Suppose you are given m halfspaces in $\Re^n$ with rational coefficients. Describe a polynomial-time algorithm to find the largest *sphere* that is contained inside the polyhedron defined by these halfspaces.

7. Let f be an n-variate convex function such that for every x, every eigenvalue of $\nabla^2 f(x)$ lies in $[m, M]$. Show that the optimum value of f is lowerbounded by $f(x) - \frac{1}{2m}|\nabla f(x)|_2^2$ and upperbounded by $f(x) - \frac{1}{2M}|\nabla f(x)|_2^2$, where x is any point. In other words, if the gradient at x is small, then the value of f at x is near-optimal. (Hint: By the mean value theorem, $f(y) = f(x) + \nabla f(x)^T(y-x) + \frac{1}{2}(y-x)^T\nabla^2 f(z)(y-x)$, where z is some point on the line segment joining x, y.)

PRINCETON UNIVERSITY FALL '14 COS 521:ADVANCED ALGORITHMS

Homework 5

Out: *Dec 5* Due: *Dec 13*

1. Prove von Neumann's min max theorem. You can assume LP duality.

2. (Braess's paradox; wellknown to transportation planners) Figure (a) depicts a simple network of roads (each is one-way for simplicity) from point s to t. The number on the edge is the time to traverse that road. When we say the travel time is x, we mean that the time scales *linearly* with the amount of traffic in it.

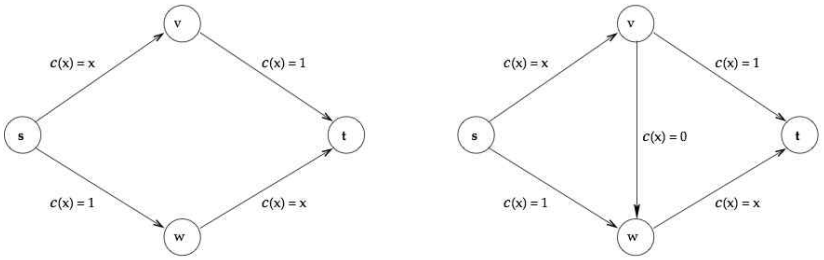

Figure 24.4: Braess's paradox

One unit of traffic (a large number of individual drivers) need to travel from s to t. (Actually assume is it just a tiny bit less than one unit.) Each driver's choice of route can be seen as a move in a multiplayer game. What is the Nash equilibrium and what is each driver's travel time to t in this equilibrium?

Figure (b) depicts the same network with a new superfast highway constructed from v to w. What is the new Nash equilibrium and the new travel time?

3. Show that approximating the number of simple cycles within a factor 100 in a directed graph is NP-hard. (Hint: Show that if there is a polynomial-time algorithm for this task, then we can solve the Hamiltonian cycle problem in directed graphs, which is NP-hard. Here the exact constant 100 is not important, and can even be replaced by, say, n.)

4. (Extra credit) (*Sudan's list decoding*) Let $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n) \in F^2$ where $F = GF(q)$ and $q \gg n$. We say that a polynomial $p(x)$ *describes* k of these pairs if $p(a_i) = b_i$ for k values of i. This question concerns an algorithm that recovers p even if $k < n/2$ (in other words, a majority of the values are wrong).

(a) Show that there exists a bivariate polynomial $Q(z, x)$ of degree at most $\lceil\sqrt{n}\rceil + 1$ in z and x such that $Q(b_i, a_i) = 0$ for each $i = 1, \ldots, n$. Show also that there is an efficient (poly(n) time) algorithm to construct such a Q.

(b) Show that if $R(z, x)$ is a bivariate polynomial and $g(x)$ a univariate polynomial then $z - g(x)$ divides $R(z, x)$ iff $R(g(x), x)$ is the 0 polynomial.

(c) Suppose $p(x)$ is a degree d polynomial that describes k of the points. Show that if d is an integer and $k > (d + 1)(\lceil\sqrt{n}\rceil + 1)$ then $z - p(x)$ divides the bivariate polynomial $Q(z, x)$ described in part (a). (Aside: Note that this places an upper-bound on the number of such polynomials. Can you improve this upperbound by other methods?)

(There is a randomized polynomial time algorithm due to Berlekamp that factors a bivariate polynomial. Using this we can efficiently recover all the polynomials p of the type described in (c). This completes the description of Sudan's algorithm for *list decoding*.)

Princeton University
COS 521: Advanced Algorithms
Final Exam Fall 2014

Sanjeev Arora

Due electronically by Jan 19 5pm.

Instructions: The test has 6 questions. Finish the test within **48 hours** after first reading it. You can consult any notes/handouts from this class and feel free to quote, without proof, any results from there. You **cannot** consult any other source or person in any way.

Do not read the test before you are ready to work on it.

Write and sign the honor code pledge on your exam (The pledge is "I pledge my honor that I have not violated the honor code during this examination.")

Sanjeev, Kevin, and Siyu will be available Jan 11–19 via email and piazza to answer any questions. We will also offer to call you if your confusion does not clear up. In case of unresolved doubt, try to explain your confusion as part of the answer and maybe you will receive partial credit. In general, stating clearly what you are trying to do can get you partial credit.

1. Consider an undirected weighted graph (no self-loops)where w_{ij} denotes the weight on edge $\{i, j\}$. Its Laplacian is the matrix whose entry (i, i) is $\sum_{j \neq i} w_{ij}$ and whose (i, j) entry is $-w_{ij}$. Then prove that the Laplacian can be written as UU^T for some $n \times n$ matrix U.

   Now consider the *positive Laplacian*, which is like above except the (i, j) entry is w_{ij}. Show that this can be written as UU^T where every entry is nonnegative.

2. *Streaming algorithms* are used in scientific or networking settings when a very long stream of numbers is whizzing by, and the algorithm lacks the space to store them all or time to process them offline. Suppose the stream consists of M integers in $\{1, \ldots, N\}$ and let m_i be the number of times i appears in the stream. *Gini's homogeneity index* G is defined as $\sum_{i=1}^{N} m_i^2$. You have to design a randomized streaming algorithm that uses $O(\frac{\log 1/\varepsilon}{\delta^2} \log(M+N))$ bits of storage and computes a number in $[(1-\delta)G, (1+\delta)G]$ with probability at least $(1 - \varepsilon)$.

   (Hint: Consider the estimator $(\sum_i s_i m_i)^2$ where $s_1, s_2, \ldots, s_N$ are 4-wise independent random variables in $\{-1, 1\}$. Argue carefully about storage. Generous credit given if you get approximately the right bound.)

3. Suppose we are given a source that produces a vector-valued signal of the following form, where $u, v \in \Re^n$ are two orthogonal unit vectors. With probability $1/2$ the source outputs a vector of the form $bu + \eta$ where b is randomly chosen from $[4 \log n, 8 \log n]$, and with probability $1/2$ it outputs $bv + \eta$. Here $\eta \sim \mathcal{N}(0, I)$ is a random spherical gaussian vector, whose each coordinate is independently distributed as a univariate gaussian of variance 1 and mean 0.

   Describe how to recover in polynomial time both u, v with error at most $1/n$ in each coordinate. Your algorithm can draw as many samples from this source as it needs.

   (Hint: Try to first identify the subspace spanned by u, v, and then recover u and v.)

4. A randomized algorithm that uses r random bits can be thought of as a distribution over 2^r deterministic algorithms, which allows it in principle to be exponentially more powerful than a deterministic algorithm. *Yao's Principle* is a way to prove lower bounds on the performance of randomized algorithms. Suppose the set of all possible deterministic algorithms running with a specified resource bound is finite, and denoted $\mathcal{A}$. The set of all inputs of a certain size is denoted $\mathcal{I}$. For $x \in \mathcal{I}$ and $A \in \mathcal{A}$ there is some cost of running A on input x, denoted $\text{cost}(A, x)$. Then a randomized algorithm is a distribution R on $\mathcal{A}$ and the *expected cost* of running it on input x is $E_{A \in R}[\text{cost}(A, x)]$.

   Now prove Yao's principle:

$$\min_R \max_{x \in \mathcal{I}} E_{A \in R}[\text{cost}(A, x)] = \max_{D:\ \text{distrib. on } \mathcal{I}} \min_{A \in \mathcal{A}} E_{x \in \mathcal{D}}[\text{cost}(A, x)].$$

   Use this principle to show that every randomized algorithm that distinguishes between the following two cases with probability at least 0.9 must examine $\Omega(1/\varepsilon)$ bits in the n-bit input. (a) Case 1: The input contains all 1's. (b) Case 2: At least ε fraction of bits are 0's.

5. In class we showed the existence of good error correcting codes. Now let us see that there actually exist good error correcting codes with linear structure. A linear error correcting code over $GF(2)^n$ is a $m \times n$ matrix M such that the encoding of a column vector $x \in GF(2)^n$ is the m-bit vector Mx. Show that if $m(1 - H(p)) > n$ then there exists such a linear error correcting code such that $E(x), E(y)$ disagree in at least pm of the bits.

6. A k-coloring of a graph $G = (V, E)$ is an assignment of one of the colors $\{1, 2, \ldots, k\}$ to each vertex, so that every two adjacent vertices get different colors. (Aside: Note that nodes that get the same color must be *independent*, i.e. have no edges amongst themselves.) This question explores an approximation algorithm for coloring using SDPs.

   (a) Show that if the graph is 3-colorable then the following SDP is feasible:

$$\begin{aligned} \langle v_i, v_j \rangle &\leq -\frac{1}{2} \qquad \forall \{i, j\} \in E \\ \langle v_i, v_i \rangle &= 1 \qquad \forall i \end{aligned}$$

   (b) Consider the following rounding algorithm: pick a random unit vector u and threshold $\tau > 0$ and select the set $S = \{i : \langle u, v_i \rangle > \tau\}$. Argue that there is a probability p such that for every i the probability that it lies in S is p.

   (c) Show that for every edge $\{i, j\}$ the probability that both i, j lie in S is at most p^4. (Hint: Reason about the plane spanned by v_i and v_j.)

   (d) Now argue that if the maximum degree is d then there is an efficient algorithm algorithm that, given a 3-colorable graph with a node of degree d finds an independent set of size $\Omega(n/d^{1/3})$. Argue that this can be turned into an algorithm that colors the graphs with $O(d^{1/3} \log n)$ colors. (Full credit also given if you have extra factors of $\log n$ in any of these bounds.)