COMS W4232: Advanced Algorithms Spring 2024

Advanced Algorithms

Prof. Alexandr Andoni *Scribe: Ekene Ezeunala*

Contents

§1 Lecture 01—17th January, 2024

§1.1 Introduction

In this class[1] we will be focusing on more modern and advanced paradigms in algorithm design and analysis. There are two main relaxations that we will make throughout the course that are slightly more relaxed than those in a typical course like COMS W4231 Analysis of Algorithms:

1. We will allow our algorithms to be randomised—that is, our algorithm A is permitted to, say, toss random coins to make decisions so that for all inputs I,

$$\Pr_A[A \text{ is correct on } I] \geq 90\%.$$

2. We will discuss algorithms whose outputs a' only approximate the correct answer a up to a multiplicative factor c—that is, for all inputs I,

$$a \leq a' \leq c \cdot a.$$

   In this case we say that the algorithm is a c-approximation algorithm.

For instance, instead of finding some minimised quantity $a_{\min}$, it may be easier and more efficient to instead output a random approximation b with

$$\Pr[\text{output } b \text{ has } a_{\min} \leq b \leq (1+\varepsilon)a_{\min}] \geq 90\%.$$

For our analyses we will mostly be concerned with the correctness of the algorithm (as demonstrated by rigorous definitions, theorems and proofs) and the algorithm's performance—perhaps in terms of its running time, the space used up by the algorithm, and other situational parameters (for instance in the case of an distributed algorithm spread out across a network for the purpose of jointly solving a problem via a consensus protocol, we may be interested in the number of messages sent between nodes in the network).

§1.2 Overview of course topics

Throughout this course, we will cover the following topics (and possibly more):

1. **Hashing.** We will work on designing hash functions $h : \mathcal{U} \to [n]$ which map items from one universe $\mathcal{U}$, a generic set, to keys in the set $[n] := \{0, 1, \ldots, n-1\}$, where $n \ll |\mathcal{U}|$ is a fixed integer. We will use these hash functions to solve the following problem:

   Problem 1.1 (*Dictionary Problem*)**.** Given a set $S \subseteq \mathcal{U}$, how can we preprocess S so that later, given an arbitrary $x \in \mathcal{U}$, we can cheaply query whether $x \in S$ as well?

   By the regular hashing procedure (which basically entails using a hash function h to map items to keys in $[n]$ and then storing the items in an array A of size n), we can solve this problem in $O(1)$ query time in expectation. We will see that with perfect hashing, we can solve this problem deterministically in $O(1)$ query time—eliminating the randomness is useful because it provides guarantees on results, which may be important in certain sensitive scenarios.

[1] Administrative details can readily be found on the course website. There will be a closed-book 75 minute midterm in class, and a final project due before the reading week.

2. **Sketching and streaming algorithms.** Roughly speaking, we can think of sketching as a kind of functional compression procedure. We will be looking to develop algorithms that take massive objects or data (perhaps a large matrix, a high-dimensional Euclidean vector, or a neural network) and compress them into a useful smaller object (perhaps a smaller matrix, etc) that preserves some useful properties of or relevant information about the original object (which in turn may then be used to compare the two objects for similarity or get a synopsis of the massive data set).
   Streaming is a related notion, but instead concerned with gathering relevant information across a "stream" containing large amounts of data, where the data is only available to us in a sequential manner (i.e. we cannot go back and look at previous data) and we only have a limited amount of memory to store the data. For example, consider a router in a massive network with gigabytes of information passing through—much larger than the memory capacity of the router—and we may want to get information (for example, was there a denial-of-service attack in the past hour? how many IP addresses have we seen? etc.) about the stream that may be useful to us.
   The algorithms we will often consider will answer approximate queries about the data, and by embedding data into high-dimensional spaces we will design algorithms that leverage properties from high-dimensional geometry to solve problems.

3. **Nearest-neighbour search in $\mathbb{R}^d$.** Given a cloud of points in $\mathbb{R}^d$ for d sufficiently large, as well as a new point p, what is the closest point in the cloud to p? This problem is a fundamental one in machine learning and computational geometry, and we will see that it has many applications in practice. Of course it can be solved by simply making a query comparing the distance between p and each point in the data space and then selecting the point with the smallest distance, but the problem is that this query will only be answered in time proportional to the size of the data set, and we'd like to obtain more efficient algorithms for this problem (perhaps sublinear in the size of the data set).

4. **Spectral graph theory.** Graphs are often represented by matrices, for example the adjacency matrix (the matrix whose (i, j)-entry is 1 if there is an edge between vertices i and j and 0 otherwise) and the Laplacian matrix (the matrix whose (i, j)-entry is 1 if $i = j$ and -1 if there is an edge between vertices i and j and 0 otherwise). We will look at the spectral properties of these matrices and their spectral decompositions (for example, the properties of the first and second largest eigenvalues, etc.) and what they mean for the underlying graph. We will see that there is a very nice relationship between the combinatorial properties of the graph (connectivity, expansion, etc.) and the algebraic properties of the graph (the eigenvalues of the Laplacian matrix, etc.).
   Our eventual goal will be to use these spectral properties to design algorithms for graph partitioning and spectral clustering on graphs.

5. **Optimisation.** We will discuss methods for solving optimisation problems of the form $\min_{x\in\mathbb{R}^d} f(x)$ where x is constrained within some set $C \subseteq \mathbb{R}^d$. We will focus on linear programs (where f is linear), semidefinite programs (where f is a linear function of a positive semidefinite matrix), the interior-point and ellipsoid methods for LPs, gradient descent, as well as second-order optimisation methods such as the Newton step method.
   We will also study multiplicative weight updates for learning from expert advice, and how this can be used to solve problems in boosting and online learning.

6. **Large-scale models for computation.** We will discuss algorithms for efficient parallel/cluster

computing (for example we could have a very large problem to be solved by a cluster of 100,000 machines, none of which have access or enough memory to store the entire problem) and how to design algorithms that are robust to failures in the cluster. A related notion is that of the I/O external models that take caching into consideration for accessing items from memory, and how we may design algorithms in this model that are robust to cache misses.

7. **Distributed algorithms.** We will discuss algorithms that are distributed across a network of machines, and how we can design algorithms that are robust to failures in the network. We will also discuss the consensus problem, where we want to design algorithms that allow the machines to agree on a common value, and how this can be used to solve problems such as leader election, mutual exclusion, and so on.
8. More stuff, depending on the schedule; time-permitting, we will think about faster exp-time algorithms for tackling larger instances of problems like 3SAT—for example, we will see how to reduce the running time of the algorithm for 3SAT from at most 2^n to at most 1.6^n (up to a constant factor, of course).

§1.3 The Morris algorithm for approximate counting

The first problem we will consider is a basic one: counting up to an integer n in a stream. Here is its formal statement:

Problem 1.2 (*Approximate Counting*)**.** Given a stream of rapid button presses observed one at a time, find the number of observed button presses using as little memory as possible.

This problem is useful in say, Google statistics, where we may want to count the number of times a particular search term has been searched for. Recall that any algorithm that counts up to n distinct items must use at least $\lceil \lg n \rceil$ bits of memory, since there are $2^{\lceil \lg n \rceil}$ possible states of the memory. Indeed, in the deterministic setting, we will see that we can only get an algorithm that uses $\lceil \lg n \rceil + O(1)$ bits of memory:

Theorem 1.1. *There is a deterministic algorithm that uses $\lceil \lg n \rceil + O(1)$ bits of memory to count the number of distinct items in a stream of length n.*

Proof. We will use a hash table H of size $2^{\lceil \lg n \rceil}$ to store the items we have seen so far. We will maintain a counter c that keeps track of the number of distinct items we have seen so far.
When we see a new item x, we will check if x is in H. If it is, we do nothing. Otherwise, we increment c and add x to H.
At the end of the stream, we output c. This algorithm clearly uses $\lceil \lg n \rceil + O(1)$ bits of memory (since H uses $\lceil \lg n \rceil$ bits of memory and c uses $O(1)$ bits of memory). We will now show that this algorithm is correct. Let S be the set of items in the stream. We will show that $|S| = c$ at the end of the stream. We will prove this by induction on the number of items in the stream. For the base case, note that if the stream is empty, then $S = \emptyset$ and $c = 0$, so $|S| = c$. Now suppose that the stream has length n and $|S| = c$. Let x be the next item in the stream. If $x \in S$, then $|S| = c$ and $x \in S$, so $|S| = c$ at the end of the stream. If $x \notin S$, then $|S| = c + 1$ and $x \notin S$, so $|S| = c + 1$ at the end of the stream. $\square$

We will discuss the MORRIS algorithm [Mor78], a randomised algorithm which obtains an exponential improvement over the deterministic algorithm above—indeed it uses $O(\lg \lg n)$ bits of memory, which works for $c = O(1)$ factor approximations. This exponential improvement is quite helpful in practice, since it allows us to use much less memory to get a good approximation of the number of distinct items in the stream (which matters more for billions of counts, for example). The constant factor approximation also makes sense here: we don't necessarily care too much about exact values but instead we care about the orders of magnitude for our counting problems.

Before we present the algorithm, why are these notions about its results feasible? Consider the algorithm—which is simpler than counting up to n—of storing numbers up to n up to a multiplicative factor of 2. (Such an algorithm is a 2-approximation algorithm.) We could write down its bit representation and find that the position of its most significant digit is b. Then to store n up to a 2-approximation, we can just note that $n \in [2^b, 2^{b+1})$, and so we only need to store b. But the value $b = O(\lg n)$, and so to store n we need $O(\lg b) = O(\lg \lg n)$ bits of memory. This is the intuition behind the algorithm.

We now present the MORRIS algorithm:

Algorithm: MORRIS algorithm for *Approximate Counting*

- Keep a counter X; initially $X \leftarrow 0$[a].
- For each input button press:
-     With probability $1/2^X$, do $X \leftarrow X + 1$.
- Return the estimator $\hat{n} = 2^X - 1$.

[a]This essentially plays the role of b above.

We will analyse this algorithm using some notions from applied probability in the next lecture. Before then, we present an extremely brief crash course in applied probability (only the areas that are useful for us in this course).

§1.4 A crash course in applied probability

For our probabilistic analyses, our main object will be a random variable X.

Definition 1.3 (Expectation)**.** *For a random variable X, the expected value of X is defined as*

$$\mathbb{E}[X] = \begin{cases} \sum_{x \in \text{Range}(X)} x \Pr[X = x] & \text{if } X \text{ is discrete,} \\ \int_{x \in \text{Range}(X)} x f(x)\, \mathrm{d}x & \text{if } X \text{ is continuous,} \end{cases}$$

where f is the probability density function of X.

A nice probability of expectations is that they are linear: for all $a, b \in \mathbb{R}$, $\mathbb{E}[aX_1 + bX_2] = a\mathbb{E}[X_1] + b\mathbb{E}[X_2]$.

Definition 1.4 (Variance)**.** *For a random variable X, the variance of X is a measure of how much X deviates from its mean, and is defined as*

$$\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

We will usually refer to the expected "property" of a randomised algorithm as a measure of its performance—we want an estimator that is close to n with good probability. For these we will need to extract performance guarantees via concentration bounds:

Definition 1.5 (Concentration bounds)**.** *For a random variable X, a concentration bound is a bound on the probability that X deviates from its mean by a certain amount. For example, we may want to bound the probability that X deviates from its mean by more than ε, i.e.* $\Pr[|X - \mathbb{E}[X]| \geq \varepsilon]$.

Here are the concentration bounds we will use in this course:

Fact 1.6 (Markov's inequality)**.** *For a non-negative random variable X and $\varepsilon > 0$, we have*

$$\Pr[X \geq \varepsilon] \leq \frac{\mathbb{E}[X]}{\varepsilon}.$$

Fact 1.7 (Chebyshev's inequality)**.** *For a random variable X and $\varepsilon > 0$, we have*

$$\Pr[|X - \mathbb{E}[X]| \geq \varepsilon] \leq \frac{\mathrm{Var}[X]}{\varepsilon^2}.$$

Fact 1.8 (Chernoff bounds)**.** *For a random variable X and $\varepsilon > 0$, we have*

$$\Pr[X \geq (1+\varepsilon)\mathbb{E}[X]] \leq e^{-\varepsilon^2 \mathbb{E}[X]/3}.$$

Fact 1.9 (Hoeffding's inequality)**.** *For a random variable X and $\varepsilon > 0$, we have*

$$\Pr[|X - \mathbb{E}[X]| \geq \varepsilon] \leq 2e^{-2\varepsilon^2 n}.$$

Corollary 1.10. *For a random variable X, with probability $1 - \delta$, it holds that*

$$X = \mathbb{E}[X] \pm \sqrt{\frac{\mathrm{Var}[X]}{\delta}}.$$

Proof. The proof is straightforward given Chebyshev's inequality:

$$\Pr\left[|X - \mathbb{E}[X]| > \sqrt{\frac{\mathrm{Var}[X]}{\delta}}\right] \leq \frac{\mathrm{Var}[X]}{\left(\sqrt{\mathrm{Var}[X]/\delta}\right)^2} = \delta,$$

and therefore the probability that X is not within $\sqrt{\mathrm{Var}[X]/\delta}$ of $\mathbb{E}[X]$ is at most δ. □

§2 Lecture 02—22nd January, 2024

§2.1 Analysis of the Morris algorithm for approximate counting

Remember the *Approximate Counting* problem from last time:

Problem 2.1 (*Approximate Counting*)**.** Given a stream of rapid button presses observed one at a time, find the number of observed button presses using as little memory as possible.

Last time we saw the MORRIS algorithm for approximate counting:

```
Algorithm: The MORRIS algorithm for Approximate Counting

• Keep a counter $X$; initially $X \leftarrow 0$.
• For each input button press:
•     With probability $1/2^X$, do $X \leftarrow X + 1$.
• Return the estimator $\hat{n} = 2^X - 1$.
```

We will now analyse the MORRIS algorithm using some notions from applied probability. First we check that the algorithm is "correct":

Claim 2.2. *The estimator for* MORRIS *algorithm, $\hat{n} = 2^X - 1$, is an unbiased estimator for n. That is, $\mathbb{E}[\hat{n}] = n$.*

Proof. Let X_n be the value of X after n presses, so that we have that $\mathbb{E}[\hat{n}] = \mathbb{E}\left[2^{X_n} - 1\right]$. We shall prove via induction that $\mathbb{E}[\hat{n}] = n$. For the base case, we have that after no presses,

$$\mathbb{E}[\hat{n}] = \mathbb{E}\left[2^{X_0} - 1\right] = \mathbb{E}\left[2^0 - 1\right] = 0.$$

Now assume that $\mathbb{E}\left[2^{n-1} - 1\right] = n - 1$. Then for the inductive step,

$$\begin{aligned}
\mathbb{E}[\hat{n}] &= \mathbb{E}_{X_1,\dots,X_n}\left[2^{X_n} - 1\right] \\
&= \mathbb{E}_{X_1,\dots,X_{n-1}}\left[\mathbb{E}_{X_n}\left[2^{X_n} - 1\right]\right] \\
&= \mathbb{E}_{X_1,\dots,X_{n-1}}\left[\underbrace{2^{-X_{n-1}} \cdot \left(2^{X_{n-1}+1} - 1\right)}_{\substack{\text{random event that} \\ X \text{ is incremented}}} + \underbrace{\left(1 - 2^{-X_{n-1}}\right) \cdot \left(2^{X_{n-1}} - 1\right)}_{\substack{\text{random event that} \\ X \text{ is not incremented}}}\right] \\
&= \mathbb{E}_{X_1,\dots,X_{n-1}}\left[2 - 2^{-X_{n-1}} + 2^{X_{n-1}} - 1 - 1 + 2^{-X_{n-1}}\right] \\
&= \mathbb{E}_{X_1,\dots,X_{n-1}}\left[2^{X_{n-1}} - 1\right] + 1 \\
&= n - 1 + 1 = n.
\end{aligned}$$

□

We will now show a bound on the amount of memory used by the MORRIS algorithm, in bits:

Claim 2.3. *The space used by the* MORRIS *algorithm is $O(\lg \lg n)$ bits.*

Proof. We already know that we need $O(\lg X)$ bits to represent X on a suitable register. Therefore it suffices to appropriately bound the value of X in terms of n.
By Markov's inequality and the fact that $\mathbb{E}[\hat{n}] = n$, we have that

$$\Pr\left[\hat{n} > 10n\right] \le \frac{\mathbb{E}[\hat{n}]}{10n} = \frac{1}{10},$$

and therefore $\hat{n} \le 10n$ with probability at least $9/10$. Therefore since $X = \lg \hat{n} + 1 = O(\lg n)$ it follows that $O(\lg \lg n)$ bits of memory are used. □

Claims 2.2 and 2.3 together give us some useful behaviour about the performance of our algorithm in expectation. But Claim 2.2 falls a bit short of the goal—showing that $\hat{n} \approx n$ with good enough probability—in the sense that it could be incorrect most of the time, as the average over $\hat{n}$ is not sharp enough. The Chebyshev inequality is helpful here, in that it gives us bounds on how much the estimator $\hat{n}$ deviates from its mean $\mathbb{E}[\hat{n}] = n$:

$$\Pr\left[|\hat{n} - n| \geq \lambda\right] \leq \frac{\mathrm{Var}[\hat{n}]}{\lambda^2}.$$

Claim 2.4. *The variance of the estimator in the* Morris *algorithm is*

$$\mathrm{Var}[\hat{n}] = \mathbb{E}[\hat{n}^2] - \mathbb{E}[\hat{n}]^2 \leq \frac{3n(n+1)}{2} + 1 = O(n^2).$$

Proof. By definition,

$$\begin{aligned}
\mathrm{Var}[\hat{n}] &= \mathbb{E}[\hat{n}^2] - \mathbb{E}[\hat{n}]^2 = \mathbb{E}\left[\left(2^{X_n} - 1\right)^2\right] - n^2 \\
&= \mathbb{E}\left[2^{2X_n} - 2^{X_n+1} + 1\right] - n^2 = \mathbb{E}\left[2^{2X_n}\right] - 2 \cdot \mathbb{E}\left[2^{X_n}\right] + 1 - n^2 \\
&= \mathbb{E}\left[2^{2X_n}\right] - 2 \cdot \mathbb{E}\left[2^{X_n} - 1\right] - 1 - n^2 = \mathbb{E}\left[2^{2X_n}\right] - 2n - 1 - n^2 \\
&= \mathbb{E}\left[2^{2X_n}\right] - (n+1)^2 \\
&\leq \mathbb{E}\left[2^{2X_n}\right].
\end{aligned}$$

Then via the same inductive procedure as before,

$$\begin{aligned}
\mathbb{E}_{X_1,\ldots,X_n}\left[2^{2X_n}\right] &= \mathbb{E}_{X_1,\ldots,X_{n-1}}\left[\mathbb{E}_{X_n}\left[2^{2X_n}\right]\right] \\
&= \mathbb{E}_{X_1,\ldots,X_{n-1}}\left[2^{-X_{n-1}} \cdot 2^{2(X_{n-1}+1)} + \left(1 - 2^{-X_{n-1}}\right) \cdot 2^{2X_{n-1}}\right] \\
&= \mathbb{E}_{X_1,\ldots,X_{n-1}}\left[4 \cdot 2^{X_{n-1}} + 2^{2X_{n-1}} - 2^{X_{n-1}}\right] \\
&= \mathbb{E}_{X_1,\ldots,X_{n-1}}\left[3 \cdot 2^{X_{n-1}}\right] + \mathbb{E}_{X_1,\ldots,X_{n-1}}\left[2^{2X_{n-1}}\right] \\
&= 3 \cdot \underbrace{\mathbb{E}_{X_1,\ldots,X_{n-1}}\left[2^{X_{n-1}}\right]}_{= n \text{ via Claim 2.2}} + \underbrace{\mathbb{E}_{X_1,\ldots,X_{n-1}}\left[2^{2X_{n-1}}\right]}_{\leq \frac{3}{2}n(n-1)+1 \text{ by the above}} \\
&\leq 3n + \frac{3}{2}n(n-1) + 1 \\
&= \frac{3n(n+1)}{2} + 1 = O(n^2).
\end{aligned}$$

□

With the above facts in mind, we may then simply apply Chebyshev's inequality and find that it is enough to get

$$\frac{\mathrm{Var}[\hat{n}]}{\lambda^2} \leq \frac{1}{10} \iff \frac{2n^2}{\lambda^2} \leq \frac{1}{10} \implies \lambda \geq \sqrt{20n^2}.$$

Then it is enough to set $\lambda = 5n$, so that by Chebyshev, with probability at least $9/10$, we have that $|\hat{n} - n| \leq 5n$, which is the same as saying that $\hat{n} \in [n - 5n, n + 5n] = [-4n, 6n]$ with probability at least $9/10$. But this is not as good as we would like—for instamce, note that an estimator that always remains unchanged at $X = 0$ for all the updates to the stream would also satisfy this guarantee.

So we have established that the estimator we get here, despite being unbiased, is not a good estimator in the sense that it is too naïve. But wehat if instead of $|\hat{n} - n| \leq 5n$, we had that $|\hat{n} - n| \leq \varepsilon n$ for a sufficiently small $\varepsilon > 0$ that we can choose ourselves? Can we expand our range of possible beliefs to include some $\hat{n} \in [n - \varepsilon n, n + \varepsilon n]$ for some $\varepsilon > 0$? Yes we can, via the Morris+ algorithm.

§2.2 Improving the Morris algorithm: the Morris+ algorithm

The goal of this algorithm will be to output some estimator $\hat{n} \in [(1-\varepsilon)n, (1+\varepsilon)n]$ with high probability, where $\varepsilon > 0$ is a small constant, say $\varepsilon = 0.1$. We will accomplish this by repeating the Morris algorithm k times and outputting the average of the k estimators:

Algorithm: Morris+ for *Approximate Counting*

- Run k independent copies of the Morris algorithm in parallel, each with X_i as the counter. Store the values $(X_1, \dots, X_k)$.
- At the end of all the instances of the Morris algorithm, return the combined estimator $\hat{n} = \frac{1}{k}\sum_{i=1}^{k} 2^{X_i} - 1$.

Fortunately many of the properties of this algorithm that we will analyse are straightforward to prove given the fact that we are running k independent copies of the Morris algorithm in parallel and we have already established the properties of the Morris algorithm.

Claim 2.5. *The estimator $\hat{n}$ for Morris+ is an unbiased estimator for n. That is, $\mathbb{E}[\hat{n}] = n$.*

Proof. The proof follows right away from the linearity of expectation:

$$\mathbb{E}[\hat{n}] = \mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{k} 2^{X_i} - 1\right] = \frac{1}{k}\sum_{i=1}^{k} \mathbb{E}\left[2^{X_i} - 1\right] = \frac{1}{k}\sum_{i=1}^{k} n = n. \qquad \square$$

Claim 2.6. *The variance of the estimator $\hat{n}$ for Morris+ is*

$$\mathrm{Var}[\hat{n}] = \mathbb{E}[\hat{n}^2] - \mathbb{E}[\hat{n}]^2 \le \frac{3n(n+1)}{2k} + 1 = O\left(\frac{n^2}{k}\right).$$

Proof. We already know that the variance of a single estimator in the original Morris algorithm is $\mathrm{Var}[\hat{n}] = \frac{3n(n+1)}{2} + 1$. Then since $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$ for any random variable X and constant a,

$$\begin{aligned}\mathrm{Var}[\hat{n}] &= \mathrm{Var}\left[\frac{1}{k}\sum_{i=1}^{k} 2^{X_i} - 1\right] = \frac{1}{k^2}\sum_{i=1}^{k} \mathrm{Var}\left[2^{X_i} - 1\right] \\ &\le \frac{1}{k^2}\sum_{i=1}^{k} \mathrm{Var}\left[\hat{n}\right] = \frac{1}{k^2} \cdot k \cdot \left(\frac{3n(n+1)}{2} + 1\right) \\ &= \frac{3n(n+1)}{2k} + \frac{1}{k} = O\left(\frac{n^2}{k}\right). \qquad \square\end{aligned}$$

Claim 2.7. *With probability $\ge 90\%$, the space used by Morris+ is $O(k \lg\lg n)$ bits.*

Proof. We already know that the space used by a single instance of the Morris algorithm is $O(\lg\lg n)$ bits. Then since we are running k independent copies of Morris algorithm in parallel, the space used by Morris+ is $O(k \lg\lg n)$ bits, as no counter is larger and occupies more space than the largest counter in any of the k instances of Morris algorithm. $\square$

Given these facts, outlining the rest of the analysis is straightforward. We will use the consequence of Chebyshev's inequality (Corollary 1.10) to get that with sufficiently high constant probability,

$$2^{X_n} - 1 = \mathbb{E}\left[2^{X_n} - 1\right] \pm \frac{1}{\sqrt{k}} \cdot O(n) \iff 2^{X_n} - 1 = n \pm \frac{1}{\sqrt{k}} \cdot O(n).$$

Now setting $k = O(1/\varepsilon^2)$, we get that with probability at least $9/10$, we have that

$$\hat{n} = \frac{1}{k}\sum_{i=1}^{k} 2^{X_i} - 1 = n \pm \frac{1}{\sqrt{k}} \cdot O(n) = n \pm \varepsilon n,$$

which is exactly what we wanted. Therefore we have shown that Morris+ is a $1 \pm \varepsilon$ approximation algorithm for *Approximate Counting*.

Morris++. What if we wanted to achieve the $(1+\varepsilon)$-approximation for *Approximate Counting* with probability $\geq 1 - \delta$? We can do this by running Morris+ with k set to $O\left(\delta^{-1}/\varepsilon^2\right)$. Then a straightforward application of Chebyshev's inequality gives us that $2^{X_n} - 1 = n \pm \varepsilon n$, except with failure probability at most

$$\frac{\operatorname{Var}\left[2^{X_n} - 1\right]}{\varepsilon^2 n^2} \leq O\left(\frac{n^2\varepsilon^2\delta}{\varepsilon^2 n^2}\right) = O(\delta),$$

and therefore if we choose the multiplicative constants appropriately, we can guarantee that a $(1+\varepsilon)$-approximation is achieved with probability $\geq 1 - \delta$. This discussion suggests an $O(\varepsilon^{-2}\delta^{-1} \lg\lg n)$-space bound for Morris++ with high probability, although we will not prove this here.

Indeed, on the first problem set we will design and analyse an algorithm that uses around at most $O(\log\log n + \log 1/\varepsilon)$ bits of memory to achieve a $(1+\varepsilon)$-approximation for *Approximate Counting* with probability $\geq 9/10$.

§2.3 Introducing hashing

Consider the following *Dictionary* problem:

Problem 2.8 (*Dictionary Problem*)**.** Given a set $S \subseteq \mathcal{U}$, how can we preprocess S into a data structure so that later, given an arbitrary $x \in \mathcal{U}$, we can cheaply query whether $x \in S$ as well?

Here are a few solutions to this problem:

1. **Linear search.** For each query, scan over S. The query time is $\Theta(n)$ (since we may have to scan over the entire set S), and the space used is $O(n)$ (since we need to at most the entire set S).
2. **Binary search via a BST.** We can store S in a binary search tree, and then query whether $x \in S$ by searching for x in the BST. The query time is $O(\lg n)$ (since we need to search for x in the BST), and the space used is $O(n)$ (since we need to store the BST).
3. **Full indexing.** We can store S in a lookup table T of size $|\mathcal{U}|$, where $T[x] = 1$ if $x \in S$ and $T[x] = 0$ otherwise. Then we can query whether $x \in S$ by simply checking $T[x]$. The query time is $O(1)$ (since we can just check $T[x]$), and the space used is $O(|\mathcal{U}|)$ (since we need to store the entire lookup table T).

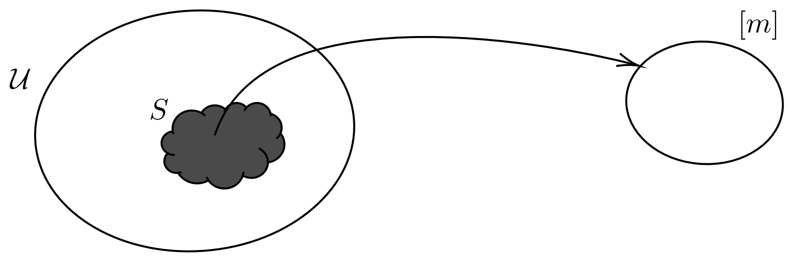
The question for us will be the following: can we combine the best features of all these algorithms to obtain an algorithm with query time $O(1)$ and space $O(n)$? The answer is yes, via hashing. We will see this in the next lecture.

§3 Lecture 03—24th January, 2024

§3.1 Hashing, universal hashing

Recall the *Dictionary* problem from last time: we are given a universe of items, say IP addresses under the IPv4 protocol, and we want to store a set of IP addresses $S \subseteq \mathcal{U}$ in a data structure so that we can quickly query whether an IP address $x \in \mathcal{U}$ is in S or not. We discussed standard solutions for this problem: the first two had query time more than a constant, and the third, full indexing, had $O(1)$ query time. Recall that the full indexing routine uses a table of size $|\mathcal{U}$ to prepare for all the possible queries, and although it has $O(1)$ query time, it uses $O(|\mathcal{U}|)$ space, which we may not be able to afford.

The notion of hashing will allow us to get the best of both worlds: $O(1)$ query time and $O(n)$ space, eventually. The reasoning is as follows: if we have a really large but sparse table, we shouldn't try to store the entire table but instead compress it somewhat. Hashing may be viewed as a kind of compression in this setting; more generally, we will look to take the domain of size $|\mathcal{U}|$ and map it to a smaller range of size m, so that it is still possible to store the range in memory. We will then store the items in the set S in the range, and then query whether an item $x \in \mathcal{U}$ is in S by checking whether x is in the range. This is the idea behind hashing.



This "compression" or *hashing* will be accomplished by means of a hash function $h : \mathcal{U} \to [m]$, where $m \ll |\mathcal{U}|$. One property we would like to have is that distinct elements of the universe $\mathcal{U}$ do not "clash", i.e. there are no/few collisions.

Definition 3.1 (Collision)**.** *Two elements $x, y \in \mathcal{U}$ with $x \neq y$ are said to collide under a hash function h if $h(x) = h(y)$.*

Here's a first idea for appropriately hashing the elements of $\mathcal{U}$. Suppose there are no collisions among elements of S. Then the dictionary problem can be tackled by the usual full-indexing routine: build a lookup table $T(1, \ldots, m)$, and for every $x \in S$, set $T(h(x)) = x$. How would queries work in this system? For any given x, look up $T(h(x))$ and check whether it is equal to x. If it is, then $x \in S$ and output `YES`, and if not, then $x \notin S$ and output `NO`.
Obviously the query time is $O(1)$—looking up T, computing $h(x)$, and checking whether $T(h(x)) = x$

all take constant time—and the space used is $O(m)$—since we need to store the entire lookup table T of size at most m.

Now this is a nice, idealistic situation. Is it reasonably achievable? Yes! Given $S = \{s_1, s_2, \ldots, s_n\}$, we can always come up with a suitable hash function $h(x)$ with no collisions by defining

$$\begin{aligned} h(s_1) &= 1 \\ h(s_2) &= 2 \\ &\vdots \\ h(s_n) &= n \\ h\left(s_i \notin S\right) &= n+1, \end{aligned}$$

and indeed this can be accomplished in $m = n + 1$ bits of space. The moral lesson here is that it is not fatal for a hash function to depend on the ground set S itself, as long as we can store the hash function in a reasonable amount of space. The problem here however is that computing h as defined above for an input x becomes expensive very quickly—we must store the hash function, and given an input x, we must verify which element of S it is equal to, which takes us back to the original *Dictionary* problem (with the problem hidden inside the hash function). So we need to refine our goal somewhat: we want to find a hash function h that is *efficiently computable* and has *few collisions.*

Here's a different idea then. Pick an exact deterministic hash function h independently of S, perhaps adopting the suggestion by Don Knuth:

$$h(x) = \left\lfloor \left\{ \frac{\sqrt{5}-1}{2} \cdot x \right\} \cdot m \right\rfloor,$$

where $\{\cdot\}$ denotes the fractional part of the argument and $\lfloor \cdot \rfloor$ denotes the usual floor function and the map $h : \mathcal{U} \to [m-1]$. The issue, of course, is that knowing that we map into a much smaller domain, we are guaranteed to gave many collisions. Thus we can always choose a sufficiently adversarial set S so that everything collides with everything else. For instance, if $S = \{1, 3, 5, 7, 9\}$, and we choose the hash function $h(x) := x \bmod 2$, then $h(1) = h(3) = h(5) = h(7) = h(9) = 1$, and therefore we have a collision for every element of S. This is a general problem with deterministic hash functions: if we know the hash function beforehand, then we can always choose a set S that entirely collides with itself under h.

The takeaway here is that h should be chosen *randomly* from a suitable family of hash functions. So we will define a universe of hash functions $\mathcal{H}$ as follows:

$$\mathcal{H} := \{\text{all hash functions } h : \mathcal{U} \to [m]\},$$

and subsequently choose a hash function $h \in \mathcal{H}$ uniformly at random from $\mathcal{H}$. There could be some collisions still, depending on the set S and the hash function h chosen, but striving to minimise the number of these collisions leads us to the notion of hashing-with-chaining and the Basic-Hash algorithm.

Algorithm: Basic-Hash

- Keep a lookup table $T(1, \ldots, m)$.
- For each $i \in [m]$, keep a linked list of the $y \in S$ such that $h(y) = i$.

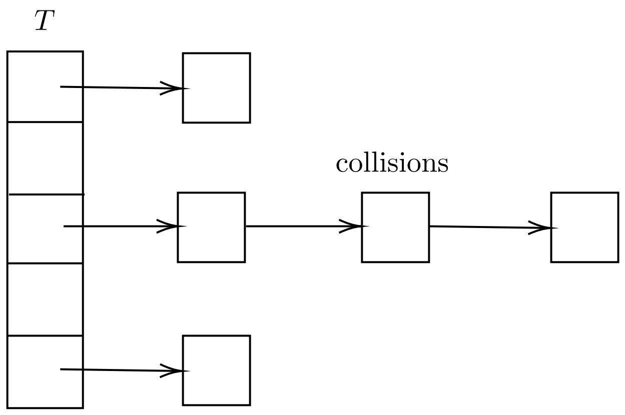


Figure 1. This is hashing-with-chaining: the buckets in the linked list contain the elements of S that collide under h.

Claim 3.2. *The space of the* Basic-Hash *algorithm is $O(m) + O(n)$ bits.*

Proof. The space used by the lookup table T is $O(m)$ bits, and the space used by the linked lists is $O(n)$ bits, since each linked list contains at most n elements. Therefore the total space used is $O(m) + O(n)$ bits. $\square$

Claim 3.3. *The expected query time of the* Basic-Hash *algorithm is $\tau[h] + O(1) + O(C_x) = \tau[h] + O(1)$ for $m = \Theta(n)$, where $\tau[h]$ is the time taken to evaluate h, and the collision count C_x is a random variable that is the number of $y \in S$ with $y \neq x$ such that $h(y) = h(x)$.*

Proof. Obviously it takes $\tau[h]$ time to evaluate h by definition and $O(1)$ time to go through the linked list. Why the dependence on C_x? Well, perhaps our element is very unlucky and collides with every other element in the set S. Then we would need to go through all the elements in one bucket of the linked list, which would be at most as long as all the collisions of x under h, that is, $O(C_x)$:

$$\begin{aligned}
\mathbb{E}_h\,[\text{size of the bucket in the linked list}] &= \mathbb{E}_h\,[C_x] \\
&= \mathbb{E}\left[\sum_{y \in S \setminus \{x\}} \mathbb{1}\{h(x) = h(y)\}\right] \\
&= \sum_{y \in S \setminus \{x\}} \underbrace{\mathbb{E}\,[\mathbb{1}\{h(x) = h(y)\}]}_{\substack{1/m \text{ as this happens} \\ \text{once for fixed } h(y)}} \\
&= \sum_{y \in S \setminus \{x\}} \frac{1}{m} = \frac{n-1}{m} \leq \frac{n}{m}.
\end{aligned}$$

The algorithm sets $m = \Theta(n)$ because of the structure of the linked list, and so the expected query time is $\tau[h] + O(1) + O(C_x) = \tau[h] + O(1)$. $\square$

Are we okay with this? The maximum possible query time (in the average-case sense) over the choice of $x \in \mathcal{U}$ is $\leq \max_x C_x + 1 \leq$ bucket size, since for every x we examine the linked list, and the longest time we spend in the linked list is the largest possible bucket available.

Claim 3.4. *With high probability,*

$$\mathbb{E}_h[\text{maximum bucket size}] = \Theta\left(\frac{\log n}{\log\log n}\right),$$

when $m = n$.

Proof. We first show that $\mathbb{E}_h[\text{maximum bucket size}] = O\left(\frac{\log n}{\log\log n}\right)$. Our high-level approach here will be to show that for every $k > 3\log n/\log\log n$,

$$\Pr[\text{bucket 1 has size exactly } k] = o(1/n^2),$$

and hence we can union bound over the n buckets, and the $< n$ values of $k \in \{3\log n/\log\log n, 1 + 3\log n/\log\log n, \ldots, n\}$. Now,

$$\Pr[\text{bucket 1 has size exactly } k] \le \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k} \le \frac{n^k}{k!}\cdot\left(\frac{1}{n}\right)^k.$$

Using the fact that $k! \ge (k/e)^k$ for $k \ge 1$, we get that if $k = c\cdot\log n/\log\log n$ for some $c > e = 2.71828\ldots$, then

$$\frac{1}{k!} \le \left(\frac{\log n}{\log\log n}\right)^{-c\cdot\log n/\log\log n} \le e^{-c(\log\log n - \log\log\log n)\log n/\log\log n} = e^{-c\log n + c\log n\cdot\frac{\log\log\log n}{\log\log n}}.$$

As the second term in the exponent is $o(\log n)$, this entire expression is at least $n^{-c+o(1)}$. As the size of the bucket can be at most n, by doing a union bound over all $K \ge c\log n/\log\log n$, we get

$$\Pr\left[\text{bucket 1 has load} > k\right] \le n^{-c+1+o(1)}.$$

Taking any $c > 3$ yields that even after a union bound over the n buckets, and $< n$ values of k between $c\log n/\log\log n$ and n, the probability that the maximum bucket size is at least $c\log n/\log\log n$ is $o(1)$, and therefore the expected maximum bucket size is $O\left(\frac{\log n}{\log\log n}\right)$.

Now we show that $\mathbb{E}_h[\text{maximum bucket size}] = \Omega\left(\frac{\log n}{\log\log n}\right)$. Consider drawing $k \sim \mathsf{Poi}(n/2)$ and then tossing k query items. Then the size of each bucket is independently drawn from $\mathsf{Poi}(1/2)$. Hence, the probability that at least one of the n buckets has size at least b is at least the probability that at least one has load exactly b (since the probability that the load is at least b is at most the probability that the load is exactly b), and this is

$$1 - \left(1 - \frac{e^{-1/2}(1/2)^b}{b!}\right)^n \ge 1 - e^{-n\cdot\frac{e^{-1/2}(1/2)^b}{b!}}.$$

By Stirling's approximation, $\log b! = b\log b - b + O(\log b)$, and therefore, for $b = c\log n/\log\log n$ this term is $c\log n + o(\log n)$. Therefore, for $c < 1$, the dominant term in the exponent of the probability we are analysing is $-n/n^c \to -\infty$, and for any constant $c < 1$, with probability $1 - o(1)$, there will be a bucket with size b for $b = c\log n/\log\log n$ in this Poissonised setting.
We now relate this to the setting where exactly n items are hashed by the random hash function h into the buckets. As the number of query items in each bucket increases monotonically with the

number of query items, we only need to argue that $k \leq n$ with high probability. This is true because by Chebyshev's inequality,

$$\Pr[k \geq n] = \Pr\left[|k - \mathbb{E}[k]| \geq \frac{n}{2}\right] \leq \frac{\mathrm{Var}[k]}{(n/2)^2} = \frac{n/2}{n^2/4} = o(1),$$

and therefore with probability $1 - o(1)$, $k \leq n$, and hence the expected maximum bucket size is $\Omega\left(\frac{\log n}{\log\log n}\right)$.

Combining the two bounds, we get that the expected maximum bucket size is $\Theta\left(\frac{\log n}{\log\log n}\right)$. $\square$

Now this result (Claim 3.4) could be a good or a bad thing, depending on the application. If the queries are fine on average, then we are okay, but if we need to guarantee that our hashing procedure is robust to adaptive, potentially adversarial queries, we could spend an extremely long time answering each query in the worst case. So there is a conditional issue here—we will address it later. First we address the issue of storing the hash function h taken randomly from $\mathcal{H}$; this is a problem because $\mathcal{H}$ is very large. In fact, $|\mathcal{H}| = m^{|\mathcal{U}|}$, and to store it we need $O(|\mathcal{U}| \log m)$ bits, which is simply too large.

To repair the exponential-size problem, we can use some hash function $h \in_r \mathcal{H}'$, where $|\mathcal{H}' \ll m^{|\mathcal{U}|}$, with the property that the functions in $\mathcal{H}'$ has more "limited" randomness but each individual function is still "random enough". Why is this possible? Remember that in the proof of Claim 3.3, we only use the randomness of the hash function once to deduce that $\mathbb{E}_h\left[\mathbb{1}\{h(x) = h(y)\}\right] = 1/m$ for $x \neq y$. So that analysis works exactly the same if our hash function purely satisfies this property, and nothing else. This is the idea behind universal hashing.

Definition 3.5 (Universal hash functions). *A family of hash functions $\mathcal{H}$ is universal if for every $x, y \in \mathcal{U}$ with $x \neq y$,*

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = 1/m.$$

Essentially, the definition above defines our problem away! It immeditely follows then that BASIC-HASH works just as well with h chosen uniformly at random from a universal family of hash functions as it does with h chosen uniformly at random from the entire universe of hash functions. But is there a family of universal hash functions which can be efficiently stored (compared to random hash functions)? Yes!

Fact 3.6. *There exists a universal hash family $\mathcal{H}$ such that the space to store $h \in \mathcal{H}$ is at most $\log|\mathcal{H}| = O(|\mathcal{U}|)$ bits, and h is efficiently evaluatable on query inputs in $O(1)$ time.*

Instead of proving this fact by construction, we will exhibit an extension to the definition of universal hash functions and present a definition of a family of hash functions that is (essentially) universal, simple, and practical.

Definition 3.7 (α-universal hash functions). *A family $\mathcal{H}$ of hash functions is α-universal for $\alpha \geq 1$ if and only if for every $x, y \in \mathcal{U}$ with $x \neq y$,*

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \alpha/m$$

It is easy to show (by exactly the same argument we used in the proof of Claim 3.3) that if h is chosen at random from some α-universal family of hash functions, then the expected number of collisions is $O(\alpha n/m)$.

As an example, Dietzfelbinger et al. [DHKP97] gave the following example of an α-universal family of hash functions:

$$\mathcal{H}_D = \{h_a : a \in \mathcal{U} \text{ chosen uniformly at random is odd}\},$$
$$h_a(x) = \left\lfloor ((a \cdot x) \mod |\mathcal{U}|) \cdot \frac{m}{|\mathcal{U}|} \right\rfloor .$$

Fact 3.8. *$\mathcal{H}_D$ is 2-universal.*

Proof. See Dietzfelbinger, Hagerup, Katajainen, and Pentonnen [DHKP97]. □

Theorem 3.1. *The Dictionary problem can be solved in $O(n)$ space and $O(1)$ expected query time using hashing with chaining with a hash function h chosen uniformly at random from an α-universal family of hash functions $\mathcal{H}$.*

Proof. Set $m = n$, so that the table takes space $O(m) + O(n) = O(n)$ bits. Then the expected size of the bucket is $\leq 1 + n/m = 2$. The expected query time, excluding the evaluation of h, is $O(1) + O(1) = O(1)$. □

§3.2 Perfect hashing

In *perfect hashing*, our goal changes: here we want to guarantee a deterministic query time of $O(1)$—the randomness relaxation is removed entirely. So in some sense we are back to the problem of designing a good h such that there are no collisions inside S.

Define the quantity $C := \sum_{x \in S} C_x$. If there are no collisions in S—which we want—then $C = 0$, which implies that the size of each bucket is at most 1. Can we do this with a random hash function? If we increase m, then perhaps we can:

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{x \in S} C_x\right] = \sum_{x \in S} \mathbb{E}[C_x] \leq n \cdot \frac{n}{m} = \frac{n^2}{m}.$$

Therefore we can indeed achieve $C = 0$ in expectation if $m \ll n^2$. Indeed,

$$m = 4n^2 \implies \mathbb{E}[C] \leq \frac{n^2}{4n^2} = \frac{1}{4}.$$

Then by Markov's inequality, $\Pr[C \geq 1] \leq \mathbb{E}[C] \leq 1/4$, and therefore with probability at least $3/4$, $C < 1$, which must mean that $C = 0$ as C is a nonnegative integer. So we can achieve $C = 0$ in expectation, and with high probability, if $m = \Theta(n^2)$, which is good. Indeed, it motivates the following corollary:

Corollary 3.9. *There is an algorithm that solves the Dictionary problem in $O(n^2)$ space and $O(1)$ deterministic query time.*

Proof. We claim that the following algorithm is sufficient:

Algorithm: QUADRATIC-PERFECT-HASH

- Set $m = 4n^2$.
- Build a hash table using random $h \in \mathcal{H}$.
- Compute the number of collisions C by iterating through and determining the size of each bucket.
- If $C = 0$, then output the hash table, and if not, then repeat the algorithm.

We now analyse the QUADRATIC-PERFECT-HASH algorithm. Let T be the number of tries it takes to build the hash table (i.e. the number of restarts of the algorithm plus the initial run). This is a random variable as it depends on the random choice of $h \in \mathcal{H}$. Thus,

$$\begin{aligned}
\mathbb{E}[T] &= \sum_{i \geq 1} i \cdot \Pr[T = i] \\
&= \sum_{i \geq 1} \Pr[T \geq i] \\
&\leq \sum_{i \geq 1} \left(\frac{1}{4}\right)^{i-1} && \text{since the failure probability} \leq 1/4 \\
&= \frac{1}{1 - 1/4} \\
&= \frac{4}{3}.
\end{aligned}$$

So in expectation, the number of tries is 4/3, and so the preprocessing time in expectation is

$$\frac{4}{3} \cdot (O(n) + O(m)) = \frac{4}{3} \cdot \big(O(n) + O(4n^2)\big) = O(n^2),$$

and the total space used by the algorithm is $O(n^2)$ bits. The query time is $O(1)$, since we can just look up the element in the hash table. □

Note that the algorithm and its analysis above works even when h is universal! We did not use any further properties of the hash function, except that each time we build a hash table, we use a fresh $h \in \mathcal{H}$ independent of the previous choices. So in some sense, the construction is randomised.
Does this algorithm depend on S? Yes! The number of collisions depends on the set S, and this number is useful for the algorithm as it determines the number of tries it takes to build the hash table.

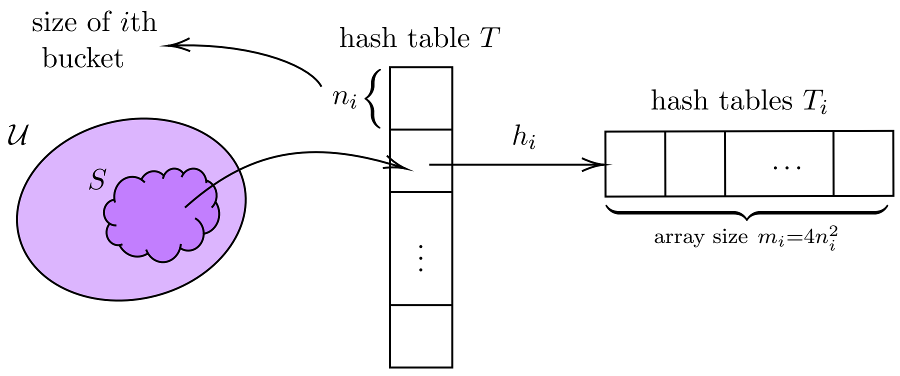
There still remains, however, the issue that this algorithm requires quadratic space; next time we will see a method by which we may improve this to a linear-in-n space algorithm with the same guarantees in query time, and then move on to the next unit of this course: sketching.

§4 Lecture 04—29th January, 2024

§4.1 Perfect hashing—continued; power of two choices

Now we turn our attention to improving the space of the algorithm for hashing by combining the impact of the BASIC-HASH and QUADRATIC-PERFECT-HASH algorithms. The resulting algorithm we shall obtain will use a second layer of hashing to achieve a constant space algorithm.

In more detail, the idea for the algorithm, which we will call PERFECT-HASH, is as follows. Note that paying more in space ensures constant deterministic time, and therefore it makes sense to replace the chain used in the linked list used by the BASIC-HASH algorithm with a second layer of hashing within which we will use the QUADRATIC-PERFECT-HASH algorithm.



We now explicitly outline the algorithm. The algorithm PERFECT-HASH is as follows:

```
Algorithm: PERFECT-HASH
  • Pick $h : \mathcal{U} \to [m]$ at random.
  • Set $n_i = |S_i \cap h^{-1}(i)|$ to be the size of bucket $i$.
  • For each $i \in [m]$:                    // run QUADRATIC-PERFECT-HASH on $n_i$
      – Build a second-level hash function $h_i : \mathcal{U} \to [m_i]$ for $m_i = 4n_i^2$.
      – Hash $|S_i \cap h^{-1}(i)|$ into the second level table $T_i$ using $h_i$.
      – Repeat until there are no further collisions.
```

Two immediate points of note:

1. Obviously the query time is deterministic $O(1)$, since each query involves two hash table lookups.
2. The algorithm is correct by construction, since we know that both of the algorithms we use, BASIC-HASH and QUADRATIC-PERFECT-HASH are correct.

We focus on the space used by this algorithm. We claim that for $m = \Theta(n)$, the expected space used by the PERFECT-HASH is $O(n)$. To see this, note that the construction time for the table is $O(n)$. This is because computing the n_i takes $O(n)$ space, as for every one of the n first level buckets, we compute where each of them hash to, and by so doing we construct the hash table.

We now prove that the space used by the PERFECT-HASH algorithm is $O(n)$ in expectation. The space used in the first-level hash is $O(n) \cdot d[h]$, where $d[h]$ is the description length of the hash function h. (Note that this will not be a problem for us as we are using the limited-randomness α-universal hash functions which are not too long.) In the second level, the space used is $O\left(\sum_{i=1}^{m} n_i^2\right)$—crucially, this is not trivially $O(n)$, as it could be that some of the n_i are almost as large as $\log n$, and if it sp happens that all the elements fall into $n/\log n$ buckets each of size $\log n$, then $\sum_{i=1}^{m} n_i^2$ will be more than linear in n, perhaps even $O(n \log n)$. However, we can show that the expected space used is $O(n)$ with high probability taken over the randomness of the hash function h. Observe that

$$\mathbb{E}_h\left[\sum_{i=1}^{m} n_i^2\right] = \mathbb{E}_h\left[\sum_{i=1}^{m} (n_i + C_i)\right],$$

where C_i is the number of collisions in the ith bucket. (To clarify this observation, consider an $n_i \times n_i$ tabular array with diagonals n_i; then the off-diagonal entries are the number of collisions in the ith bucket, and the entire "area" of the array is the sum of the $n_i + C_i$.) Therefore we can set C to be the total number of collisions in all the buckets, we can directly apply the linearity of expectation to obtain

$$\begin{aligned}
\mathbb{E}_h\left[\sum_{i=1}^{m} n_i^2\right] &= \mathbb{E}_h\left[\sum_{i=1}^{m} (n_i + C_i)\right] \\
&= \mathbb{E}_h\left[\sum_{i=1}^{m} n_i\right] + \mathbb{E}_h\left[\sum_{i=1}^{m} \sum_{\substack{x \text{ in} \\ \text{bucket } i}} C_x\right] \\
&= n + \mathbb{E}[C] \\
&= n + \frac{n^2}{m} = O(n) \quad \text{for } m = \Theta(n).
\end{aligned}$$

This is a nice result; as usual, a few remarks are in order:

Remark 4.1. 1. For the QUADRATIC-PERFECT-HASH algorithm, it is sufficient to use a 2-universal hash function to accomplish $O(n)$ space.

2. We only need the α-universal hash function for the first level of the PERFECT-HASH algorithm to provide sufficient "randomness" to ensure that the expected space used is $O(n)$.

Altogether, the analyses we have supplied for the BASIC-HASH, QUADRATIC-PERFECT-HASH, and PERFECT-HASH algorithms yield the following theorem:

Theorem 4.1. *The algorithm* PERFECT-HASH *achieves $O(n)$ space in expectation, $O(1)$ deterministic query time, and $O(1)$ construction time in expectation.*

Remark 4.2. We can make the space essentially deterministic $O(n)$, since we know that the space is $O(n)$ with probability $\geq 90\%$, and we may simply apply Markov's inequality and repeat the construction $O(1)$ times to achieve guaranteed $O(n)$ space.

Power of two choices. If we use one hash function, both the expected query time and the expected bucket size are $O(1)$ if $m = n$. We can achieve a tight high probability bound on the bucket size by using this hash function:

Claim 4.3. *The maximum load in any bucket is* $\Theta(\log n / \log\log n)$ *with probability* 50%.

Proof. For the proof of the upper bound, fix the bucket size as S_i for the ith bucket. Then

$$\Pr[S_i \geq k] \leq \sum_{\substack{T \subseteq K \\ |T|=k}} \Pr\left[\bigcap_{t \in T} h(t) = i\right] = \binom{n}{k} \cdot \left(\frac{1}{n}\right)^k \leq \left(\frac{en}{k}\right)^k \left(\frac{1}{n}\right)^k = \left(\frac{e}{k}\right)^k.$$

We want $\Pr[S_i \geq k] \leq 1/(4n)$, so that

$$\mathbb{E}[\text{number of buckets of size} \leq k] \leq n \cdot \Pr[S_i \geq k] \leq \frac{1}{4} \leq 1,$$

with probability $1/2$ from Markov's inequality. Hence we can infer for k that

$$\left(\frac{e}{k}\right)^k \leq \frac{1}{4n} \implies k \leq C \cdot \frac{\log n}{\log\log n},$$

for a sufficiently large constant C. The lower bound is similar. □

Now suppose we had two hash functions $h_1, h_2 : \mathcal{U} \to [n]$. For any $x \in \mathcal{U}$, we can compute $h_1(x)$ and $h_2(x)$ and insert x into the lesser loaded bucket. Such load balancing ensures the maximum load for any bucket is $O(\log\log n)$ with probability 50%.

§4.2 Introduction to sketching and streaming, the *Distinct-Count* problem

Now we start a new unit of the course where our focus is similar to the model we used for *Approximate Counting*. In particular, consider some recording device—perhaps an internet router—with massive amounts of data passing through it. Our goal is to output a much smaller summary—a "sketch"—of the data, perhaps with the goal of solving some computational task. So for example we might want to find the unique number of IP addresses of packets that a router processes in order to determine the existence of a DDoS attack.

Today, we focus on the following problem:

Problem 4.4 (*Distinct-Count*)**.** Given a stream of input data composed of a sequence given by $x_1, \ldots, x_m \in [n]$, output the number d of distinct x_i observed at the end of the stream.

What strategies can we immediately think of to solve this problem? Here are a few:

1. *Hashing.* Store the distinct x_i; given a new element, check if it has already been seen. If it has, then discard it; if not, then add it to the hash table of size $\approx m$. This approach uses $O(m \log n)$ space (since we are merely storing the entire stream) and $O(1)$ query time.
2. *$O(n)$ bit-table.* If $n \ll m$, then we can have a bit-vector q of length n where $q_i = 1$ if there exists j such that $x_j = i$ and $q_i = 0$ otherwise. This approach uses $O(n)$ space and $O(1)$ query time.

These are perhaps not ideal. Sadly, they are also near-optimal for deterministic and exact algorithms. Therefore in order to reduce the desired space complexity we must resort to probabilistic and approximate algorithms. In particular, our overarching goal will be to improve on these algorithms by estimating d up to a factor of $1 \pm \varepsilon$ with probability $1 - \delta$ using $O(\log(1/\delta)/\varepsilon^2)$ space (often we will merely take $\delta = 1/10$).

In fact, for $0 < \varepsilon < 1$, we can get space $O(1/\varepsilon^2)$ words for this problem, where each word is $O(\log n)$ bits by using hashing or a similarly motivated procedure. So we will focus our attention on the cases where $d > \Omega(1/\varepsilon^2)$.

§4.2.1 The Flajolet-Martin algorithm

The algorithm due to Philippe Flajolet and Nigel Martin [FM85] is usually credited for being the first to introduce this family of streaming algorithms. We present the algorithm right away:

Algorithm: Flajolet-Martin

- Pick a hash function $h : [n] \to [0, 1]$ at random.
- Initialise a counter $z \leftarrow 1$.
- For a new item $i \in [m]$, update $z \leftarrow \min\{z, h(x_i)\}$.
- Output the estimate $\hat{d} = (1 - d)/d$.

Let us try to gain some intuition for why this algorithm works, via the following observations:

1. *As the algorithm progresses, z cannot increase.* Observe that as our algorithm sees more of the stream and possibly encounters even more unseen elements, it should only be able to increase the estimate it makes and by so doing further decrease z.

2. The update z decreases only if we see a new value in the stream. Notice that prior to seeing the ith item, $z \leq h(x_j)$ for all $j \leq i$. Thus a repeating value in the stream cannot affect the value of z.

3. *The more unique values we see, the likelier it is that the algorithm gives a higher value,* as we would want. Since the output of the hash function is uniformly distributed over $[0, 1]$, the more new elements we see, the more chances we have to draw a smaller number from h, which should reduce z and consequently give a smaller estimate.

We now analyse the Flajolet-Martin algorithm.

Claim 4.5. *Let d be the number of dstinct elements in the stream $x_1, \ldots, x_m$. Then for h the randomly chosen hash function $\mathbb{E}_h[d] = 1/(d+1)$.*

Proof. We can observe that z is exactly $\min_{i=1,\ldots,n} h(x_i)$. Based on our observations above, we can reduce the minimum to the d unique elements of the stream $x_{j_1}, \ldots, x_{j_d}$, and so z is merely a minimum of d randomly selected numbers in the range $[0, 1]$. Pick another number $a \in [0, 1]$ at random. Then

$$\Pr[a \le z] = \Pr\left[a \le \min_{j=1,\dots,d} h(x_{j_d})\right] = \Pr[a = \min\{a, h(x_{j_1}), \dots, h(x_{j_d})\}] = \frac{1}{d+1}.$$

$$\Pr[a \le z] = \int_0^1 \left(\int_0^x f_a(y)\,\mathrm{d}y\right) f_z(x)\,\mathrm{d}x = \int_0^1 x \cdot f_z(x)\,\mathrm{d}x = \mathbb{E}[z],$$

since the $d+1$ numbers are identically distributed and $f_a(y) = 1$ for $y \in [0,1]$ from uniformity. Combining these two results gives the desired conclusion. □

This is nice, but $\mathbb{E}[1/z] \ne 1/\mathbb{E}[z]$, and so our estimate is not unbiased. However, we will prove a concentration bound on z in order to get lower and upper bounds on our estimate $\hat{d}$. To get a concentration bound from Chebyshev's inequality, we need a bound on the variance of z; fortunately, we can readily obtain this bound:

Claim 4.6. *For a fixed fully random hash function $h : [n] \to [0,1]$, we have* $\mathrm{Var}[z] \le 2/d^2$.

Proof. To compute $\mathrm{Var}[z]$, we can use the fact that $\mathrm{Var}[z] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$. We have by the previous claim that $\mathbb{E}[z]^2 = 1/(d+1)^2 > 0$, and so $\mathrm{Var}[z] \le \mathbb{E}[z^2]$. Now fix an $x \in [0,1]$. Since h is fully random and h takes the independent values in the range $[0,1]$, we have that

$$\Pr\left[x < z^2\right] = \Pr\left[\sqrt{x} < z\right] = \prod_{i=1}^{d} \Pr\left[\sqrt{x} < h(x_i)\right] = (1-\sqrt{y})^d,$$

as h is fully random and therefore takes independent values. Furthermore,

$$\mathbb{E}[z^2] = \Pr\left[a \le z^2\right] = \int_0^1 \Pr\left[x \le z^2\right] f_a(x)\,\mathrm{d}x = \int_0^1 (1-\sqrt{x})^d\,\mathrm{d}x = \frac{2}{(d+1)^2+1} \le \frac{2}{d^2}.$$

This completes the proof. □

By Chebyshev then, we have that with probability $\ge 1/2$,

$$z \in \left[\frac{1}{d+1} - \frac{2\sqrt{2}}{d}, \frac{1}{d+1} + \frac{2\sqrt{2}}{d}\right].$$

We can get an equivalent interval for $1/z - 1$, but it will not be centred around d.

§4.2.2 The Flajolet-Martin+ algorithm

Similar to Morris's algorithm, we can linearly increase the space used by a factor of k and reduce the variance by a factor of $1/k$, which translates to a $(1+\varepsilon)$ approximation of $1/(d+1)$ when $k = O(1/\varepsilon^2)$. This leads to the following algorithm:

```
Algorithm: Flajolet-Martin+

  • For $i = 1, \dots, k$:                              // run Flajolet-Martin $k$ times
      – Get $z_i \leftarrow$ Flajolet-Martin $\left(\{x_i\}_{j=1}^m\right)$
  • Return the estimate $\hat{d} = \frac{1}{k}\sum_{i=1}^k \left(\frac{1}{z_i} - 1\right)$.
```

§4.2.3 The Bottom-k algorithm

Instead of the Flajolet-Martin+ algorithm, which employs k hash functions, we can use the Bottom-k algorithm, which updates the k minimal values seen up to the current point of just one run of the simple FM algorithm. This algorithm is as follows:

```
Algorithm: Bottom-k
  • Set $z \leftarrow (1, \dots, 1)$
  • For $i = 1, \dots, m$:
      – Update $z \leftarrow k\text{-MIN}\left\{\{z\}_{j=1}^{k}, h(x_i)\right\}$[a]  // here k is the # of minimal hashes
  • Return the estimate $\hat{d} = \frac{k}{z_{\max}}$, where $z_{\max} = \max\{z_1, \dots, z_k\}$.
```

[a] This step can also be framed as maintaining $z_1 < z_2 < \ldots < z_k$ smallest k hash values seen in the stream

Now we can prove with high probability that we have a $(1+\varepsilon)$ approximation of d when $k = O(1/\varepsilon^2)$ with high probability:

Claim 4.7. *The following are true:*

$$\Pr\left[\hat{d} > (1+\varepsilon)d\right] \le 0.05,$$
$$\Pr\left[\hat{d} < (1-\varepsilon)d\right] \le 0.05.$$

We will prove the first inequality; the proof for the second is similar. For that inequality we define

$$X_i = \begin{cases} 1 & \text{if } h\left(x_{j_i}\right) < \frac{k}{(1+\varepsilon)d}, \\ 0 & \text{otherwise.} \end{cases}$$

We can use this definition to somewhat simplify the requested probability.

Fact 4.8. *The estimate $\hat{d} > (1+\varepsilon)d$ if and only if $\sum_{i=1}^{d} X_i > k$.*

Proof. We proceed directly:

$$\begin{aligned}\sum_{i=1}^{d} X_i > k &\iff \text{there exist at least } k \text{ numbers for which } h(x_{j_i}) < \frac{k}{(1+\varepsilon)d} \\ &\iff \text{there exist at least } k \text{ numbers for which } z_i < \frac{k}{(1+\varepsilon)d} \\ &\iff z_k < \frac{k}{(1+\varepsilon)d} \iff \frac{k}{z_k} > (1+\varepsilon)d \\ &\iff \hat{d} > (1+\varepsilon)d.\end{aligned}$$

□

Proof of Claim 4.7. It then remains to analyse $\Pr\left[\sum_{i=1}^{d} X_i > k\right]$. We have,

$$\mathbb{E}\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} \mathbb{E}\left[X_i\right] = \sum_{i=1}^{d} \frac{k}{(1+\varepsilon)d} = \frac{k}{1+\varepsilon}.$$

and the variance

$$\operatorname{Var}\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} \operatorname{Var}[X_i] = \sum_{i=1}^{d} \left(\mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2\right) \le \frac{k}{1+\varepsilon} \le k,$$

where in the last step we have $\mathbb{E}[X_i^2] = \mathbb{E}[X_i]$ since $X_i \in \{0,1\}$. We can now apply Chebyshev's inequality to obtain

$$\Pr\left[\left|\sum_{i=1}^{d} X_i - \frac{k}{1+\varepsilon}\right| > \sqrt{20k}\right] \le 0.05 \implies \Pr\left[\sum_{i=1}^{d} X_i > \frac{k}{1+\varepsilon} + \sqrt{20k}\right] \le 0.05.$$

Setting $\varepsilon < 1/2$ and $k = O(1/\varepsilon^2)$, we have that

$$\begin{aligned} \frac{i=1}{d} X_i &\le \frac{k}{1+\varepsilon} + \sqrt{20k} \\ &\le k\left(1 - \varepsilon + \varepsilon^2\right) + \sqrt{20k} && \text{by the Taylor series expansion} \\ &\le k \underbrace{- \frac{c}{\varepsilon} + \varepsilon + \sqrt{20}\frac{\sqrt{c}}{\varepsilon}}_{\text{for large } c \text{, this is } \le 0} \\ &\le k \text{ for large } c > 100. \end{aligned}$$

Therefore,

$$\Pr\left[\sum_{i=1}^{d} X_i > k\right] \le \Pr\left[\sum_{i=1}^{d} X_i > \frac{k}{1+\varepsilon} + \sqrt{20k}\right] \le 0.05. \qquad \square$$

§5 Lecture 05—31st January, 2024

§5.1 Selecting the hash function for the Bottom-k algorithm

In the last class we introduced the basic model of streaming problems: we have a stream of objects $x_1, x_2, \ldots, x_m$, and for example they come from some universe of numbers from 1 to n, namely $x_i \in [n]$. We introduced the following problem:

Problem 5.1 (*Distinct-Count*)**.** Given a stream of input data composed of a sequence given by $x_1, \ldots, x_m \in [n]$, output the number d of distinct x_i observed at the end of the stream.

We also gave the following algorithm for this problem:

```
Algorithm: Bottom-$k$
  • Set $z \leftarrow (1, \ldots, 1)$
  • For $i = 1, \ldots, m$:
      – Maintain the ordered $z_1 < z_2 < \ldots < z_k$ smallest $k$ hash values seen in the stream
        // here $k$ is the # of minimal hashes
  • Return the estimate $\hat{d} = \frac{k}{z_{\max}}$, where $z_{\max} = \max\{z_1, \ldots, z_k\}$.
```

There are a few matters we glossed over in the last class, which we will now address.

1. Since the output of the hash function is any real number on the $[0,1]$ interval, the number of buckets of our hash function is uncountably infinite (i.e. one bucket for each real number in $[0,1]$). This is not practical and cannot be stored in memory. So we need to discretise the hash function: instead of having $h : [n] \to [0,1]$, we will have $h : [n] \to \{0, 1/M, 2/M, \ldots, (M-1)/M\}$ for some sufficiently large M.

   Note that when we have real outputs, the probability of collision (i.e. that $h(x) = h(y)$ for random $x \neq y$) is zero. But if we switch to the discretisation, the probability of collision is non-zero. Suppose that there are n distinct items in the stream. If we pick $M = n^3$, then the expected number of collisions is at most $n^2/M = 1/n$. So if we have M sufficiently large, the number of collisions will be very small in expectation.

2. In the Bottom-k algorithm, the hash function must be fully random. Like we mentioned, it is possible to use hash functions with "limited randomness" to achieve our desired goal.

   Definition 5.2 (k-wise independence)**.** *Let H be a family of hash functions $h : [n] \to [M]$. We say that H is k-wise independent if for any $a_1, \ldots, a_k \in \{0, 1/M, \ldots, (M-1)/M\}$ and for any distinct $x_1, \ldots, x_k \in [n]$ and any hash function $h \in H$, we have*

   $$\Pr\left[h(x_1) = a_1 \wedge \ldots \wedge h(x_k) = a_k\right] = \frac{1}{M^k}.$$

   Remark 5.3. If H is 2-wise independent, then it is also universal. Furthermore, for all $k \geq 2$, we can construct a k-wise independent hash function class H of size $\log|H| = O(k \log n)$.

   Theorem 5.1 (Thorup [Tho13])**.** *For the* Bottom-k *algorithm, it is sufficient to use a 2-wise independent hash function.*

§5.2 The *Most-Frequent-Object* problem and heavy hitters

We now focus on a different problem. Given as input a stream $x_1, \ldots, x_m \in [n]$, define a *frequency vector* $f = (f_1, \ldots, f_n) \in \mathbb{R}^n$, where f_i is the number of times i appears in the stream. We are interested in the following problem:

Problem 5.4 (*Most-Frequent-Object*)**.** Given a stream of input data composed of a sequence given by $x_1, \ldots, x_m \in [n]$, output the most frequent object x where f_x is maximal among all the dimensions of the frequency vector f.

Note that the Distinct-Count problem is a special case of the *Most-Frequent-Object* problem; there we output the number of distinct objects, which is the dimension of the frequency vector f.

A simple solution is to store the vector f explicitly and just output the argument of the maximum. However, this is not feasible in the streaming model, as the space complexity of this solution is $\Theta(n)$, which is too large for the streaming model (assuming f is not too sparse). Unfortunately there is some bad news here:

Theorem 5.2. *Any deterministic streaming algorithm for the Most-Frequent-Object problem requires $O(n)$ space. In fact, even 2-approximations require $\Omega(n)$ space, and the hard case (where $f = (0,1,0,1,1,\ldots,1)$ or f contains a 2) requires more than $\Omega(n)$ space already.*

Proof. The proof is by reduction from the DISTINCT-COUNT problem. Suppose we have a streaming algorithm for the *Most-Frequent-Object* problem that uses $o(n)$ space. We can use this algorithm to solve the DISTINCT-COUNT problem as follows: for each $i \in [n]$, we set $f_i = 1$ if i appears in the stream and $f_i = 0$ otherwise. Then we run the *Most-Frequent-Object* algorithm on this frequency vector f. If the output is 1, then we output n; otherwise we output 0. This algorithm uses $o(n)$ space, but it solves the DISTINCT-COUNT problem, which is a contradiction. □

To get past these issues, we will seek to only report items that are *sufficiently* frequent. This is the relaxation that motivates the definition of the *heavy hitter*:

Definition 5.5 (ϕ-heavy-hitter)**.** *Given a stream of input data composed of a sequence given by $x_1, \ldots, x_m \in [n]$, suppose $\phi \in (0, 1)$. Then item $x \in [n]$ is called a* ϕ-heavy-hitter *if $f_x \geq \phi \cdot \sum_y f_y = \phi \cdot m$. In other words, the frequency of item x is at least a ϕ-fraction of the entire length of the stream.*

We can now present the *Heavy-Hitters* problem as a formal relaxation of the *Most-Frequent-Object* problem:

Problem 5.6 (*Heavy-Hitters*)**.** Given a stream of input data composed of a sequence given by $x_1, \ldots, x_m \in [n]$, and a parameter $\phi \in (0, 1)$, find all items that are ϕ-heavy-hitters.[2]

§5.2.1 The BUCKETING algorithm

The eventual goal of the BUCKETING algorithm is to obtain some estimate $\hat{f}_x$ for f_x. In order to reduces the space needed to store the vector, we need to reduce the number of dimensions in the vector. In other words, we neeed to "transform" the original n-dimensional frequency vector f into a smaller vector S with $w \ll n$ dimensions. Now, if we do this, we run the risk of having collisions, since we are reducing from something of dimension n to something of much smaller dimension w. We will take that as a trade-off of less space.

Intuitively it makes sense to use a hash function $h : [n] \to [w]$ to store this information for f in S. Suppose f_{i_1}, f_{i_2}, and f_{i_3} are the dimensions of f that hash to the same bucket i in S. Then we would like to store the sum of f_{i_1}, f_{i_2}, and f_{i_3} in S_i. This is because whenever any of f_{i_1}, f_{i_2}, or f_{i_3} changes, we want to be able to update S_i in $O(1)$ time. This is the basic idea behind the BUCKETING algorithm, which returns an approximate count of how many times each object $x \in [n]$ appears in the stream. The algorithm is as follows:

```
Algorithm: BUCKETING

  • Pick a hash function $h : [n] \to [w]$ with $w \ll n$      // $w$ is the number of buckets
  • Set $S \leftarrow (0, \ldots, 0)$          // initially $S$ is the $w$-dimensional zero vector
  • For each time we see an object $x_i$ in the stream, do the following:
      – Update $S[h(x_i)] \leftarrow S[h(x_i)] + 1$                    // update the count
  • Return the estimators $\hat{f}_x = S[h(x)]$ for each $x \in [n]$.
```

[2]Here we allow the space used by the algorithm to depend on ϕ, maybe something like $O(1/\phi)$ or $O(\log(1/\phi))$.

We will prove that this estimator $\hat{f}_x$ is good. Right away, note that our estimator takes into account the occurrences of x and other elements for which their hash value is $h(x)$, and therefore the estimator must be at least f_x—it never underestimates the frequency of x. The estimator $\hat{f}_x$ can be much larger than the actual frequency f_x if there are many collisions, but we can intuitively see that it is not too large. Note that we care about ϕ-heavy-hitters, that is, about frequencies f_x that are at least $\phi \cdot m$. Intuitively, there is an approximation, as we will never be able to be precisely certain about whether the item's frequency is slightly more than some threshold $\phi \cdot m$ or slightly less, and we need to be able to detect f_x whose magnitude is about $\phi \cdot m$ up to some additive error. This leads to the following claim:

Claim 5.7. *Whenever $w \geq \Omega(1/(\varepsilon\phi))$, the* Bucketing *algorithm outputs an estimate $\hat{f}_x$ such that*

$$\Pr_h\left[\hat{f}_x > f_x + \varepsilon\phi m\right] \leq \frac{1}{10}.$$

Proof. Suppose that h is a fully random hash function. Then for any $x \in [n]$, we have

$$\begin{aligned}
\mathbb{E}\left[\hat{f}_x\right] &= \mathbb{E}\left[\sum_{y\in[n]} \mathbb{1}\{h(x)=h(y)\}\cdot f_y\right] = \mathbb{E}_h\left[f_x + \sum_{y\neq x}\mathbb{1}\{h(x)=h(y)\}\cdot f_y\right] \\
&= f_x + \sum_{y\neq x}\Pr[h(x)=h(y)]\cdot f_y = f_x + \sum_{y\neq x}\frac{f_y}{w} \\
&\leq f_x + \frac{1}{w}\sum_{y\neq x} f_y \leq f_x + \frac{1}{w}\sum_{y} f_y \\
&\leq f_x + \frac{1}{w}\cdot m.
\end{aligned}$$

We can rewrite this as $\mathbb{E}\left[\hat{f}_x - f_x\right] \leq m/w$. Then $\hat{f}_x - f_x$ is a positive random variable and we can use Markov's inequality to get, for $w = 10/(\varepsilon\phi)$,

$$\Pr\left[\hat{f}_x - f_x > 10\cdot\frac{m}{w}\right] \leq \frac{1}{10} \implies \Pr\left[\hat{f}_x > f_x + \varepsilon\phi m\right] \leq \frac{1}{10}. \qquad \square$$

The Bucketing algorithm is a good start, but it is not perfect. The space complexity of the algorithm is $O(w)$, and we would like to reduce this to $O(\log n)$ or $O(\log(1/\phi))$. We will see how to do this in the next class.

§6 Lecture 06—05th February, 2024

§6.1 The ϕ-Heavy-Hitter algorithm

Now the Bucketing algorithm from last time gives us an estimator for the frequency of every fixed x. But this is not an algorithm for outputting all the heavy hitters of the stream. We will not be solving the exact *Heavy-Hitters* problem, where we need to report all the heavy hitters precisely, but will only solve it approximately—as it is very hard to distinguish between elements that are exactly ϕ-heavy-hitters and those that are just a little bit less.

A natural algorithm is the following:

Algorithm: ϕ-HEAVY-HITTER

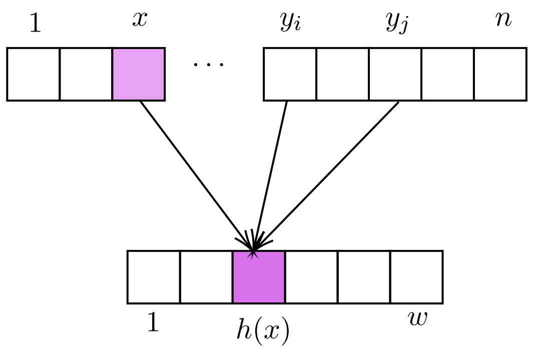
- For each $x \in [n]$, do the following:
  - Compute $\hat{f}_x$ using the sketch S
  - If $\hat{f}_x \geq \phi m$, then report x as a ϕ-heavy-hitter.

We obtain the following guarantees right away:

Corollary 6.1.
- *If $f_x \geq \phi m$, then the algorithm definitely reports x as a ϕ-heavy-hitter since $\hat{f}_x \geq f_x$.*
- *If $f_x < \phi m - \varepsilon\phi m = (1-\varepsilon)\phi m$, then with probability $\geq 90\%$ the algorithm does not report x as a ϕ-heavy-hitter since $\hat{f}_x < \phi m$.*

By Claim 5.7, $\hat{f}_x$ is less than $f_x + \varepsilon\phi m$ with greater than 90% probability. Hence if $f_x < \phi m - \varepsilon\phi m$, then $\hat{f}_x < f_x + \varepsilon\phi m < \phi m$ with probability $\geq 90\%$.

The issue with this algorithm is that with probability $\leq 10\%$, some non-heavy-hitters x are reported as ϕ-heavy-hitters. Indeed this is not just an issue in theory; we have a very large frequency vector f and we map a large number of n items into a relatively small vector of size $w \ll n$. Now say we have a frequent item x that falls into a bucket $h(x)$. Then in expectation, a fraction of $1/w$ of all the other items in the stream will also fall into the bucket $h(x)$, and so we cannot distinguish which one contributed to the heavy frequency corresponding to this bucket.



In fact, all the y such that $h(y) = h(x)$ will be considered heavy, which is problematic because there are a lot of items. How might we solve such an issue? We only need to make sure that there are no collisions but then we need $w = \Omega(n)$, which does not save the space. We now turn to something different.

§6.2 The COUNT-MIN algorithm

The idea behind the COUNT-MIN algorithm is to repeat the basic BUCKETING algorithm $O(\log n)$ times and do some aggregation for sufficiently many experiments to obtain high confidence. The COUNT-MIN algorithm is as follows:

```
Algorithm: Count-Min
  • Initialise $L = O(\log n)$ tables $S_i = S_i[1, \dots, w]$ for $i = 1, \dots, L$.
  • Initialise $L$ independent hash functions $h_i : [n] \to [w]$.
  • Upon receiving an item $x$, for each $i = 1, \dots, L$, do the following:
      – Increment $S_i[h_i(x)] \leftarrow S_i[h_i(x)] + 1$.
  • Return frequency estimator $\hat{f}_x = \min_{i=1,\dots,L} S_i[h_i(x)]$ for each $x \in [n]$.
```

Immediately we observe that just as before, $\hat{f}_x \geq f_x$ for all $x \in [n]$, because none of the L hash tables S_i underestimate, and so their minimum also does not underestimate. We also have the following guarantee:

Claim 6.2. *For $L = O(\log n)$ with $w = 10/(\varepsilon\phi)$, we have*

$$\Pr_h\left[\left\{\forall\, x \in [n] : \hat{f}_x \leq f_x + \varepsilon\phi m\right\}\right] \geq 1 - \frac{1}{n}.$$

Proof. Fix some $x \in [n]$. Then we have that since $S_i[h_i(x)] \geq f_x$ for any fixed x,

$$\begin{aligned}\Pr_{h_1,\dots,h_L}\left[\hat{f}_x > f_x + \varepsilon\phi m\right] &= \prod_{i=1}^{L} \Pr_{h_i}\left[S_i[h_i(x)] > f_x + \varepsilon\phi m\right] \qquad \text{since the } h_i \text{ are independent}\\ &\leq \left(\frac{1}{10}\right)^L = e^{-L\ln 10}\\ &= \frac{1}{n^2},\end{aligned}$$

as long as $L = 2\ln n/\ln 10 = O(\log n)$. Then by the union bound over all $x \in [n]$, we have that

$$\Pr_h\left[\left\{\exists\, x \in [n] : \hat{f}_x > f_x + \varepsilon\phi m\right\}\right] \leq \frac{1}{n}.$$

Hence we have that

$$\Pr_h\left[\left\{\forall\, x \in [n] : \hat{f}_x \leq f_x + \varepsilon\phi m\right\}\right] \geq 1 - \frac{1}{n}.$$ □

The idea for our proof was to show that for every fixed x, the probability that we overestimate the frequency is extremely small, and so the failure probability has to be driven down to be poly-small in n, and then we can apply the union bound over all the $x \in [n]$, and we can conclude that the fraction of items incorrectly estimated is $\leq 1/n$. The ϕ-Heavy-Hitter algorithm is the same as before:

```
Algorithm: $\phi$-Heavy-Hitter for Count-Min
  • For each $x \in [n]$, do the following:
      – Compute $\hat{f}_x$ using the sketch $S$
      – If $\hat{f}_x \geq \phi m$, then report $x$ as a $\phi$-heavy-hitter.
```

Our concluding theorem is then the following:

Theorem 6.1. *The* COUNT-MIN *algorithm with the* ϕ-HEAVY-HITTER *algorithm reports a set* $\mathcal{L} \subseteq [n]$ *such that:*

- *If* f_x *is a* ϕ*-heavy-hitter, i.e.* $f_x \geq \phi m$*, then* $x \in \mathcal{L}$.
- *If* $f_x < (1-\varepsilon)\phi m$*, then* $x \notin \mathcal{L}$,

with probability $\geq 1 - 1/n$.

Furthermore, the algorithm uses (optimal) space $O(w\mathcal{L}) = O\left(\log n/(\varepsilon\phi)\right)$ *words to secure a* $1 \pm \varepsilon$ *multiplicative approximation to the frequency of every item.*

The way we use the hash functions leads to the following remark:

Remark 6.3. It is enough to take each h_i from a family of universal hash functions, and so we need $O(\log n)$ space to store the hash functions h_i.

Example 6.4. Consider a situation where we have a network with several routers $f^{(1)}, \ldots, f^{(k)}$. Suppose we have a separate stream for each router, and assume we are interested in the entire frequency vector f for the network, and not frequency vectors per router. Note then that $f = f^{(1)} + \cdots + f^{(k)}$, which implies that $f_x = f_x^{(1)} + \cdots + f_x^{(k)}$ for any $x \in [n]$. How might we compute a sketch that is able to estimate $\hat{f_x}$ for every $x \in [n]$?
Each rounter computes a COUNT-MIN sketch of its frequency vector (using the same hash function) for its own stream, and then we sum the sketches to obtain the sketch for the entire network. In particular, a COUNT-MIN sketch of some x is a vector from $\mathbb{R}^{\mathcal{L}w}$, and each coordinate in S_i is a linear combination of the coordinates of the frequency vector f. The COUNT-MIN sketch can then be written in the form $A \cdot f$, where $A \in \mathbb{R}^{\mathcal{L}w \times n}$ has, for $h_i(x) = j$ for $i \in [L]$ and $j \in [w]$,

$$A_{(i,j),x} = \begin{cases} 1 & \text{if } h_i(j) = h_i(x), \text{ for } h_i(x) = j \text{ for } i \in [L] \text{ and } j \in [w] \\ 0 & \text{otherwise,} \end{cases}$$

and this is another way to write the sketch function. So the sketch can be described as a linear transformation of the frequency vector f:

$$\underbrace{Af^{(1)}}_{\text{sketch for router 1}} + \cdots + \underbrace{Af^{(k)}}_{\text{sketch for router } k} = \underbrace{A(f^{(1)} + \cdots + f^{(k)})}_{\text{sketch for the network}},$$

which suggests then that COUNT-MIN sketches are linear sketches of the frequency vector.

Now consider the size of $\mathcal{L}$, which we write as $|\mathcal{L}|$. We know that with high probability, $\mathcal{L}$ only contains elements that have $f_x \geq (1-\varepsilon)\phi m$, which means that $|\mathcal{L}|$ is at most

$$|\mathcal{L}| \leq \frac{m}{(1-\varepsilon)\phi m} \leq \frac{1}{(1-\varepsilon)\phi} \leq \frac{1}{\phi},$$

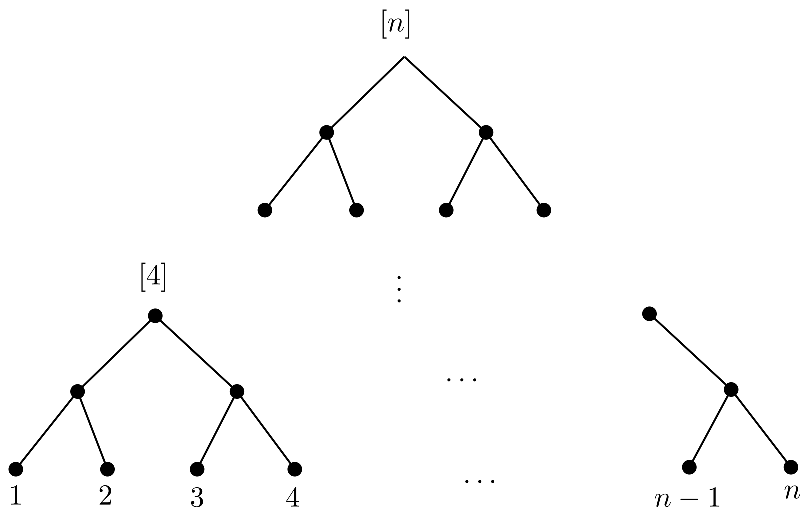
and so we have to go through all the x in $[n]$ to check if they are ϕ-heavy-hitters. This is an issue, because then the time to construct $\mathcal{L}$ is $\Omega(n)$. We'd like to reduce this time to get a faster algorithm.

§6.3 The Fast-Count-Min sketch

We will now describe a faster algorithm for constructing $\mathcal{L}$, which we call the Fast-Count-Min algorithm, and which is presented as follows.

Theorem 6.2. *There exists an algorithm, the* Fast-Count-Min *algorithm, that has the exact same guarantees as the* Count-Min *algorithm, but runs in time $O\left(\phi^{-1}\log^2 n\right)$ and uses space $O\left((\varepsilon\phi)^{-1}\log^2 n\right)$.*

Proof. Let n be a power of 2, and fix level $\ell = 0, 1, 2, \dots, \log_2 n$, and imagine constructing a binary tree on the final levels, each corresponding to a different level ℓ:



These are the dyadic intervals; they are intervals of the form $I_i^\ell = [i \cdot 2^\ell + 1, (i+1)2^\ell]$ for $i = 0, 1, \dots, n/2^\ell - 1$. In particular, the first level $\ell = 0$ is the entire set $[n]$, and the last level $\ell = \log_2 n$ is a set of singletons, the root node; also, the size of the set N^ℓ containing all such intervals is $|N^\ell| = n \cdot 2^{-\ell}$. We will now describe the Fast-Count-Min sketch:

<u>*Algorithm*: Fast-Count-Min sketch</u>

- For each level $\ell = 0, 1, \dots, \log_2 n$, do the following:
  - Store a Count-Min sketch S^ℓ for the set N^ℓ. Here, we can write Count-Min$_\ell$ to denote the Count-Min sketch for the universe N^ℓ.

In some sense we can think of this as splitting the stream into $O(\log n)$ streams, and then we can run Count-Min on each of these streams. Note that in that case, for fixed ℓ, $f^\ell_{I_i^\ell} = \sum_{x \in I_i^\ell} f_x$, and furthermore, $f^\ell \in \mathbb{R}^{n/2^\ell}$.

We have the following very simple observation. Fix some interval I_i^ℓ with children $I_j^{\ell-1}, I_{j+2^{\ell-1}}^{\ell-1}$. If $I_j^{\ell-1}$ or $I_{j+2^{\ell-1}}^{\ell-1}$ is a ϕ-heavy-hitter, then its parent I_i^ℓ is also a ϕ-heavy-hitter. To see this, note that if $I_j^{\ell-1}$ is a ϕ-heavy-hitter, then

$$f^{\ell-1}_{I_j^{\ell-1}} = \sum_{x \in I_j^{\ell-1}} f_x \geq \phi m,$$

since by the way the tree above is constructed, the sum of the frequencies at each level is exactly the same as the sum of the frequencies at the level below, and so the meaning of the ϕ-heavy-hitter is preserved. Then

$$f^{\ell-1}_{I^{\ell-1}_j} \geq \phi m \implies f^{\ell}_{I^{\ell}_i} = f^{\ell-1}_{I^{\ell-1}_j} + f^{\ell-1}_{I^{\ell-1}_{j+2^{\ell-1}}} \geq \phi m,$$

and so the algorithm for finding the heavy hitters is as follows:

Algorithm: ϕ-Heavy-Hitter for Fast-Count-Min

- On node v with children v_1, v_2, do the following:
  - If v is a leaf and a ϕ-heavy-hitter, then report v as a ϕ-heavy-hitter and continue.
  - If v_1 or v_2 is a ϕ-heavy-hitter, then recurse on v_1 or v_2.

The time bound here is modulo how much it takes us to count the Count-Min sketch for the frequency in each level; i.e. the product of $c \cdot \log n$, the time taken to estimate the frequency $\hat{f}$ in each level, and the maximum number of ϕ-heavy-hitters at each level:

$$T(n) \leq (c \cdot \log n) \cdot \log n \cdot \frac{1}{\phi} = O\left(\phi^{-1} \log^2 n\right),$$

and the space used is

$$O\left(\log n \cdot \frac{1}{\varepsilon\phi} \cdot \log n\right) = O\left((\varepsilon\phi)^{-1} \log^2 n\right). \qquad \square$$

§7 Lecture 07—07th February, 2024

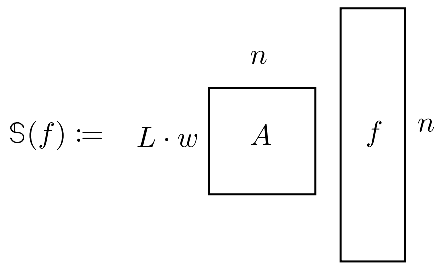
§7.1 The power of linearity

Last time we saw the Count-Min sketch:

Sketch: Count-Min

- Initialise $L = O(\log n)$ tables $S_i = S_i[1, \ldots, w]$ for $i = 1, \ldots, L$.
- Initialise L independent hash functions $h_i : [n] \to [w]$.
- Store $S_i[j] \leftarrow \sum_{k \in [n], h_i(k) = j} f_k$.

We can think of the table $\mathbb{S}(f)$ with elements $S_i[j]$ as a *sketching function* that depends on the input f and maps an n-dimensional real vector to an Lw-dimensional real vector. An observation that then follows is that the Count-Min sketch is linear in the linear algebraic sense:



Here, A is a matrix such that $S_i[j] = A_{ij} \cdot f$, where $A_{(ij),k} = 1$ if $h_i(k) = j$. We can then think of the COUNT-MIN sketch as a linear transformation of the input vector f.

We have seen an example in Example 6.4; we give another example below.

Example 7.1. Consider a situation where we have a network with one input router with frequency f_{input} and one output router with frequency f_{output}, where $f_{\text{output}} \leq f_{\text{input}}$, i.e. some of the packets are dropped in the network.
Suppose we wanted to find the heavy hitters of the dropped packets, i.e. the heavy hitters of $\Delta f = f_{\text{input}} - f_{\text{output}}$. To do this, we pick a fixed hash function h_i at random, and also store the sketch S_i of f_{input} and f_{output}. Then, we can compute the sketch of Δf as $S_i[\Delta f] = S_i[f_{\text{input}}] - S_i[f_{\text{output}}]$, so that $\mathbb{S}(\Delta f) = \mathbb{S}(f_{\text{input}}) - \mathbb{S}(f_{\text{output}})$, by linearity. To find the heavy hitters of Δf, we can then use the COUNT-MIN sketch to find the heavy hitters of $\mathbb{S}(\Delta f)$.

Note that, in some sense, linearity is also the reason why we can update the sketch as we see the traffic:

$$\mathbb{S}\left(f + \mathrm{e}_x\right) = \mathbb{S}(f) + \mathbb{S}(\mathrm{e}_x),$$

where e_x is the basis vector with all zeros except for a 1 at position x.

§7.2 Frequency moments and the TUG-OF-WAR algorithm

So far we have discussed the problem of finding $\|f\|_\infty$, the most frequent item in a stream, via the COUNT-MIN sketch. Before that, we saw the problem of finding $\|f\|_0$, the number of distinct elements in a stream, via the algorithm due to Flajolet and Martin. We now turn to other norms, say $\|f\|_1$ (the sum of the frequencies, i.e. the number of packets in the stream) and $\|f\|_2$ (the sum of the squares of the frequencies, i.e. the sum of the number of packets in the stream). The first is easy; the second is much more interesting, and is in fact the most natural norm to consider in the context of frequency moments (for mathematical reasons, such as the fact that it is the only norm with an inner product).

The p-th frequency moment of a stream f is defined as

$$F_p(f) \coloneqq \|f\|_p^p = \sum_x f_x^p.$$

Our attention will be focused on the second frequency moment, $F_2 = \sum_x f_x^2$, which is the sum of the squares of the frequencies. We will present a very classical algorithm developed in 1998 by Alon, Matias and Szegedy, called the TUG-OF-WAR algorithm.

Motivation The motivation for the TUG-OF-WAR algorithm is as follows. Suppose we want to distinguish the following two cases:

- Either a frequency vector $f_1 = (0/1, 0/1, \ldots, 0/1)$, i.e. all elements randomly either appear once or don't appear at all, or
- The frequency vector $f_2 = (0/1, 0/1, \ldots, 0/1, 10\sqrt{n}, 0/1, \ldots, 0/1)$, i.e. all elements randomly either appear once or don't appear at all, except for one random element that appears $10\sqrt{n}$ times.

Can we solve the problem of determining $\|f\|_2$ in both cases with space better than $O(n)$? We know that $m \leq 2n$, and in f_2 we know hat i is a ϕ-heavy-hitter for $\phi = 10\sqrt{n}/n = 10/\sqrt{n} \approx 1/\sqrt{n}$. We can then use Count-Min with $\phi \approx 1/\sqrt{n}$ to solve the problem with space $O(\sqrt{n}\log n)$ since ε is constant. Now can we do better? Note that $F_2(f_1) \leq n$ and $F_2(f_2) \geq (10\sqrt{n})^2 = 100n$. Now, we cannot distinguish f_1 from f_2 using Count-Min unless we have space $O(\sqrt{n})$, because the one term in f_2 that is $10\sqrt{n}$ is a ϕ-heavy-hitter for $\phi = 10\sqrt{n}/n = 10/\sqrt{n} \approx 1/\sqrt{n}$. Count-Min may mao a lot of values with frequency 1 in the same bucket as this element, particularly if we map to a small table. In the extreme case for example, if we have only one bucket, the frequencies of 1 overwhelm the $\sqrt{n}$ frequency, and it will be tough to sufficiently distinguish our two frequency vectors, unless we have a table of space $O(\sqrt{n})$.

This discussion motivates the following idea: we multiply all our frequencies in the frequency vector by a random variable $r_x \in \{-1, 1\}$, and then we look at the expected value of the sum of the squares of the frequencies multiplied by the random variables. Why would we do this? The sum of n random $\{-1, 1\}$ variables has expectation 0, variance $\approx n$, and standard deviation $\approx \sqrt{n}$. So, if we look at the expected value of the sum of the squares of the frequencies multiplied by the random variables, we shouldn't expect to be too far from 0. If we have any outlier frequencies, or nonrandom frequencies, then these influence our sums more and are easier to track. This is the idea behind the Tug-of-War algorithm.

Theorem 7.1 (Alon, Matias, Szegedy [AMS96])**.** *There is an algorithm that can estimate F_2 up to a $(1 \pm \varepsilon)$ factor using space $O(\varepsilon^{-2}\log n)$ (i.e. $O(1/\varepsilon^2)$ bits) and one pass over the stream.*

Proof. The algorithm whose existence is claimed here is the Tug-of-War algorithm:

```
Algorithm: Tug-of-War

  • Initialise $z \leftarrow 0$.
  • Choose $\sigma : [n] \to \{\pm 1\}$                                // pick the signs at random
  • Set $z \leftarrow \sum_{x\in[n]} \sigma(x) \cdot f_x$, and upon seeing an item $x$, update $z \leftarrow z + \sigma_x$   // sketch
  • Return the second frequency moment estimate $\hat{F}_2 \leftarrow z^2$.
```

We now proceed with the usual analyses. The estimator $\hat{F}_2$ is unbiased, i.e. $\mathbb{E}[\hat{F}_2] = \mathbb{E}[z^2] = F_2$, as we show below:

$$\mathbb{E}\left[z^2\right] = \mathbb{E}\left[\left(\sum_{x\in[n]} \sigma_x \cdot f_x\right)^2\right] = \mathbb{E}\left[\sum_{x,y\in[n]} \sigma_x\sigma_y f_x f_y\right] = \sum_{x,y\in[n]} \mathbb{E}\left[\sigma_x\sigma_y\right] f_x f_y = \sum_{x\in[n]} f_x f_x = F_2,$$

since $\mathbb{E}[\sigma_x\sigma_y] = 1$ if $x = y$ and 0 otherwise. Furthermore, we claim that the variance $\mathrm{Var}[\hat{F}_2] = \mathrm{Var}[z^2] \leq 3F_2^2$, and prove the claim below.

$$\begin{aligned}\mathrm{Var}[\hat{F}_2] = \mathrm{Var}[z^2] &= \mathbb{E}[z^4] - \mathbb{E}[z^2]^2 = \mathbb{E}[z^4] - F_2^2 \\ &= \mathbb{E}\left[\left(\sum_{x\in[n]} \sigma_x f_x\right)^4\right] - F_2^2\end{aligned}$$

$$
\begin{aligned}
&= \mathbb{E}\left[\left(\sum_{x,y,u,v\in[n]} \sigma_x\sigma_y\sigma_u f_v f_x f_y f_u f_v\right)\right] - F_2^2 \\
&= \sum_{x,y,u,v\in[n]} \mathbb{E}\left[\sigma_x\sigma_y\sigma_u f_v\right] f_x f_y f_u f_v - F_2^2 \\
&= \sum_x f_x^4 + 3\sum_{x\neq y} f_x^2 f_y^2 - F_2^2 \\
&= F_2^2 + 3F_2^2 - F_2^2 = 3F_2^2,
\end{aligned}
$$

since $\mathbb{E}[\sigma_x\sigma_y\sigma_u f_v] = 1$ if $x = y = u = v$ or $x = y \neq u = v$. Now by Chebyshev's inequality, we have

$$
\Pr\left[\left|\hat{F}_2 - F_2\right| \geq \underbrace{\sqrt{3\cdot 10F_2^2}}_{\leq 6F_2}\right] \leq \frac{1}{10},
$$

so that $z^2 = F_2 \pm 6F_2$ with probability at least $9/10$. □

§7.2.1 The Tug-of-War+ algorithm

To obtain a better, more concentrated $1 \pm \varepsilon$ estimate for the second frequency moment, we can use the same technique of averaging over multiple independent runs of the Tug-of-War algorithm. This is exactly what the Tug-of-War+ algorithm does. We present the algorithm below.

Algorithm: Tug-of-War+

- Keep $k = \Theta(1/\varepsilon^2)$ counters $z_1, \ldots, z_k$.
- Pick $\sigma_1, \ldots, \sigma_k : [n] \to \{\pm 1\}$ independently at random.
- Upon seeing z, update each counter $z_i \leftarrow z_i + \sigma_i(x)$.
- Return the second frequency moment estimate $\hat{F}_2 \leftarrow \frac{1}{k}\sum_{i=1}^k z_i^2$.

The analyses of the Tug-of-War+ algorithm are similar to those of the Tug-of-War algorithm, and we present them below.

Theorem 7.2 (Alon, Matias, Szegedy [AMS96]). *There is an algorithm that can estimate F_2 up to a $(1 \pm \varepsilon)$ factor using space $O(\varepsilon^{-2}\log n)$ (i.e. $O(1/\varepsilon^2)$ bits) and one pass over the stream.*

Proof. The estimator $\hat{F}_2$ is unbiased, i.e. $\mathbb{E}[\hat{F}_2] = \mathbb{E}\left[\frac{1}{k}\sum_{i=1}^k z_i^2\right] = \frac{1}{k}\sum_{i=1}^k \mathbb{E}[z_i^2] = F_2$, and the variance $\mathrm{Var}[\hat{F}_2] = \mathrm{Var}\left[\frac{1}{k}\sum_{i=1}^k z_i^2\right] = \frac{1}{k^2}\sum_{i=1}^k \mathrm{Var}[z_i^2] \leq \frac{3F_2^2}{k}$, so that by Chebyshev's inequality,

$$
\Pr\left[\left|\hat{F}_2 - F_2\right| \geq \sqrt{10\cdot\frac{3F_2^2}{k}}\right] \leq \frac{1}{10},
$$

so that with $k = 36/\varepsilon^2$, we have $\hat{F}_2 = F_2 \pm \sqrt{10\cdot\frac{3F_2^2}{k}} = F_2 \pm \varepsilon\left(\frac{5}{6}\right)^{1/2} F_2$ with probability at least $9/10$. □

Recall now that our sketch $\mathbb{S} : \mathbb{R}^n \to \mathbb{R}^k$ is linear, and that for a vector z of length k,

$$\mathbb{S}(f) = z = A \cdot f,$$

where A is a $k \times n$ matrix. In the case of the Tug-of-War+ algorithm, $A_{ij} = \sigma_i(j) \in \{-1, 1\}$, and in the case of the Count-Min sketch, $A_{ij} = 1$ if $h_i(j) = x$ and 0 otherwise. We can then think of the Tug-of-War+ algorithm as a linear transformation of the input vector f, as always.

But what do we really need from the σ? We used that $\mathbb{E}\left[\sigma_i(j)\right] = 0$, as well as $\mathbb{E}\left[\sigma_i(j)^2\right] = 1$, and $\mathbb{E}\left[\sigma_i(j)^4\right] \le O(1)$. Indeed, the p-Rademacher uniform random variable $\sigma_i(j)$ is not the only random variable that satisfies these properties; there are many others, such as the (μ, σ^2)-Gaussian, which is perhaps the most natural. We can then use the Gaussian random variable in place of the Rademacher random variable in the Tug-of-War+ algorithm, and we will still have the same guarantees (up to constant factors).

We end today with a final remark. We already said that our estimate satisfies

$$\Pr\left[\hat{F}_2 = (1 \pm \varepsilon) F_2\right] \ge 0.9 \iff \frac{1}{k}\|z\|_2^2 = \|f\|_2^2(1 \pm \varepsilon),$$

so that

$$\frac{1}{k}\|\mathbb{S}(f)\|_2^2 = (1 \pm \varepsilon)\|f\|_2^2.$$

Now this is a dimension-reducing map from ℓ_2 to ℓ_2! We are interested in the ℓ_2-norm of $f \in \mathbb{R}^n$, and we have a relationship between k-dimensional function $\mathbb{S}(f)$ and the ℓ_2-norm of f up to scaling by a factor of $1 \pm \varepsilon$; in particular, k does not depend on the original dimension *at all.* This is a very important component of dimension reduction, and we will see more of it in the next lecture.

§8 Lecture 08—12th February, 2024

§8.1 Dimension reduction

Last time, we discussed how the Tug-of-War+ algorithm connects to the dimension-reducing map

$$\frac{1}{k}\|\mathbb{S}(f)\|_2^2 = (1 \pm \varepsilon)\|f\|_2^2.$$

By linearity, we have that for all $x, y \in \mathbb{R}^n$ and with probability $\ge 90\%$,

$$\|\mathbb{S}(x) - \mathbb{S}(y)\|_2^2 = \|\mathbb{S}(x - y)\|_2^2 = (1 \pm \varepsilon)\|x - y\|_2^2,$$

which is a dimension-reducing map $\mathbb{S} : \mathbb{R}^n \to \mathbb{R}^k$ with $k = O(\varepsilon^{-2} \log n)$.

Consider now the following question: can we boost the probability of success of guarantee for the Tug-of-War+ algorithm from 0.9 to $1 - \delta$ for any $\delta > 0$? The answer is yes:

Theorem 8.1. *Given a failure probability $\le \delta$ where $\delta \in (0, 1)$ is small, we can obtain an estimate $\hat{F}_2$ such that*

$$\Pr\left[\hat{F}_2 = (1 \pm \varepsilon) F_2\right] \ge 1 - \delta$$

as long as $k = O(\varepsilon^{-2} \log(1/\delta))$.

Proof. Ungraded exercise. □

However, using Chebyshev we can only prove the above theorem for $k \le O\left((\delta\varepsilon^2)^{-1}\right)$, which is much worse than the bound $O(\varepsilon^{-2}\log(1/\delta))$; we want our δ to be very small, say polynomially small in n.

§8.2 The Johnson-Lindenstrauss Lemma

The Johnson-Lindenstrauss Lemma is a fundamental result in dimension reduction; before we state it, we first need to define precisely the map whose existence it postulates.

Definition 8.1 (Sketching function). *For an $x \in \mathbb{R}^n$, a* sketching function *is a map $\varphi : \mathbb{R}^n \to \mathbb{R}^k$ such that for $\sigma \in \mathbb{R}^{k\times n}$, we have that*

$$\varphi(x) = \sigma \cdot x = \frac{1}{\sqrt{k}}\left(\sum_{i=1}^{n}\sigma_{1i}x_i, \sum_{i=1}^{n}\sigma_{2i}x_i, \ldots, \sum_{i=1}^{n}\sigma_{ki}x_i\right).$$

As we were discussing above, we have that φ is an estimator of the ℓ_2 norm of x:

$$\|\varphi(x)\|_2^2 = (1\pm\varepsilon)F_2(x),$$

and that for a linear sketching function $\varphi(x) = \sigma\cdot x$, it is true that $\varphi(x\pm y) = \varphi(x)\pm\varphi(y)$.

We need some preliminaries before we introduce the Johnson-Lindenstrauss Lemma.

Definition 8.2 (Gaussian random variable). *A random variable X is said to be standard* Gaussian *if its probability density function is given by*

$$f(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}.$$

An important property of standard Gaussian random variables is the spherical symmetry property:

Fact 8.3 (Spherical symmetry of the Gaussian). *For a vector $\overline{g} = (g_1,\ldots,g_n)\in\mathbb{R}^n$ where each g_i is an i.i.d. standard Gaussian random variable, we have that the probability density function $f(\overline{g})$ is invariant under any orthogonal transformation of $\overline{g}$, i.e.*

$$f(\overline{g}) = \prod_{i=1}^{n}\frac{1}{\sqrt{2\pi}}\cdot e^{-g_i^2/2} = \left(\frac{1}{\sqrt{2\pi}}\right)^n\cdot e^{-\|\overline{g}\|_2^2/2}.$$

The spherical symmetry property merely states that if we visualise the distribution of $\overline{g}$ as contour lines in $\mathbb{R}^n$, then the distribution is rotationally symmetric about the origin, and so the probability density function is invariant under any orthogonal transformation of $\overline{g}$.

A corollary of this fact is the following:

Fact 8.4 (Stability of the Gaussian). *Given the vector $\overline{g} = (g_1,\ldots,g_n)\in\mathbb{R}^n$ where each g_i is an i.i.d. standard Gaussian random variable, the inner product $\langle\overline{g}, z\rangle = \sum_{i=1}^n g_iz_i$ is also a Gaussian random variable; in particular $\langle\overline{g}, z\rangle$ is distributed as $g'\cdot\|z\|_2$, where $g'\sim\mathcal{N}(0,1)$ and $\|z\|_2 = \sqrt{\sum_{i=1}^n z_i^2}$.*

Proof. Take an arbitrary rotation $R(z)$ of z; then $\langle \overline{g}, z\rangle \sim \langle \overline{g}, R(z)\rangle$. Take the rotation $R(z) \triangleq (\|z\|_2, 0, \ldots, 0)$—the ℓ_2 norm is preserved, so this is a rotation. Then $\langle \overline{g}, z\rangle \sim \overline{g} \cdot \|z\|_2 = g' \cdot \|z\|_2$, where $g' \sim \mathcal{N}(0,1)$. $\square$

We will also need the chi-squared distribution, which is a generalisation of the Gaussian distribution.

Definition 8.5 (Chi-squared distribution)**.** *A random variable X is said to have a chi-squared distribution with n degrees of freedom if its probability density function is given by*

$$f(x) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2},$$

where $\Gamma(n/2)$ is the gamma function evaluated at $n/2$.

The chi-squared distribution is the distribution of the sum of the squares of n independent standard Gaussian random variables.

The final fact we need is the following:

Fact 8.6. *Given a chi-squared random variable χ_k with k degrees of freedom, we have that* $\Pr\left[1-\varepsilon \le \chi_k^2 \le 1+\varepsilon\right] \le e^{-\varepsilon^2 k/9}$.

We now state the Johnson-Lindenstrauss Lemma.

Lemma 8.7 (Johnson-Lindenstrauss, [JL84])**.** *For any $\varepsilon > 0$, there exists a distribution over linear maps $\varphi : \mathbb{R}^n \to \mathbb{R}^k$ such that for all $x, y \in \mathbb{R}^n$,*

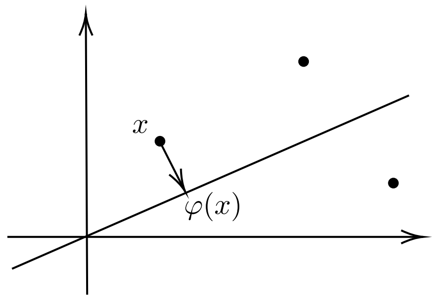
$$\Pr_{\varphi}\left[(1-\varepsilon)\|x-y\|_2^2 \le \|\varphi(x)-\varphi(y)\|_2^2 \le (1+\varepsilon)\|x-y\|_2^2\right] \ge 1 - e^{-\varepsilon^2 k/9}.$$

Proof. If φ is linear, then $\|\varphi(x)-\varphi(y)\|_2^2 = (1\pm\varepsilon)\|\varphi(x-y)\|_2^2$, and with $z = x-y$, we have that $\|\varphi(z)\|_2^2 = (1\pm\varepsilon)\|z\|_2^2$. So it is enough to prove this statement for an arbitrary fixed z:

$$\Pr_{\varphi}\left[(1-\varepsilon)\|z\|_2^2 \le \|\varphi(z)\|_2^2 \le (1+\varepsilon)\|z\|_2^2\right] \ge 1 - e^{-\varepsilon^2 k/9}.$$

This is a constructive lemma; we will construct concrete φ. There are a few options for which φ can prove this theorem:

- For φ being the TUG-OF-WAR+ algorithm, we have that $\|\varphi(z)\|_2^2 = (1\pm\varepsilon)\|z\|_2^2$ with probability $\ge 1-\delta$ as long as $k = O(\varepsilon^{-2}\log(1/\delta))$, as we have seen before.
- Take φ a projection on a random k-dimensional linear subspace.



- Take φ to be the matrix vector product $\varphi(x) = G \cdot x$ where G is a random rescaled matrix with i.i.d. Gaussian entries $G_{ij} \sim \mathcal{N}(0,1)$.

We will adopt the third implementation, which satisfies the conditions $\mathbb{E}[G_i] = 0$, $\mathbb{E}\left[G_i^2\right] = 1$, and $\mathbb{E}\left[G_i^4\right] = 1$ that we used in the proof of the Tug-of-War+ algorithm.

(Now can we get some k for which this works for any x, y with a fixed φ, without any randomness whatsoever? No; this is not possible even when $k = n-1$ (if we have n-dimensional space and throw in one extra dimension, it is mathematically impossible for all the distances to be preserved since an n-dimensional space is not isometric to an $n-1$-dimensional space). So probability plays a crucial role in this lemma.)

Now let us explicitly write out the rescaled form of φ:

$$\begin{aligned}
\varphi(z) &= \frac{1}{\sqrt{k}} G z \\
&= \frac{1}{\sqrt{k}} \left(\sum_{i=1}^{n} G_{1i} z_i, \sum_{i=1}^{n} G_{2i} z_i, \ldots, \sum_{i=1}^{n} G_{ki} z_i \right) && \text{by the matrix-vector product} \\
&= \frac{1}{\sqrt{k}} \left(\|z\|_2 g_1, \|z\|_2 g_2, \ldots, \|z\|_2 g_k \right) && \text{by Fact 8.4 and with } g_i \sim \mathcal{N}(0,1),
\end{aligned}$$

so that

$$\|\varphi(z)\|_2^2 = \frac{1}{k} \left(\sum_{i=1}^{k} \|z\|_2^2 g_i^2 \right) = \|z\|_2^2 \left(\frac{1}{k} \sum_{i=1}^{k} g_i^2 \right).$$

We have that $\frac{1}{k}\sum_{i=1}^{k} g_i^2$ is a chi-squared random variable with k degrees of freedom, so by Fact 8.6, we have that

$$\Pr\left[1 - \varepsilon \leq \frac{1}{k} \sum_{i=1}^{k} g_i^2 \leq 1 + \varepsilon\right] \leq e^{-\varepsilon^2 k/9}.$$

Therefore,

$$\Pr\left[(1-\varepsilon)\|z\|_2^2 \leq \|\varphi(z)\|_2^2 \leq (1+\varepsilon)\|z\|_2^2\right] \geq 1 - e^{-\varepsilon^2 k/9},$$

as required. □

The following corollary gives us a way to tune the dimension k to achieve a desired failure probability δ, which is a very useful result.

Corollary 8.8. *For any $\delta \in (0,1)$, there exists a distribution over linear maps $\varphi : \mathbb{R}^n \to \mathbb{R}^k$ such that for all $x, y \in \mathbb{R}^n$,*

$$\Pr_{\varphi}\left[(1-\varepsilon)\|x-y\|_2^2 \leq \|\varphi(x) - \varphi(y)\|_2^2 \leq (1+\varepsilon)\|x-y\|_2^2\right] \geq 1 - \delta$$

as long as $k = (9/\varepsilon^2) \cdot \ln(1/\delta)$.

Proof. By the Johnson-Lindenstrauss Lemma, we have that

$$\Pr\left[(1-\varepsilon)\|x-y\|_2^2 \leq \|\varphi(x) - \varphi(y)\|_2^2 \leq (1+\varepsilon)\|x-y\|_2^2\right] \geq 1 - e^{-\varepsilon^2 k/9},$$

so we need to solve the equation $e^{-\varepsilon^2 k/9} = \delta$ for k to obtain the desired result. □

With a view towards nearest neighbour search, we present the following corollary of the Johnson-Lindenstrauss Lemma.

Corollary 8.9. *Fix N vectors $x_1, \dots, x_N \in \mathbb{R}^d$. Then there exist vectors $y_1, \dots, y_N \in \mathbb{R}^k$ such that $k = O(\varepsilon^{-2} \log N)$ and for all $i, j \in [N]$,*

$$\Pr_{\varphi} [(1-\varepsilon)\|x_i - x_j\| \leq \|y_i - y_j\| \leq (1+\varepsilon)\|x_i - x_j\|] \geq 1 - \frac{1}{N}.$$

Proof. Take $\varphi : \mathbb{R}^d \to \mathbb{R}^k$ from Lemma 8.7 with $\delta = 1/N^3$ and define $y_i = \varphi(x_i)$ for all $i \in [N]$. By the Johnson-Lindenstrauss Lemma, we have that

$$\Pr_{\varphi} \left[\text{not} \left\{(1-\varepsilon)\|x_i - x_j\|^2 \leq \|y_i - y_j\|^2 \leq (1+\varepsilon)\|x_i - x_j\|^2\right\}\right] \leq \frac{1}{N^3},$$

so that with $k = \Theta(\varepsilon^{-2} \log(1/\delta))$,

$$\Pr_{\varphi} \left[\text{not} \left\{(1-\varepsilon)\|x_i - x_j\|^2 \leq \|y_i - y_j\|^2 \leq (1+\varepsilon)\|x_i - x_j\|^2\right\}\right] \leq \frac{1}{N^3}.$$

Now by the union bound over all pairs $i, j \in [N]$,

$$\Pr_{\varphi} \left[\text{there exist pairs } i, j \text{ such that not} \left\{(1-\varepsilon)\|x_i - x_j\|^2 \leq \|y_i - y_j\|^2 \leq (1+\varepsilon)\|x_i - x_j\|^2\right\}\right] \leq N^2 \cdot \frac{1}{N^3} = \frac{1}{N},$$

so that with probability $\geq 1 - 1/N$, we have that

$$(1-\varepsilon)\|x_i - x_j\|^2 \leq \|y_i - y_j\|^2 \leq (1+\varepsilon)\|x_i - x_j\|^2$$

for all $i, j \in [N]$. □

Sometimes the Johnson-Lindenstrauss Lemma is stated in the form of Corollary 8.9. The gist is that given n vectors in a (possibly infinite-dimensional) space, we can find a low-dimensional representation of these vectors such that the metric properties of the original vectors are preserved with high probability and up to a small error.

§8.2.1 Discussion of the Johnson-Lindenstrauss Lemma

With N vectors in an infinite-dimensional space, we can always find $y_1, \dots, y_N \in \mathbb{R}^k$ such that the metric properties of the original vectors are preserved exactly, with $k = N + 1$.

But we have only shown dimension reduction in the ℓ_2 norm—what about other norms? For the ℓ_1 norm, consider the map $\varphi \colon \mathbb{R}^n \to \mathbb{R}^k$; can we preserve metrics with good probability and approximation or reduce dimensions of any given N points? No:

Theorem 8.2 (Brinkman and Chakar [BC05]). *There exist N points in ℓ_1 that require $k \geq N^{\Omega(1/\alpha)}$ for an α-approximation.*

Indeed, a stronger version of the above theorem is due to Naor and Johnson:

Theorem 8.3 (Johnson and Naor [JN10])**.** *For every $D, K > 0$, there exists a constant $c = c(K, D) > 0$ with the following property. Let X be a Banach space such that for every $n \in \mathbb{N}$ and every $x_1, \ldots, x_n \in X$, there exists a linear subspace $F \subseteq X$ of dimension at most $K \log n$, and a linear mapping $\varphi : X \to F$ such that $\|x_i - x_j\| \le \|\varphi(x_i) - \varphi(x_j)\| \le D\|x_i - x_j\|$ for all $i, j \in [n]$. Then for every $k \in \mathbb{N}$ and every k-dimensional subspace $E \subseteq X$, we have that the Banach-Mazur distance between X and a Hilbert space satisfies*

$$c_2(E) \le 2^{2^{c \log^*(k)}}.$$

In essence, what this theorem says is that any space that admits good dimension reduction (that is, $k \le \text{polylog}(N)$ dimension) must be very close to ℓ_2, and that in some sense, dimension reduction is an intrinsic property of ℓ_2. (By very close, we mean close in the notion of metric embeddings, which is a very strong notion of closeness.)

What about other geometric properties, such as the angle between vectors? Can we make $\varphi \colon \mathbb{R}^n \to \mathbb{R}^k$ more efficient, e.g. computable in time much less than $k \cdot n$? (Yes.) We will investigate these questions in the homework.

Can we replace the Gaussian with another distribution more specific to ℓ_1 for ℓ_1 dimension reduction? Yes; the equivalent of the Gaussian distribution for ℓ_1 is not the Laplace distribution but rather the Cauchy distribution, with very similar stability properties. However, the tails of the Cauchy are very weak, and we do not get something like $\log n$ dimension at the end of the day.

Johnson-Lindenstrauss is a very powerful tool, and it is used in many areas of algorithms to speed up computations. For example, consider least-squared regression for the overconstrained linear system $Ax = b$ where $A \in \mathbb{R}^{m \times n}$ and $m \gg n$:

$$x = \arg\min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2.$$

For exact solutions, we can solve this problem iin time $O(n^\omega)$, where ω is the matrix multiplication exponent, and this is the best runtime for exact computation. However, we can use Johnson-Lindenstrauss to reduce the runtime with a more efficient dimension-reducing map φ to get an approximate solution in time

$$O\left(\text{nnz}(A) + \log(1/\varepsilon) + \text{poly}(d, \log(1/\varepsilon))\right).$$

Finally, can we get $k \ll \log N$ for ℓ_2 dimension reduction? No; $k = \Theta(\varepsilon^{-2} \log N)$ is optimal, according to a result by Larsen and Nelson [LN17].

§9 Lecture 09—14th February, 2024

Last time we saw the Johnson-Lindenstrauss Lemma, which states that for any set of n points in $\mathbb{R}^d$, there exists a linear map $f : \mathbb{R}^d \to \mathbb{R}^k$ such that for any two points $x, y \in \mathbb{R}^d$, we have

$$(1-\varepsilon)\|x-y\|_2 \leq \|f(x)-f(y)\|_2^2 \leq (1+\varepsilon)\|x-y\|_2$$

with $\varepsilon \in (0,1)$ where $k = O(\varepsilon^{-2}\log n)$ and probability $1-\delta$ for appropriately selected small $\delta \in (0,1)$.

Today, we start with the problem of finding the nearest neighbour of a point in a set of points, which has connections to the Johnson-Lindenstrauss Lemma.

§9.1 Nearest neighbour search

Problem 9.1 (*Nearest Neighbour Search*)**.** Given a data set $D \subset \mathbb{R}^d$, preprocess D so that given a query point $q \in \mathbb{R}^d$, we report $p^* \in D$ such that

$$p^* = \arg\min_{p \in D} \|p-q\|_2.$$

In this course, we take $n = |D|$ and we will work in high dimensions and take $d = \dim(D)$ to be very large, say more than 400. The problem of finding these points that are nearest to the queried point is useful in many applications, such as in databases, computer graphics, and machine learning. In fact, we can find similar points in a database of images, and use this to find similar images via the nearest neighbour search on a set of features of the images.

For our performance metrics for solutions to this problem, we will clear about the query time, the space used, and the preprocessing time. What solutions can we think of for this problem?

1. **Linear scan.** The lazy approach is simply to skip the preprocessing and just scan through the entire data set D to find the nearest neighbour. This requires space $O(nd)$ (good!), query time $O(d)$ (bad, but the algorithm works), and no preprocessing time.
2. **Slashing down the search space.** Here the rough general idea is to simply keep searching through a smaller subset of the data set D within which we are guaranteed to find the nearest neighbour:
   - For $d = 1$, we can use a binary search tree to find the nearest neighbour in $O(\log n)$ time (good!) and $O(n)$ space (good!), and
   - For $d = 2$, we will use a Voronoi diagram to partition the space into cells such that each cell contains a point in D, and we can solve several low-dimensional nearest neighbour problems in each cell.

     Then given a query point, we must find the cell of the Voronoi diagram to which the query point belongs—we do this via the point location data structure—and then we can solve the nearest neighbour problem in that cell.

     The space used via this procedure is proportional to the decision complexity of the Voronoi diagram—i.e. how many lines we need to clearly delineate the cells—and this is in turn at most $O(n \log n)$. Also, we get query time $O(\log n)$, and preprocessing time $O(n \log n)$.

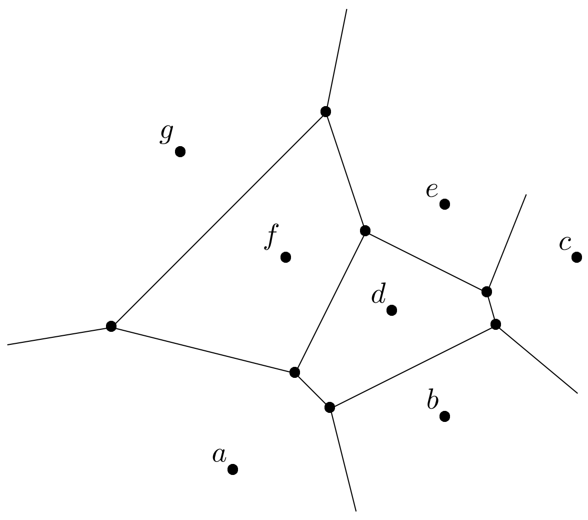


Figure 9.1: An example of a Voronoi diagram.

- For $d > 2$, we can use the same idea of partitioning the space into cells, but the main issue here is that the decision complexity of the Voronoi diagram is $O\left(n^{\lfloor d/2 \rfloor}\right)$, which is bad. Even to write down the diagram, let alone to build a data structure on top of it, will be exponential in the dimension—this is the curse of dimensionality, which we would like to avoid.

All of these are infeasible in some sense, and we'd like to do this much better. The following theorem pours cold water on all our hopes.

Theorem 9.1 (Bringmann [Bri21]). *If we can solve Nearest Neighbour Search with space and preprocessing time $n^{O(1)}$, and query time $n^{1-o(1)}$, then* SETH *is false.*

The strong exponential time hypothesis (SETH) is a conjecture that states that for all $\varepsilon > 0$, there is some $k \geq 3$ such that k-SAT cannot be solved for a Boolean formula with N variables and $N^{k-\varepsilon}$ clauses in time $O\left(2^{(1-\varepsilon)N}\right)$, which basically is a stronger version of $\mathsf{P} \neq \mathsf{NP}$ and says that, say, for a CNF formula on N variables, then there is nothing better to do than enumerating over all possible assignments to the variables and for each assignment, checking if it satisfies the formula.

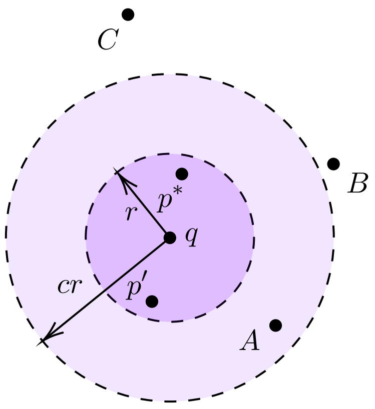
So we have nothing better to do than an approximate nearest neighbour search.

§9.2 Approximate nearest neighbour search

The idea here is that instead of being fixated on the ideal point p^* that is the nearest neighbour of the query point q, we will be happy with a point p that is close enough to q, perhaps within a factor of $1 + \varepsilon$ or 2 or 3, etc.

Problem 9.2 (*Near Neighbour Search*). For a fixed $r > 0$, preprocess the data set $D \subset \mathbb{R}^d$ so that given a query point $q \in \mathbb{R}^d$, we report any $p \in D$ such that $\|q - p\| \leq r$.

Problem 9.3 (*Approximate Near Neighbour Search*)**.** For a fixed $r > 0$ with $c > 1$, preprocess the data set $D \subset \mathbb{R}^d$ so that given a query point $q \in \mathbb{R}^d$, if there exists a point $p^* \in D$ such that $\|q - p^*\| \leq r$, then we report some $p \in D$ such that $\|q - p\| \leq cr$. If there is no such p^*, then we report `fail`.



As shown above, suppose we have a point p such that the distance between q and p^* is less than r, the algorithm should output either p' or point A, but never point B or point C—if both p^* and p' do not exist, then the algorithm may report A or `fail`.

The problem statement is a bit weird; the reason we state it in this way is because for many problems it's easier to solve that problem directly. Indeed, we can solve *Approximate Nearest Neighbour Search* by solving *Approximate Near Neighbour Search*:

Remark 9.4.

1. There is a formal parsimonious reduction from *Approximate Near Neighbour Search* to *Approximate Nearest Neighbour Search*, with very few parameter losses. The idea here is that we need to guess the right $r = \min_{p \in D} \|q - p\|_2$ and then we can solve the *Approximate Near Neighbour Search* problem with r as the parameter.
2. Most algorithms can be modified to output a list $L \subseteq D \subset \mathbb{R}^d$ such that:
   - if $p \in D$ is such that $\|p - q\|_2 \leq r$, then $p \in L$, and
   - if $p \in L$, then $\|p - q\|_2 \leq cr$,

   and so we can think of the algorithms for this problem as methods by which we can filter the data set D to get a smaller set L which we can scan through to filter out the points that are approximate nearest neighbours but not exact neighbours.
3. We will focus on randomised algorithms for this problem, and we will see that the Johnson-Lindenstrauss Lemma will be useful in this context.

§9.2.1 A first algorithm for Approximate Near Neighbour Search

Our goal now is to get any sort of improvement over the linear scan algorithm with space $O(nd)$ and query time $O(nd)$, by Johnson-Lindenstrauss dimension reduction. The idea here is that since we can do dimension reduction, we can have all the points represented by vectors of smaller dimension k, and instead of computing the distance between two points exactly, we will compute the distances approximately using dimension-reduced vectors, with the advantage being that those vectors are shorter, distances are preserved (in ℓ_2), and so we can do the computation faster. We present the algorithm below:

```
Algorithm: ANNS-Solution-1

  • Pick $\varphi\colon \mathbb{R}^d \to \mathbb{R}^k$ at random, where $k = O(\varepsilon^{-2}\log n)$
  • Compute $\varphi(p)$ for $p \in D$
  • At query $q$, compute, for all $p \in D$, the distance $\|\varphi(p) - \varphi(q)\|_2$ and $\varphi(q)$
  • Report $p$ such that $\|\varphi(p) - \varphi(q)\|_2 \le (1+\varepsilon)r$. If no such $p$ exists, report fail.
```

Theorem 9.2. *The algorithm* ANNS-Solution-1 *solves the Approximate Near Neighbour Search problem for approximation $c = 1 + O(\varepsilon)$ with space $O(nd + dk)$ and query time $O(nk + dk)$. In particular, for $d \ll n$, the query time is $O(nd)$ and the space is $O(\varepsilon^{-2} n \log n)$.*

Proof. We use space $O(nd)$ to store the original data set, and space $O(dk)$ to store the map φ. For the time, we use time $O(dk)$ to compute the embedding $\varphi(q)$, and time $O(nk)$ to compute the distances between $\varphi(q)$ and all the points in the data set. The total time is $O(nk + dk)$, and the space is $O(nd + dk)$.

We only need to argue that the algorithm is correct. Remember that for a random $\varphi\colon \mathbb{R}^d \to \mathbb{R}^k$, with $\delta = 1/n^2$, we have that for all $p, q \in \mathbb{R}^d$,

$$(1-\varepsilon)\|p - q\|_2^2 \le \|\varphi(p) - \varphi(q)\|_2^2 \le (1+\varepsilon)\|p - q\|_2^2$$

with probability at least $1 - 1/n^2$ by the Johnson-Lindenstrauss Lemma. Therefore, by the union bound all the distances $\|\varphi(p) - \varphi(q)\|_2$ are preserved up to a factor of $1+\varepsilon$ with high probability $1 - 1/n$, and so if something was within distance r, then the new distance will be within distance $(1+\varepsilon)r$, so such a point p will be reported by the algorithm, and in the worst case, the real distance of any point passing through the threshold is

$$\|p - q\| \le \frac{(1+\varepsilon)r}{1-\varepsilon} \le (1 + O(\varepsilon))r,$$

and so the algorithm is correct. □

Observe that, crucially, we used an oblivious map φ, that is, the embedding φ is independent of the data set D and the query q. This is important because we want to be able to preprocess the data set and then use an out-of-sample extension to apply the map on an unseen query point.

The general principle used in this algorithm are still used in many nearest-neighbour-search type algorithms today, except with the use of a learned embedding (that somehow depends on the data set but can still take out-of-sample query points) as opposed to a random one.

Here's a generalised version of ANNS-Solution-1. So far we have used embeddings that are dimension-reducing maps from $\mathbb{R}^d$ to $\mathbb{R}^k$ in the space ℓ_2, but we cannot do this in the space ℓ_1. Note however that when we build the algorithm, we don't really need the fact that the estimator of the original distance between p and q is an ℓ_2 distance, as we don't need to preserve the geometry itself. Therefore it is okay to have a sketch $\varphi\colon \mathbb{R}^d \to \{0,1\}^k$, and we can use, say, the Hamming distance to compare the sketches of the points.

Theorem 9.3 (Sketching for ℓ_1). *For any $r > 0$, there exists a distribution over $\varphi\colon (\mathbb{R}^d, \ell_1) \to \{0,1\}^k$ such that for any $p, q \in \mathbb{R}^d$ and for some procedure Π and with probability $1-\delta$ taken over the choice of φ,*

- *If $\|p-q\|_1 \le r$, then $\Pi(\varphi(p), \varphi(q)) = 1$, and*
- *If $\|p-q\|_1 > cr$, then $\Pi(\varphi(p), \varphi(q)) = 0$,*

where $k = O(\varepsilon^{-2}\log(1/\delta))$ and $c = 1+\varepsilon$.

So we can take the ANNS-Solution-1 algorithm and replace φ from the Johnson-Lindenstrauss Lemma with the sketching map φ for ℓ_1 and we will get a similar result, and this indeed works for ℓ_p for $p < 2$.

We won't prove this theorem, but to get a sense of how ϕ looks like, we mention the following. If we focus on the Hamming space $\varphi\colon \{0,1\}^d \to \{0,1\}^k$, then we can take a random $k \times d$ matrix A with entries in $\{0,1\}$, and then for any $p \in \{0,1\}^d$, we can take $\varphi(x) = Ap$, where the multiplication is over $\mathbb{F}_2$. This is a very simple and efficient way to get a sketching map for ℓ_1, similar to the Johnson-Lindenstrauss construction. Furthermore, the procedure Π computes $\|\varphi(p) - \varphi(q)\|_p$, and outputs 1 if and only if the distance is less than some threshold θ.

§10 Lecture 10—19th February, 2024

Last time, we saw the c-approximate near neighbour search problem:

Problem 10.1 (*Approximate Near Neighbour Search*). *For a fixed $r > 0$ with $c > 1$, preprocess the data set $D \subset \mathbb{R}^d$ so that given a query point $q \in \mathbb{R}^d$, if there exists a point $p^* \in D$ such that $\|q - p^*\| \le r$, then we report some $p \in D$ such that $\|q - p\| \le cr$. If there is no such p^*, then we report* `fail`.

In fact, via Hamming spaces, we can get a similar result for $\{0,1\}^d$.

§10.1 A second algorithm for Approximate Near Neighbour Search

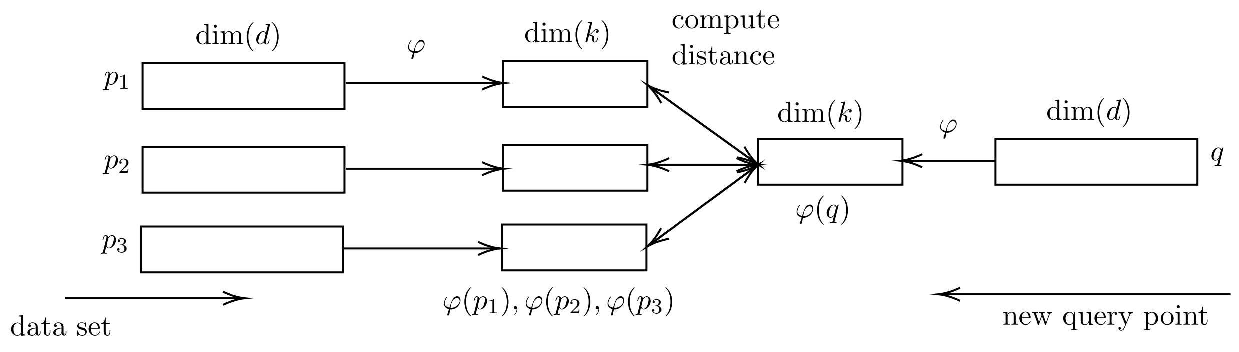
We saw the following theorem last time:

Theorem 10.1 (Kushilevitz, Ostrovsky, Rabani [KOR98]). *For any $\varepsilon > 0$, and for any threshold $r > 0$, there exists a distribution $\varphi\colon \{0,1\}^d \to \{0,1\}^k$ for $k = O\left(\varepsilon^{-2}\log n\right)$ such that for any $p, q \in \{0,1\}^d$,*

- *if $\|p-q\|_1 \le r$, then $\Pr\left[\|\varphi(p) - \varphi(q)\|_1 \le \theta\right] \ge 1 - 1/n^3$,*
- *if $\|p-q\|_1 > (1+\varepsilon)r$, then $\Pr\left[\|\varphi(p) - \varphi(q)\|_1 > \theta\right] \le 1 - 1/n^3$,*

for some threshold $\theta > 0$.

Note that this algorithm implies ANNS-Solution-1 under the assumption that the distance function is the Hamming distance.



We can use this theorem to solve the *Approximate Near Neighbour Search* problem via ANNS-Solution-1 with space just over $O(nd)$ and time $O(dk + nk) = O(n\varepsilon^{-2} \log n)$. We would like to obtain an algorithm with query time $\ll n$, and we will do this via the algorithm ANNS-Solution-2 which uses $n^{O(1/\varepsilon^2)}$ space and has query time $O(d\varepsilon^{-2} \log n)$. The algorithm we have already seen, ANNS-Solution-1, suggests that $\varphi(q)$ is enough to answer the query. Our new algorithm will use exhaustive storage to solve *Approximate Near Neighbour Search*.

Algorithm: ANNS-Solution-2

- Prepare for all the possible $\varphi(q)$. That is, construct a table $T[\sigma]$ for every possible sketch $\sigma \in \{0, 1\}^k$.
- $T[\sigma]$ stores the answer that ANNS-Solution-1 would give for the query $\varphi(q) = \sigma$.
- On query q, we look up $T[\varphi(q)]$ and return the answer.

We analyse this algorithm now. The correctness follows directly from the correctness of ANNS-Solution-1; we are only aiming to amplify its performance. Now, we use space $O(nd)$ to store the original data set, and the table T has $2^k = 2^{O(\varepsilon^{-2} \log n)} = n^{O(1/\varepsilon^2)}$ entries—since the sketch is binary—each of which is a constant-sized answer. The query time is the same as the time it takes to compute $\varphi(q)$, which is $O(dk) = O(d\varepsilon^{-2} \log n)$.

So we have seen that the space is $n^{O(1/\varepsilon^2)}$, and the query time is $O(d\varepsilon^{-2} \log n)$. For query time as low as this, this is the best space known, up to constant factors.

§10.2 Locality-sensitive hashing: an algorithmic technique for ANNS

This is really an approach to designing algorithms due to Indyk and Motwani [IM98]. Locality-sensitive hashing (LSH) is a form of geometric hashing for approximate near neighbour search algorithms to give a solution with better space complexity (in particular, space $O(n^{1+\varrho})$ for $0 < \varrho < 1$), albeit not as good a query time (in particular, $O(n^\varrho)$ for $\varrho = \varrho(c) < 1$). The idea is to specify a hash family for which it is actually good to collide, a slightly different treatment with all our previous encounters with hashing. Once we take a space (perhaps the entire Euclidean space), and decide on a map that discretises the space, we have an induced partition of the space where each partition corresponds to all the points with exactly the same hash code.

Definition 10.2 (Locality-sensitive hash family). *Fix $r > 0$ and some approximation $c = 1 + \varepsilon > 1$. A family $\mathcal{H}$ of hash functions which map $\mathbb{R}^d$ to a universe U is called (r, cr, p_1, p_2)-locality-sensitive* *if for any $h \in \mathcal{H}$, and for any $p, q \in \mathbb{R}^d$,*

- *if $\|p - q\| \le r$, then $\Pr[h(p) = h(q)] \ge p_1$ (i.e. for close points, we want a high probability of collision),*
- *if $\|p - q\| > cr$, then $\Pr[h(p) = h(q)] \le p_2$ (i.e. for far points, we want a low probability of collision),*

where $p_1 > p_2$.

Here we want points that are far away to not collide, but we want points that are close to collide with high probability—this makes sense given our geometric interpretation of LSH.

§10.2.1 The Basic-LSH algorithm

Suppose, for starters, that $p_1 = 1$ and $p_2 = 0$ (this is not possible, but suppose it was the case). How can we design an algorithm for this problem then?

Algorithm: Basic-LSH

- For the preprocessing step, pick $h \in \mathcal{H}$ at random, compute $h(p)$ for $p \in P$, and store the hash function in a dictionary data structure; in particular, given $h(q)$, return all p such that $h(p) = h(q)$.
- Query the dictionary data structure with $h(q)$, enumerate over P such that $h(p) = h(q)$, and return the first p such that $\|p - q\| \le cr$.

Excluding storing the entire data set P and the hash function h, the space complexity is $O(n)$, and the query time is the time taken to compute h, $T(h)$, look up the dictionary, and enumerate the p (which is proportional to the number of far points which happen to collide with q, and the expected number of such points is $n \cdot p_2$), which is $T(h) + O(1) + n \cdot p_2$ in total.

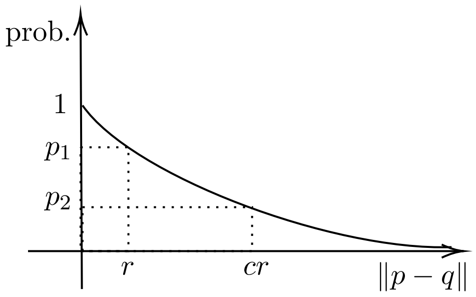
We now prove correctness. We will show that the probability of success is at least p_1. Intuitively, we are successful if there is a point p^* such that $\|p^* - q\| \le r$, so suppose that there is such a point; so we fail only when the entire bucket $h(q)$ has no approximate near neighbour. Then

$$\begin{aligned}\Pr[\text{failure}] &= \Pr[p^* \text{ is not colliding with } q] \\ &\le \Pr[p^* \text{ is not colliding with } q \mid \|p^* - q\| \le r] \\ &\le 1 - p_1,\end{aligned}$$

so that the probability of success is at least p_1.

So we have an algorithm with space that is $O(n)$, query time $O(np_2d + 1)$, and success probability at least p_1, excluding all the details for the hash function. If we have efficient hash functions h (which we describe later), then the space is exactly linear, and if $p_2 = 0$, then the query time is constant.

However, it is not possible to have $p_1 = 1$ and $p_2 = 0$! Consider the space partition of points p, q in a hash function h. If $h(p) \ne h(q)$, then there is a border separating p and q in the space partition. Consider then two points x and y that are neighbours on either side of the space partition. These points are close, but their hash codes are different, so we have a contradiction. So we need to relax our requirements for the hash family to much smoother guarantees.



§10.2.2 The full LSH-FOR-ANNS algorithm

Suppose now that we have more control over p_1 and p_2. There is a nice trick that allows us to take in any LSH with fixed p_1 and p_2 and obtain another LSH with boosted p_1 and reduced p_2.

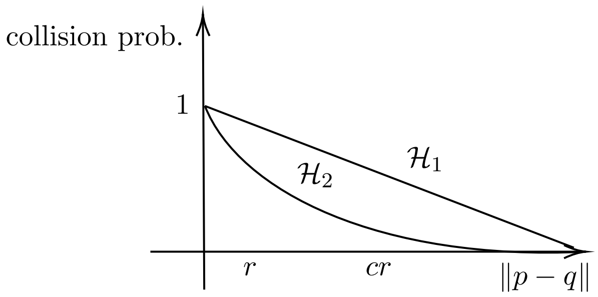
Claim 10.3. *Suppose $\mathcal{H}$ is an (r, cr, p_1, p_2)-locality-sensitive hash family. Fix some integer $k > 1$, and define the k-fold tensorization of the hash family $\mathcal{H}$ as the family $\mathcal{H}^k$ of hash functions g which map $\mathbb{R}^d$ to U^k, where $g(x) \triangleq (h_1(x), h_2(x), \ldots, h_k(x))$, with $h_1, \ldots, h_k \in_r U$. Then $\mathcal{H}^k$ is an $\left(r, cr, p_1^k, p_2^k\right)$-locality-sensitive hash family.*

Proof. Fix close points p, q. Then

$$\Pr_{g\in\mathcal{H}^k}[g(p) = g(q)] = \prod_{i=1}^{k} \Pr_{h_i}[h_i(p) = h_i(q)] \geq \left(\Pr_{h}[h(p) = h(q)]\right)^k \geq p_1^k,$$

and similarly for far points p, q. □

Essentially the idea is that by, say, taking $k = 2$, we can depress the probability of collision for far points, and depress even further the probability of collision for close points:



Corollary 10.4. *Suppose there exists a (r, cr, p_1, p_2)-locality-sensitive hash family $\mathcal{H}$ for some $r > 0$ and $c > 1$. Then* BASIC-LSH *with $\mathcal{H}^k$ for some $k > 1$ has success probability at least p_1^k, space complexity $O(n)$, and query time $O(np_2^k d + 1)$.*

We now present an algorithm that uses this fact. Essentially the idea is to repeat the basic LSH algorithm ℓ times, and return the first approximate near neighbour found with sufficiently high probability (i.e. with failure probability at most 10%).

Algorithm: LSH-FOR-ANNS

- Fix k, and take $\ell = (10/p_1)^k$.
- Build ℓ hash tables, each with a fresh, random $g_i \in G_k$.
- Store all the points $g_i(p)$ in a dictionary data structure.
- On query q and for tables $i = 1, \ldots, \ell$, retrieve points $p \in D$ where $g_i(p) = g_i(q)$, and return the first p such that $\|p - q\| < cr$.

We obtain the following guarantees on this algorithm:

Theorem 10.2. *We can solve c-approximate near neighbour search in* $\mathbb{R}^d$ *with space complexity*

$$O(nd) + O\left(p_1^{-1} n^{1+\varrho} d\right) + \ell k \cdot (\textit{space used to store } h),$$

and expected query time

$$O\left(p_1^{-1} n^{\varrho} \cdot (d + k \cdot (\textit{time taken to compute } h))\right),$$

where

$$\varrho = \frac{\log(1/p_1)}{\log(1/p_2)} \in (0,1).$$

Proof. We will use the full LSH algorithm, where we set k such that $p_2^k \le 1/n$, that is, take $k = \lceil \log(n)/\log(1/p_2) \rceil$. The parameter ℓ can be written as

$$\ell = \frac{10}{p_1^k} = 10 \cdot 2^{k \cdot \log(1/p_1)} = 10 \cdot \left(\frac{1}{p_2}\right)^{k \cdot \frac{\log(1/p_1)}{\log(1/p_2)}} = \left(\frac{10}{p_1}\right) \cdot n^{\varrho},$$

so that the number of hash tables is $O(p_1^{-1} n^{\varrho})$. The space complexity then follows immediately: we use $O(nd)$ to store the data set, and $O(\ell n)$ to store the hash tables, and the total space excluding the hash function is $O(nd) + O\left(p_1^{-1} n^{1+\varrho} d\right)$. The query time is the time taken to compute h via ℓk hash function evaluations plus the time taken for the number of all the far points colliding with q in all ℓ hash functions. First we find that the expected number of far points colliding with q in one hash table is $\le np_2^k$, and so we can expect

$$\mathbb{E}\left[\text{query time}\right] = O(\ell k \cdot T(h)) + O(\ell n p_2^k),$$

where $T(h)$ is the total time required to evaluate the hash function and $k = \lceil \log(n)/\log(1/p_2) \rceil$. Fixing this k, take the $\ell = \left(\frac{10}{p_1}\right) \cdot n^{\varrho}$ from before, so that

$$\begin{aligned}\mathbb{E}\left[\text{query time}\right] &\le O\left(\left(\frac{10}{p_1}\right) \cdot n^{\varrho} \left\lceil \frac{\log(n)}{\log(1/p_2)} \right\rceil \cdot T(h)\right) + O\left(\left(\frac{10}{p_1}\right) \cdot n^{\varrho} \cdot n p_2^{\left\lceil \frac{\log(n)}{\log(1/p_2)} \right\rceil}\right), \\ &\le O\left(p_1^{-1} n^{\varrho} \cdot (d + k \cdot T(h))\right).\end{aligned}$$

It now remains to show that the success probability holds as claimed:

$$\begin{aligned}\Pr\left[\text{success}\right] &\ge 1 - \left(1 - p_1^k\right)^{\ell} \ge 1 - e^{-\ell p_1^k} \\ &\ge 1 - e^{-10} \ge 0.90.\end{aligned}$$

□

§10.2.3 Existence of LSH families

Do these LSH families exist? The answer is yes, and we will see how to construct them. Consider the Hamming spaces $\{0,1\}^d$; we will construct a locally-sensitive hash family for the Hamming space. The idea is to take a random hyperplane and use it to partition the space into two halves, and then use the parity of the number of bits in the input to determine which half the input is in.

In particular, we can select a random hash function $h_i \colon \{0,1\}^d \to \{0,1\}$ by examining a random bit, that is, take $h_i(x) = x_i$ for some $i \in [d]$. Then for fixed p and q, and Hamming distance $\|p - q\| = r$, we have

$$\Pr_{h \in \mathcal{H}} [h_i(p) = h_i(q)] = \frac{d-r}{d} = 1 - \frac{r}{d},$$

so that

$$p_1 \geq 1 - \frac{r}{d} \approx e^{-r/d}$$
$$p_2 \leq 1 - \frac{cr}{d} \approx e^{-cr/d},$$

and

$$\varrho = \frac{\log(1/p_1)}{\log(1/p_2)} \approx \frac{r/d}{cr/d}.$$

The above discussion leads to the following theorem:

Theorem 10.3 (Indyk and Motwani, [IM98]). *Algorithms that can solve Approximate Near Neighbour Search in Hamming space with space $O\left(nd + n^{1+1/c}\right)$, and query time $O\left(n^{1/c}d\right)$ exist.*

Indeed, $\varrho = 1/c$ is a tight LSH for Hamming spaces, and one can obtain $\varrho = 1/c^2$ for Euclidean spaces, which is tight for those spaces as well.

We have discussed a particular variety of LSH algorithms, but is this the best possible for nearest neighbour search? The answer is no; we can do data-dependent versions of LSH, just like we did for perfect hashing—it turns out that we can design an LSH that first looks at the data set, extracts some information, and then partition the data set. For these methods, we can obtain $\varrho = 1/(2c^2 - 1)$ for Euclidean spaces, and $\varrho = 1/(2c - 1)$ for Hamming spaces, and we believe that these are the best obtainable explainable for nearest neighbour search, barring any new breakthroughs.

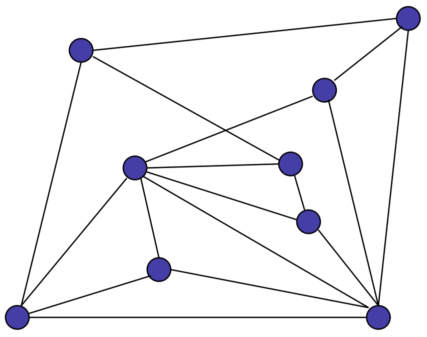
§11 Lecture 11—21th February, 2024

§11.1 Introduction to spectral graph theory

Let us start by recalling some of the things we already know.

Definition 11.1. *A graph $G = (V, E)$ is a pair of sets V and E where $E \subseteq V \times V$. The elements of V are called vertices and the elements of E are called edges.*

We say that $e = (u, v) \in E$ is an edge between u and v. We say that u and v are adjacent if $(u, v) \in E$. We say that u and v are neighbours (or $u \sim v$) if they are adjacent. Here's a graph:



We will (mostly) consider cases where G is finite, has no isolated vertices (i.e. no $v \in V$ such that $\deg(v) = 0$), could have self-loops and parallel edges, is not necessarily connected, and is unweighted. However, from time to time we will consider graphs that do not satisfy some of these properties (this doesn't change things too much; we can deal with typical weights w by using parallel unweighted edges).

In spectral graph theory we are particularly interested in the matricial representations of graphs. One of the most natural ones is the adjacency matrix:

Definition 11.2 (Adjacency matrix)**.** *Let $\mathcal{G}$ be an unweighted graph with vertices $v_1, v_2, \ldots, v_n$. Then the adjacency matrix of $\mathcal{G}$ is the symmetric matrix $A \in \mathrm{Mat}(n \times n; \{0,1\})$, whose (i,j) entry, denoted by $[A]_{i,j}$, is defined by*

$$[A]_{i,j} = \begin{cases} 1 & \textit{if } v_i \textit{ and } v_j \textit{ are adjacent,} \\ 0 & \textit{otherwise.} \end{cases}$$

This is merely a spreadsheet of the graph. It is quite natural, but we will see that it is not the most useful representation of a graph. Here's a more useful one:

Definition 11.3 (Laplacian matrix)**.** *Define the degree matrix $D \in \mathrm{Mat}(n \times n; \mathbb{R})$ as*

$$D_{i,j} = \begin{cases} \deg(v_i), & \textit{if } i = j \\ 0, & \textit{otherwise.} \end{cases}$$

The Laplacian matrix is then defined as $L = D - A$.

§11.2 Random walks and diffusion on graphs

Now consder a walker walking randomly on the graph G; it starts from some node i, and at each step moves to a neighbour of the current node, chosen uniformly at random. We can describe the walker's position at time t by a probability distribution $x^{(t)} \in \mathbb{R}^n$, where $x^{(t)}(i)$ is the probability that the walker is at node i at time t.

Starting from the initial position $x^{(0)} = (1, 0, \ldots, 0)$, the evolution of $x^{(t)}$ is given by

$$x_i^{(t+1)} = \sum_{j \sim i} \frac{1}{\deg(i)} x_j^{(t)} \implies x^{(t+1)} = \left(A_G \cdot D_G^{-1}\right) \cdot x^{(t)},$$

where $W_G \triangleq A_G \cdot D_G^{-1}$ is the diffusion operator A_G and D_G are the adjacency and degree matrices of G, respectively.

Note that $x^{(t)} = (W_G)^t \cdot x^{(0)}$. We can also write $W_G = I - D_G^{-1} \cdot L_G$, where I is the identity matrix. This is the continuous-time analogue of the discrete-time random walk. Furthermore, in the evolution process $x^{(t+1)} = A_G \cdot D_G^{-1} \cdot x^{(t)}$, we can see that the vector $D_G^{-1} \cdot x^{(t)}$ is a special vector indicating the level of contribution from each node i to the next step of the random walk.

Example 11.4. Consider the line graph L_5 on five vertices. The adjacency and degree matrices of L_5 are

$$A_{L_5} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad D_{L_5} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Suppose the walker starts at node 2. Then the probability distribution of the walker at time t is

$$x^{(1)} = \frac{1}{2}\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad x^{(2)} = \frac{1}{4}\begin{bmatrix} 0 \\ 3 \\ 0 \\ 1 \\ 0 \end{bmatrix}.$$

A useful intuition to have is that the diffusion operator W_G is a linear operator that smooths out the initial distribution $x^{(0)}$ over the graph; say our graph is a conductive material, then W_G is the operator that describes the diffusion of heat over the material until it reaches equilibrium. This notion of an equilibrium motivates the following definition:

Definition 11.5 (Stationary distribution)**.** *A probability distribution $x^* \in \mathbb{R}^n$ is called a* stationary distribution *of the diffusion operator W_G if $W_G \cdot x^* = x^*$.*

From linear algebra, we know that the stationary distribution of a linear operator is the eigenvector associated with the eigenvalue 1, and so we will want to use more linear algebra as we proceed.

Example 11.6. The distribution $x^* = \frac{1}{8}\begin{bmatrix} 1 & 2 & 2 & 2 & 1 \end{bmatrix}^\top$ is a stationary distribution of the diffusion operator W_{L_5}.

Some of the questions one may ask are the following: Does x^* always exist? If existence is guaranteed, is it always unique? When $t \to \infty$, does $x^{(t)}$ always converge to x^*? If so, how fast does this convergence happen? We will answer some of these questions using the spectral decomposition of the diffusion operator W_G.

§11.3 Theory of symmetric matrices: the spectral decomposition

We start by formally defining the eigenvalues and eigenvectors of a matrix.

Definition 11.7 (Eigenvalues and eigenvectors)**.** *Let $A \in \mathrm{Mat}(n \times n; \mathbb{R})$. A scalar $\lambda \in \mathbb{R}$ is called an* eigenvalue *of A if there exists a non-zero vector $v \in \mathbb{R}^n$ such that $Av = \lambda v$. The vector v is called an* eigenvector *of A associated with the eigenvalue λ. The set of all eigenvalues of A is called the* spectrum *of A.*

The following fact should also be very familiar:

Fact 11.8. *Let $A \in \mathrm{Mat}(n \times n; \mathbb{R})$. Then the following are equivalent:*

1. *λ is an eigenvalue of A.*
2. $\det(A - \lambda I) = 0$.
3. *λ is a root of the degree n characteristic polynomial of A.*

Another fact we should know is that the determinant is defined by:

$$\det(A) = \sum_{\sigma \in S_n} \mathrm{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)},$$

where S_n is the symmetric group on n elements, and $\mathrm{sgn}(\sigma)$ is the sign of the permutation σ. Furthermore, for a symmetric matrix M, all its n roots as well as its spectrum are always real.

We now state the spectral theorem.

Theorem 11.1 (Spectral Theorem)**.** *If $M \in \mathrm{Mat}(n \times n; \mathbb{R})$ is an n-by-n, real, symmetric matrix, then there exist real numbers $\lambda_1, \dots, \lambda_n$ and n mutually orthogonal unit vectors $\nu_1, \dots, \nu_n$ such that ν_i is an eigenvector of M of eigenvalue λ_i, for each i. That is, for each tuple (i, j) such that $i \neq j$, we have $\nu_i \cdot \nu_j = 0$ and $\|\nu_i\| = 1$ (and hence the v_i form an orthonormal basis), as well as $M \cdot \nu_i = \lambda_i \nu_i$.*

We do not prove this theorem here, but we will make the following remarks.

1. The ordering $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ is unique.
2. The eigenvectors $\nu_1, \dots, \nu_n$ are not necessarily unique (especially up to sign).

   Example 11.9. Suppose we have for some $i \in \mathbb{N}$, the eigenvalue $\lambda_i = \lambda_{i+1}$. Then the orthogonal eigenvalue-eigenvector pair is $(\lambda_i, \nu_i), (\lambda_i, \nu_{i+1})$, but so is the pair

   $$\left(\lambda_i, \frac{\nu_i + \nu_{i+1}}{\|\nu_i + \nu_{i+1}\|}\right), \left(\lambda_i, \frac{\nu_i - \nu_{i+1}}{\|\nu_i - \nu_{i+1}\|}\right).$$

   However, if the eigenvalues $\lambda_1 > \lambda_2 > \ldots > \lambda_n$, then the eigenvectors are unique up to sign.
3. M can be expressed as the sum of outer products of eigenvectors and eigenvalues:

   $$M = \sum_{i=1}^{n} \lambda_i \nu_i \nu_i^\top,$$

   or in a more familar form, we can write $M = V \Lambda V^\top$, where $V = [\nu_1 \ \dots \ \nu_n]$ and $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$.

Observe that $V^{-1} = V^\top$, and so $M = V\Lambda V^{-1}$.

Example 11.10. If $M = \mathbf{0}$, then $\lambda_1 = \lambda_2 = \ldots = \lambda_n = 0$, and any basis of $\mathbb{R}^n$ is a set of eigenvectors of M.
If $M = I$, then $\lambda_1 = \lambda_2 = \ldots = \lambda_n = 1$, and again any orthonormal basis of $\mathbb{R}^n$ is a set of eigenvectors of M.

Since we know that n eigenvectors form a basis of $\mathbb{R}^n$, we can write any vector $x \in \mathbb{R}^n$ as a linear combination of the eigenvectors of M:

$$x = \sum_{i=1}^n \alpha_i \nu_i,$$

and so the result of applying M to x is the summation of scaled eigenvectors:

$$Mx = \left(\sum_{i=1}^n \lambda_i u_i \nu_i \right) \left(\sum_{j=1}^n \alpha_j \nu_j \right) = \sum_{i=1}^n \lambda_i \alpha_i \nu_i.$$

The fact that this matrix vector product is merely a scaling of the eigenvectors by the eigenvalues leads to the next section.

§11.3.1 The Rayleigh quotient

Intuitively, the Rayleigh quotient is a measure of how much a vector x is aligned with an eigenvector ν_i of a matrix M. It is defined as follows:

Definition 11.11. *Let $M \in \mathrm{Mat}(n \times n; \mathbb{R})$ be a real, symmetric matrix, and let $x \in \mathbb{R}^n$ be a non-zero vector. The* Rayleigh quotient *of x with respect to M is defined as*

$$R_M(x) = \frac{x^\top M x}{x^\top x}.$$

Observe that for an eigenvector ν_i of M with eigenvalue λ_i, we have

$$R_M(\nu_i) = \frac{\nu_i^\top M \nu_i}{\nu_i^\top \nu_i} = \frac{\lambda_i \nu_i^\top \nu_i}{\nu_i^\top \nu_i} = \lambda_i.$$

The following theorem due to Courant and Fischer helps us use the Rayleigh quotient to characterise the eigenvalues of a matrix.

Theorem 11.2 (Courant-Fischer). *Let $M \in \mathrm{Mat}(n \times n; \mathbb{C})$ be Hermitian. Then for each $1 \le k \le n$, let $\{S_k^\alpha\}_{\alpha \in I_k}$ be the set of all k-dimensional linear subspaces of $\mathbb{C}^n$. Also, enumerate the n eigenvalues $\lambda_1, \ldots, \lambda_n$ of M (counting multiplicities) in increasing order, i.e. $\lambda_1 \le \cdots \le \lambda_n$. Then we have*

$$\lambda_k \overset{(i)}{=} \min_{\alpha \in I_k} \max_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top M x}{x^\top x} \overset{(ii)}{=} \max_{\alpha \in I_{n-k+1}} \min_{x \in S_{n-k+1}^\alpha \setminus \{0\}} \frac{x^\top M x}{x^\top x},$$

where λ_k is the kth largest eigenvalue of M and S, T are subspaces of $\mathbb{C}^n$.

The above is a more general form of the Courant-Fischer theorem; we will use the following simpler version.

Theorem 11.3 (Courant-Fischer)**.** *Let $M \in \mathrm{Mat}(n \times n; \mathbb{R})$ be a real, symmetric matrix, and let $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ be its eigenvalues. Then for each $i \in \{1, 2, \ldots, n\}$, we have*

$$\lambda_i = \min_{\substack{x \in \mathbb{R}^n \\ x \neq \mathbf{0}}} R_M(x) \quad \text{and} \quad \lambda_i = \max_{\substack{V \subseteq \mathbb{R}^n \\ \dim(V) = i}} \min_{\substack{x \in V \\ x \neq \mathbf{0}}} R_M(x).$$

We can use this theorem for separate i:

> **Example 11.12.** The Fiedler value of a graph G is the second smallest eigenvalue of its Laplacian matrix. It is characterised by the Rayleigh quotient as
>
> $$\lambda_2 = \min_{\substack{x \in \mathbb{R}^n \\ x \neq \mathbf{0}}} R_L(x) \quad \text{and} \quad \lambda_2 = \max_{\substack{V \subseteq \mathbb{R}^n \\ \dim(V) = 2}} \min_{\substack{x \in V \\ x \neq \mathbf{0}}} R_L(x).$$

We will now prove the harder version of Theorem 11.3. (This was not covered in lecture, but the proof covered in lecture is popular (show that the maximum is both at least ad at most the eigenvalue), and so I chose to cover the harder version here.)

Proof of Theorem 11.2. Denote an orthonormal basis for the eigenvectors as $u_1, \ldots, u_n$ corresponding to eigenvalues $\lambda_1, \ldots, \lambda_n$. We will first prove equation (i), and then (ii):

(i) First let $W = \mathsf{span}\{u_1, \ldots, u_n\}$, so that $\dim(W) = n - k + 1$. Thus for any k-dimensional subspace S_k^α, we should have $\dim(S_k^\alpha \cap W) \geq 1$; this is because $\dim(S_k^\alpha + W) = \dim(S_k^\alpha) + \dim(W) - \dim(S_k^\alpha \cap W)$, and of course $\dim(S_k^\alpha + W) \leq n$.
Now choose $x \in (S_k^\alpha \cap W) \setminus \{0\}$, and note that $x = \sum_{j=k}^n \langle x, u_j \rangle u_j$ and $Mu_j = u_j$, so that

$$\begin{aligned} \frac{x^\top M x}{x^\top x} = \frac{\langle Mx, x \rangle}{\|x\|^2} &= \frac{\langle \sum_{j=k}^n \lambda_j \langle x, u_j \rangle u_j, x \rangle}{\|x\|^2} \\ &= \frac{\sum_{j=k}^n \lambda_j |\langle x, u_j \rangle|^2}{\|x\|^2} \geq \lambda_k \frac{\sum_{j=k}^n \lambda_j |\langle x, u_j \rangle|^2}{\|x\|^2} \\ &= \lambda_k. \end{aligned}$$

Note that we have used the fact that $\lambda_k \leq \lambda_{k+1} \leq \ldots \leq \lambda_n$, and $\|x\|^2 = \sum_{j=k}^n |\langle x, u_j \rangle|^2$.
Thus for any S_k^α,

$$\sup_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top M x}{x^\top x} \geq \lambda_k.$$

Indeed for any $x \in S_k^\alpha$, we have

$$\sup_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top M x}{x^\top x} = \sup_{x \in S_k^\alpha, \|x\| = 1} x^\top M x,$$

and since $\{x \in S_k^\alpha : \|x\| = 1\}$ is compact, the supremum is attained at some $x \in S_k^\alpha$ with $\|x\| = 1$. Thus we have

$$\max_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top M x}{x^\top x} = \sup_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top M x}{x^\top x} \geq \lambda_k.$$

On the other hand, consider a particular k-dimensional subspace $S_k^\alpha = \mathsf{span}(u_1, \dots, u_k)$, so that

$$\frac{x^\top Mx}{x^\top x} = \frac{\sum_{j=k}^n |\langle x, u_j\rangle|^2}{\|x\|^2} \le \lambda_k.$$

Choosing $x = u_k$, we see that the inequality is tight, so that we have

$$\max_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top Mx}{x^\top x} = \lambda_k.$$

This also implies that the minimum of $\max_{x \in S_k^\alpha \setminus \{0\}} \dfrac{x^\top Mx}{x^\top x}$ over all the $\alpha \in I_k$ is also attained, and we conclude that

$$\lambda_k = \min_{\alpha \in I_k} \max_{x \in S_k^\alpha \setminus \{0\}} \frac{x^\top Mx}{x^\top x}$$

.

(ii) The proof of (ii) is similar to that of (i)—we will omit similar details. Choose again $W = \mathsf{span}\{u_1, \dots, u_k\}$, so that $\dim(W) = k$. Then for any subspace S_{n-k+1}^α, we should have $\dim(S_k^\alpha \cap W) \ge 1$.

Next, choose any $x \in (S_{n-k+1}^\alpha \cap W) \setminus \{0\}$, such that $\dfrac{x^\top Mx}{x^\top x} \le \lambda_k$, and therefore

$$\min_{x \in S_{n-k+1}^\alpha \setminus \{0\}} \frac{x^\top Mx}{x^\top x} \le \lambda_k.$$

We again choose a particular $S_{n-k+1}^\alpha = \mathsf{span}(u_{k+1}, \dots, u_n)$, so that

$$\min_{x \in S_{n-k+1}^\alpha \setminus \{0\}} \frac{x^\top Mx}{x^\top x} = \lambda_k \implies \max_{\alpha \in I_{n-k+1}} \min_{x \in S_{n-k+1}^\alpha \setminus \{0\}} \frac{x^\top Mx}{x^\top x} = \lambda_k,$$

and we are done. □

§11.4 Random walks in a graph G

We now apply these concepts to analyse random walks on graphs. Let A be an adjacency matrix of a graph G on n vertices, and let D be the degree matrix of G. Let $X^{(0)}$ be a starting distribution over the n vertices of G, and let $X^{(t)}$ be the distribution over the vertices at time t. Then we have that $X^t = \left(A \cdot D^{-1}\right)^t \cdot X^{(0)}$. We will use the spectral theorem to characterize $X^{(t)}$—which is not necessarily symmetric—as $t \to \infty$, which is a problem because Theorem 11.1 only holds for real symmetric matrices. With the symmetric matrix $\hat{A} = D^{-1/2} \cdot A \cdot D^{-1/2}$ however, we can write

$$\begin{aligned}
X^{(t)} &= \underbrace{\left(A \cdot D^{-1}\right) \cdot \left(A \cdot D^{-1}\right) \cdots \left(A \cdot D^{-1}\right)}_{t \text{ times}} \cdot X^{(0)} \\
&= \underbrace{\left(A \cdot D^{-1/2} \cdot D^{-1/2}\right) \cdot \left(A \cdot D^{-1/2} \cdot D^{-1/2}\right) \cdots \left(A \cdot D^{-1/2} \cdot D^{-1/2}\right)}_{t \text{ times}} \cdot X^{(0)} \\
&= \left(D^{-1/2} \cdot A \cdot D^{-1/2}\right)^t \cdot X^{(0)} \\
&= D^{-1/2} \cdot \hat{A}^t \cdot D^{1/2} \cdot X^{(0)}.
\end{aligned}$$

Then with $Y^{(t)} = D^{-1/2} \cdot X^{(t)}$, we have $Y^{(t)} = \hat{A}^t \cdot Y^{(0)}$. We can now use the spectral theorem to write

$$\hat{A} = V \Lambda V^\top,$$

where $V = [\nu_1 \ \ldots \ \nu_n]$ and $\Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$ with $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$.

Fact 11.13. *1. The symmetric matrix $\hat{A} = \sum_{i=1}^n \lambda_i \nu_i \nu_i^\top$ for orthogonal unit vectors $\nu_1, \ldots, \nu_n$.*

2. The matrix $Y^{(0)} = \sum_{i=1}^n \alpha_i \nu_i$, where the $\alpha_i \in \mathbb{R}$ and are uniquely determined, and

3. In particular, $Y^{(t)} = \hat{A}^t \cdot Y^{(0)} = \sum_{i=1}^n \alpha_i \lambda_i^t \nu_i$.

Proof. We only need to prove the last fact; we will use an inductive argument to do this. Clearly for $t = 1$, we have

$$Y^{(1)} = \hat{A} \cdot Y^{(0)} = \sum_{i=1}^n \alpha_i \lambda_i \nu_i = \sum_{i=1}^n \alpha_i \lambda_i^1 \nu_i.$$

Now suppose that $Y^{(t)} = \sum_{i=1}^n \alpha_i \lambda_i^t \nu_i$. Then

$$Y^{(t+1)} = \hat{A} \cdot Y^{(t)} = \left(\sum_{i=1}^n \lambda_i \nu_i \nu_i^\top \right) \cdot \left(\sum_{i=1}^n \alpha_i \lambda_i^t \nu_i \right) = \sum_{i=1}^n \alpha_i \lambda_i^{t+1} \nu_i,$$

and we are done. □

§12 Lecture 12—26th February, 2024

§12.1 More spectral graph theory

Recall our story from last time: we have a possibly weighted undirected graph G, and we build up its adjacency matrix as $A_G = \{a_{ij}\}$, where a_{ij} is the weight of the edge between i and j if it exists, and 0 otherwise. We also have a vector d of length n where d_i is the degree of vertex i. From these we can define the Laplacian matrix $L_G = D_G - A_G$, where D_G is the diagonal matrix with d_i on the diagonal. We also have the normalised Laplacian $L_G^N = D_G^{-1/2} L_G D_G^{-1/2}$.

Indeed we have a spectral decomposition of the adjacency matrix $A_G = U \Lambda U^\top$, where U is the orthonormal matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues. We also saw the Rayleigh quotients, which are the eigenvalues of the normalised adjacency matrix. More specifically, we have

$$R_M(x) = \frac{x^\top M x}{x^\top x} = \frac{\sum_{i,j} a_{ij} x_i x_j}{\sum_i x_i^2}.$$

We also saw that the Rayleigh quotient is minimized by the eigenvector corresponding to the smallest eigenvalue of M, by means of the Courant-Fischer optimisation correspondence:

$$\lambda_1(M) = \min_{x \neq 0} R_M(x) = \min_{x \neq 0} \frac{x^\top M x}{x^\top x}, \quad \lambda_n(M) = \max_{x \neq 0} R_M(x) = \max_{x \neq 0} \frac{x^\top M x}{x^\top x}.$$

We ended last time with the fact that a matrix raised to an integer power interacts very nicely with its eigendecomposition:

Fact 12.1. *1. The symmetric matrix $\hat{A} = \sum_{i=1}^n \lambda_i \nu_i \nu_i^\top$ for orthogonal unit vectors $\nu_1, \ldots, \nu_n$.*

2. The matrix $Y^{(0)} = \sum_{i=1}^n \alpha_i \nu_i$, where the $\alpha_i \in \mathbb{R}$ and are uniquely determined, and

3. In particular, $Y^{(t)} = \hat{A}^t \cdot Y^{(0)} = \sum_{i=1}^n \alpha_i \lambda_i^t \nu_i$.

Indeed, this fact lends itself to the following corollary:

Corollary 12.2. *1. If $\lambda_1 > 1$, then we must have that as the number of time steps $t \to \infty$, then the norm $\|Y^{(t)}\| \to \infty$. Similarly, if $\lambda_1 < 1$, then $\|Y^{(t)}\| \to \infty$.*

2. If $\lambda_i \in (-1, 1)$, then as $t \to \infty$, the norm $\|Y^{(t)}\| \to 0$.

Proof. Immediate. □

So it must either be that $\lambda_1 = 1$ or $\lambda_n = -1$, as well as the following theorem:

Theorem 12.1. *Let $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$ be the eigenvalues of $\hat{A}$ choosen according to the spectral theorem. Then $\lambda_1 = 1$.*

Proof. All of our proofs will follow from the Rayleigh quotient. First we show that $\lambda_1 \ge 1$. Recall that

$$\lambda_1 = \max_{x \neq 0} R_{\hat{A}}(x) = \max_{x \neq 0} \frac{x^\top \hat{A} x}{x^\top x}.$$

Therefore, to show that $\lambda_1 \ge 1$, it suffices to find some vector x such that $x \neq 0$ and $R_{\hat{A}}(x) \ge 1$. We claim that

$$x = \left(\sqrt{d_1}, \sqrt{d_2}, \ldots, \sqrt{d_n}\right)^\top = D_G^{1/2} \cdot \mathbf{1},$$

where $\mathbf{1}$ is the vector of all ones, will do the trick and in particular give us $R_{\hat{A}}(x) = 1$. Indeed, $A \cdot \mathbf{1}$ is a vector of all the row sums of A corresponding to the degrees of the vertices, and so we can use this fact in the Rayleigh quotient to get

$$\begin{aligned}
R_{\hat{A}}(x) &= \frac{x^\top \hat{A} x}{x^\top x} = \frac{\mathbf{1}^\top D_G^{1/2} \hat{A} D_G^{1/2} \mathbf{1}}{\mathbf{1}^\top D_G \mathbf{1}} \\
&= \frac{\mathbf{1}^\top D_G^{1/2} D_G^{-1/2} D_G A_G D_G^{-1/2} D_G^{1/2} \mathbf{1}}{\mathbf{1}^\top D_G \mathbf{1}} = \frac{\mathbf{1}^\top D_G^{1/2} D_G^{-1/2} L_G D_G^{-1/2} D_G^{1/2} \mathbf{1}}{\sum_{i=1}^n x_i^2} \\
&= \frac{\mathbf{1}^\top A_G \mathbf{1}}{\sum_{i=1}^n x_i^2} = \frac{\mathbf{1}^\top (d_1, \ldots, d_n)^\top}{\sum_{i=1}^n x_i^2} \\
&= \frac{\sum_{i=1}^n d_i}{\sum_{i=1}^n d_i} = 1.
\end{aligned}$$

Therefore, $\lambda_1 \ge 1$. Now we show the harder direction, that $\lambda_1 \le 1$ by showing that for all $x \neq 0$, $R(x) \le 1$. We have

$$\begin{aligned}
R_{\hat{A}}(x) &= \frac{x^\top \hat{A} x}{x^\top x} = \frac{x^\top D^{-1/2} A D^{1/2} x}{x^\top x} = \frac{\left(\frac{x_1}{\sqrt{d_1}}, \ldots, \frac{x_n}{\sqrt{d_n}}\right) A \left(\frac{x_1}{\sqrt{d_1}}, \ldots, \frac{x_n}{\sqrt{d_n}}\right)^\top}{\|x\|_2^2} \\
&= \frac{\sum_{1 \le i,j \le n} a_{ij} \cdot \frac{x_i}{\sqrt{d_i}} \cdot \frac{x_j}{\sqrt{d_j}}}{\sum_{i=1}^n x_i^2} = \frac{\sum_{(i,j) \in E} \frac{x_i}{\sqrt{d_i}} \cdot \frac{x_j}{\sqrt{d_j}}}{\sum_{i=1}^n x_i^2}.
\end{aligned}$$

Recalling Cauchy-Schwarz (for all $p, q \in \mathbb{R}^n$, we have $p^\top q \le \|p\|_2 \|q\|_2$), we can define $p, q \in \mathbb{R}^{2|E|}$ such that for all $(i,j) \in E$ where $a_{ij} \neq 0$, we have $p_{(i,j)} = \frac{x_i}{\sqrt{d_i}}$ and $q_{(i,j)} = \frac{x_j}{\sqrt{d_j}}$. Then we have

$$\begin{aligned}
R_{\hat{A}}(x) &= \frac{\sum_{(i,j)\in E} \frac{x_i}{\sqrt{d_i}} \cdot \frac{x_j}{\sqrt{d_j}}}{\sum_{i=1}^n x_i^2} \\
&\le \frac{\left(\sum_{(i,j)\in E} \left(\frac{x_i}{\sqrt{d_i}}\right)^2\right)^{1/2} \left(\sum_{(i,j)\in E} \left(\frac{x_j}{\sqrt{d_j}}\right)^2\right)^{1/2}}{\sum_{i=1}^n x_i^2} \\
&= \frac{\left(\sum_{i=1}^n \frac{x_i^2}{d_i}\right)^{1/2} \left(\sum_{i=1}^n \frac{x_i^2}{d_i}\right)^{1/2}}{\sum_{i=1}^n x_i^2} = \frac{\sum_{(i,j)\in E} \frac{x_i^2}{d_i}}{\sum_{i=1}^n x_i^2} = \frac{\sum_{i=1}^n x_i^2}{\sum_{i=1}^n x_i^2} = 1.
\end{aligned}$$

Therefore, $R_{\hat{A}}(x) \le 1$ for all $x \neq 0$, and so $\lambda_1 \le 1$. Putting things together, $\lambda_1 = 1$. □

Based on this proof, we have the following remark and corollary:

Remark 12.3. If $\lambda_1 = 1$ is unique (that is, $\lambda_2 < 1$), then its corresponding eigenvector is $\nu_1 = (\sqrt{d_1}, \ldots, \sqrt{d_n})^\top$.

Corollary 12.4. *Fix a nonzero vector x such that $R(x) = 1$. Then for any edge (i,j), we have $\frac{x_i}{\sqrt{d_i}} = \frac{x_j}{\sqrt{d_j}}$. Furthermore, $R(x) \ge -1$ for all x, with equality if and only if $\frac{x_i}{\sqrt{d_i}} = -\frac{x_j}{\sqrt{d_j}}$ for all $(i,j) \in E$.*

Proof. Consider the $\lambda_1 \le 1$ part of the proof of Theorem 12.1 above in the case that the x is such that $R(x) = 1$. The resolution we follow is due to Cauchy-Schwarz, which we must then replace with the equality case of Cauchy-Schwarz, which is that for all $p, q \in \mathbb{R}^n$, we have $p^\top q = \|p\|_2 \|q\|_2$, so that $R(x) = 1$ implies $p = \alpha q$ for some $\alpha \in \mathbb{R}_+$. By the definition of p and q, we have that for all $(i,j) \in E$,

$$\frac{x_i}{\sqrt{d_i}} = \alpha \frac{x_j}{\sqrt{d_j}}, \qquad \frac{x_j}{\sqrt{d_j}} = \alpha \frac{x_i}{\sqrt{d_i}},$$

by symmetry. Dividing one equation by another gives the result. The proof for the $R(x) \ge -1$ part is similar. □

We mentioned before that the core idea of spectral graph theory is to form interrelations between the algebraic aspects of the graph and its combinatorial properties. The following theorem is a classic instance of this:

Theorem 12.2. $\lambda_2 = 1$ *if and only if the graph G is disconnected.*

Proof. We first show the reverse direction: that if G is disconnected, then $\lambda_2 = 1$. Suppose that G is disconnected. We can think of G as a collection of two non-empty graphs G_1 and G_2 with no paths between them. Reordering the rows and columns of A_G to list the vertices of G_1 first and then the vertices of G_2, we have

$$\hat{A}_G = \begin{pmatrix} \hat{A}_{G_1} & 0 \\ 0 & \hat{A}_{G_2} \end{pmatrix},$$

where $\hat{A}_{G_1}$ and $\hat{A}_{G_2}$ are the normalised adjacency matrices of G_1 and G_2 respectively. Suppose now that G_1 has k vertices, and let the tuples $\{(\lambda_i, \nu_i)\}_{i=1}^n$ be the eigenpairs of $\hat{A}_{G_1}$, where each $\nu_i \in \mathbb{R}^k$. Furthermore, let $\{(\lambda_{i+k}, \nu_{i+k})\}_{i=1}^n$ be the eigenpairs of $\hat{A}_{G_2}$, where each $\nu_{i+k} \in \mathbb{R}^{n-k}$. Then we have that the eigenpairs of $\hat{A}_G$ are $\{(\lambda_i, \nu_i')\}_{i=1}^n$, where

$$\nu_i' = \begin{cases} (\nu_i, 0, \dots, 0) & \text{if } i \leq k, \\ (0, \dots, 0, \nu_i) & \text{if } i > k. \end{cases}$$

By Theorem 12.1, the first eigenvalue of both $\hat{A}_{G_1}$ and $\hat{A}_{G_2}$ is 1; in particular, $\lambda_1 = 1$ with $\nu_1' = (\nu_1, 0, \dots, 0)$ and $\lambda_{k+1} = 1$ with $\nu_{k+1}' = (0, \dots, 0, \nu_1)$ are the eigenpairs of $\hat{A}_G$. Therefore, $\lambda_2 = 1$ since ν_1' and ν_{k+1}' are orthogonal.

We show the other dierection by contradiction. Suppose that G is connected, and that $\lambda_2 = 1$. Let ν_2 be an eigenvector with eigenvalue λ_2. We already know that $\nu_2 \perp \nu_1$, and so $R(\nu_2)\lambda_2 = 1$. By Corollary 12.4, we have that for every edge (i, j), we have

$$\frac{x_1}{\sqrt{d_i}} = \frac{x_j}{\sqrt{d_j}}.$$

Now set $\beta = \frac{x_i}{\sqrt{d_i}}$ for all $i \in V$. Then since G is connected, there is a path from vertex 1 to every other vertex i, and the chain of equalities along each edge of this path gives us that for all vertices i, $\frac{x_i}{\sqrt{d_i}} = \frac{x_1}{\sqrt{d_1}} = \beta$. Therefore, $x_i = \beta\sqrt{d_i}$ for all $i \in V$, and so $x = \nu_1 \cdot \beta$, since $\nu_1 = (\sqrt{d_1}, \dots, \sqrt{d_n})^\top$. But then we have a contradiction, since $\nu_1 \perp \nu_2$. $\square$

There are other nice connections between the eigenvalues of the normalised adjacency matrix and the combinatorial properties of the graph. For example, the multiplicity of $\lambda_1 = 1$ is the number of connected components of the graph, and if $\lambda_n = -1$, then the graph is bipartite (we will see these and more in the homework assignment). Next time we will see how to make some of these connections tighter.

§13 Lecture 13—28th February, 2024

§13.1 The graph Laplacian

As usual, let $G = (V, E)$ be an graph with n nodes and m edges, and let A and D be its adjacency and degree matrices respectively.

Definition 13.1 (Graph Laplacian)**.** *The* graph Laplacian *of G is defined as $L := D - A$.*

Of course,

$$L_{ij} = \begin{cases} \deg(\nu_i) & \text{if } i = j, \\ -1 & \text{if } i \neq j \text{ and } \nu_i \sim \nu_j, \\ 0 & \text{otherwise.} \end{cases}$$

Naturally this suggests that it might be useful to decompose the Laplacian L as a sum of m matrices L_e corresponding to different edges $e \in E$. For any edge $e = (i, j) \in E$, we define L_e as an $n \times n$

matrix with all zero entries except $(L_e)_{ii} = (L_e)_{jj} = 1$ and $(L_e)_{ij} = (L_e)_{ji} = -1$. Then we have $L = \sum_{e \in E} L_e$. This decomposition lends the following nice representation of the quadratic form of the Laplacian:

Fact 13.2. *For any vector $x \in \mathbb{R}^n$, we have $x^\top L x = \sum_{e \in E}(x_i - x_j)^2$.*

Proof. Via the sum decomposition above, $L = \sum_{e \in E} L_e$, and so for $e = (i, j) \in E$, we have

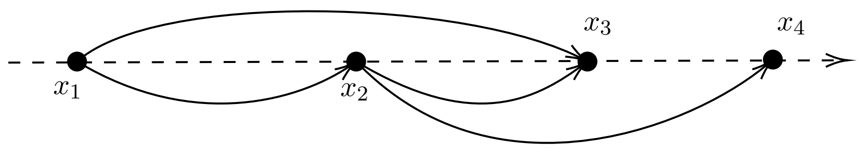
$$\begin{aligned} x^\top L x = \sum_{e \in E} x^\top L_e x &= \sum_{e \in E} \left(x_i^2 + x_j^2 - x_j x_i - x_i x_j\right) \\ &= \sum_{e \in E} (x_i - x_j)^2. \end{aligned}$$ □

The notion of the Laplacian comes directly from physics. To gain some intuition for its quadratic form, consider a modelling scenario where x_i is the voltage at node i, each edge $e \in E$ corresponds to a resistor with resistance $R_{ij} = 1$, and between any two nodes i and j there is a potential difference of $V_{ij} = x_i - x_j$. Then for each edge $e = (i, j) \in E$, the energy loss per edge is

$$I_{ij} V_{ij} = \frac{V_{ij}^2}{R_{ij}} = (x_i - x_j)^2,$$

and the total energy loss in the network is $\sum_{e \in E}(x_i - x_j)^2 = x^\top L x$. So the types of problems that appear in these scenarios are those like *Suppose we had some constraints on the potentials x_i at certain nodes, how can we find the potentials at the other nodes that minimise the total energy loss?* The Laplacian is a natural way to model this problem.

Another physical analogy to have in mind is that of a network of springs.



Here each of the edges is a spring that is trying to pull the nodes together. The (potential) energy of the spring corresponding to edge $e = (i, j)$ is proportional to $k(x_i - x_j)^2$ for some spring constant k, and so the total potential energy of the network given fixed positions x is

$$\sum_{e \in E} k(x_i - x_j)^2 = k x^\top L x = x^\top L x,$$

if we take[3] $k = 1$. So again the Laplacian is a natural way to model this problem.

We now present a few (mostly obvious) properties of the Laplacian.

Proposition 13.3. *The Laplacian L is symmetric and positive semidefinite. In particular, $x^\top L x \geq 0$ for all $x \in \mathbb{R}^n$, and all eigenvalues of L are nonnegative.*

[3]Otherwise, since the spring constant k is a positive constant, we can absorb it into the Laplacian by scaling the Laplacian by $\sqrt{k}$.

Proposition 13.4. *The smallest eigenvalue $\lambda_{\min}$ of L is 0, and the corresponding eigenvector is the all-ones vector $\mathbf{1} = (1, 1, \dots, 1)$.*

Proposition 13.5. *The second smallest eigenvalue λ_2 of L is positive if and only if the graph G is connected, and the multiplicity of λ_2 is equal to the number of connected components of G.*

§13.2 Graph drawing in two dimensions

Now we will focus our attention on the problem of drawing a graph $G = (V, E)$ in two dimensions; in particular, we want to find functions $x, y \colon [n] \to \mathbb{R}$ such that (x_i, y_i) is a good representation of the nodes ν_i and (x_i, y_i)-(x_j, y_j) is a good representation of the edge (ν_i, ν_j).

Example 13.6 (Square graph)**.** Consider the graph G with $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (2, 3), (3, 4), (4, 1)\}$. We want to find a mapping

$$\begin{aligned} f \colon V &\to \mathbb{R}^2, \\ i &\mapsto (x_i, y_i), \end{aligned}$$

where (x_i, y_i) is the position of node i in the plane.

A formalisation of the "optimal drawing" can be obtained in any number of ways via different optimisation functions that correspond to different functions. We will choose one that is convenient and gives the following simple heuristic to visualise the graph G in $\mathbb{R}^2$.

Algorithm: 2D SPECTRAL DRAWING

- Compute the Laplacian L of the graph G.
- Compute the eigenvectors ν_2, ν_3 corresponding to the second and third smallest eigenvalues of L.
- Let $X = (\nu_2, \nu_3)$ be the matrix with columns ν_2 and ν_3.
- For each row i of X, let (x_i, y_i) be the coordinates of the point in $\mathbb{R}^2$ corresponding to the i-th node.
- Draw the graph G with nodes at (x_i, y_i) and edges as straight lines between the corresponding points.

Essentially we have taken $f(i) = (x_i, y_i) = (\nu_2[i], \nu_3[i])$ as the mapping. Note that $\nu_1 = (1, 1, \dots, 1)$ is the eigenvector corresponding to the smallest eigenvalue of L—indeed $L_{v_1} = (D - A)e = 0$ for any edge e—and so ν_2 and ν_3 are orthogonal to ν_1. This means that the spectral drawing algorithm will always place the nodes in a way that is invariant under translation and scaling, and so the algorithm is not unique, like we suggested earlier.

In any case, let us discuss how we arrived at this drawing. Consider the following instance of drawing in one dimension, i.e. $x \in \mathbb{R}^n$ is the vector of positions of the nodes along a line, and each coordinate x_i is the position of node $i \in V$. We make the following attempts at determining the best drawing we can get:

1. First we formulate the problem as the following optimisation problem:
$$\min_{x\in\mathbb{R}^n} \sum_{(i,j)\in E} (x_i - x_j)^2 = \min_{x\in\mathbb{R}^n} x^\top Lx.$$
In this case, we are minimising distance between nodes, and so the optimal solution is to place the nodes at the origin, i.e. $x = (0, 0, \ldots, 0)$. This is not a very useful drawing.

2. To correct this degenerate collapse, we force the nodes to be spread out by adding a constraint that there is some non-zero "location measure" for the positions of the nodes, perhaps, $\sum_{i=1}^n x_i^2 = \|x\|_2^2 = 1 \implies \|x\|_2 = 1$. This gives us the following optimisation problem:
$$\min_{x\in\mathbb{R}^n} x^\top Lx \quad \text{s.t.} \quad \|x\|_2 = 1,$$
which is satisfied by taking $x = \left(\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}, \ldots, \frac{1}{\sqrt{n}}\right)$. This however is merely a translation of the result above.

3. To get a more adequate drawing, we add a second constraint that the nodes should be orthogonal to the all-ones vector, i.e. $\mathbf{1}^\top x = 0$. This gives us the following optimisation problem:
$$\min_{x\in\mathbb{R}^n} x^\top Lx \quad \text{s.t.} \quad \|x\|_2 = 1, \quad \mathbf{1}^\top x = 0.$$
By the Rayleigh quotient description of the second eigenvector, the solution to this problem is the eigenvector ν_2 corresponding to the second smallest eigenvalue of L. This option doesn't suffer from the faults of the first two, and this is what we want to adopt in one dimension.

Using ideas from what we have seen in this one dimensional case, we will try to minimise the sum of squared distances from the locations of directly connected nodes in the two-dimensional case where we take $x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$. Consider the following options:

1. We formulate the optimisation problem as
$$\min_{\substack{x,y\in\mathbb{R}^n \\ \|x\|_2=1,\, \|y\|_2=1 \\ \mathbf{1}^\top x=0,\, \mathbf{1}^\top y=0}} \sum_{(i,j)\in E} ((x_i - x_j)^2 + (y_i - y_j)^2) = \min_{\substack{x,y\in\mathbb{R}^n \\ \|x\|_2=1,\, \|y\|_2=1 \\ \mathbf{1}^\top x=0,\, \mathbf{1}^\top y=0}} x^\top Lx + y^\top Ly.$$
Now, for all x such that $\|x\|_2 = 1$ and $x^\top \nu_1 = 0$, we have $x^\top Lx \geq \nu_2^\top L\nu_2$, and likewise for y, $y^\top Ly \geq \nu_2^\top L\nu_2$. So the optimal solution to this problem is bounded below by $2\nu_2^\top L\nu_2$, and the lower bound is achieved by taking $x = \nu_2$ and $y = \nu_2$, which is the optimal solution. But this is not desirable, as the option does not use all the two dimensions available for visualisation.

2. To avoid the issue in the option above, we impose the additional constraint that the vectors x and y should be orthogonal to each other, i.e. $x^\top y = 0$. This gives us the following optimisation problem:
$$\min_{\substack{x,y\in\mathbb{R}^n \\ \|x\|_2=1,\, \|y\|_2=1 \\ \mathbf{1}^\top x=0,\, \mathbf{1}^\top y=0 \\ x^\top y=0}} \left(\sum_{(i,j)\in E} (x_i - x_j)^2 + (y_i - y_j)^2 \right) = \min_{\substack{x,y\in\mathbb{R}^n \\ \|x\|_2=1,\, \|y\|_2=1 \\ \mathbf{1}^\top x=0,\, \mathbf{1}^\top y=0 \\ x^\top y=0}} x^\top Lx + y^\top Ly.$$
As we are independently minimising $x^\top Lx$ and $y^\top Ly$, the optimal solution is to take $x = \nu_2$ and $y = \nu_3$.

The second option is the one we will adopt in two dimensions, and this is the idea behind the spectral drawing algorithm.

§13.3 The normalised Laplacian

In the next class we will explore the connection between the second smallest eigenvalue of the Laplacian and the connectivity of the graph. To do this, we will need to consider the normalised Laplacian.

Definition 13.7 (Normalised Laplacian matrix)**.** *The* normalised Laplacian *of a graph G is defined as $\hat{L} := D^{-1/2}LD^{-1/2} = D^{-1/2}(D-A)D^{-1/2} = I - D^{-1/2}AD^{-1/2} = I - \hat{A}$, where $\hat{A} = D^{-1/2}AD^{-1/2}$ is the normalised adjacency matrix.*

Let the tuples $\{(\lambda_i, \nu_i)\}_{i=1}^n$ be the eigenpairs of the normalised adjacency matrix $\hat{A}$. Then we have

$$\hat{L}\nu_i = (I - \hat{A})\nu_i = \nu_i - \hat{A}\nu_i = \nu_i - \lambda_i\nu_i = (1-\lambda_i)\nu_i,$$

$$\hat{L} = \begin{bmatrix} 1-\lambda_1 & 0 & \cdots & 0 \\ 0 & 1-\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1-\lambda_n \end{bmatrix},$$

when we express $\hat{L}$ in the basis of the eigenvectors of $\hat{A}$. So the eigenvalues of the normalised Laplacian are $1-\lambda_i$ for $i = 1, 2, \ldots, n$. From the properties of the normalised adjacency matrix, we know that $1 = \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, and so $0 = 1-\lambda_1 \leq 1-\lambda_2 \leq \cdots \leq 1-\lambda_n$. This means that, since we know that $\lambda_n \geq -1$, if the eigenvalues of $\hat{L}$ are $\mu_i = 1-\lambda_i$, then $0 = \mu_1 \leq \mu_2 \leq \cdots \leq \mu_n \leq 2$. Furthermore, the eigenvector corresponding to the smallest eigenvalue of the normalised Laplacian is $\hat{\nu}_1 = D^{1/2}\nu_1 = (\sqrt{d_1}, \sqrt{d_2}, \ldots, \sqrt{d_n})^\top$

By Proposition 13.5, an algebraic criterion for connectivity of the graph is that the second smallest eigenvalue of the normalised Laplacian $\mu_2 > 0$. We will need to define a combinatorial notion of connectivity (related to the sparsest cut) and then show the connection between μ_2 and this notion. Furthermore, we will show that if μ_2 is small, then the graph can be split into two parts by removing a small number of edges. This will be the focus of the next class.

§14 Lecture 14—04th March, 2024

§14.1 Combinatorial connectivity

Fix a set $S \subseteq V$ such that $S \neq \varnothing$.

Definition 14.1. *A* cut *is a partition of V into two sets S and $V \setminus S$. The* boundary *of the cut is the set of edges with one endpoint in S and the other in $V \setminus S$, i.e.*

$$\partial S = \{(u,v) \in E \mid u \in S, v \in V \setminus S\}.$$

We want to say that a graph is more disconnected if there is a large section of the graph which is very poorly connected to the rest of the graph. So it makes sense to develop a quantitative statement for how close-to-disconnected a graph is—this is the conductance.

Definition 14.2. *Let the volume of a cut S be $\mathrm{vol}(S) = \sum_{v \in S} \deg(v)$, where $\deg(v)$ is the degree of vertex v. The* conductance *of a cut S is*

$$\phi(S) := \frac{|\partial S|}{\mathrm{vol}(S)}.$$

The conductance *of a graph G is then*

$$\phi(G) := \min_{S \subseteq V} \phi(S) \text{ s.t. } 0 < \mathrm{vol}(S) \leq \frac{1}{2}\mathrm{vol}(V),$$

that is,

$$\phi(G) = \min_{S \,:\, \mathrm{vol}(S) \neq 0} \frac{|\partial S|}{\min\{\mathrm{vol}(S), \mathrm{vol}(V) - \mathrm{vol}(S)\}}.$$

Note that as we are dividing by the total number of edges in S, we will focus our attention on sets that are less than $\frac{1}{2}$ multiple of the edge volume. From this discussion we can also consider the following variation of the definition for the conductance of a cut S:

$$\phi(S) = \frac{\text{number of edges crossing from } S \text{ to } V \setminus S}{2 \times \text{number of internal edges in } S + \text{number of edges crossing from } S \text{ to } V \setminus S} \leq 1.$$

The conductance $\phi(G)$ is equal to 0 if and only if the graph is disconnected, and is equal to 1 if and only if the graph is a complete graph. We cannot get a graph that is more connected than a complete graph, and so $0 \leq \phi(G) \leq 1$. This notion of conductance will be our combinatorial notion of connectivity.

The problem of finding the set S that gives us the conductance $\phi(G)$ is NP-hard, so much of the focus is on designing approximation algorithms for this problem. Note also that the conductance is defined independently of the spectral properties of the graph. We will now see how the conductance is related to the eigenvalues of the Laplacian of the graph via a celebrated inequality which we now discuss.

§14.2 Cheeger's inequality

Theorem 14.1 (Cheeger's inequality)**.** *Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Let $\hat{L}$ be the normalised Laplacian of G. Then the conductance of G is related to the second smallest eigenvalue of $\hat{L}$ by the inequality*

$$\frac{\lambda_2}{2} \leq \phi(G) \leq \sqrt{2\lambda_2},$$

where λ_2 is the second smallest eigenvalue of $\hat{L}$.

This inequality gives a reasonable approximation to the conductance of a graph when μ_2 is relatively high. The moral lesson of this inequality is that just like we were able to discuss algebraic properties of a graph using discussions about its combinatorial properties, we can achieve a bound on this combinatorial property (how "connected" a graph is) merely by examining its Laplacian.

§14.2.1 Easy direction of the proof of Cheeger's inequality: the lower bound

We first prove that $\phi(G) \geq \frac{\lambda_2}{2}$.

Fix any set $S \subseteq V$ and let $s = \mathrm{vol}(S)/\mathrm{vol}(V)$ be the fraction of the volume of V that is in S. We will prove the lower bound by proving that $\phi(S) \geq (1-s)\mu_2$. We already know from the definition of the Rayleigh quotient that

$$\mu_2 = \min_{\substack{x\neq 0\\ x\perp\hat{\nu}_1}} \frac{x^\top \hat{L} x}{x^\top x} = \min_{\substack{x\neq 0\\ x\perp\hat{\nu}_1}} \frac{x^\top D^{-1/2} L D^{-1/2} x}{\|x\|_2^2}$$

where D is the degree matrix of G and $\hat{\nu}_1 = (\sqrt{d_1}, \sqrt{d_2}, \ldots, \sqrt{d_n})$ is the normalised eigenvector corresponding to the smallest eigenvalue of $\hat{L}$. Now assume that there is no isolated vertex, and let $y = D^{-1/2}x$. By changing x into y, we get

$$\mu_2 = \min_{\substack{y\neq 0\\ y\perp d}} \frac{y^\top L y}{y^\top D y},$$

where $d = (d_1, \ldots, d_n)$.

Since μ_2 is the smallest value of the Rayleigh quotient for all vectors under these constraints, we will construct a vector y below that satisfies these constraints and use the fact that μ_2 is no more than y's Rayleigh's quotient to prove the bound. Let $y' = \mathbb{1}\{i \in S\}$. We have

$$y'^\top L y' = \sum_{(i,j)\in E} \left(y'_i - y'_j\right)^2 = \sum_{(i,j)\in\partial S} 1 = |\partial S|.$$

Now set $y = y' - s\mathbf{1}$. Note that $L\mathbf{1} = 0$, so

$$\begin{aligned} y^\top L y &= (y' - s\mathbf{1})^\top L (y' - s\mathbf{1}) \\ &= y'^\top L y' - 2sy' L\mathbf{1} + s^2 \mathbf{1}^\top L \mathbf{1} \\ &= |\partial S| - 2s\mathrm{vol}(S) + s^2\mathrm{vol}(V) = |\partial S| - s^2\mathrm{vol}(V) + s^2\mathrm{vol}(V) = |\partial S|. \end{aligned}$$

We also have

$$\begin{aligned} y^\top D y &= \sum_{i\in V} y_i^2 d_i = \sum_{i\in S}(1-s)^2 d_i + \sum_{i\notin S}(-s)^2 d_i \\ &= (1-s)^2\mathrm{vol}(S) + s^2\left(\mathrm{vol}(V) - \mathrm{vol}(S)\right) = (1-s)\mathrm{vol}(V). \end{aligned}$$

Furthermore, y is orthogonal to d, because

$$y^\top d = \sum_{i\in S}(1-s)d_i - s\sum_{i\notin S} d_i = (1-s)\mathrm{vol}(S) + s\left(\mathrm{vol}(V) - \mathrm{vol}(S)\right) = 0.$$

Therefore we get that for any cut S,

$$\mu_2 \leq \frac{|\partial S|}{(1-s)\mathrm{vol}(S)} = \frac{|\partial S| \cdot \mathrm{vol}(V)}{\mathrm{vol}(S)\mathrm{vol}(V\setminus S)} \leq 2\phi(S),$$

since $\max(\mathrm{vol}(S), \mathrm{vol}(V\setminus S)) \leq \mathrm{vol}(V)/2$. Therefore $\mu_2/2 \leq \min_{\substack{S\subseteq V\\ \mathrm{vol}(S)\neq 0}} \phi(S) = \phi(G)$, which is the desired result.

§14.2.2 Hard direction of the proof of Cheeger's inequality: the upper bound

Now we show that $\phi(G) \le \sqrt{2\lambda_2}$.

Let y be the eigenvector corresponding to μ_2. By re-indexing vertices in the graph we can assume that $y_1 \le y_2 \le \ldots \le y_n$. Let $k \in [n]$ be the minimum index such that $\sum_{i=1}^{k} d_i \ge \frac{1}{2}\text{vol}(V)$. We define $z := y - y_k \mathbf{1}$, and then we rescale z such that $z_1^2 + z_n^2 = 1$. Note that $z_1 \le z_k = 0 \le z_n$. We claim that we still have $\frac{z^\top L z}{z^\top D z} \le \frac{y^\top L y}{y^\top D y}$. Indeed,

$$z^\top D z = (y - y_k \mathbf{1})^\top D (y - y_k \mathbf{1}) = y^\top D y - 2 y_k \mathbf{1}^\top D y + y_k^2 \mathbf{1}^\top D \mathbf{1} \ge y^\top D y,$$

where the last inequality holds because in the second term $\mathbf{1}^\top D y = d^\top y = 0$, as $y \perp d$ and the third term is non-negative. We also have

$$z^\top L z = (y - y_k \mathbf{1})^\top L (y - y_k \mathbf{1}) = y^\top L y$$

since $L\mathbf{1} = 0$ and so the second and third terms in the expansion of $z^\top L z$ are zero. Therefore $\frac{z^\top L z}{z^\top D z}$ has a bigger denominator while the numerator is the same, so it is smaller.

For $t \in \mathbb{R}$, define $S_t := \{i \in V : z_i \le t\}$. We will prove that there exists a number $t \in \mathbb{R}$ such that $\phi(S_t) \le \sqrt{2\mu_2}$. Then we will get that $\phi(G) \le \phi(S_t) \le \sqrt{2\mu_2}$. As the choice of t is not obvious, we randomly pick t from a distribution given by the PDF p, where $p(t) = 2|t|$ if $t \in [z_1, z_n]$, and $p(t) = 0$ otherwise. Indeed $p(\cdot)$ is a PDF:

$$\sum_{z_1}^{z_n} p(t)\,\mathrm{d}t = \int_{z_1}^{0} -2t\,\mathrm{d}t + \int_{0}^{z_n} 2t\,\mathrm{d}t = z_1^2 + z_n^2 = 1,$$

and so this is a valid distribution. If we can prove that $0 \le \mathbb{E}\left[\sqrt{2\mu_2}\min\{\text{vol}(S), \text{vol}(\overline{S})\} - |\partial S|\right]$, then we know that there must be some t for which the expresion inside the expectation is non-negative. This implies that, for that t, $\phi(S_t) = \frac{|\partial S_t|}{\min\{\text{vol}(S_t), \text{vol}(\overline{S_t})\}} \le \sqrt{2\mu_2}$, and then the proof of the upper bound is completed. We will need the following lemma.

Lemma 14.3. *For any random $t \in \mathbb{R}$, we have $\mathbb{E}\left[|\partial S_t|\right] \le \sqrt{2\mu_2} \cdot \mathbb{E}_t\left[\min\{\text{vol}(S_t), \text{vol}(\overline{S_t})\}\right]$.*

Before we prove this lemma, we first show that it implies the desired result. We have

$$0 \le \sqrt{2\mu_2} \cdot \mathbb{E}_t\left[\min\{\text{vol}(S_t), \text{vol}(\overline{S_t})\}\right] - \mathbb{E}\left[|\partial S_t|\right] = \mathbb{E}\left[\sqrt{2\mu_2}\min\{\text{vol}(S_t), \text{vol}(\overline{S_t})\} - |\partial S_t|\right],$$

and by the probabilistic method, there exists $t \in [z_1, z_n]$ such that

$$0 \le \sqrt{2\mu_2}\min\{\text{vol}(S_t), \text{vol}(\overline{S_t})\} - |\partial S_t| \implies \frac{|\partial S_t|}{\min\{\text{vol}(S_t), \text{vol}(\overline{S_t})\}} \le \sqrt{2\mu_2},$$

and so now we can focus our attention on proving the lemma. We will prove two claims that separately bound the left and right sides of the inequality.

Claim 14.4. *For any random $t \in \mathbb{R}$, $\mathbb{E}_t\left[|\partial S_t|\right] \le \left(z^\top L z\right)^{1/2} \cdot \sqrt{2 z^\top D z} \le \sqrt{2\mu_2} \cdot z^\top D z$.*

Proof. The second inequality is immediate since

$$\left(z^\top Lz\right)^{1/2} \cdot \sqrt{2z^\top Dz} = \left(\frac{z^\top Lz}{z^\top Dz}\right)^{1/2} \cdot \sqrt{2} \cdot z^\top Dz \le \sqrt{2\mu_2} \cdot z^\top Dz.$$

We will now prove the first inequality. Fix an edge $(i,j) \in E$. We assume $z_i \le z_j$ w.l.o.g. (otherwise, we can swap i and j). It can be shown by considering various cases depending on the signs of z_i and z_j that $\Pr[z_i \le t \le z_j] = |\mathrm{sign}(z_j)z_j^2 - \mathrm{sign}(z_i)z_i^2|$. Therefore if S is a cut, we have

$$\begin{aligned}\Pr[(i,j) \in S] &= \Pr[z_i \le t \le z_j] = |\mathrm{sign}(z_j)z_j^2 - \mathrm{sign}(z_i)z_i^2| \\ &= \begin{cases} |z_i^2 - z_j^2| & \text{if } \mathrm{sign}(z_i) = \mathrm{sign}(z_j) \\ z_j^2 + z_i^2 & \text{if } \mathrm{sign}(z_i) \ne \mathrm{sign}(z_j)\end{cases} \le \begin{cases} |z_i - z_j|(|z_i| + |z_j|) & \text{if } \mathrm{sign}(z_i) = \mathrm{sign}(z_j) \\ (z_i - z_j)^2 & \text{if } \mathrm{sign}(z_i) \ne \mathrm{sign}(z_j)\end{cases} \\ &\le |z_i - z_j|(|z_i| + |z_j|).\end{aligned}$$

Here we have merely applied the triangle inequality twice. To get the last inequality when $\mathrm{sign}(z_i) \ne \mathrm{sign}(z_j)$, we upper-bound one of the terms $z_i - z_j$ by $|z_i| + |z_j|$, and now writing the random variable $|\partial S_t|$ as a sum over indicator random variables denoting whether each edge is in the cut, we observe that

$$\begin{aligned}\mathbb{E}_t\left[|\partial S_t|\right] &= \sum_{(i,j)\in E} \Pr[(i,j) \in S] \le \sum_{(i,j)\in E} |z_i - z_j|(|z_i| + |z_j|) \\ &\le \sqrt{\sum_{(i,j)\in E} (z_i - z_j)^2} \cdot \sqrt{\sum_{(i,j)\in E} (|z_i| + |z_j|)^2} \le \left(z^\top Lz\right)^{1/2} \cdot \left(2 \cdot \sum_{(i,j)\in E} (z_i^2 + z_j^2)\right)^{1/2} \\ &= \left(z^\top Lz\right)^{1/2} \cdot \sqrt{2\sum_{i\in V} d_i z_i^2} = \left(z^\top Lz\right)^{1/2} \cdot \sqrt{2z^\top Dz},\end{aligned}$$

where the penultimate inequality is obtained by changing the sum over edges to a sum over vertices. (For each edge (i,j), we assumed that $z_i \le z_j$, and thus we consider each edge only once in the sum.) Now fix a vertex $i \in V$. For each edge incident on i, the term z_i^2 appears exactly once, and hence the total number of times that this term appear is the number of edges that are incident on it, which is d_i. Therefore we have

$$\mathbb{E}_t\left[|\partial S_t|\right] \le \left(z^\top Lz\right)^{1/2} \cdot \sqrt{2z^\top Dz} \le \sqrt{2\mu_2} \cdot z^\top Dz. \qquad \square$$

Claim 14.5. *For any random $t \in \mathbb{R}$, $\mathbb{E}_t\left[\min\{\mathrm{vol}(S_t), \mathrm{vol}(\overline{S_t})\}\right] = z^\top Dz$.*

Proof. Exercise. $\square$

Proof of Lemma 14.3. Combining the above two claims, we have

$$\mathbb{E}_t\left[|\partial S_t|\right] \le \sqrt{2\mu_2} \cdot z^\top Dz = \sqrt{2\mu_2} \cdot \mathbb{E}_t\left[\min\{\mathrm{vol}(S_t), \mathrm{vol}(\overline{S_t})\}\right],$$

which is the desired result. $\square$

So we have shown that there exists a t such that $\phi(S_t) \le \sqrt{2\mu_2}$, and so we have proved the upper bound of Cheeger's inequality. This completes the proof of Cheeger's inequality—but not quite. We need an efficient algorithm to construct such a set S_t that achieves the desired conductance.

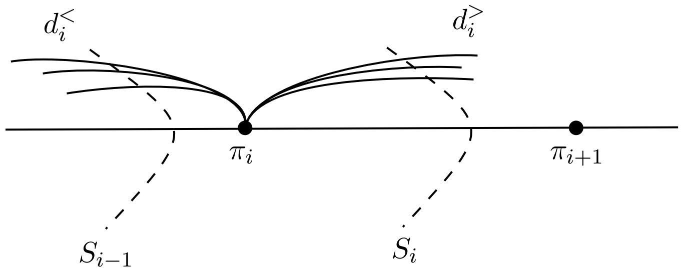
§14.2.3 Spectral partitioning

We present an algorithm that, given a graph $G = (V, E)$, finds a cut $S \neq \varnothing, V$ such that $\phi(S) \leq \sqrt{2\mu_2}$. The algorithm is as follows:

```
Algorithm: Spectral Partitioning

  • Let $x$ be the second eigenvector of the normalised Laplacian $\hat{L}$ of $G$.
  • Sort all $x_i$ to get a permutation $\pi\colon [n] \to [n]$ such that $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$.
  • For $t = 1, 2, \ldots, n-1$:
      – Define $S_i := \{\pi_1, \pi_2, \ldots, \pi_i\}$.
      – Compute $\phi(S_i) = \frac{|\partial S_i|}{\min\{\mathrm{vol}(S_i), \mathrm{vol}(V \setminus S_i)\}}$.
```

The runtime of this algorithm is $T_x + O(mn)$, where T_x is the time taken to compute the eigenvector $x = \hat{\nu}_2$, since the conductance of a set can be naïvely computed in $O(m)$ time. However, we can compute the conductance iteratively and spend much less time per iteration, to get a better runtime of $T_x + O(m)$. The idea for improving this time is to compute both ∂S_i and $\mathrm{vol}(S_i)$ from ∂S_{i-1} and $\mathrm{vol}(S_{i-1})$ in $O(1)$ time. Consider the following picture:



Here $d_i^<$ is the number of edges that are incident on π_i and have the other endpoint beyond S_{i-1}, and $d_i^>$ is the number of edges that are incident on π_i and have the other endpoint beyond S_i. We may then check to see that

$$\begin{aligned} \mathrm{vol}(S_i) &= \mathrm{vol}(S_{i-1}) + d_{\pi_i} \\ \partial S_i &= \partial S_{i-1} + d_i^> - d_i^<, \end{aligned}$$

Power iteration method Now we discuss the best algorithm for finding the eigenvalues. Of course we can use Gaussian elimination to find the eigenpairs, but that is somewhat slow for us. We can do the following to find the second eigenvector of any matrix, say the normalised adjacency matrix $\hat{A}$. Start with $x^{(0)}$, a normalised Gaussian random vector. Then compute $\hat{x}^{(i)} = \hat{A}x^{(i-1)}$, where $\hat{A}$ is the normalised adjacency matrix. Then set $x^{(i)} = \left(\hat{x}^{(i)} - \left(\hat{x}^{(i)} \cdot \nu_1\right)\mu_1\right) / \left\|\hat{x}^{(i)}\right\|$, where ν_1 is the first eigenvector of $\hat{A}$, and repeat this process a bunch of times. So, modulo the normalisation, we notice that $x^{(t)} \propto \sum_i \alpha_i \cdot \lambda_i^t \cdot \nu_i$, where ν_i is the i-th eigenvector of the adjacency matrix. So what happens is that as we increase $t \to \infty$, the components that don't correpond to the eigenvector we desire will decay.

§15 Lecture 15—06th March, 2024

§15.1 Introducing linear optimisation

The optimisation problems we will study have the generic form

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in \mathcal{F} \subseteq \mathbb{R}^n, \end{aligned}$$

where $\mathcal{F}$ is a feasible set of constraints.

Here's an example of such a problem that follows on our discussion from the last class:

Example 15.1 (Conductance)**.** The min-conductance problem in the graph $G = (V, E)$ that we discussed before is the following optimisation problem:

$$\begin{aligned} \min \quad & \frac{|\partial S|}{\sum_{i \in S} d_i} \\ \text{s.t.} \quad & S \neq \varnothing, \sum_{i \in S} d_i \leq \frac{1}{2} \sum_{i \in V} d_i, \end{aligned}$$

where d_i is the degree of node i.
We can regard this problem as a linear optimisation problem by defining the following variables:

$$x_i = \begin{cases} 1 & \text{if } i \in S, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the problem can be written as

$$\begin{aligned} \min \quad & \frac{\sum_{(i,j) \in E} (x_i - x_j)^2}{\sum_{i \in V} d_i x_i} \\ \text{s.t.} \quad & \sum x_i \geq 1, x_i \in \{0, 1\} \text{ i.e. } x_i(1 - x_i) = 0, \\ & \sum_{i \in V} d_i x_i \leq \frac{1}{2} \sum_{i \in V} d_i. \end{aligned}$$

This optimisation problem is NP-hard in general.

There are situations, however, where we can solve linear optimisation problems efficiently. We will study one of these kinds of problems, linear programs.

Definition 15.2. *A* linear program *is an optimisation problem of the form*

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \geq b, \end{aligned}$$

where $c, x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. That is, linear programs have linear objective functions and linear constraints.

Note that for maximisation problems, we can convert the objective function to a minimisation problem by negating the objective function. We can also convert inequalities to equalities by introducing slack variables, denoted by s. For example, the inequality $Ax \geq b$ can be converted to $Ax + s = b, s \geq 0$.

Many problems in algorithm design can be formulated as linear programs. Indeed, every problem that can be solved in polynomial time can be formulated as a linear program.

Example 15.3 (Fractional knapsack)**.** The *fractional* knapsack problem asks: given n divisible items (that is, items that can be "split into parts"), where item i has weight $w_i > 0$ and value $v_i > 0$, as well as a knapsack of capacity $B > 0$, what is the maximum value of items (or fractions thereof) that can be put into the knapsack (respecting its capacity B) to maximise the total value taken?
We will formulate this problem as an LP. Let $x_i \in (0, 1]$ be the fraction of item i taken. Then,

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq B, \\ & 0 \leq x_i \leq 1, \quad i = 1, \ldots, n. \end{aligned}$$

Indeed, every problem that can be solved in polynomial time can be formulated as a linear program.

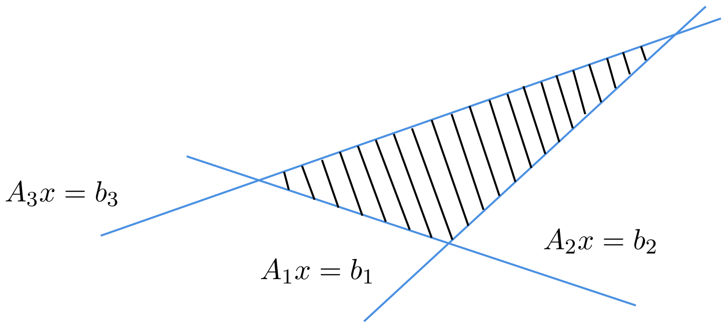
§15.2 Structure of solutions to linear programs

Definition 15.4. *The* feasible set *of an LP is the set $F = \{x : Ax \geq b\} \subseteq \mathbb{R}^n$. The* set of optimal solutions *is the set of points in F that achieve the minimum value of the objective function (i.e. there is no better solution in F).*

Indeed we can think of

$$F = \{x : A_1 x \geq b_1\} \cap \{x : A_2 x \geq b_2\} \cap \ldots \cap \{x : A_m x \geq b_m\} = \bigcap_{i=1}^{m} \{x : A_i x \geq b_i\},$$

where each $\{x : A_i x \geq b_i\}$ is a half-space in $\mathbb{R}^n$.



In $\mathbb{R}^n$, we call this intersection (shaded above) a *polyhedron* or a *polytope*. The feasible set of an LP is a polyhedron P, and we say that P is bounded if there exists a (large enough) high-dimensional box containing P; it is unbounded otherwise.

Take $P = F$. There are three possibilities for the solution x to an LP:

1. $P = \varnothing$, in which case the LP is infeasible, and there is no solution.
2. P is unbounded, in which case:
   a) The solution is infinite, e.g. in the case $\max_{x_1 \geq 0} x_1$.
   b) The solution is finite, e.g. in the case $\max_{x_1 \geq 0} -x_1$.
3. P is bounded, in which case there is a finite solution.

§15.3 Finding solutions to linear programs

Now we try to answer the following question: given an objective function $f = c^\top x$, how do we find the optimal values for x that minimise (or maximise) the objective function? Our initial attempt will be to come up with an algorithm that isn't particularly realistic (as it is difficult to implement), but it will help us understand the structure of the solution space and lead us to more efficient algorithms.

Algorithm:

- Guess $v^* = \min_{x \in F} c^\top x$.
- Start with a very large $v \gg v^*$.
- Decrease v continuously until there exists $x^* \in F \cap \{c^\top x \leq v\}$.
- Return $x^* \in H_v = \{x : c^\top x = v\}$.

Geometrically speaking, this algorithm is equivalent to moving a hyperplane H_v in the direction of $-c$ until it touches the feasible set F. But this algorithm is inefficient because we have to continuously decrease v and check whether there exists a solution in H_v. Notice that the first instance H_v intersects the feasible set F will be at a corner or vertex of the polytope P. We formally define a corner below.

Definition 15.5. *A* corner *or* vertex *of a polytope is defined by a linear system of n (the dimension of the space) equations of the form:*

$$\begin{aligned} A_{i_1} x &= b_{i_1}, \\ A_{i_2} x &= b_{i_2}, \\ &\vdots \\ A_{i_n} x &= b_{i_n}, \end{aligned}$$

where A_{i_j} is a row of A and b_{i_j} is the corresponding element of b.

Thus the optimal solution x^* being in the corner of the polytope means that it is defined by the solution to a system of n tight constraints. Thus the number of optimal solutions can potentially be n, because we can take n different hyperplanes. Based on this discussion, we can design an implementable—but still inefficient—algorithm for solving linear programs as follows.

Algorithm:

- Iterate over all $\binom{m}{n}$ sets $S = \{i_1, \ldots, i_n\}$ of the original n inequality constraints.
- Solve the linear system of equations $A_S x = b_S$.
- Check if the solution x satisfies the remaining $m - n$ constraints.
- If it is feasible, compute the objective function $f = c^\top x$.
- Return the solution x that minimises $c^\top x$.

Now for each of the $\binom{m}{n}$ corners of the polytope, we spend time solving the system of n equations, checking the feasibility of the remaining constraints, and computing the optimal value of the objective function. So if T_s is the time taken to solve a system of n equations, the time complexity of this algorithm is

$$\binom{m}{n} \cdot (T_s + m \cdot n + n) \simeq \left(\frac{m}{n}\right)^n \cdot (T_s + m \cdot n + n)\,,$$

which is very slow. Also note that it is not immediately clear that the time taken to write down a solution to the linear program is polynomial in the input size—for instance if the solution involves non-algebraic numbers, it may take some time to write down the solution. In the next lecture, we will say more about the time taken to solve a linear system of equations.

§16 Lecture 16—18th March, 2024

§16.1 Finding a solution to a system of linear equations

So given $Ax = b$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, we want to solve $Ax = b$ for the case where $m = n$. Recall the rule due to Crámer: $x_i = \frac{\det(A_i)}{\det(A)}$, where $\det(\cdot)$ is the mapping defined as

$$\begin{aligned} \det \colon \mathbb{R}^{n \times n} &\to \mathbb{R} \\ A &\mapsto \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)} \end{aligned}$$

where S_n is the set of all permutations of n elements, and $\operatorname{sign}(\sigma)$ is the sign of the permutation. The following theorem comes from elementary linear algebra:

Theorem 16.1. *Given a matrix A, the following are equivalent:*

1. *A is invertible (i.e. $\exists\, A^{-1}$ such that $AA^{-1} = I$),*
2. $\det(A) \neq 0$,

3. *The columns of A are linearly independent,*
4. *The rows of A are linearly independent,*
5. *The rank of A is n.*
6. *The equation $Ax = b$ has a unique solution for all b, namely $x = A^{-1}b$.*

Recall what we know from COMS W 4231 Analysis of Algorithms I that we can compute A^{-1} naïvely in time $O(n^3)$ by Gaussian elimination. (We can do better than this, but we won't discuss that.)

Corollary 16.1. *Suppose $A_{i,j}$ and b_j are both integers described by at most B bits. Then the solution x_i is rational and can be described by only $O(Bn + n \log n)$ bits.*

Proof. Let the maximum value of $\det(A_i)$ (resp. $\det(A)$) be α_i (resp. α). Then to describe x_i, we need $\log \alpha_i + \log \alpha$ bits. Since there are $n!$-many permutations and the maximum value of each $A_{i,j}$ of length B bits is 2^B, we have $\alpha_i \le n!2^{Bn}$ and $\alpha \le n!2^{Bn}$. Thus, we need $\log n! + Bn + \log n! + Bn = O(Bn + n \log n)$ bits, applying Stirling's approximation. $\square$

Note that the exact same holds for the solution to any instance of a linear program. Thus, describing the entire solution x to a linear program may require $\Omega(n^2)$ bits, even when $B = O(1)$.

Now suppose $m \neq n$ or $\det(A) = 0$. Recall from linear algebra that $\mathsf{col}(A)$ is the column space of A, and that

$$\mathsf{span}(\mathsf{col}(A)) = \left\{ y \in \mathbb{R}^m : y = \sum_{i=1}^{n} x_i A_i = Ax \text{ for all } x \in \mathbb{R}^n \right\}.$$

In particular, the span of the column space is exactly the same set as that of all possible b such that $Ax = b$ has a solution. This is crucial because we want to solve $Ax = b$, and if $b \notin \mathsf{span}(\mathsf{col}(A))$, then $Ax = b$ has no solution, and if $b \in \mathsf{span}(\mathsf{col}(A))$, then $Ax = b$ has one or more solutions.

We will now discuss how to find a solution. Let $S = \{S_1, \dots, S_k\}$ (with $k \le n$) be the maximal set of linearly independent columns of A. Then S is a basis for $\mathsf{col}(A)$, and $|S| = \mathsf{rank}(A)$. Furthermore, $\mathsf{span}(S) = \mathsf{span}(\mathsf{col}(A))$. We will exhibit a completion of $S \subseteq \mathbb{R}^m$ to a full basis of $\mathbb{R}^m$, namely $S' = \{S_1, \dots, S_k, S'_{k+1}, \dots, S'_m\}$, such that S' is a basis for $\mathbb{R}^m$. We can then write $b = \sum_{i=1}^m y_i S'_i$ for some y_i. Then $Ax = b$ has a solution if and only if $x = \sum_{i=1}^m y_i S'_i$ for some x. We can then write $x = \sum_{i=1}^k x_i S_i$ for some x_i, and then $Ax = b$ if and only if $x = \sum_{i=1}^k x_i S_i$ for some x_i. This is equivalent to $x = \sum_{i=1}^k x_i S_i + \sum_{i=k+1}^m x_i S'_i$ for some x_i. Thus, $Ax = b$ if and only if $x = \sum_{i=1}^k x_i S_i + \sum_{i=k+1}^m x_i S'_i$ for some x_i.

(Needed to leave class for a bit, so I missed a bit of this part.)

But what if there is no solution? We have the following proto-Farkas lemma.

Claim 16.2. *The following are equivalent:*

(i) *There is no solution to $Ax = b$,*

(ii) *There exists $y \in \mathbb{R}^m$ such that $y^\top A = 0$ and $y^\top b \neq 0$.*

Proof. First we show that (ii) implies (i) by contradiction. If there exists x such that $Ax = b$, then $0 \neq y^\top b = y^\top Ax = (y^\top A)x = 0$, a contradiction. Now we show that (i) implies (ii). Suppose there is no solution to $Ax = b$. Then $b \notin \mathsf{span}(\mathsf{col}(A))$; take $y = b - \text{proj}_A(b)$. Then we have that $b^\top y = y^\top(y + \text{proj}_A(b)) = \|y\| \neq 0$ and since $y \perp \mathsf{span}(\mathsf{col}(A))$, it must be that $y^\top A = 0$, and that by normalisation, $b^\top y = 1$. □

But how can we find y quickly then? All we need is to solve the system of linear equations $y^\top A = 0$ and $y^\top b = 1$. We can do this by solving the system of linear equations $A^\top y = 0$ and $b^\top y = 1$.

§16.2 Retvrn to linear programming: the Simplex algorithm

Recall now the general form of a linear program:

$$\begin{aligned} &\max\, c^\top x \\ &\text{s.t. } Ax \leq b, \end{aligned}$$

and the standard form of the linear program:

$$\begin{aligned} &\max\, c^\top x \\ &\text{s.t. } Ax = b, \\ &\quad\;\; x \geq 0. \end{aligned}$$

Say that an inequality $A_i x \leq b_i$ is *tight* if $A_i x = b_i$, that a point x is *basic* if it is the solution to n linearly independent tight constraints, and a basic feasible solution (bfs) x is both basic and feasible.

Definition 16.3. *A real vector x is a convex combination of vectors $y^{(1)}, \ldots, y^{(n)}$ if there exist $\alpha_1, \ldots, \alpha_n \geq 0$ such that $\sum_{i=1}^n \alpha_i = 1$ and $x = \sum_{i=1}^n \alpha_i y^{(i)}$.*

A solution $x \in F = \{x : Ax = b, x \geq 0\}$ is a *basic feasible solution* (bfs) if x is not a convex combination of other solutions in F. A bfs is *degenerate* if it is the solution to n linearly independent tight constraints. Indeed, basic feasible solutions are the vertices of the feasible polytopic region F.

Claim 16.4. *If x^* is a linear programming solution to a system of n equations that is feasible and bounded, then there exists an optimal solution that is a basic feasible solution.*

Proof. Suppose that x^* is not a basic feasible solution. Then there are less than n tight linearly independent constraints. Let C be a space defined by these $n-1$ tight linearly independent constraints. Then $\dim C \geq 1$, and there exists a direction $d \in \mathbb{R}^n$ with $d \neq 0$, such that for all $\alpha \in \mathbb{R}$, $x^* + \alpha d \in C$. Now consider a small $\varepsilon > 0$ so that $x^* \pm \varepsilon d \in C$. Note that C takes care of all the tight constraints on x^* and all other constraints have the form $x_i > 0$. This implies that there exists a small enough $\varepsilon > 0$ such that $x^* + \varepsilon d$ is still feasible. Now we consider

$$c^\top(x^* \pm \varepsilon d) = c^\top x^* \pm \varepsilon c^\top d.$$

Since x^* is optimal, $c^\top d = 0$. One of the directions decreases some coordinates of x^*, we want to push as much as allowed so that some coordinate of $x^* \pm \varepsilon d$ becomes 0. This makes one more constraint tight. We can repeat this process until we have n tight constraints, and we will have a basic feasible solution. □

Given our discussion we can now present a naïve algorithm for solving linear programs.

```
Algorithm: NAÏVE-LP-SOLVER($A, b$)

  • Iterate through all $\binom{n+m}{m}$ possible subsets of constraints; let the set of constraints be $T$.
  • For each $T$, solve the system of linear equations $Ax = b$ for $x$.
  • For each $x$, check if $x$ is feasible.
  • Return the optimal basic feasible solution.
```

The running time of this algorithm is $O(\binom{n+m}{m} n^3)$, which is exponential in n. Ideally we want to *not* do this. Instead, we will use the SIMPLEX algorithm which runs in poly-time in the average-case scenario.

```
Algorithm: SIMPLEX

  • Choose the starting vertex $x^{(0)} \in F$ and set $t = 0$.
  • Let $N(x^{(t)})$ be the set of neighbouring vertices of $x^{(t)}$.    // i.e. vertices that
    differ from $x^{(t)}$ in only one tight constraint
  • If $x^{(t)}$ is optimal, return $x^{(t)}$.
  • Otherwise, choose $x^{(t+1)} \in N(x^{(t)})$ such that $c^\top x^{(t+1)} < c^\top x^{(t)}$ and set $t \leftarrow t + 1$.
```

Here the *pivoting rule* is the rule that chooses $x^{(t+1)}$ from $N(x^{(t)})$—we're keeping $n-1$ constraints tight, throwing out one constraint, and adding another constraint. The SIMPLEX algorithm is a family of algorithms depending on which pivoting rule is used. To solve the problem of choosing $x^{(t+1)}$, we first need to know the starting vertex $x^{(0)} \in F$. It is a common trick that to find a starting point for this, we will need to solve another linear program which is easier to initialise. Given the general form of an LP, we only care about the $Ax \leq b$ if we want to find a feasible vertex point. To achieve this, we wish to minimise t subject to the constraint that $Ax - t\mathbf{1} \leq b$. Note that for optimal t with $t > 0$, the original LP is infeasible, i.e. $F = \varnothing$. For optimal $t \leq 0$, x is a starting vertex. Thus we can initialise this by setting $x = 0$ and $t = -\min_i b_i$.

Remark 16.5. The SIMPLEX algorithm is a polynomial-time algorithm in the average case, but there is no known polynomial-time algorithm for the worst-case scenario. In fact, the SIMPLEX algorithm can take exponential time in the worst-case scenario (i.e. where the pivoting rule is as bad as it can be).

Remark 16.6. Taking y^* which has the best possible improvement is not optimal; in other words, greedily choosing the neighbour that improves the objective function the most is not necessarily the best strategy. This is because the pivoting rule that improves the objective function the best is not known, and in fact, there is no known pivoting algorithm with a polynomial number of steps.

Remark 16.7. In practice, the SIMPLEX algorithm works well.

Conjecture 16.8 (Hirsch (1957), see [Dan16])**.** *For any starting vertex in the polytope F and any other optimal vertex V, there exists a shortest path of length* $\mathrm{poly}(n, m)$ *from the starting vertex to V.*

In fact the original "superconjecture" by Hirsch proposes a really tight upper bound (of $\leq m - n$) for the shortest path; however this was disproved by Santos in 2010, who found a concrete polytope of dimension 43 and 86 faces.

§17 Lecture 17—20th March, 2024

§17.1 Smoothed analysis for Simplex

This is due to Spielman and Teng [ST03]. In particular, they proved that for any matrix A and any pair of vectors b and c, the expected running time of the simplex method on the linear program

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & (A + G) \cdot x \leq b + h \end{aligned}$$

is bounded by $\mathrm{poly}\left(n, d, \frac{1}{\sigma}\right)$. Here G and h are a matrix and a vector respectively, consisting of independent Gaussian random variables with mean 0 and standard deviation $\sigma \cdot \max_{i \in [n]} \|(A_i, b_i)\|_2$, where A_i denotes the i-th row of the matrix A. Thus if we perturb the instance only very slightly, i.e. if we choose $\sigma = 1/\mathrm{poly}(n, d)$, the simplex method has polynomial expected running time for any A, b, and c that an adversary can choose.

§17.2 Linear programming duality and complementary slackness

Duality is one of the key concepts in linear programming: given a solution x to an LP of value z, how do we decide if x is optimal? In other words, how can we calculate a lower bound on $\min c^\top x$ given that the LP is in the following "standard form":

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0. \end{aligned}$$

Suppose now that we have y such that $A^\top y \leq c$, and observe that $y^\top b = y^\top Ax \leq c^\top x$ for any feasible solution x. Thus $y^\top b$ provides a lower bound on the value for our linear program. This conclusion is true for all y satisfying $A^\top y \leq c$, and so in order to find the best lower bound, we need to solve the following linear program:

$$\begin{aligned} \max \quad & y^\top b \\ \text{s.t.} \quad & A^\top y \leq c. \end{aligned}$$

This is called the *dual* (D) LP of the original linear program, which is called the *primal* (P) LP. Indeed, we have

Theorem 17.1 (Weak duality)**.** *If the primal (P) is a minimisation linear program with optimum value z, then it has a dual (D) which is a maximisation problem with optimum value w, where $w \leq z$.*

Proof. Let x be a feasible solution to (P) and y be a feasible solution to (D). Then $c^\top x \geq y^\top Ax = y^\top b$, and so $z \geq c^\top x \geq y^\top b$. Thus $w \leq y^\top b \leq z$. $\square$

Note that this is true if either the primal or dual is infeasible or unbounded, provided that we use the following convention: for an infeasible minimisation problem, the optimum value is $+\infty$; for an unbounded minimisation problem, the optimum value is $-\infty$; for an infeasible maximisation problem, the optimum value is $-\infty$; for an unbounded maximisation problem, the optimum value is $+\infty$. What is remarkable is that one even has strong duality—namely that both the primal and dual have the same values—if at least one of them is feasible.

Theorem 17.2 (Strong duality)**.** *If the primal (P) is a minimisation linear program with optimum value z, then it has a dual (D) which is a maximisation problem with optimum value w, where $w = z$ if at least one of the primal or dual is feasible.*

Proof. We assume that (P) is feasible (the argument is analogous if (D) is feasible). If (P) is unbounded, then $z = -\infty$, and by weak duality, $w \leq z$, and so it must be that $w = -\infty$. Otherwise (i.e. (P) is not unbounded), let x^* be the optimum solution to (P), i.e.

$$\begin{aligned} z &= c^\top x^* \\ Ax^* &= b \\ x^* &\geq 0. \end{aligned}$$

We would like to find a dual feasible solution with the same value as (or no worse than that of) x^*. That is, we are looking for y such that

$$\begin{aligned} A^\top y &\leq c \\ b^\top y &\geq z. \end{aligned}$$

If no such y exists, we can use Farkas' lemma to derive that there exists $x \in \mathbb{R}^n$ nonnegative and $\lambda in \mathbb{R}$ nonnegative such that $Ax - \lambda b = 0$ and $c^\top x < \lambda z$. We distinguish two cases:

- If $\lambda \neq 0$, then we can scale by λ and assume that $\lambda = 1$. Then $Ax = b$ and $c^\top x < z$, which contradicts the fact that x^* is optimal.
- Here $\lambda = 0$, so $Ax = 0$ and $c^\top x < 0$. Consider now $x^* + \varepsilon x$ for $\varepsilon > 0$ small enough. This is feasible for (P), and so $z \leq c^\top x^* + \varepsilon c^\top x < 0$, which is a contradiction.

Thus there exists y such that $A^\top y \leq c$ and $b^\top y \geq z$, and so $w \geq z$. By weak duality, $w \leq z$, and so $w = z$. $\square$

Complementary slackness If we introduce a slack variable z, we note that changing the value of z_j does not affect our objective function, and we are allowed to pick any positive z. Hence if the corresponding Lagrange multiplier is λ_j, then we must have $(z^*(\lambda))_j \lambda_j = 0$. This is since by definition $z^*(\lambda)_j$ minimises $z_j \lambda_j$. Hence if $z_j \lambda_j \neq 0$, we can tweak the values of z_j to make a smaller $z_j \lambda_j$.

This makes our life easier since our search space is smaller.

Example 17.1. Consider the following problem:

$$\text{maximize } x_1 - 3x_2 \text{ subject to}$$

$$\begin{aligned} x_1^2 + x_2^2 + z_1 &= 4 \\ x_1 + x_2 + z_2 + z_2 &= 2 \\ z_1, z_2 &\geq 0, \end{aligned}$$

where z_1, z_2 are slack variables. The Lagrangian is

$$L(x, z, \lambda) = ((1 - \lambda_2)x_1 - \lambda_1 x_1^2) + ((-3 - \lambda_2)x_2 - \lambda_1 x_2^2) - \lambda_1 z_1 - \lambda_2 z_2 + 4\lambda_1 + 2\lambda_2.$$

To ensure finite minimum, we need $\lambda_1, \lambda_2 \leq 0$.
By complementary slackness, $\lambda_1 z_1 = \lambda_2 z_2 = 0$. We can then consider the cases $\lambda_1 = 0$ and $z_1 = 0$ separately, and save a lot of algebra.

(Missed the rest of the lecture because I left for some reason.)

§18 Lecture 18—25th March, 2024

§18.1 The Ellipsoid method

The Ellipsoid algorithm, due to Khachiyan [Kha79], is the first polynomial-time algorithm for linear programming. It solves the feasibility problem, which is that of finding $x \in \mathbb{R}^n$ such that $Ax \le b$. Why is this enough for linear programming? The idea is that the feasible region of a linear program is the intersection of halfspaces, and so if we can find a point in each halfspace, we can find a point in the feasible region. The Ellipsoid algorithm we will present solves the feasibility of $Q_v = \{x : Ax \le b, c^\top x \le v\}$ for a given v, where

$$\begin{aligned} v^* = \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \le b. \end{aligned}$$

The key observation is that Q_v is non-empty if and only if $v \ge v^*$, and so we can use binary search on v^* to find the optimal value of the linear program. Let $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$, where and let R be the maximum possible value of $|v^*|$. When A, b, and c are B-bit numbers, $R \le 2^{\text{poly}(n,m,B)}$.

```
Algorithm: Feasibility

  • If $Q_{+R}$ is empty, return "infeasible".
  • If $Q_{-R}$ is non-empty, return "unbounded".        // since then $v^* = -\infty$
  • Run binary search:
      – Start the interval $[\ell, r] = [-R, R]$.
      – Compute the center $q = (r + \ell)/2$.
      – If $Q_q$ is feasible then recurse on $[s,t] = [s,q]$, else recurse on $[s,t] = [q,t]$.
  • Stop when the interval is of size $|t - s| \le \frac{1}{R}$.   // this is the smallest possible
    difference between two numbers in $[-R, R]$
  • Return $v^* \approx s$.
```

The running time of the Feasibility algorithm is the product of the time per iteration of the algorithm and the number of iterations, which is

$$O\left(\log \frac{2R}{1/R}\right) = O\left(\log R^2\right) = O\left(\log R\right) = \text{poly}(n, m, B).$$

The takeaway here, is that checking the feasibility of a linear program is not an easier problem than solving the linear program itself.

For the Ellipsoid algorithm, we need to be able to check if a point $x \in F = \{x : Ax \le b\}$ or report that F is empty. We will solve a relaxed problem where either we find an $x \in F$ or report that $\text{vol}(F) \le \varepsilon$, where ε goes like $1/\text{poly}(n, m, B)$.

Question: How does this carry over to the binary search in the Feasibility algorithm?
▷ Going from the main problem to the relaxed problem is a bit of a complication in the sense that the feasible region could be such that it is nonempty but has essentially zero volume. For example, it could be that the feasible region is an $(n-1)$-dimensional subspace of $\mathbb{R}^n$, or that it contains

exactly one point of dimension zero. So there is a way to adapt to this from our second form with one more iteration that reports infeasibility if the volume is too small, and so if the feasible region is not empty it must be a lower-dimensional subspace, which we can then go down to and solve the problem in that (affine) subspace until we get to zero dimensions.

Before we describe the algorithm, we introduce a few definitions.

Definition 18.1 (Ball)**.** *Fix a radius $r > 0$ and a center $x \in \mathbb{R}^n$. The* ball *of radius r centered at x is the set $B(x, r) = \{y \in \mathbb{R}^n : \|y - x\|_2 \le r\}$.*

Definition 18.2 (Axis-aligned ellipsoid)**.** *Fix $\lambda_1, \dots, \lambda_n, r > 0$ and $x \in \mathbb{R}^n$. The* axis-aligned ellipsoid *with radii $\lambda_1, \dots, \lambda_n$ centred at y is the set*

$$E_{r,\lambda}(y) = \left\{ x \in \mathbb{R}^n : \sum_{i=1}^{n} \left(\frac{x_i - y_i}{\lambda_i} \right)^2 \le r^2 \right\}.$$

Definition 18.3 (General ellipsoid)**.** *Fix $r > 0$ and take a rank-n matrix $A \in \mathbb{R}^{n \times n}$ and $x \in \mathbb{R}^n$. The* general ellipsoid *with radius r centred at y is the set*

$$\begin{aligned} E_{r,A}(y) &= \left\{ x \in \mathbb{R}^n : x^\top A^\top A x \le r^2 \right\} = \left\{ x \in \mathbb{R}^n : \|Ax\|_2^2 \le r^2 \right\} \\ &= \left\{ x + y : \left\| \Lambda^{-1} \cdot \mathscr{R} x \right\|^2 \le r^2 \right\}, \end{aligned}$$

for

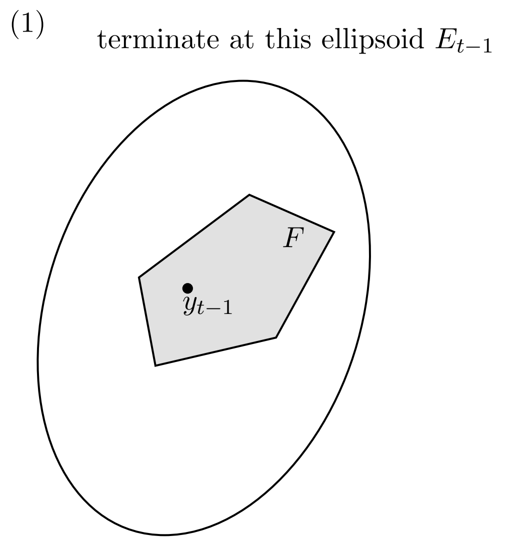
$$\Lambda = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix},$$

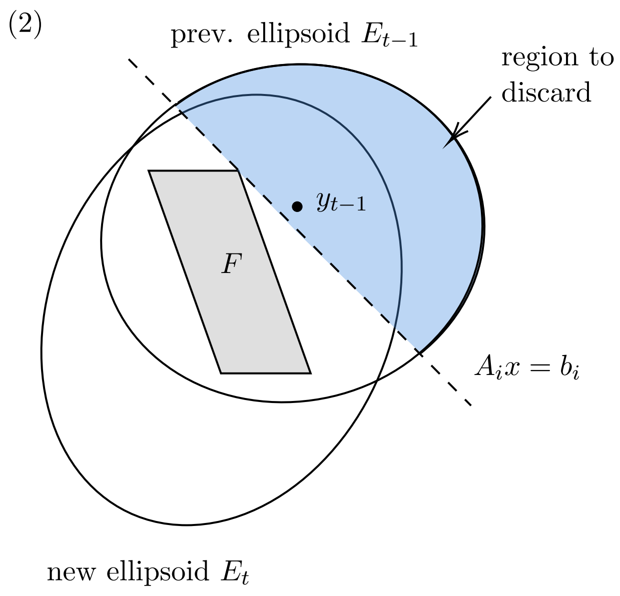
and where $\mathscr{R}$ is a unitary rotation matrix (i.e. $\mathscr{R}^\top \mathscr{R} = I$).

```
Algorithm: Ellipsoid

• Start with $E^{(0)} \supset F$, where $E^{(0)}$ is the ball $B_r(y_0)$ of radius $r$ centred at $y_0$.
• For $t = 1, \dots, T$ iterations do:
    – If the centre $y^{(t-1)} \in F$, return $y^{(t-1)}$.
    – Otherwise, there exists a violated constraint level $i$ such that $A_i y^{(t-1)} > b_i$ such
      that $y^{(t-1)}$ and $F$ are on opposite sides of $A_i x = b_i$.
    – Define $E^{(t)}$ with centre $y^{(t)}$ to be the smallest ellipsoid containing
      $E^{(t-1)} \cap \{x : A_i x \le b_i\}$. so that $F \subseteq E^{(t)}$ and $\mathrm{vol}(E^{(t)}) < \mathrm{vol}(E^{(t-1)})$.
• Return "$\mathrm{vol}(E^{(T)}) \le \varepsilon$".                                  // must be infeasible
```

The correctness of the Ellipsoid algorithm follows by definition.





The following progress measure is crucial to the running time of the algorithm.

Claim 18.4. *It is possible to construct $E^{(t)}$ in poly-time such that* $\mathrm{vol}(E^{(t)}) \le \mathrm{vol}(E^{(t-1)}) \cdot \left(1 - \frac{1}{2n}\right)$.

Proof sketch. The worst case is when $E^{(t)}$ is a ball, and the halfspace $A_i x \le b_i$ goes through the centre of the ball. □

By this claim, we know that

$$\mathrm{vol}(E^{(T)}) \le \mathrm{vol}(E^{(0)}) \left(1 - \frac{1}{2n}\right)^T \le (2R)^n \cdot e6{-}T/2n \le \varepsilon,$$

where wwe set $T = O(n) \cdot \left(\log \frac{1}{\varepsilon} + n \cdot \mathrm{poly}(n,m) \cdot B\right)$, and so the overall runtime is $\mathrm{poly}(n,m) \cdot \left(\log \frac{1}{\varepsilon} + B\right)$. In particular, it is enough to take $\varepsilon = \exp\left(\frac{1}{\mathrm{poly}(n,m,B)}\right)$ to solve the feasibility problem in polynomial time.

Note that the only access to F that the Ellipsoid algorithm has is through the oracle that either has access to $y \in F$ or finds some separating hyperplane $A_i x = b_i$ such that F and y are on opposite sides of the hyperplane. We call this oracle $\mathscr{O}_F$ a *separation oracle*. In partocular, the maximum number of calls made by the Ellipsoid algorithm to $\mathscr{O}_F$ is $\Theta(n) \cdot \left(\log \frac{1}{\varepsilon} + n \log R\right)$. In fact, it is okay if, for instance, m is exponential in n and B, as long as we have an efficient oracle $\mathscr{O}_F(y)$. This is useful for solving some LPs, for instance for some combinatorial problems with polynomially many variables but exponentially many constraints.

§19 Lecture 19—27th March, 2024

§19.1 Optimising general functions—gradient descent

The general question for us in this area is the following: given a function $f : \mathbb{R}^d \to \mathbb{R}^n$, how do we find the minimum of f? We have seen that we can formulate this as a constrained optimisation problem according to some feasible set $S \subseteq \mathbb{R}^d$; we will focus on how to solve problems of this kind without requiring any special structure on f or S. Indeed we can just take f to be $+\infty$ outside of S and S to be a general set.

Unfortunately, we don't have too much information about f, so we must do something a bit more general; in this instance, and motivated by the connection between minima of a function and the gradient of the function, we will generally use an iterative method called *gradient descent* which is conceptually similar to the SIMPLEX algorithm and follows the following generic algorithmic template:

1. Start with some $x^{(0)} \in \mathbb{R}^d$.
2. For $t = 0, 1, 2, \ldots, T$:
   - Compute the new iterate $x^{(t)}$ as a function of the previous iterate $x^{(t-1)}$ and properties of the function f evaluated at $x^{(t-1)}$ (e.g. the gradient of f at $x^{(t-1)}$, or possibly more).
3. Hope that as we progress towards T, the value of $f(x)$ decreases, and after T steps, we declare that $x^{(T)}$ is a good approximation to the minimum of f.

Some pressing questions arise immediately: (a) how do we compute $x^{(t)}$? (b) given this procedure from (a), how do we know the number of iterations T necessary to get close to the optimum $x^* := \arg\min_{x \in \mathbb{R}^d} f(x)$? Before we answer these questions, note that we can only hope to get close to x^*, and not necessarily to x^* itself, since we may not have enough information about f to do so. We say that x is an ε-close approximation to x^* if $f(x) \leq \min_{x \in \mathbb{R}^d} f(x) + \varepsilon$.

Slightly different choices in the procedure for computing $x^{(t)}$ can lead to different algorithms with improved (faster) bounds on the number of iterations required in (b), as we will see. We will start with a minimal set of assumptions, strengthen these as we go, and observe that our guarantees improve concurrently.

Assumption 0: f is smooth In particular, we assume that the function f has first and second derivatives as well as Taylor expansions.

Proposition 19.1 (Taylor expansions)**.** *For any $x, \delta \in \mathbb{R}^d$, the Taylor expansion of f at x is given by*

$$f(x+\delta) = f(x) + \nabla f(x)^\top \cdot \delta + \frac{1}{2}\delta^\top \cdot \nabla^2 f(y) \cdot \delta,$$

where $y \in [x, x+\delta] \subset \mathbb{R}^d$ is a point in the vicinity of x, and

$$\nabla f(x)_i := \frac{\partial f}{\partial x_i}(x) \in \mathbb{R}^d, \quad \nabla^2 f(x)_{ij} := \frac{\partial^2 f}{\partial x_i \partial x_j}(x) \in \mathbb{R}^{d\times d}$$

are respectively the gradient and Hessian of f at x.

Obviously, under this assumption, the next iterate will possibly depend on the previous iterate as well as the gradient and Hessian of f at the previous iterate. Our goal is to find a δ that locally minimises f; an initial option is to try to find

$$\delta^* := \arg\min_{\delta \in \mathbb{R}^d} f(x+\delta) = \arg\min_{\delta \in \mathbb{R}^d} \left\{ f(x) + \nabla f(x)^\top \cdot \delta + \frac{1}{2}\delta^\top \cdot \nabla^2 f(x) \cdot \delta \right\}. \tag{‡}$$

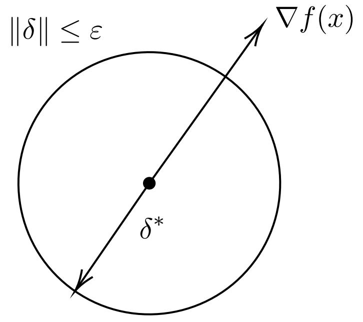
But this is as hard as the original question; we have made exactly zero progress. Instead, we will try to come up with good-enough heuristics for (‡) that are easier to solve.

Although we know $y \in [x, x+\delta]$, we don't have good control for y; if it were a function of δ for instance, the problem becomes very difficult, and so directly solving (‡) is not feasible. But notice that the last two terms in the objective are linear and quadratic in δ, so if $\|\delta\| \le \varepsilon$,

$$\nabla f^\top \cdot \delta \asymp \varepsilon, \quad \delta^\top \cdot \nabla^2 f \cdot \delta \asymp \varepsilon^2,$$

and the second term is much smaller than the first term. So our heuristic is to ignore the second term and solve

$$\delta^* = \arg\min_{\delta \text{ s.t. } \|\delta\| \le \varepsilon} f(x) + \nabla f(x)^\top \cdot \delta = \arg\min_{\delta \text{ s.t. } \|\delta\| \le \varepsilon} \nabla f(x)^\top \cdot \delta.$$



Clearly, δ^* must oppositively balance $\nabla f(x)$, and so we can take $\delta^* = -\eta \cdot \nabla f(x)$, where $\eta > 0$ is a constant such that $\|\delta^*\| \le \varepsilon$, i.e.

$$\|\eta \cdot \nabla f(x)\| \le \varepsilon \implies \eta \le \frac{\varepsilon}{\|\nabla f(x)\|}.$$

This gives us the usual gradient descent algorithm:

Algorithm: GD

- Start with $x^{(0)} \in \mathbb{R}^d$.
- For $t = 1, \dots, T$ iterations compute:

$$\begin{cases} x^{(t)} = x^{(t-1)} + \delta^* = x^{(t-1)} - \eta \cdot \nabla f(x^{(t-1)}), \\ \eta = \dfrac{\varepsilon}{\|\nabla f(x^{(t-1)})\|}. \end{cases}$$

- Return $f(x^{(T)})$.

This algorithm is simple enough, but it is hard to say much more about what it is actually doing. It isn't even clear that the function value $f(x^{(t)})$ is decreasing at each iteration, so we will try to rectify this with a more principled assumption on f.

Assumption 1: f is β-smooth This assumption "granularises" the previous one:

Definition 19.2 (Smoothness)**.** *We say that f is β-smooth for $\beta > 0$ if and only if for all $x, y \in \mathbb{R}^d$, the gradient function ∇f is β-Lipschitz continuous, i.e.*

$$\|\nabla f(x) - \nabla f(y)\| \leq \beta \cdot \|x - y\|.$$

Claim 19.3. *The function f is β-smooth if and only if for all $x, \delta \in \mathbb{R}^d$, we have $\delta^\top \cdot \nabla^2 f \cdot \delta \leq \beta \cdot \|\delta\|^2$; that is, all eigenvalues of $\nabla^2 f$ are bounded above by β.*

Proof. Suppose first that f is β-smooth, i.e. $\|\nabla f(x) - \nabla f(y)\| \leq \beta\|x - y\|$. Let $\delta \neq 0$ and consider $x \mapsto x + \theta\delta$ with $\theta \in [0, 1]$. Using the mean value theorem,

$$\nabla f(x + \delta) - \nabla f(x) = \int_0^1 \nabla^2 f(x + t\delta)\, \delta\, dt.$$

Taking the inner product with δ and applying the Lipschitz bound, we obtain

$$\delta^\top \nabla^2 f(x + t\delta)\, \delta \leq \beta\, \|\delta\|^2,$$

implying $\nabla^2 f$ has eigenvalues at most β.

Conversely, if $\delta^\top \nabla^2 f(x)\, \delta \leq \beta\, \|\delta\|^2$ for all x, δ, then

$$\|\nabla f(x + \delta) - \nabla f(x)\| \leq \int_0^1 \left\|\nabla^2 f(x + t\delta)\, \delta\right\| dt \leq \int_0^1 \beta\, \|\delta\|\, dt = \beta\, \|\delta\|,$$

showing ∇f is β-Lipschitz. □

Given the above, we have our Taylor expansion as

$$f(x + \delta) \leq f(x) + \nabla f(x)^\top \cdot \delta + \frac{\beta}{2}\|\delta\|^2 =: A_{\delta,x},$$

and so our direction under this new assumption is to find the δ minimising $A_{\delta,x}$. This objective is a nice quadratic function that we're able to optimise analytically, so

$$\begin{aligned}
\delta^* &= \arg\min_{\delta \in \mathbb{R}^d} \nabla f(x)^\top \cdot \delta + \frac{1}{2}\beta\|\delta\|^2 = \arg \min_{\substack{\delta = -\eta \cdot \nabla f(x) \\ \mathbb{R} \ni \eta \geq 0}} \frac{1}{2}\beta\eta^2\|\nabla f(x)\|^2 - \eta\|\nabla f(x)\|^2 \\
&= \arg\min_{\eta \geq 0} \frac{1}{2}\beta\eta^2 - \eta = \arg\min_{\eta \geq 0} \frac{1}{2}\beta\eta^2 - \eta \\
&= -\frac{1}{\beta}\nabla f(x).
\end{aligned}$$

So we have the following algorithm:

```
Algorithm: β-SMOOTH GD
    • Start with $x^{(0)} \in \mathbb{R}^d$.
    • For $t = 1, \dots, T$ iterations compute $x^{(t)} = x^{(t-1)} - \frac{1}{\beta}\nabla f(x^{(t-1)})$.
    • Return $f(x^{(T)})$.
```

Now do we make progress as this algorithm runs? Observe that

$$\begin{aligned} f(x^{(t)}) &= f(x^{(t-1)} - \frac{1}{\beta}\nabla f(x^{(t-1)})) \\ &\le f(x^{(t-1)}) - \frac{1}{\beta}\|\nabla f(x^{(t-1)})\|^2 + \frac{\beta}{2}\cdot\frac{1}{\beta^2}\|\nabla f(x^{(t-1)})\|^2 \\ &= f(x^{(t-1)}) - \frac{1}{2\beta}\|\nabla f(x^{(t-1)})\|^2. \end{aligned}$$

This gives the following immediate observations:

(i) $f(x^{(t)}) \le f(x^{(t-1)})$ for all t,

(ii) if $\nabla f(x^{(t-1)}) \neq 0$, then $f(x^{(t)}) < f(x^{(t-1)})$.

In particular, our algorithm gets stuck at a point $x^{(t)}$ if and only if $\nabla f(x^{(t)}) = 0$, which happens when one of the following occurs:

(a) *a global minimum* of f is reached, which is what we wanted anyway,

(b) *a local minimum* of f is reached, in which case our small δ^* steps aren't large enough to escape the local minimum, or

(c) *a saddle point* is reached, in which case f increases in some directions and decreases in others, and so the gradient is zero.

The saddle point issue can be resolved by using random perturbations in the direction of the gradient to escape the saddle point, amd although this is a heuristic and not a guarantee, it works most times. We can also use second-order information such as the Hessian to escape saddle points, but this is slightly more involved and we will not discuss it here. The second, however, is harder to resolve, and we will need to make an additional assumption to address it.

Assumption 2: f is convex This assumption wishes away the issue of local minima immediately:

Definition 19.4 (Convexity). *We say that f is convex if and only if $\lambda_{\min}\left(\nabla^2 f(x)\right) \ge 0$ for all $x \in \mathbb{R}^d$.*

Fact 19.5. *The function f is convex if and only if for all $x, y \in \mathbb{R}^d$ and $\lambda \in (0,1)$, the convex interpolation $f(\lambda x + (1-\lambda)y) \le \lambda f(x) + (1-\lambda) f(y)$.*

Proof. First, assume $\nabla^2 f(x)$ is positive semidefinite for all x. For any $x, y \in \mathbb{R}^d$ and $\lambda \in (0,1)$, consider $g(t) = f\big((1-t)x + ty\big)$. By the chain rule,

$$g''(t) = \nabla^2 f\big((1-t)x + ty\big)\big(y - x,\, y - x\big) \ge 0,$$

so g is convex in t, implying

$$f(\lambda x + (1-\lambda)y) = g(\lambda) \leq (1-\lambda)g(0) + \lambda g(1) = (1-\lambda)f(x) + \lambda f(y).$$

Conversely, if $f(\lambda x + (1-\lambda)y) \leq (1-\lambda)f(x) + \lambda f(y)$ holds for all x, y, λ, then $g(t)$ is convex for all line segments, ensuring $\nabla^2 f(x)$ is positive semidefinite everywhere. $\square$

Fact 19.6. *If f is convex, any local minimum is also a global minimum, i.e. $\nabla f(x) = 0 \implies x = x^*$.*

Proof. Fix $x \in \mathbb{R}^d$. For all $\delta \in \mathbb{R}^d$, we have the Taylor expansion

$$f(x+\delta) = f(x) + \nabla f(x)^\top \cdot \delta + \frac{1}{2}\delta^\top \cdot \nabla^2 f(y) \cdot \delta \geq f(x),$$

since $\nabla f(x)^\top \cdot \delta = 0$ and $\delta^\top \cdot \nabla^2 f(y) \cdot \delta \geq 0$ for all $y \in [x, x+\delta]$ since f is convex. Hence $f(x) \leq f(x+\delta)$ for all $\delta \in \mathbb{R}^d$, and so x is a global minimum. $\square$

With the assumption of β-smoothness and convexity, we can start talking about the rate of convergence, i.e. how many steps T are needed to to get $f(x^{(T)}) - f(x^*) \leq \varepsilon$.

Theorem 19.1. *If f is β-smooth and convex, then the β-SMOOTH GD algorithm converges—i.e. $f(x^{(T)}) - f(x^*) \leq \varepsilon$ is satisfied—after $T = O(D^2\beta/\varepsilon)$ iterations, where*

$$D := \max_{x \text{ s.t. } f(x) \leq f(x^{(0)})} \|x - x^{(0)}\|$$

is the diameter of the feasible set.

Proof. We start with the following claim:

Claim 19.7. $\|\nabla f(x)\| \geq \dfrac{f(x) - f(x^*)}{\|x - x^*\|}$ *for all $x \in \mathbb{R}^d$.*

Proof. We will look to relate how far we are from the optimal solution to how much progress we're making, vis a vis the relation between $f(x') - f(x)$ to $\|\nabla f(x)\|$ where x' is ε-close to x^*. We have

$$\begin{aligned} f(x^*) = f(x + (x^* - x)) &= f(x) + \nabla f(x)^\top \cdot (x^* - x) + \frac{1}{2}(x^* - x)^\top \cdot \nabla^2 f(y) \cdot (x^* - x) \\ &\geq f(x) + \nabla f(x)^\top \cdot (x^* - x). \end{aligned}$$

Rearranging this gives $\nabla f(x)^\top (x^* - x) \geq f(x^*) - f(x)$, which is

$$f(x) - f(x^*) \leq \|\nabla f(x)\| \cdot \|x - x^*\| \implies \|\nabla f(x)\| \geq \frac{f(x) - f(x^*)}{\|x - x^*\|}. \qquad \square$$

For example, in the claim above, if we have $x^{(t)}$ such that $f(x^{(t)}) - f(x^*) > \varepsilon$, then we also have $\|\nabla f(x^{(t)})\| \geq \varepsilon / \|x^{(t)} - x^*\|$. We can now proceed with the proof of the theorem. Set $\mathsf{d}_t = f(x^{(t)}) - f(x^*) > \varepsilon$, so that $\|\nabla f(x^{(t)})\| \geq \mathsf{d}_t / \|x^{(t)} - x^*\| \geq \mathsf{d}_t / D$.

Let T_1 be the number of iterations until $\mathsf{d}_{T_1} \leq \mathsf{d}_0/2$. Before reaching $t = T_1$, we have $\|\nabla f(x^{(t)})\| \geq \mathsf{d}_t/D \geq \mathsf{d}_0/(2D)$; in each iteration, $f(x^{(t)}) - f(x^*)$ drops by

$$\geq \frac{1}{2\beta} \cdot \|\nabla f(x^{(t)})\|^2 \geq \frac{1}{2\beta} \cdot \frac{\mathsf{d}_0^2}{4D^2} \implies T_1 \leq \frac{\mathsf{d}_0/2}{\mathsf{d}_0^2/(8\beta D^2)} = \frac{4\beta D^2}{\mathsf{d}_0}.$$

Now let T_2 be the number of iterations until $\mathsf{d}_{T_1+T_2} \le \mathsf{d}_{T_1}/2$. Before reaching $t = T_1 + T_2$, we have $\|\nabla f(x^{(t)})\| \ge \mathsf{d}_t/D \ge \mathsf{d}_{T_1}/(2D)$; in each iteration, $f(x^{(t)}) - f(x^*)$ drops by

$$\ge \frac{1}{2\beta} \cdot \|\nabla f(x^{(t)})\|^2 \ge \frac{1}{2\beta} \cdot \frac{\mathsf{d}_{T_1}^2}{4D^2} \implies T_2 \le \frac{\mathsf{d}_{T_1}/2}{\mathsf{d}_{T_1}^2/(8\beta D^2)} = \frac{4\beta D^2}{\mathsf{d}_{T_1}} = \frac{8\beta D^2}{\mathsf{d}_0}.$$

We can continue in this manner to T_k, the number of iterations until $\mathsf{d}_{T_1+\ldots+T_k} \le \mathsf{d}_{T_{k-1}}/2$. Before reaching $t = T_1 + \ldots + T_k$, we have $\|\nabla f(x^{(t)})\| \ge \mathsf{d}_{T_{k-1}}/D$; in each iteration, $f(x^{(t)}) - f(x^*)$ drops by

$$\ge \frac{1}{2\beta} \cdot \|\nabla f(x^{(t)})\|^2 \ge \frac{1}{2\beta} \cdot \frac{\mathsf{d}_{T_{k-1}}^2}{4D^2} \implies T_k \le \frac{\mathsf{d}_{T_{k-1}}/2}{\mathsf{d}_{T_{k-1}}^2/(8\beta D^2)} = \frac{8\beta D^2}{\mathsf{d}_{T_{k-1}}} = \frac{16\beta D^2}{\mathsf{d}_{T_{k-2}}} = \ldots = \frac{2^{k+1}4\beta D^2}{\mathsf{d}_0}.$$

The question then is how to set k so that at time $T = \sum_{i=1}^k T_i$, the value $f(x^{(T)}) - f(x^*) \le \varepsilon$, i.e. $\varepsilon/2 \le \mathsf{d}_0/2^k$; the choice $k = \log_2(\mathsf{d}_0/\varepsilon)$ suffices. Hence the total time is then

$$\begin{aligned} T &= T_1 + T_2 + \ldots + T_k \\ &\le \frac{2\beta D^2}{\mathsf{d}_0}\left(2 + 4 + 8 + \ldots + 2^k\right) \\ &\le \frac{2\beta D^2}{\mathsf{d}_0} \cdot 2^{k+1} = \frac{2\beta D^2}{\mathsf{d}_0} \cdot 2 \cdot \frac{\mathsf{d}_0}{\varepsilon} \\ &= \frac{4\beta D^2}{\varepsilon}. \end{aligned}$$

So convergence requires $T = O(D^2\beta/\varepsilon)$ iterations, as desired. □

§20 Lecture 20—01st April, 2024

April Fools'!

§20.1 Domain-free basic GD

Last time we proved the following theorem:

Theorem 20.1. *If f is β-smooth and convex, then the β-SMOOTH GD algorithm converges—i.e. $f(x^{(T)}) - f(x^*) \le \varepsilon$ is satisfied—after $T = O(D^2\beta/\varepsilon)$ iterations, where*

$$D := \max_{x \text{ s.t. } f(x) \le f(x^{(0)})} \left\|x - x^{(0)}\right\|$$

is the diameter of the feasible set.

This gives us a generic bound on the convergence of gradient descent. But it isn't really a good bound, not least because (1) it requires us to understand the geometry of the optimisation landscape to obtain D, and (2) we have the annoying dependence on $1/\varepsilon$ in the number of iterations. The second is particularly annoying: if we want to get a very high precision, we need to run the algorithm for a very long time, and in fact, the algorithm doesn't terminate in poly-time in general.

Example 20.1. For LPs, what should be ε? Well, for exact LP solvers, we need ε small enough so that we can round to the exact solution as the nearest integer. But for LPs, the optimal value is an integer multiple of $1/2^{\text{poly}(n)\cdot B}$, where B is the number of bits in the input. So we must set $\varepsilon < 1/2^{\text{poly}(n)\cdot B}$, in which case our runtime is $O\left(2^{\text{poly}(n)\cdot B} D^2 \beta\right)$, which is exponential in n and B.

So while it makes sense to think of the algorithm from before as an approximation algorithm, it isn't poly-time in general because of the $1/\varepsilon$ depenence; to fix this, we'd ideally want a $\log(1/\varepsilon)$ dependence instead, which introduces our final assumption for GD:

Assumption 3: f is α-strongly convex Here we basically say that f isn't just convex, but has additional "strict" curvature; this curvature allows us to remove the dependence on D and improve the dependence on ε in one fell swoop.

Definition 20.2 (Strong convexity)**.** *We say that f is α-strongly convex if $\lambda_{\min}(\nabla^2 f(x)) \geq \alpha$ for all x, where $\lambda_{\min}$ is the smallest eigenvalue and $\alpha > 0$.*

Theorem 20.2. *If f is β-smooth and α-strongly convex, then the β-*SMOOTH GD *algorithm converges—i.e. $f(x^{(T)}) - f(x^*) \leq \varepsilon$ is satisfied—after*

$$T = O\left(\frac{\beta}{\alpha} \log\left(\frac{f(x^{(0)}) - f(x^*)}{\varepsilon}\right)\right)$$

iterations.

Proof. Check that

$$\begin{aligned} f(x) = f(x^* + (x - x^*)) &= f(x^*) + \nabla f(x^*)^\top (x - x^*) + \frac{1}{2}(x - x^*)^\top \nabla^2 f(y)(x - x^*) \\ &\geq f(x^*) + \frac{\alpha}{2}\|x - x^*\|^2, \end{aligned}$$

where the inequality follows from the fact that $\nabla f(x^*) = 0$ and $\nabla^2 f(y) \succeq \alpha I$ for some y. So if we are close enough to the optimum, x is close to x^*, and indeed,

$$f(x) - f(x^*) \geq \frac{\alpha}{2}\|x - x^*\|^2.$$

So we look to remove the dependency of T on $\|x - x^*\|^2$. In each iteration we move from $x^{(t)}$ to $x^{(t+1)}$, and our aim is to appropriately quantify the progress towards optimality per iteration. From the last lecture, since we decrease $f(x)$ by $\|\nabla f(x)\|^2/(2\beta)$ in each iteration, we have from the inequality above that

$$\begin{aligned} f(x^{(t+1)}) - f(x^*) &\leq f(x^t) - f(x^*) - \frac{1}{2\beta}\|\nabla f(x^{(t)})\|^2 \\ &\leq f(x^{(t)}) - f(x^*) - \frac{1}{2\beta}\left(\frac{f(x^{(t)}) - f(x^*)}{\|x^{(t)} - x^*\|}\right)^2 \\ &\leq f(x^{(t)}) - f(x^*) - \frac{1}{2\beta} \cdot \frac{(f(x^{(t)}) - f(x^*))^2}{2\alpha^{-1}(f(x^{(t)}) - f(x^*))} \leq (f(x^{(t)}) - f(x^*))\left(1 - \frac{\alpha}{4\beta}\right), \end{aligned}$$

where we are using the fact that $\|\nabla f(x)\| \leq \beta\|x - x^*\|$ and $\|x - x^*\| \leq \sqrt{2(f(x) - f(x^*))/\alpha}$. This effectively says that, per iteration, the distance of the current point to the optimum decreases by a factor of $1 - \alpha/(4\beta)$; starting from $x = x^{(0)}$, we have in T iterations that

$$f(x^{(T)}) - f(x^*) \leq \left(1 - \frac{\alpha}{4\beta}\right)^T (f(x^{(0)}) - f(x^*)) \leq \varepsilon,$$

where we have taken T as

$$T = \frac{4\beta}{\alpha} \log\left(\frac{f(x^{(0)}) - f(x^*)}{\varepsilon}\right). \qquad \square$$

Here we have made two grand improvements. First, the number of iterations the algorithm takes no longer depends on $\|x - x^*\|$, but instead on the function values themselves; this is a big deal because it means we don't need to understand the geometry of the feasible set to get D^2 which gives a bound on the number of iterations required. Second, the number of iterations T has a $\log(1/\varepsilon)$ dependence, which is a huge improvement over the $1/\varepsilon$ dependence we had before. What we may yet find unsatisfying is the dependence of T on β/α, or, the *condition number* of the Hessian:

$$\kappa\left(\nabla^2 f(x)\right) := \frac{\beta}{\alpha} = \frac{\lambda_{\max}(\nabla^2 f(x))}{\lambda_{\min}(\nabla^2 f(x))}.$$

We can think of this as a measure of how much the curvature of f varies across the feasible set; if κ is large, then the function is very flat in some directions and very steep in others, which makes it hard to optimise. In the worst case, κ can be as large as 2^n for n-dimensional LPs, which is why we have the exponential dependence on n in the runtime of the algorithm. So we may want to ask: can we do even better?

§20.2 Newton's **method**

The gradient dexcent algorithm we just discussed just uses the gradient as a first derivative to guide the search for the optimum—it is a *first-order* method. But we can improve the cost of the condition number on convergence by using *second-order* information such as the Hessian, which is what Newton's method does. The general idea is to approximate the function f by a quadratic function q at each point x, and then minimise q instead of f.

We will not prove all the results, but will provide general intuition. To understand this method, recall the Taylor expansion of f around x:

$$f(x + \delta) = f(x) + \nabla f(x)^\top \delta + \frac{1}{2}\delta^\top \nabla^2 f(y)\delta,$$

where y is some point between x and $x + \delta$. In GD, we found bounds on $\delta^\top \nabla f(x)\delta$ using α-strong convexity for a lower bound and β-smoothness for an upper bound. But in Newton's method, we try to optimise this quadratic form directly instead of finding bounds for it.

The intuition for our technique is as follows: if we are at a point x and we want to move to a point $x + \delta$ such that $f(x + \delta) < f(x)$, then we can exhibit a change of (basis) variables in such a way that the correct direction to move is the one in which the quadratic form $\delta^\top \nabla^2 f(x)\delta$ is minimised.

To this end, define $\Gamma = A \cdot \delta$, where A is a full rank basis transformation matrix, so that $\delta = A^{-1} \cdot \Gamma$. Then plugging this into the Taylor expansion above, we have

$$f(x+\delta) = f(x) + \nabla f(x)^\top A^{-1}\Gamma + \frac{1}{2}\Gamma^\top \left(A^{-1}\right)^\top \nabla^2 f(y) A^{-1}\Gamma.$$

Remember that in gradient descent, the number of iterations T was determined by the condition number of the Hessian term. Furthermore, in the strongly convex case, we can say that T is proportional to the condition number of the matrix $\left(A^{-1}\right)^\top \nabla^2 f(y) A^{-1}$, which is the Hessian of f in the new basis. So the idea is to choose A in such a way that the condition number (which is always at least 1) is as small as possible, so that

$$(A^{-1})^\top \nabla^2 f(y) A^{-1} \approx I.$$

Suppose equality for a moment. In the discussion above, we have changed variables from δ to Γ, so the question is now about generating the right basis transformation A to make the Hessian of f in the new basis is exactly the identity:

$$\left(A^{-1}\right)^\top \nabla^2 f(y) A^{-1} = I \implies \nabla^2 f(y) = A^\top A \implies A = \left(\nabla^2 f(y)\right)^{1/2},$$

where A is defined as long as $\lambda_{\min}(\nabla^2 f(y)) \geq 0$. This A is our best change of variables.

Next, we will look to selecting the best δ direction to move in. Recall that can do this by minimising the quadratic form

$$\delta = \arg \min_{\delta \text{ s.t. } \Gamma = A\delta} \Delta f(x)^\top \cdot A^{-1}\Gamma + \frac{1}{2}\Gamma^\top\Gamma.$$

We can solve this by setting the gradient to zero, which gives us

$$\Gamma = -\eta \left(\nabla f(x)^\top A^{-1}\right)^\top = -\eta \left(A^{-1}\right)^\top \nabla f(x),$$

where η is a step size. Under our usual "nice function" assumption, Γ is symmetric and therefore so is A and A^{-1}, so we can write $\Gamma = -\eta \cdot \left(A^{-1}\right)^\top \cdot \nabla f(x) = -\eta \cdot \left(\nabla^2 f(y)\right)^{-1} \cdot \nabla f(x)$.

Claim 20.3. *The above quadratic form is optimised when $\eta = 1$.*

Proof. Recall that we have defined $\Gamma = -\eta(A^{-1})^\top \nabla f(x)$, and, by our choice of basis, we have $A = \left(\nabla^2 f(y)\right)^{1/2}$ so that $A^{-1} = \left(\nabla^2 f(y)\right)^{-1/2}$. Substituting Γ into the form $\nabla f(x)^\top A^{-1}\Gamma + \frac{1}{2}\Gamma^\top\Gamma$, we first compute

$$\nabla f(x)^\top A^{-1}\Gamma = -\eta\, \nabla f(x)^\top A^{-1}(A^{-1})^\top \nabla f(x) = -\eta\, \nabla f(x)^\top (A^{-1})^2 \nabla f(x).$$

A^{-1} is symmetric and $(A^{-1})^2 = \left(\nabla^2 f(y)\right)^{-1}$, so $\nabla f(x)^\top A^{-1}\Gamma = -\eta\, \nabla f(x)^\top \left(\nabla^2 f(y)\right)^{-1} \nabla f(x)$. Similarly,

$$\frac{1}{2}\Gamma^\top\Gamma = \frac{1}{2}\eta^2\, \nabla f(x)^\top \left((A^{-1})^\top A^{-1}\right) \nabla f(x) = \frac{1}{2}\eta^2\, \nabla f(x)^\top \left(\nabla^2 f(y)\right)^{-1} \nabla f(x).$$

Denote $c \coloneqq \nabla f(x)^\top \left(\nabla^2 f(y)\right)^{-1} \nabla f(x) \geq 0$. Then the expression becomes $-\eta\, c + \frac{1}{2}\eta^2\, c = c\left(\frac{1}{2}\eta^2 - \eta\right)$, which is a quadratic function in η. Differentiating with respect to η yields

$$\frac{\mathrm{d}}{\mathrm{d}\eta}\left[c\Big(\frac{1}{2}\eta^2 - \eta\Big)\right] = c(\eta - 1).$$

Setting the derivative equal to zero gives $c(\eta - 1) = 0$, and since $c > 0$, it follows that $\eta = 1$. Thus, the quadratic form is optimised when $\eta = 1$. □

We know $\delta = A^{-1}\Gamma$, so we plug in Γ to get $\delta = -\left(\nabla^2 f(y)\right)^{-1}\nabla f(x)$. The algorithm is summarised below:

Algorithm: NEWTON'S METHOD

- Start with $x^{(0)} \in \mathbb{R}^d$.
- For $t = 1, \dots, T$ iterations, compute $x^{(t)} = x^{(t-1)} - \left(\nabla^2 f(x^{(t-1)})\right)^{-1}\nabla f(x^{(t-1)})$.
- Return $x^{(T)}$.

Why do we describe the algorithm in this way instead? Note that there's an issue we glossed over: when we compute the best δ, it depends on the gradient of point x but also the Hessian at some point y, where y also depends on δ as well and we don't know y. Therefore, we cannot actually use the solution directly. Instead, we can assume that y is close enough to x so that $\nabla^2 f(y) \approx \nabla^2 f(x)$, so then

$$\delta = \arg\min_{\delta \text{ s.t. } \Gamma = A\delta} \nabla f(x)^\top A^{-1}\Gamma + \frac{1}{2}\left(A^{-1}\right)^\top \nabla^2 f(x) A^{-1}\Gamma = -\left(\nabla^2 f(x)\right)^{-1}\nabla f(x),$$

and this is why we describe the algorithm in the way we do.

Remark 20.4. Computing δ is harder here, because when we compute $\delta = -\left(\nabla^2 f(x)\right)^{-1}\nabla f(x)$, we need to invert the Hessian, which is a $n \times n$ matrix. This is a $O(n^\omega)$ operation, where ω is the matrix multiplication exponent, which is about 2.4. So this is a bit more expensive than computing the gradient, which is $O(n)$.

§21 Lecture 21—03rd April, 2024

Eclipse day!

Last time we finished with the guarantees for the final form of gradient descent (that we will discuss), and began discussing the heuristic for NEWTON'S method as a way to optimise functions (or precisely, the Taylor expansion thereof) by taking advantage of the extra second order information that we have about the function via the Hessian matrix.

§21.1 Guarantees for NEWTON'S method

Now we discuss the advantages gained by using NEWTON'S method.

Theorem 21.1. *Suppose there exists $r > 0$ such that every x, y is at most r away from x^*, and that*

1. $\lambda_{\min}\left(\nabla^2 f(x)\right) \geq \alpha$, *i.e. the function is α-strongly convex,*
2. $\|\nabla^2 f(x) - \nabla^2 f(y)\|_{\text{op}} \leq L\|x - y\|_2$, *where $\|\cdot\|_{\text{op}}$ is the operator norm, the largest eigenvalue of the difference of the Hessian matrices.*

Then for all $x^{(0)}$ at distance $\leq r$ away from x^*, if $x^{(1)} := x^{(0)} - \left(\nabla^2 f(x^{(0)})\right)^{-1}\nabla f(x^{(0)})$, then $\|x^{(1)} - x^*\|_2 \leq \dfrac{L}{2\alpha}\|x^{(0)} - x^*\|_2^2$.

This theorem says that under the Newton step update, the distance between the current iterate and the optimal solution will drop roughly quadratically. To clarify the convergence here, define a new variable $\gamma = 2\alpha/L$, so that the inequality above becomes

$$\left\| \frac{x^{(1)} - x^*}{\gamma} \right\|_2 \le \frac{L^2}{4\alpha^2} \cdot \|x^{(0)} - x^*\|_2^2 = \left\| \frac{x^{(0)} - x^*}{\gamma} \right\|_2^2.$$

If, say, $\dfrac{x^{(0)} - x^*}{\gamma} \le 0.9$, after T iterations, we have

$$\left\| \frac{x^{(T)} - x^*}{\gamma} \right\|_2 \le \left\| \frac{x^{(0)} - x^*}{\gamma} \right\|_2^{2^T} \le 0.9^{2^T}.$$

To make this $\le \varepsilon$, it is enough to take $T = \log_2 \left(\dfrac{\log(0.1/\varepsilon)}{\log(0.9)} \right) = O\left(\log\log \dfrac{1}{\varepsilon} \right)$ iterations.

Note that to apply the above, we need that $\|x^{(0)} - x^*\|_2 \le \min\{r, 0.9\gamma\}$, so as long as we're close enough to x^*, we can zoom in on the optimal solution extremely fast. This is called a *warm start*; Newton's method works well as long as we're close enough to the optimal solution in this sense.

§21.2 The INTERIOR POINT method

Let us go back to the LP setting where we had a problem of the form

$$\begin{aligned} \min_{x \in K} \quad & c^\top x \\ \text{s.t.} \quad & Ax \le b, \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $K = \{x : Ax \le b\}$ is a feasible set. This is a constrained optimisation problem; as we have seen in our previous considerations, we can relax the minimisation so that we have an unconstrained optimisation problem similar to that considered in gradient descent. The idea is to change the objective to penalise the constraint violations, so that we instead naively define the function to be optimised as

$$f(x) = \begin{cases} c^\top x & \text{if } x \in K, \\ +\infty & \text{otherwise,} \end{cases}$$

which is well-defined and can be computed for any x; the problem is that at the boundary ∂K of K, this isn't a nice function (it isn't even differentiable), so we cannot apply gradient descent as is and will need to find something else, a "smoother" function that approximates this one. So rather than having such a sharp penalty, we can instead consider a function that is smooth and differentiable; one classic choice is

$$F(x) = \begin{cases} < \infty & \text{if } x \in K, \\ +\infty & \text{if } x \notin K, \end{cases} \qquad \text{such that } \lim_{x \to \partial K} F(x) = +\infty.$$

We call such a function a *barrier function*; it is a smooth approximation of the indicator function of K. Using ideas from usual algebra, we can incorporate this barrier function into the objective function in this way: for a parameter $\eta > 0$, we define the new objective function as

$$f_\eta(x) = \eta c^\top x + F(x).$$

Note that $f_0(x) = F(x)$, which depends on the feasible set but not in the direction we seek to optimise. Also observe that as $\eta \to \infty$, $f_\eta/\eta \to c^\top x$ for $x \in K$ and $+\infty$ otherwise. This is a way to interpolate between the original objective function and the barrier function. One possible choice for the barrier function is the logarithmic barrier function, which is defined as

$$F(x) = \log\left(\prod_{i=1}^{m} \frac{1}{b_i - A_i x}\right) = -\sum_{i=1}^{m} \log(b_i - A_i x),$$

where A_i is the i-th row of A and b_i is the i-th element of b. So our goal is to optimise

$$x_\eta^* = \arg\min_{x\in\mathbb{R}^n} f_\eta(x) = \arg\min_{x\in\mathbb{R}^n} \eta c^\top x - \sum_{i=1}^{m} \log(b_i - A_i x) \xrightarrow{\eta\to\infty} \arg\min_{x\in K} c^\top x.$$

We will be applying gradient-type methods to this problem, so we must show at least that the function is convex.

Claim 21.1. *The function $f_\eta(x)$ is convex.*

Proof. It suffices to show that the Hessian $\nabla^2 f_\eta$ is positive semidefinite, i.e. for all $x \in \mathbb{R}^n$, $v^\top \nabla^2 f_\eta(x) v \geq 0$ and all eigenvalues are nonnegative. We have

$$f_\eta(x) = \eta c^\top x - \sum_{i=1}^{m} \log(b_i - A_i x), \quad \nabla f_\eta(x) = \eta c^\top + \sum_{i=1}^{m} \frac{A_i}{b_i - A_i x}, \quad \nabla^2 f_\eta(x) = \sum_{i=1}^{m} \frac{A_i^\top A_i}{(b_i - A_i x)^2}.$$

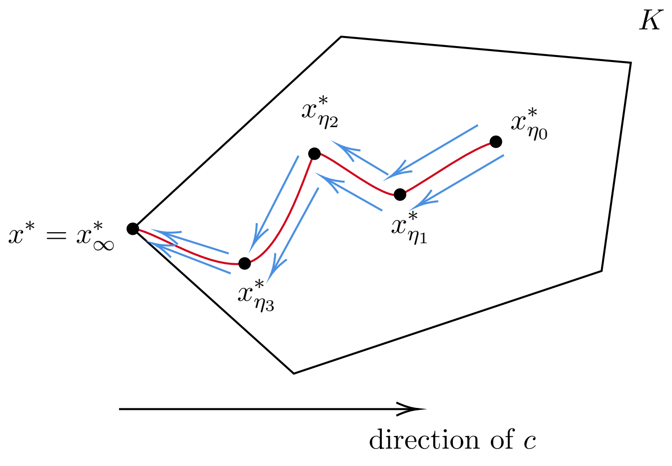
For any y arbitrary vector, we have

$$y^\top \nabla^2 f_\eta(x) y = \sum_{i=1}^{m} \frac{(A_i y)^\top A_i y}{(b_i - A_i x)^2} = \sum_{i=1}^{m} \frac{\|A_i y\|_2^2}{(b_i - A_i x)^2} \geq 0,$$

so the Hessian is positive semidefinite, and f_η is convex. □

Remark 21.2. If the rows of the matrix A span $\mathbb{R}^n$—that is, $\mathsf{vol}(K) > 0$, then $y^\top \nabla^2 f_\eta(x) y > 0$ for all $y \neq 0$, that is, f_η is strongly convex.

One can think of the optimal values as sequences of functions in η that converge to the optimal value of the original problem from a point called the *analytic centre*.

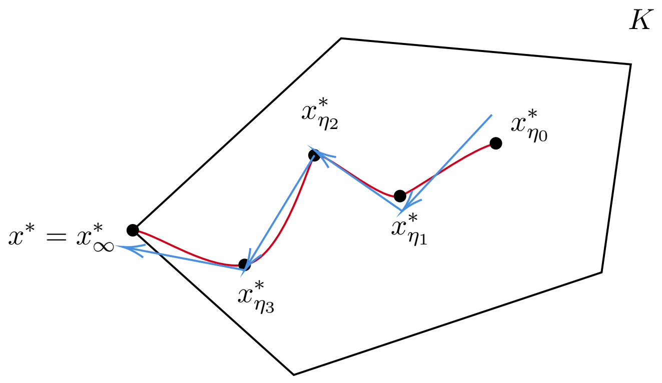


Definition 21.3. *We say tha x_0^* is the* analytic centre *of K if $x_0^* = \arg\min_{x \in K} f_0(x)$. Furthermore, the set $\{x_\eta^*\}_{\eta>0}$ is called the* central path *of the LP.*

The naive algorithm is to solve for x_η^* for very large η via Newton's method (and not gradient descent due to the issue with the condition number which we raised before). The problem is that Newton's method needs a warm start, and we don't have a good warm start for this problem. The main intuition is to walk along the central path (the path of optimal solutions) using the previous x_η^* as a warm start for the next $x_{\eta'}^*$, where $\eta' > \eta$. This is the idea behind the interior point method.

Algorithm: INTERIOR-POINT-1

- Start at $x_{\eta_0}^* \in \mathbb{R}^n$ for some $\eta_0 > 0$, which is very close to x_0^*, which we assume is known. `// finding` x_0^* `is a nontrivial task`
- For $t = 1, \dots, T$ iterations, let $\eta_{t+1} = \eta_t(1+\alpha)$ for small $\alpha > 0$; compute $x_{\eta_{t+1}}^*$ using NEWTON's method with warm start $x_{\eta_t}^*$.
- Return the stopping point $x_{\eta_T}^*$ as the approximate solution to the LP.



But must we always compute the optimal values in the intermediate steps if we only care about the final solution? The answer is no:

Claim 21.4. *The value $x_{\eta_{i+1}}^*$ is contained within the radius of convergence of Newton's method around $x_{\eta_i}^*$ for all i. In particular, these values are separated according to a small perturbation distance.*

Proof. Define the function $F(\eta, x) := \nabla f_\eta(x) = \eta\, c - \sum_{j=1}^m \frac{A_j^\top}{b_j - A_j x}$. By definition, the optimal point x_η^* satisfies $F(\eta, x_\eta^*) = 0$. Since f_η is twice continuously differentiable and strongly convex on the interior of $K = \{x : Ax \le b\}$, its Hessian $\nabla^2 f_\eta(x) = \sum_{j=1}^m \frac{A_j^\top A_j}{(b_j - A_j x)^2}$ is positive definite. In particular, the Jacobian with respect to x, $\nabla_x F(\eta, x) = \nabla^2 f_\eta(x)$, is invertible for all $x \in \mathrm{int}(K)$.

Thus, by the implicit function theorem, there exists a continuously differentiable mapping $x^*(\eta)$ defined in a neighborhood of any fixed η_i, such that $F(\eta, x^*(\eta)) = 0$ and $x^*(\eta_i) = x_{\eta_i}^*$.

Differentiating the identity $F(\eta, x^*(\eta)) = 0$ with respect to η yields

$$\frac{\partial F}{\partial \eta}(\eta, x^*(\eta)) + \nabla_x F(\eta, x^*(\eta)) \frac{\mathrm{d}x^*(\eta)}{\mathrm{d}\eta} = 0.$$

Noting that
$$\frac{\partial F}{\partial \eta}(\eta, x) = c \implies \frac{\mathrm{d}x^*(\eta)}{\mathrm{d}\eta} = -\Big(\nabla^2 f_\eta(x^*(\eta))\Big)^{-1} c,$$
we obtain by the mean value theorem that for some ξ between η_i and η_{i+1} we have
$$\|x^*_{\eta_{i+1}} - x^*_{\eta_i}\| = \|x^*(\eta_{i+1}) - x^*(\eta_i)\| \le \left\|\frac{\mathrm{d}x^*(\xi)}{\mathrm{d}\eta}\right\| |\eta_{i+1} - \eta_i| \le \left\|\Big(\nabla^2 f_\xi(x^*(\xi))\Big)^{-1}\right\| \|c\| \, |\Delta\eta|.$$
By choosing $|\Delta\eta|$ (or, equivalently, a small multiplicative increase, e.g., $\eta_{i+1} = \eta_i(1+\alpha)$ with small $\alpha > 0$) sufficiently small, we can ensure that $\|x^*_{\eta_{i+1}} - x^*_{\eta_i}\| \le \delta$, where δ is the radius of convergence for Newton's method about $x^*_{\eta_i}$. Thus, $x^*_{\eta_{i+1}}$ lies within the region where the quadratic convergence is guaranteed. $\square$

With this fact, we can present a faster algorithm:

Algorithm: INTERIOR-POINT-2

- Start at $x^*_{\eta_0} \in \mathbb{R}^n$ for some $\eta_0 > 0$, which is very close to x^*_0, which we assume is known.
- For $t = 1, \ldots, T$ iterations, let $\eta_{t+1} = \eta_t(1+\alpha)$ for small $\alpha > 0$ at step $t+1$ only; compute $x_{\eta_{t+1}}$ using one step of NEWTON's method with warm start x_{η_t}.
- Return the stopping point $x^*_{\eta_T}$ as the approximate solution to the LP.

Clearly the approximation error is cruder than in the first algorithm—as we only take one step of Newton's method and within the radius of convergence, the error is small enough for some purposes. However, the algorithm is much faster.

§22 Lecture 22—10th April, 2024

§22.1 The stopping criterion and starting point for the INTERIOR POINT method

We begin with an analysis of the terminal condition η_T.

Claim 22.1. *With all the assumptions and notations of the previous lecture, $c^\top x_\eta - c^\top x^* \le m/\eta$.*

Proof. By the definition of x^*_η (the minimum of the function under consideration), the gradient convex function $\nabla f_\eta(x^*) = 0$, so that $\eta c + \nabla f_\eta(x^*) = 0$. Thus, $c = -\nabla f_\eta(x^*)/\eta$. Take any $x, y \in K$ the feasible set, and apply the convexity of f_η to get
$$\begin{aligned}\nabla f_\eta(x)^\top (y - x) &= \sum_{i=1}^m \frac{A_i}{b_i - A_i x} \cdot (y - x) = \sum_{i=1}^m \frac{A_i y - A_i x}{b_i - A_i x} \\ &= \sum_{i=1}^m \frac{b_i - A_i x - (b_i - A_i y)}{b_i - A_i x} = \sum_{i=1}^m 1 - \frac{b_i - A_i y}{b_i - A_i x} \le m.\end{aligned}$$
Therefore with $x = x^*_\eta$, and $y = x^*$, we have
$$\nabla f_\eta(x^*_\eta)^\top (x^* - x^*_\eta) \le m \implies \frac{c^\top x^* - c^\top x^*_\eta}{\eta^{-1}} \le m \implies c^\top x^* - c^\top x^*_\eta \le m/\eta. \qquad \square$$

Corollary 22.2. *To achieve $\pm\varepsilon$ additive error, set $m/\eta_T = \varepsilon \implies \eta_T = m/\varepsilon$. Equivalently,*

$$T = \Theta\left(\frac{1}{\alpha} \cdot \log \frac{m}{\varepsilon\eta_0}\right) = O\left(\log_{1+\alpha} \frac{m}{\varepsilon\eta_0}\right),$$

and it suffices to take $\alpha = 1/\text{poly}(n, m)$.

Now we consider the starting point $x*_0$. This is a doubly hard task; this point must (1) be in the feasible set K (2) and be its true analytic centre. To make this easier, we divide the task into two halves. First, we assume there is some x strictly in the interior of K such that x is the analytic centre of K. for a particular arrangement of f_η.

Claim 22.3. *For all $x \in K \setminus \partial K$, there exists $\eta. \geq 0$ and $c' \in R^n$ such that x is equal to $x^*_{\eta'}$ for $f'_{\eta'} = \eta' {c'}^\top x + F(x)$.*

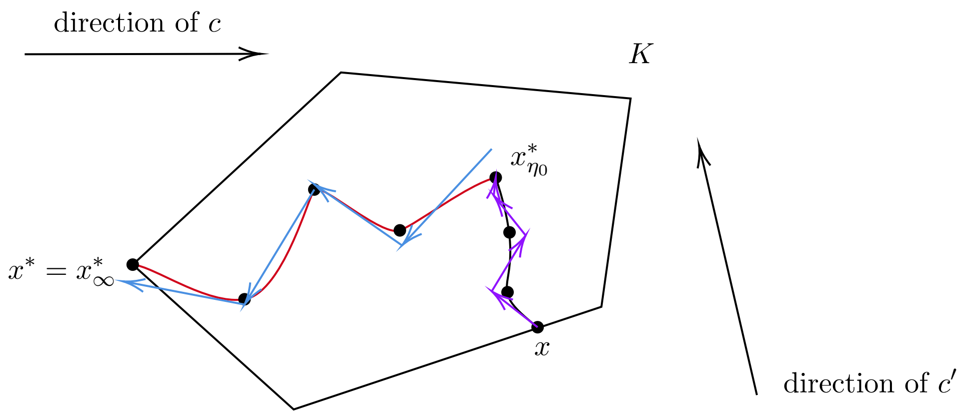
Proof. We want $\nabla f'_{\eta'}(x) = 0$. But we already have $\nabla f_{\eta'}(x) = \eta' c' + \nabla F(x) = 0 \implies c' = -\eta'^{-1} \nabla F(x)$. So we can merely fix $\eta' = 1$ and find $c' = -\nabla F(x)$. This is possible because x is strictly in the interior of K. □

Now we can give an algorithm for finding the analytic centre x^*_0 of K from x, simply by walking back along the central path.

Algorithm

- Given $x \in K \setminus \partial K$, compute $c' = -\nabla F(x)$ with $\eta' = 1$.
- At iteration $t = 1, \ldots, T$, take the Newton step on $\eta_t = \eta_{t-1}(1 - \alpha)$, and set x_t to be the result of the NEWTON method applied to f'_{η_t} with warm start x_{t-1}, where $x_0 = x$.
- Return $x_T \approx x^*_{\eta_0} \in \mathbb{R}^n$.

The purple path below shows the trajectory of the algorithm.



Now we move to the second step, finding some $x \in K \setminus \partial K$. (We only assumed its existence before.) We know that checking the feasibility of the linear program is not a simpler question than solving the linear program itself. So we don't expect this step to be simpler, and it isn't. However, we have a neat idea; we can reduce it to solve another LP where finding $x \in K \setminus \partial K$ is much easier. The new LP is as follows:

$$\begin{aligned} \min \quad & t \\ \text{s.t.} \quad & A_i x \leq b_i + t, \quad i = 1, \dots, m \\ & t \in \mathbb{R}, x \in \mathbb{R}^n. \end{aligned}$$

Suppose t^* is the optimal value of this new LP. If $t^* > 0$, then $K \cap \{x : A_i x \leq b_i\} \neq \emptyset$, so $K = \emptyset$. If $t^* < 0$, then x is a point in the interior of K. If $t^* = 0$, then $K = \partial K$, which we don't need to worry about. So for this linear program, a feasible solution can be $x = 0$ and $t = \max_{i=1,\dots,m} -b_i + 1$ large enough.

Claim 22.4. *It is enough to take $\alpha = \Theta(1/\sqrt{m})$ to achieve ε additive error in the* INTERIOR POINT *method.*

In general, the INTERIOR POINT method is a very powerful tool for solving linear programs in large scale; it performs better than the simplex / ellipsoid method, and the number of steps has been improved to something on the order of $\widetilde{O}(n)$.

§22.2 Multiplicative weights and the WEIGHTED MAJORITY algorithm

Suppose we wish to determine whether it will rain or not on a given day, and we have access to n weather experts who are willing to give us their answers to the question "will it rain tomorrow?" on every day over the course of T days. For each $i \in [n]$, define an indicator

$$f_i^{(t)} = \begin{cases} 1 & \text{if expert } i \text{ on day } t \text{ is wrong,} \\ 0 & \text{otherwise.} \end{cases}$$

We are not experts in meteorology, so we don't know the true probability of rain on any given day or how to even estimate it. However, we know that the predictions of the experts should be better than ours, so if there is some expert that is pretty good, we want to narrow down on that expert only and make as many errors as that expert does.

Definition 22.5 (Best expert)**.** *Let $m_i^{(t)} = \sum_{s=1}^t f_i^{(s)}$ be the number of mistakes expert i has made up to day t, and let $M^{(t)}$ be the number of mistakes we make up to day t. The best expert i is the expert who has made the fewest mistakes up to day t, i.e., $i = \arg\min_{j \in [n]} m_j^{(t)}$.*

Our goal, the best we can hope for, is to make $M^{(t)}$ as close as possible to the number of errors of the best expert, i.e. $M^T \leq \min_{i \in [n]} m_i^{(T)}$. We can describe some bad algorithms towards achieving this goal:

1. *Follow the majority:* This is basically just mimicking the majority prediction of whether it is going to rain today. Obviously this is a bad idea, because there could be $n - 1$ wrong experts and one correct expert everyday, and we would make the wrong decision every time.
2. *Follow the expert who was correct last time:* This is still bad, obviously. Suppose that we have two experts, and measured over four days. Expert 1 says that it will rain only on

odd-numbered days and expert 2 says that it will rain only on even-numbered days. However, the truth could be that it actually never rains over the four days. If we follow the expert who was correct last time starting from day 2, we'd go with expert 2 who predicts rain, but it was sunny. Then we would switch to expert 1 who predicts rain, but it was sunny again. We would keep switching between the two experts and never make even a single near-accurate prediction.

Our intuition now is that we shouldn't trust any single expert, but rather we should try to adapt to the experts who tend to be more correct. So we want to use historical data on experts to determine our decision, penalising experts who tend to be wrong and listening more to experts who tend to be right. This idea leads to the WEIGHTED MAJORITY algorithm.

```
Algorithm: WEIGHTED-MAJORITY

• For $i = 1, \dots, n$, for day $t = 0$ assign each expert a weight $w_i^{(0)} = 1$.
  // Initially, we trust all experts equally.
• For $t = 1, \dots, T$:
    – Define the sums
      $$\sigma_R^{(t)} = \sum_{i \text{ s.t. expert } i \text{ says rain}} w_i^{(t)} \quad \text{and} \quad \sigma_S^{(t)} = \sum_{i \text{ s.t. expert } i \text{ says sun}} w_i^{(t)}.$$
    – If $\sigma_R^{(t)} \geq \sigma_S^{(t)}$, predict it will rain; otherwise, predict it will be sunny.
      // We take a majority prediction of the weights.
    – For each $t \geq 0$, set $w_i^{(t+1)} = w_i^{(t)}(1 - \varepsilon f_i^{(t)})$ for all $i \in [n]$, where $\varepsilon$ is t.b.d.
      // We penalise only the experts who are wrong by a factor of $1 - \varepsilon$.
```

We have an immediate guarantee on this algorithm.

Theorem 22.1. *For all $i \in [n]$ and $\varepsilon \in (0, 1/2)$, we have that the weighted majority algorithm yields*

$$M^{(T)} \leq 2(1+\varepsilon)m_i^{(T)} + \frac{2\ln n}{\varepsilon} \implies M^{(T)} \leq 2(1+\varepsilon)\min_{i\in[n]} m_i^{(T)} + \frac{2\ln n}{\varepsilon}.$$

Note that we cannot be strictly multiplicative because of the initial condition; we have the two sources of error in $1+\varepsilon$ and $1/\varepsilon$. Before we discuss how this result relates to broader considerations about multiplicative weights and optimisation, we will prove the theorem.

Proof of Theorem 22.1. The idea for the proof is to track a suitably selected "potential function" that will allow us to bound the number of mistakes made per round. For $t = 0, 1, \dots, T$, define the potential function

$$\Phi_t = \sum_{i=1}^{n} w_i^{(t)},$$

so that $\Phi_0 = n$. We will prove the following lemma.

Lemma 22.6. *For $t \geq 0$, we have that $\Phi_{t+1} \geq w_i^{(t+1)} = (1-\varepsilon)^{m_i^{(t)}}$.*

Proof. The inequality follows by definition, since $w_i^{(t)} \geq 0$ for all $i \in [n]$. The weight of the (perpetually wrong) expert i at time $t+1$ is $w_i^{(0)} = 1$ multiplied by $1-\varepsilon$ for each mistake made by expert i up to time t; since we only decrease the weight of expert i when they make a mistake, we have $w_i^{(t+1)} = (1-\varepsilon)^{m_i^{(t)}}$. $\square$

Next, we will look to relate the weights of the predictions to our the number of mistakes we make, by means of the following lemma.

Lemma 22.7. *For $t \geq 0$, we have that*

$$\Phi_{t+1} \leq n\left(1-\frac{\varepsilon}{2}\right)^{M^{(t)}}.$$

Proof. For each $t \geq 0$, note that if we choose the correct action on day t, then clearly $\Phi_{t+1} \leq \Phi_t$, since by definition we have $w_i^{(t+1)} \leq w_i^{(t)}$ for all $i \in [n]$ with equality if and only if $f_i^{(t)} = 0$ for all $i \in [n]$. This inequality is tight, since it may be possible that all experts choose correctly on day t. However, suppose we chose wrong. Then by our algorithm, the weighted majority made the wrong decision. By our algorithm, this meaans that the weighted majority must have chosen the wrong decision. Suppose that the set of experts was $A \subset [n]$; since it was the weighted majority, we have that

$$\sum_{i\in A} w_i^{(t)} \geq \sum_{i\in[n]\setminus A} w_i^{(t)} \geq \frac{1}{2}\sum_{i\in[n]} w_i^{(t)} = \frac{\Phi_t}{2}.$$

Moreover, note that for each expert $i \in A$, since it chose wrong, we decrease its weight, and for each expert $i \notin A$, we keep its weight the same. Therefore, we have that

$$\begin{aligned}
\Phi_{t+1} &= \sum_{i=1}^{n} w_i^{(t+1)} \leq \sum_{i\in A} w_i^{(t+1)} + \sum_{i\notin A} w_i^{(t+1)} \\
&\leq \sum_{i\in A} w_i^{(t)}(1-\varepsilon) + \sum_{i\notin A} w_i^{(t)} = \sum_{i\in A} w_i^{(t)}(1-\varepsilon) + \sum_{i\notin A} w_i^{(t)} \\
&= \sum_{i=1} w_i^{(t)} - \varepsilon\sum_{i\in A} w_i^{(t)} = \Phi_t - \varepsilon\sum_{i\in A} w_i^{(t)} \\
&\leq \Phi_t - \frac{\varepsilon}{2}\Phi_t = \left(1-\frac{\varepsilon}{2}\right)\Phi_t.
\end{aligned}$$

Continuing this argument, we have that

$$\Phi_{t+1} \leq \left(1-\frac{\varepsilon}{2}\right)\Phi_t \leq \left(1-\frac{\varepsilon}{2}\right)^2\Phi_{t-1} \leq \ldots \leq \left(1-\frac{\varepsilon}{2}\right)^{M^{(t)}}\Phi_0 = n\left(1-\frac{\varepsilon}{2}\right)^{M^{(t)}},$$

which completes the proof. $\square$

Finally, we can combine the two lemmas to prove the theorem. We have that for each $t \geq 0$,

$$(1-\varepsilon)^{m_i^{(t)}} \leq \Phi_{t+1} \leq n\left(1-\frac{\varepsilon}{2}\right)^{M^{(t)}} \leq ne^{-\varepsilon M^{(t)}/2},$$

where the last inequality follows from the fact that $1-x \leq e^{-x}$ for all $x \in \mathbb{R}$. Noting that

$$\ln(1-\varepsilon) \geq -\varepsilon-\varepsilon^2, \qquad \varepsilon \in (0, 1/2),$$

we obtain

$$\begin{aligned}(1-\varepsilon)^{m_i^{(t)}} \le ne^{-\varepsilon M^{(t)}/2} &\implies m_i^{(t)}\ln(1-\varepsilon) \le \ln n - \frac{\varepsilon}{2}M^{(t)} \\ &\implies -m_i^{(t)}\varepsilon(1+\varepsilon) \le \ln n - \frac{\varepsilon}{2}M^{(t)} \\ &\implies M^{(t)} \le 2(1+\varepsilon)m_i^{(t)} + \frac{2\ln n}{\varepsilon}.\end{aligned}$$

This holds for any $t \ge 0$, so we can take $t = T$ to obtain the desired result. $\square$

§23 Lecture 23—10th April, 2024

§23.1 The multiplicative weights update method: algorithm and analysis

Recall the model we've been discussing: there are n experts, and on the t-th day, the function

$$f_i^{(t)} = \begin{cases} 1 & \text{if expert } i \text{ is wrong on day } t \\ 0 & \text{otherwise} \end{cases}$$

is revealed. We have $m_i^{(T)} = \sum_{t=1}^T f_i^{(t)}$, and our cumulative mistakes are given by $M^{(T)} = \sum_{t=1}^T f_{i_t}^{(t)}$ where i_t is the expert we choose on day t. We wish to minimize $M^{(T)}$.

To get the multiplicative weights update, we can assume that for each day t and expert i, $f_i^{(t)}$ isn't just an indicator variable, but it is generalising the "loss" of expert i on day t, within the range $[-1,+1]$. It turns out that with this new formulation, we can reduce the factor of $2(1+\varepsilon) \approx 2$ in the regret bound we proved last time; the only remaining technique is using *randomisation.*

Algorithm: MWU

- For $i = 1, \dots, n$, for day $t = 0$ assign each expert a weight $w_i^{(0)} = 1$.
  `// Initially, we trust all experts equally.`
- For $t = 1, \dots, T$:
  - For each expert i,
  $$p_i^{(t)} = \frac{w_i^{(t)}}{\sum_{j=1}^n w_j^{(t)}}.$$
  - Choose expert i from the distribution $(p_1^{(t)}, \dots, p_n^{(t)})$ and follow this expert.
    `// Select an expert with probability proportional to its weight.`
  - After learning $f_i^{(t)}$, set $w_i^{(t+1)} = w_i^{(t)}(1-\varepsilon f_i^{(t)})$ for all $i \in [n]$, where ε is t.b.d.
    `// We penalise the experts who are wrong, and reward the experts who are correct.`

How do we understand the mistakes we're making? For $t \in [T]$, define $p^{(t)} = (p_1^{(t)}, \dots, p_n^{(t)})$ and

$f^{(t)} = (f_1^{(t)}, \dots, f_n^{(t)})$. Then, the expected number of mistakes we make on day t is

$$M^{(T)} = \sum_{t=1}^{T} \sum_{i=1}^{n} p_i^{(t)} f_i^{(t)} = \sum_{t=1}^{T} \langle p^{(t)}, f^{(t)} \rangle,$$

and of course we have $m_i^{(t)} = \sum_{j=1}^{T} f_i^{(j)}$ (note that these are both expectations). We can now prove the following theorem.

Theorem 23.1. *For all $i \in [n]$ and $\varepsilon \in (0, 1/2)$, the* MWU *algorithm satisfies, for any T,*

$$M^{(T)} \le m_i^{(t)} + \frac{\ln n}{\varepsilon} + \varepsilon T.$$

Proof. Define $Q^{(T+1)} = \sum_{i \in [n]} w_i^{(T+1)}$. Then

$$\begin{aligned} Q^{(T+1)} &= \sum_{i \in [n]} w_i^{(T)} (1 - \varepsilon f_i^{(T)}) = \sum_{i \in [n]} w_i^{(T)} - \varepsilon \sum_{i \in [n]} w_i^{(T)} f_i^{(T)} = \sum_{i \in [n]} w_i^{(T)} - \sum_{i \in [n]} Q^{(T)} p_i^{(T)} \varepsilon f_i^{(T)} \\ &= Q^{(T)} - \varepsilon \langle Q^{(T)} p^{(T)}, f^{(T)} \rangle = Q^{(T)} (1 - \varepsilon \langle p^{(T)}, f^{(T)} \rangle). \end{aligned}$$

Using the bound $1 - x \le e^{-x}$, the last inequality gives that

$$\begin{aligned} Q^{(T+1)} = \sum_{i \in [n]} w_i^{(T+1)} &\le Q^{(T)} e^{-\varepsilon \langle p^{(T)}, f^{(T)} \rangle} = Q^{(0)} e^{-\varepsilon \sum_{i=1}^{n} p_i^{(T)} f_i^{(T)}} \\ &= n e^{-\varepsilon \sum_{i=1}^{n} p_i^{(T)} f_i^{(T)}} = n e^{-\varepsilon \langle p^{(T)}, f^{(T)} \rangle} \\ &= n e^{-\varepsilon M^{(T)}}. \end{aligned}$$

On the other hand, we have that $Q^{(T+1)} = \sum_{i \in [n]} w_i^{(T+1)} \ge \max_{i \in [n]} w_i^{(T+1)} \ge w_i^{(T+1)}$, and so following the algorithm gives that

$$w_i^{(T+1)} \ge \prod_{t=1}^{T} (1 - \varepsilon f_i^{(t)}) w_i^{(0)} = \prod_{t=1}^{T} (1 - \varepsilon f_i^{(t)}).$$

Given that $\prod_{t=1}^{T} (1 - \varepsilon f_i^{(t)}) \le Q^t \le e^{-\varepsilon \sum_{t=1}^{T} f_i^{(t)}}$, we have that

$$\varepsilon M^{(T)} \le \ln n - \sum_{t=1}^{T} \ln(1 - \varepsilon f_i^{(t)})$$

for all $i \in [n]$. But $-\ln(1 - x) \le x + x^2$ for all $x \in [0, 1/2)$, and so

$$M^{(T)} \le \frac{\ln n}{\varepsilon} + \sum_{t=1}^{T} \varepsilon f_i^{(t)} + \sum_{t=1}^{T} \varepsilon^2 f_i^{(t)} \le \frac{\ln n}{\varepsilon} + \varepsilon \sum_{t=1}^{T} f_i^{(t)} + \varepsilon T,$$

since $\varepsilon \le 1/2$ and $(f_i^{(t)})^2 \le 1$ for all $i \in [n]$ and $t \in [T]$. $\square$

Remark 23.1. 1. The theorem above gives a bound on the expected number of mistakes we make, but we can also get a bound on the expected number of mistakes we make in the worst case. This is done by using the fact that $f_i^{(t)} \in [-1, 1]$ for all $i \in [n]$ and $t \in [T]$.

2. Note that the result immediately implies that $M^{(T)} \leq \varepsilon^{-1} \ln n + (1+\varepsilon) \min_{i \in [n]} m_i^{(T)}$.

The discussion about multiplicative weights is part of the broader topic of online learning and bandit problems. In the context of online learning, we can think of the experts as being the actions we can take, and the loss of each expert as the reward we get for taking that action.

§23.2 Multiplicative weights update for solving LPs approximately

Now we present a generic framework using the multiplicative weights update method to solve linear program feasibility approximately.

Problem 23.2. *Determine whether there exists a vector $x \in \mathbb{R}^n$ such that $Ax \geq b$.*

We will relax this feasibility problem by the following approximative version.

Problem 23.3. *Distinguish between the cases for which there exists $x \in \mathbb{R}^n$ such that $Ax \geq b - \varepsilon \mathbf{1}_m$, and the case for which there does not exist any x such that $Ax \geq b$.*

To solve the relaxed version of this problem, we will use a clever agent mechanism; towards this assume we have an access to an oracle $\mathcal{O}$ which "solves the original LP on average", defined as follows.

Definition 23.4 (Oracle for solving LPs approximately)**.** *Say that $\mathcal{O}_{A,b}$ is an oracle that, given as input a distribution $p \in \mathbb{R}^m$ over constraints, returns x if there exists $x \in \mathbb{R}^n$ such that $\langle p, Ax \rangle \geq \langle p, b \rangle$, and returns $\perp$ otherwise.*

Note that if there exists a solution to our initial problem, it is desirable for the oracle to output it or something close to it. We can now present the algorithm for solving the relaxed version of the LP feasibility problem.

Theorem 23.2. *There is an algorithm* MWU-LP *that, given an oracle $\mathcal{O}_{A,b}$, solves the relaxed LP feasibility problem to an ε-additive error using $O(\varepsilon^{-2} \log n)$ calls to the oracle.*

Proof. The idea is just to run an MWU scheme where every inequality plays the role of an expert. We will maintain a distribution $p^{(t)}$ over the constraints at each time t, and we will query the oracle with this distribution. The algorithm is as follows.

```
Algorithm: MWU-LP

  • Start with $w_i^{(0)} = 1$ and $p_i^{(0)} = 1/m$ for all $i \in [m]$, where $m$ is the number of different
    inequalities in the system.
  • For $t = 1, \dots, O(\varepsilon^{-2} \log n)$:
      – Set $p^{(t)} = (p_1^{(t)}, \dots, p_m^{(t)})$, where $p_i^{(t)} = \frac{w_i^{(t)}}{\sum_{j=1}^m w_j^{(t)}}$.
      – Query the oracle $\mathcal{O}_{A,b}$ with $p^{(t)}$ to get $x^{(t)}$. If $\mathcal{O}_{A,b}(p^{(t)}) \in \mathbb{R}^n$, set $x^{(t)} = \mathcal{O}_{A,b}(p^{(t)})$;
      otherwise, output "no solution".
    – Set $f_i^{(t)} = A_i x^{(t)} - b_i$ for all $i \in [m]$ to be the error made by the $i$-th constraint at
      iteration $t$. // This is how far we are from satisfying the constraint.
      Note that the easily satisfied constraints are less important, so
      the loss function's value is larger.
    – Set $w_i^{(t+1)} = w_i^{(t)}(1 - \varepsilon f_i^{(t)})$ for all $i \in [m]$.
  • Return the average $\overline{x} = \frac{1}{T}\sum_{t=1}^{T} x^{(t)}$.
```

Now we argue its correctness. Note that without loss of generality, $f_i^{(t)} \in [-1, +1]$ by the guarantees of the oracle; indeed, suppose we can solve $\mathcal{O}_{A,b}$ and get $\max_{i\in[n]} |A_i x - b_i| \leq k$ and $Ax \geq b$. Then $\max_{i\in[n]} |(A_i/k)x - b_i/k| \leq 1$ and $(A/k)x \geq b/k$. This can be solved approximately as well to get $Ax \geq b - \varepsilon k \mathbf{1}_m$, and would require $T = O(k^2\varepsilon^{-2}\log n)$ calls to the oracle. Check that

$$\begin{aligned}
M^{(T)} &= \sum_{i=1}^{(T)} \langle p^{(t)}, f^{(t)} \rangle = \sum_{t=1}^{T}\sum_{i=1}^{m} p_i^{(t)} f_i^{(t)} = \sum_{t=1}^{T} \langle p^{(t)}, f^{(t)} \rangle \\
&= \sum_{t=1}^{T} \langle p^{(t)}, Ax^{(t)} - b \rangle = \sum_{t=1}^{T} \langle p^{(t)}, Ax^{(t)} \rangle - \sum_{t=1}^{T} \langle p^{(t)}, b \rangle \\
&= \sum_{t=1}^{T} \langle p^{(t)}, Ax^{(t)} \rangle - \sum_{t=1}^{T} \langle p^{(t)}, b \rangle = \sum_{t=1}^{T} \Big(\langle p^{(t)}, Ax^{(t)} \rangle - \langle p^{(t)}, b \rangle \Big) \\
&\geq 0,
\end{aligned}$$

where the last inequality follows from the fact that the oracle is correct. Now we know that if $T = O(\varepsilon^{-2}\log n)$, then the MWU guarantee becomes $M^{(T)} \leq \min_{i\in[m]} m_i^{(T)} + 2\varepsilon T$, and so we have that

$$m_i^T \geq -2\varepsilon T \implies \sum_{t=1}^{T} f_i^{(t)} \geq -2\varepsilon T \implies \sum_{t=1}^{T} A_i x^{(t)} - b_i \geq -2\varepsilon T,$$

and moving things around yields

$$A_i \sum_{t=1}^{T} x^{(t)} \geq T b_i - 2\varepsilon T(\mathbf{1}_m)_i \implies A_i \overline{x} \geq b_i - 2\varepsilon(\mathbf{1}_m)_i.$$

This completes the proof. □

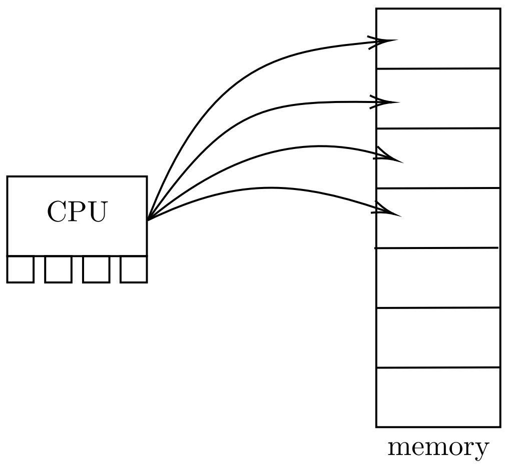
§24 Lecture 24—15th April, 2024

§24.1 Large-scale models

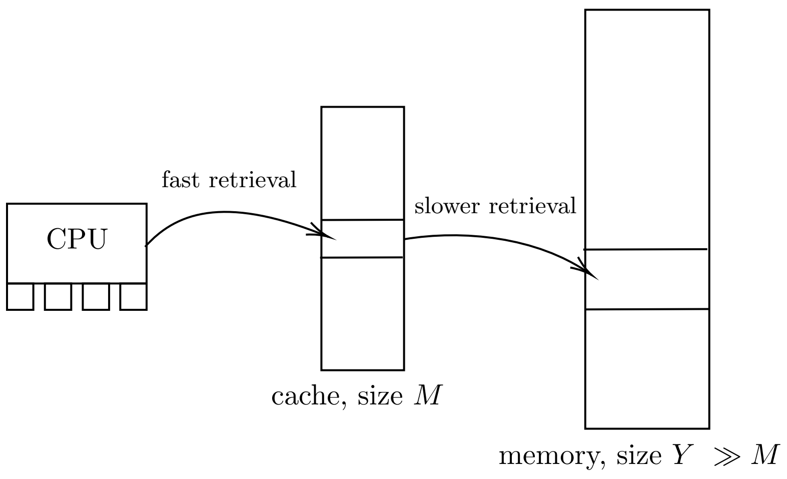
Now we will switch gears completely and take a slightly different perspective on what we have been discussing so far. In particular, we will talk more about models of computation and how to select the model that helps us solve rather standard problems with large data sets.

§24.1.1 The I/O model

For a large part of the class, we have been using the standard RAM model of computation, where we have the CPU serving as the primary computational engine with a constant number of registers, and the data (graphs, arrays, etc.) stored in a large memory unit containing a bunch of cells each of which can be accessed in constant time. The CPU then can read/write to these cells and use the data to perform computations.



Although this may be useful from the algorithmicist's or programmer's perspective, this model is far too primitive and unrealistic for large-scale data sets. Moving closer to reality, we have the cache model below:



The CPU interacts with the memory through a cache, which is a smaller, faster memory unit that stores a subset of the data from the larger memory. The cache is designed to speed up data retrieval by keeping frequently accessed data closer to the CPU. But sometimes the data we need is not in the cache, in which case we have to retrieve it from the larger memory, which is slower (and the difference between "fast" and "slow" can be up to several orders of magnitude).

Sometimes the above happens over multiple sublevels (with a disc, distributed storage, etc.). In any case, our gaol is to design algorithms that are efficient in terms of the number of I/O operations (i.e., data transfers between the cache and the memory).

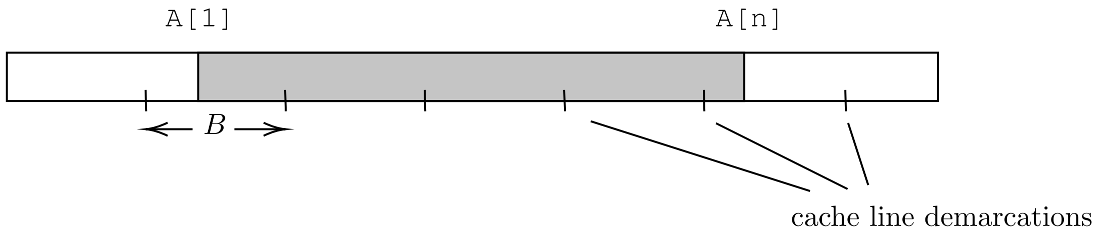
Like we said, the cache has several *cache lines* or *blocks*, which are contiguous segments of memory of size $B \leq M$ (each cache line can hold B words). If the accessed memory location is not in the cache, then we have a *cache line miss* and need to retrieve the cache line from memory first. We write CLMs to denote the number of cache line misses. We will also need some kind of replacement policy that specifies which cache line to evict when a new cache line is brought in; some of those policies include:

1. LRU (least recently used): evict the cache line that was accessed the longest time ago;
2. FIFO (first in, first out): evict the cache line that has been in the cache the longest;
3. Random: evict a random cache line.

Note further that every cache line is either in the cache already or in memory, so we can always find it in one of the two places (we're ignoring situations with data faults or corruption, and we're working in fully non-parallel environments). Finally, our notion of efficiency will be in terms of the number of cache line misses; we discount the cost of accessing data in the cache since it is so cheap that it is negligible compared to the cost of accessing data in memory.

Now we will go through some examples of algorithms in the I/O model.

(A1) Scanning and summing the entries of an array `A[1,...,n]` Suppose we have an array of n elements loaded in memory:



The processor reads from left to right, one element at a time. Every time we access an element, we check if it is in the cache. If it is, we can read it quickly; if not, we have a cache line miss and need to load the entire cache line containing that element from memory into the cache. Since the array is stored contiguously in memory, every B consecutive elements of the array fit into one cache line. Thus, the number of cache line misses is approximately $\lceil n/B \rceil$, since every time we access an element that is not in the cache, we load the entire cache line containing it (which brings in B new elements), and we need an extra cache line if n is not a multiple of B. So the total number of cache line misses is CLMs $\leq 1 + \left\lceil \frac{n}{B} \right\rceil = O\left(1 + \frac{n}{B}\right)$. Note that this is constant time when $n \ll B$, and runs in time $O(n/B) \approx O(n)$ when $n \gg B$. So this captures the two situations well, and communicates the intuition that scanning through an array is efficient if we can afford to load the entire array into the cache (or at least a significant portion of it).

(A2) Scanning an array `A[1,...,n]` in arbitrary/worst-case order This kind of problem is encountered in scenarios where we need to do some kind of "pointer chasing" or where the access pattern is not known in advance, like reading in a linked list or walking on a graph via BFS or something like this. That is, after we read in the i-th element, an adversary can choose the next element to read in arbitrarily to potentially misdirect our focus, and we have no control over it. As long as $M \ll n$, then in the worst case the location is not in the cache and we have a cache line miss. So CLMs $= \Omega(n)$.

In this way the difference between scanning A in linear fashion versus arbitrary fashion is but a factor of the size B of each cache line; for $B \gg 1$, this could be a significant difference. This is why cache lines are designed as such; it is a basis of the so-called "locality of reference" principle, which states that if we access a memory location, then it is likely that we will access the same or a nearby location soon. This is quite important, as it is where we typically get the most cache performance gains in practice.

Remark 24.1.

1. In the above scenario (A2), if $M \gg n$ then CLMs $= O(1 + n/B)$, since each time we see something not in the cache, we bring it into the cache until we've brought in all the cache lines describing the entire array; if it fits, then we're done just as in (A1).
2. To some degree our algorithms are more about how we represent things in memory than about algorithms deployed; naturally, we take allocated arrays as being contiguous in memory, and we can represent graphs as adjacency lists or adjacency matrices, which are also contiguous in memory.

(A3) Searching in a sorted array The natural algorithm here is binary search, which works as follows: first, we check the middle element of the array; if it is equal to the target, then we're done; if it isn't, then we check if the target is smaller or larger than the middle element, and then we recurse on the left or right half of the array, respectively. As we execute this algorithm, the region of validity for the location of the query item x progressively shrinks by a factor of two, so at iteration t, the distance between the new query and the old query is at most $n/2^t$. This is at least B if we have a cache-line miss, and in the worst case $n/2^t = B$, so $t = O(\log(n/B))$. If $n \leq B$, then the number of cache line misses is at most 2, since we can load the entire array into the cache and then just do a linear scan through it. The number of cache line misses is thus $O(2 + \log(n/B))$.

Example 24.2. Consider the case where $B < \sqrt{n}$. Then the number of cache line misses is

$$\text{CLMs} = O\left(2 + \log(n/B)\right) = O\left(\log(n/\sqrt{n})\right) = O(\log(\sqrt{n})) = O(\log n).$$

In particular, if the size of the cache line is quadratically smaller than the size of the array, then we can accomplish search in logarithmic time.

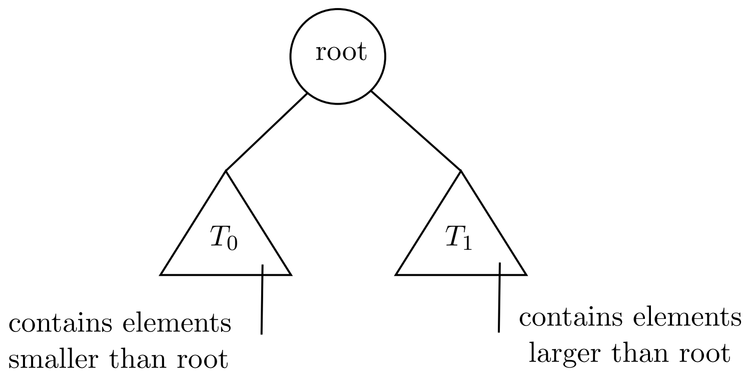
But can we do better? Yes!

(A4) Searching in a sorted array, but better To improve on (A3), we will need to change the way we store the array in memory. Instead of storing it contiguously, we will store it in a *B-tree* structure, which is a generalization of a binary search tree where each node has up to B children (instead of just 2). Recall that the BST property states that for each node v,

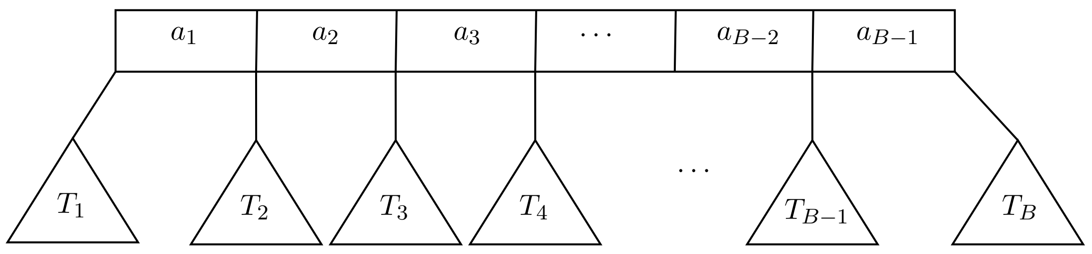
- all the elements in the left subtree of v are smaller than v;
- all the elements in the right subtree of v are larger than v.

- the left and right subtrees are also BSTs.

So a BST looks like:


root
T_0
T_1
contains elements smaller than root
contains elements larger than root

So looking up one unit of data in a BST data structure with n elements on an architecture of with cache lines of size B requires $O(\log n - \log B)$ many cache line misses as per (A3). B-trees have a larger branching factor than 2; each B-tree node contains up to $B-1$ elements, and stores up to B pointers to sub-B-trees with one pointer each between each pair of elements in a node and before the first element and after the last element in a node, unless the tree does not grow in a particular direction from this node (for example, if the node is a leaf node, in which case it doesn't grow in any direction). A node contains elements $a_1, \ldots, a_{B-1}$ and subtrees $T_0, T_1, \ldots, T_{B-1}$, so the B-tree has the following structure:



Sortedness is an invariant maintained at every node of a B-tree, so that:

1. the elements within a node are sorted, i.e. for all $i \in [1, B-2]$, we have $a_i < a_{i+1}$;
2. for any subtree whose pointer is stored to the left of any specific element in a node, all elements of that subtree are smaller than that element, that is, for all $i \in [1, B-1]$ and for every $b_i \in T_i$, we have $b_i < a_i$;
3. all elements in the subtree to the right of the largest element in a node are larger than that element, that is, for all $b_i \in T_{B-1}$, we have $a_{B-1} < b_i$.

There are B^d elements that can be stored in a tree of size d with B-ary branching as described above, so a B-tree needs to be of depth $O(\log_B n)$ to store n elements. Given this, how do we find an element x in this data structure? We start at the root, and then we check if x is an element of the root node; if it is, then we're done. Otherwise, we find the element a_i of the root node such that $a_i < x < a_{i+1}$ (and note that here we have predefined $a_0 := -\infty$ and $a_{B+1} := +\infty$), and then we recurse into T_i, the subtree whose pointer is stored between a_i and a_{i+1}. We repeat this process until we either find the element or reach a leaf node, in which case we conclude that the element is not in the tree. The number of cache line misses is $O(\log_B n - \log_B B) = O(\log_B n)$, since each time we recurse into a subtree, we load the entire node of that subtree into the cache, and the number of nodes we traverse is at most the depth of the tree, which is $O(\log_B n)$.

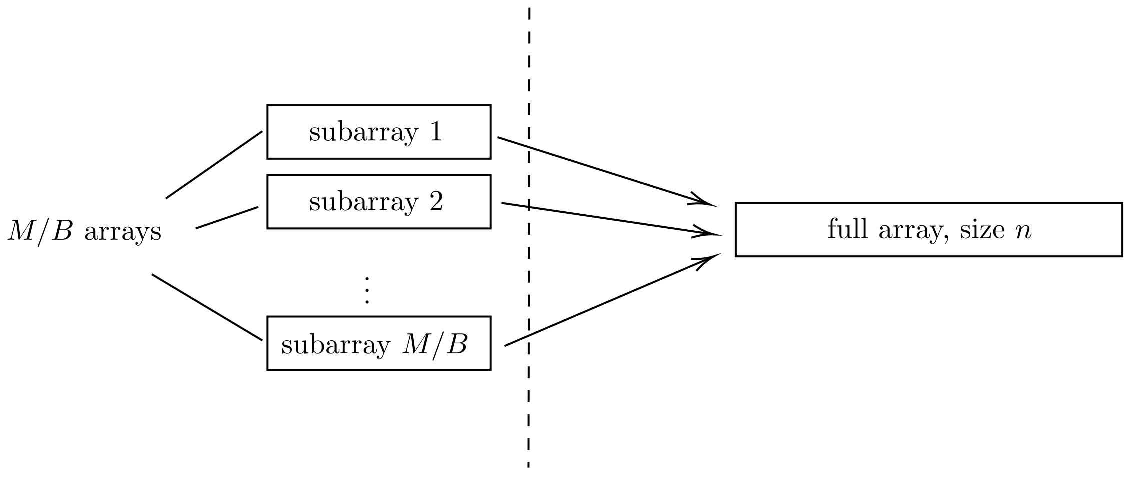
Example 24.3. Suppose $n = 2^{20}$ and $B = 2^{10}$, about 1kB. Then (A3) gives us $O(\log(n/B)) = O(\log(2^{20}/2^{10})) = O(\log(2^{10})) = O(10)$ cache line misses, while (A4) gives us $O(\log_B n) = O(\log_{2^{10}} 2^{20}) = O(2)$ cache line misses. So we can do better by using a B-tree instead of a binary search tree.

So this is the first, in some sense, non-trivial algorithm that is cache-aware and uses this particular data structure to organise things properly and achieve the improved performance.

Remark 24.4. 1. To get $O(\log n)$ CPU time for the above algorithm, we need to do binary search on each node, so the CPU time is $O(\log B) \cdot O(\log_B n) = O(\log n)$.

2. Can we do better? No! There are results that show that this is the best runtime we can accomplish for this problem; see [AV88].

(A5) Sorting an array `A[1,...,n]` We can sort an array in time $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$. This is because since the size of each cache line is B and we have total cache size M, we can create M/B new subarrays stored in each of the cache lines (and there are no cache line misses in each), and then we can execute an (M/B)-way merge-sort on these subarrays.



This sort of trick actually works for a number of computational primitives; often to get around hard bounds against computation, we can break up a larger thing into many smaller things and then process each of them in parallel, and then merge the results together. We will see this in more detail in the next lecture.

§24.1.2 The cache-oblivious model

For many reasons we may want to not have to worry about the cache size M and the cache line size B when designing algorithms, not least because:

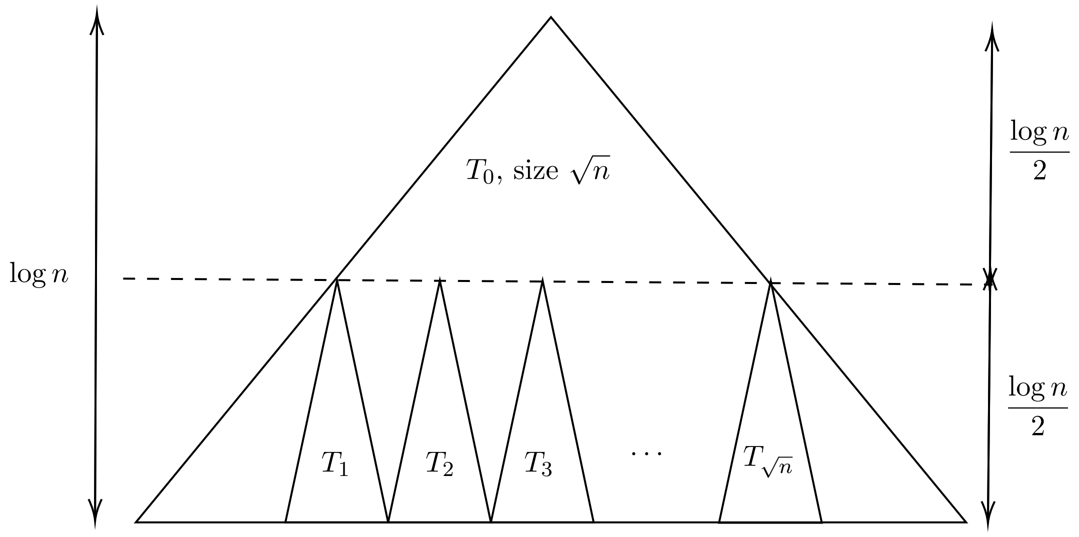
1. We typically want to design algorithms that work on a variety of hardware architectures each of which may have different cache sizes and cache line sizes;
2. The cache size and cache line size may change over time, so we want to design algorithms that are agnostic to these parameters;
3. There may be multiple levels of (coherent) caches, each with different sizes and cache line sizes; in fact the picture from earlier is a bit over-simplified; it's actually more like

   $$\text{CPU} \longrightarrow \text{L1 cache} \longrightarrow \text{L2 cache} \longrightarrow \text{L3 cache} \longrightarrow \text{main memory} \longrightarrow \text{SSD}$$

   where each of the caches has a different size and a different cache line size, and we want to design algorithms that work well across all of them by not worrying about the specifics of each level.

So we'd like to design algorithms that are *cache-oblivious*, i.e. algorithms that do not depend on the cache size M or the cache line size B. The cache-oblivious model is a theoretical model of computation that captures this idea. In this model, we assume that the cache has an infinite number of cache lines, each of size B, and that the cache is large enough to hold any subarray of the input array that we are currently processing.

(A6) Binary search in the cache-oblivious model We can do binary search as fast as possible in the cache-oblivious model, using the obvious idea of storing the BST as a heap. Recall that a heap is a BST stored in an array A where the root of the tree is stored in the zeroth array, and for each node v stored in $A[i]$, its left child is stored in $A[2i+1]$ and its right child is stored in $A[2i+2]$. But this still takes $O(\log n)$ time to search for a value in the BST. So how can we use the B-tree so that we can achieve better cache performance for all B? We store the array in a van Emde Boas layout—recall the formal definition and structure from COMS W3134 Data Structures—of the binary search tree. The vEB has the following structure:

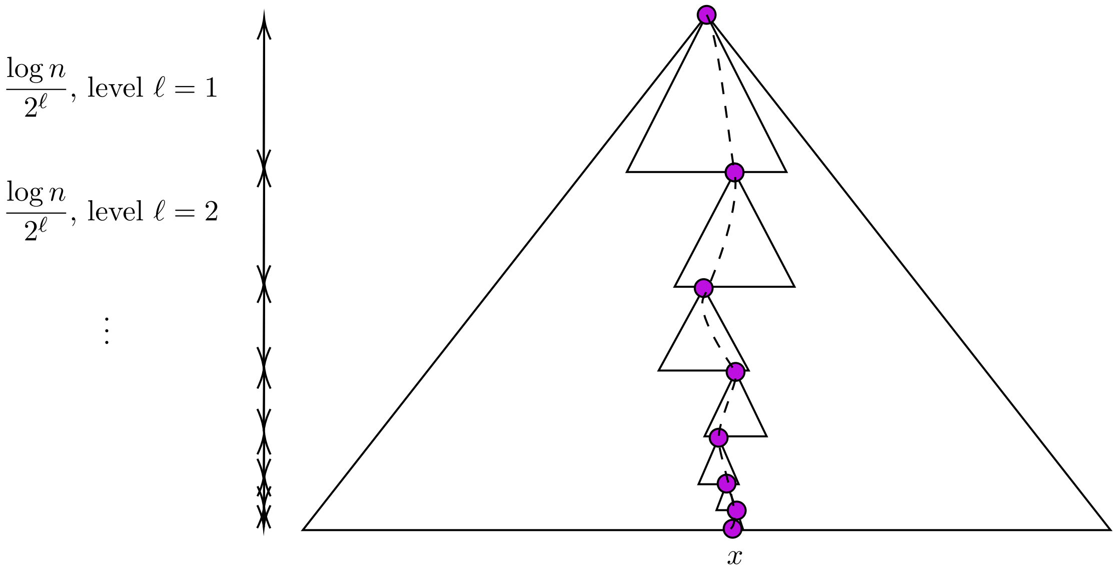


vEB layout: tree $T = T_0 \cup T_1 \cup \ldots \cup T_{\sqrt{n}}$

As a quick heuristic, we can take, for a BST T:

$$\text{vEB}(T) = \begin{cases} \text{if } n \text{ is suff. large, e.g. } n > 16, \text{ then store } (\text{vEB}(T_0), \text{vEB}(T_1), \dots, \text{vEB}(T_{\sqrt{n}})) \\ \text{otherwise, use a standard BST layout} \end{cases}$$

Search is exactly the same as before with the B-tree, but on this new data structure. So how many cache line misses? Each level of recursion halves the height of the tree, so fix a level $\ell \in [1, \log\log n]$ of the vEB recursion trees of size $n^{2^{-\ell}}$, so that the size of the minitree is $\in [\sqrt{B}, B]$. So we have a bunch of these minitrees of fixed size.


$\frac{\log n}{2^\ell}$, level $\ell = 1$
$\frac{\log n}{2^\ell}$, level $\ell = 2$
$\vdots$
x

Since each minitree is stored consecutively, the number of cache line misses per minitree is at most 2 (after all, each minitree fits into at most two cache lines since the size of the minitree is $\in [\sqrt{B}, B]$), and the total number of cache line misses is at most $O(2^\ell)$. Now

$$n^{2^{-\ell}} = B \implies \frac{1}{2^\ell} \cdot \log n = \log B \implies 2^\ell = \frac{\log n}{\log B},$$

so the total number of cache line misses is $O(2^\ell) = O(\log_B n)$.

§25 Lecture 25—17th April, 2024

§25.1 Parallel algorithms and the PRAM model

The foundational model for parallel computation is the the parallel random-access machine (PRAM) model. In this model, we have a shared memory that can be accessed by multiple processors in parallel; we can view it as an analogue for the sequential RAM model, but for parallel computation. In this model, we have:

- a collection of p processors $P_0, P_1, \dots, P_{p-1}$, each functioning as a standard RAM possessing its own private memory (i.e. registers, cache, etc.) and aware of its unique processor ID;

- a global shared memory of (theoretically) unbounded size, accessible to all processors;
- each processor operating in a synchronous manner so that each instruction, whether it involves reading from shared memory, performing a local computation, or writing to shared memory, is assumed to complete in a single, uniform time unit.

For our discussion, we will wish away practical engineering challenges such as memory access latency, network contention, cache coherence, and the overhead associated with synchronization primitives. In this way, we will see that it makes sense to only have the depth of the dependency graph of a PRAM algorithm as a measure of its time efficiency.

§25.1.1 Memory access protocols: EREW, CREW, and CRCW

Allowing shared memory causes a critical issue: memory access conflicts. A conflict occurs when multiple processors attempt to access the same memory location within the same instruction cycle. The PRAM framework is not a single model but a family of models, stratified into a hierarchy based on the rules governing the resolution of these conflicts. This hierarchy spans from the most restrictive model, Exclusive Read Exclusive Write (EREW), to the most powerful and abstract, Concurrent Read Concurrent Write (CRCW).

- *EREW (Exclusive Read, Exclusive Write).* This is the most restrictive and most common PRAM variant. It prohibits any form of simultaneous access to a single memory location. In any given time step, a memory cell can be read by at most one processor and written to by at most one processor. Algorithms designed for the EREW model must explicitly guarantee that no two processors ever access the same memory address concurrently.
- *CREW (Concurrent Read, Exclusive Write).* This model relaxes the read constraint of EREW by allowing any number of processors to read from the same memory location simultaneously. All processors participating in a concurrent read are assumed to receive the same value from the memory cell, but write access remains exclusive; only one processor may write to a given memory location at a time.
- *CRCW (Concurrent Read, Concurrent Write).* This is the most powerful PRAM model; it allows both concurrent reads and concurrent writes to the same memory location. While concurrent reads are straightforward, the outcome of a concurrent write must be well-defined. This leads to several sub-models based on the chosen write-conflict resolution protocol:
  - *Common CRCW.* A concurrent write is permitted only if all processors attempting to write to the same location are writing the exact same value. If they attempt to write different values, raise an error.
  - *Arbitrary CRCW.* Any processor can write to a memory location, and the value written is determined arbitrarily from among the values proposed by the processors; the algorithms in this submodel must be correct regardless of the winning processor.
  - *Priority CRCW.* Each processor has a unique priority, and in the event of a write conflict, the processor with the highest priority is allowed to write its value, while others are ignored.
  - *Combining CRCW.* The value ultimately written to the memory cell is the result of applying a specified associative and commutative operator (such as SUM, MAX, MIN,

AND, OR, or XOR) to all the values being written concurrently.

Remark 25.1. 1. CRCW > CREW > EREW in terms of computational power.

2. Simulating a stronger model on a weaker one incurs a penalty in parallel time, e.g. a single step of a p-processor CRCW PRAM can be simulated on an EREW PRAM in $O(\log p)$ time.

A typical technique for PRAM algorithms is parallel reduction, wherein which we take a set of n elements and combine them into a single result using a binary associative operator.

Problem: XOR of n bits. Given an input array $X = [x_1, \dots, x_n]$ of n bits, compute the XOR of all the bits in X.

The PRAM algorithm for parallel reduction employs a strategy that mirrors the structure of a complete binary tree. The n input bits are considered the leaves of this conceptual tree. The computation proceeds upwards from the leaves to the root, with each level of the tree corresponding to a parallel step:

Algorithm: Computing the XOR of n bits in parallel

- (Round 1.) Start with $n/2$ processors, with each processor P_i (for $i \in [1, n/2]$) assigned a pair of adjacent bits. Processor P_i computes $x_{2i-1} \oplus x_{2i}$. `// This takes constant time and reduces the number of values from` n `to` $n/2$.
- (Round 2.) Now, $n/4$ processors are active, each assigned a pair of results from the previous round. Each processor P_i computes the XOR of its two inputs. `// This takes constant time and again halves the problem size, to` $n/4$.
- (Round k.) In the k-th round of the algorithm, there are $n/2^{k-1}$ values remaining, and $n/2^k$ processors are active. Each processor P_i computes the XOR of its two inputs.
- (Termination.) The algorithm terminates when there is only one value remaining, which is the final XOR result.

The computational structure of this algorithm is a balanced binary tree with n leaves (the input bits) and height $O(\log n)$; since the work at each level of the tree can be performed in a single parallel step, the total parallel time complexity is the depth of the reduction tree, which is $O(\log n)$. We use $O(n)$ processors in total. (In general, we want to use the smallest parallel time with the fewest possible processors.)

If we can perform the ℓ-way XOR, then the computational structure is that of a complete ℓ-ary tree with depth $\log_\ell n$, and hence the parallel time complexity is $O(\ell \log_\ell n) = O((\log n) \cdot \ell / \log \ell)$. So we see a trade-off: increasing the arity of the tree decreases its depth, which in turn reduces the number of parallel stages; however, it also increases the amount of sequential work that must be performed at each node so each stage takes longer to complete.

As it turns out, the parallel time obtained above isn't too far from the best possible. In fact, even in the strongest PRAM model, we have the following lower bound:

Theorem 25.1 (Lower bound for PRAM algorithms, [BH89]). *Any CRCW PRAM algorithm that computes the parity of n bits in time T must use at least a combination of processors $p(n)$*

or memory cells $c(n)$ that is exponential in $n^{1/T}$; that is, $p(n) + c(n) \geq 2^{\Omega(n^{1/T})}$. As a consequence, if we constraine the number of processors to only be poly(n)*, then we require parallel time* $\Omega(\log n/\log\log n)$.

We can think of PRAMs as being roughly equivalent to the circuit model of computation, where the depth of the circuit corresponds to the parallel time and the size of the circuit corresponds to the number of processors.

§25.2 The massively parallel computing (MPC) model

Moore's law has been breaking down, so we needed to do better, and the massively parallel computing (MPC) model was developed (see [KSV10, ANOY14, BKS17, GSZ11]) as a more realistic abstraction for frameworks like MapReduce, Hadoop, Spark, etc, which themselves were designed to help perform large scale computations across various machines. A system in the MPC model consists of

- the total number of independent machines m in the system,
- the total amount of local memory S available to each machine,

so that the total memory available to the system is mS. Computation is synchronously executed in stages, and between computations, machines exchange data. Typically the MPC parameters are additionally constrained by

1. a total memory constraint, which posits that the collective memory of all machines must suffice to store the entire input.
2. a sublinear local memory constraint, which requires that the local memory of each machine is $S = O(N^\delta)$ for $0 < \delta < 1$.

This structure leads to a unique cost model. Because modern distributed systems are typically communication-bound rather than compute-bound, the MPC model simplifies analysis by treating local computation as effectively "free." The primary performance metric, or cost, of an MPC algorithm is the number of synchronous communication rounds required to solve the problem.

Remark 25.2 (MPC is at least as powerful as PRAM)**.** We can check that a PRAM with N processors is about as powerful as an MPC with $m = O(N)$ machines and $S = O(1)$ local memory.

§25.2.1 An MPC algorithm for summing N numbers

This is the basically the same as computing the XOR of N bits. There are two cases:

Case 1: Large local memory regime, $S \geq \sqrt{N}$ Let the total N-bit input be distributed among $m \leq \sqrt{N}$ machines. The algorithm is a 2-round algorithm:

Algorithm: MPC-computing the XOR of N bits in the large local memory regime

- (Round 1, local reduction.) Each of the m machines computes the XOR of all bits stored in its local memory.

- (Round 2, global aggregation.) : Each of the m machines sends its single-bit partial result to a designated coordinator machine, say M_1. M_1 then computes the final XOR of these m bits. This step is feasible in a single round because the total amount of data M_1 needs to receive is m bits. Since $m \leq \sqrt{N}$ and we are in the regime $S \geq \sqrt{N}$, it follows that $m \leq S$. The coordinator machine has sufficient memory to receive all partial results simultaneously.

Case 2: Small local memory regime, $S < \sqrt{N}$ In this scenario $m > \sqrt{N}$, so a single coordinator machine cannot receive the partial results from all other machines in one round, as this would violate the communication constraint. To overcome this, the algorithm must employ a multi-round, tree-based aggregation strategy:

Algorithm: MPC-computing the XOR of N bits in the small local memory regime

- The m machines are treated as the leaves of a conceptual communication tree. In each round, machines at a given level of the tree send their partial results "up" to parent machines at the next level. The number of child machines that can send their results to a single parent is limited by the parent's memory/communication capacity S. This limit determines the arity (or fan-in) of the tree.

The arity of the aggregation tree is $O(S)$. The number of leaves is $m \approx N/S$. The depth of a tree with m leaves and arity k is $\log_k m$. Therefore, the number of communication rounds required is

$$O(\log_S m) = O\left(\log_S \frac{N}{S}\right) = O\left(\frac{\log(N/S)}{\log S}\right) = O\left(\frac{\log N - \log S}{\log S}\right) = O\left(\frac{\log N}{\log S}\right).$$

With $S = N^\delta$, this becomes

$$O\left(\frac{\log N}{\log N^\delta}\right) = O\left(\frac{\log N}{\delta \log N}\right) = O\left(\frac{1}{\delta}\right).$$

So more memory per machine leads to fewer communication rounds.

§25.2.2 An MPC algorithm for computing prefix sums

Given an input array of N numbers $a_1, \ldots, a_N$, we want to compute the output array $\sigma_1, \ldots, \sigma_N$, where $\sigma_i = \sum_{j=1}^{i} a_j$. As before, we start with the case of large local memory:

Algorithm: MPC-computing the prefix sum of N numbers in the large memory regime

- Each machine M_i performs two local computations; first it computes the prefix sums for the elements within its own block, and then it computes the total sum β_i of its block, then it sends this single value β_i to the designated coordinator machine M_1.
- The coordinator machine M_1 receives all the m values of $\{\beta_i\}_{i=1}^{m}$ and computes the prefix sums of these block sums, namely $\pi_i = \sum_{j=1}^{i} \beta_j$, where the value π_{i-1} represents the prefix sum of the entire array up to the start of block i.

- M_1 sends this offset value π_{i-1} back to each machine M_i, and retains offset 0 for M_1 itself.
- Having received its offset π_{i-1}, each machine M_i can compute the final prefix sums for its own block as the sum of its offset and the local prefix sums computed earlier; this corrects the local values to their true global prefix sum values.

This algorithm clearly requires 3 rounds of communication in the high-memory regime where a single coordinator can handle the m block sums ($m \leq S$). If the system is in a low-memory regime ($m > S$), the aggregation in Round 1 and the distribution in Round 2 cannot be done with a single coordinator. Instead, each of these steps would require a tree-based communication pattern, similar to the XOR reduction, leading to a round complexity of $O(\log_S N)$.

§25.2.3 Approximating *Distinct-Count* for a stream in the MPC model

Now we tackle the problem of approximating the number of distinct elements in a stream of N elements from before. For distributed computation, the sketch in question must be "combinable"; that is, there must exist some efficient merge function M such that

$$\texttt{sketch}(\texttt{data}_A \cup \texttt{data}_B) = \mathsf{M}(\texttt{sketch}(\texttt{data}_A), \texttt{sketch}(\texttt{data}_B)).$$

Remark 25.3. *Every linear sketch is combinable.*

The Bottom-k sketch is one such sketch that meets our requirements; the union of two Bottom-k sketches is simply the set of k smallest hash values from the combined set of items in both individual sketches. Recall that the size ρ of the Bottom-k sketch is $O(1/\varepsilon^2)$, where ε is the desired error parameter.

Algorithm: MPC-computing the prefix sum of N numbers in the large memory regime

- Each machine M_i processes its local data partition and computes a local sketch (e.g., a Bottom-k sketch) of size ρ. This is a local computation step.
- The m local sketches are aggregated using a communication tree. In each round, machines at one level of the tree send their ρ-sized sketches to their designated parent machine. The parent machine receives multiple sketches, merges them into a single sketch of the same size ρ, and passes this new sketch up the tree in the next round.
- After a series of rounds equal to the tree's depth, the root machine holds the final, global sketch representing the entire dataset. It then applies the statistical estimator function to this final sketch to produce an approximation of the total number of distinct elements.

This algorithm's structure is identical to that of the low-memory XOR reduction, with the only difference being that the messages are sketches of size ρ instead of single bits. The number of sketches a parent machine can receive is limited by its memory S, so the fan-in of the communication tree is $\lfloor S/\rho \rfloor$. The number of rounds is therefore $O(\log_{S/\rho} m) = O(\log_S N)$; this holds as long as the memory per machine is large enough to hold at least a few sketches (e.g., $S > \rho$).

§26 Lecture 26—22nd April, 2024

I was unfortunately not able to make it today, but we covered some more MPC algorithms, namely `terasort` for sorting and a graph connectivity algorithm.

§27 Lecture 27—24th April, 2024

§27.1 Beating exhaustive search for algorithms

We start today by talking about the 3-SAT problem.

Problem 27.1 (*3-SAT*)**.** Take n variables $x_1, \dots, x_n$ and m clauses of the form $C_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ where each l_{ij} is a literal (a variable or its negation). The 3-SAT problem is to determine if there exists an assignment of truth values to the variables such that all clauses are satisfied, i.e. the CNF formula $\phi = C_1 \wedge \cdots \wedge C_m$ is satisfiable taking $m = n^{O(1)}$ without loss of generality.

We can define the k-SAT problem similarly, where each clause has k literals. This is a foundational problem; it is the prototypical NP-hard (and indeed NP-complete) problem, and so under the assumption that $\mathsf{P} \neq \mathsf{NP}$, we can't expect to solve it in polynomial time. However, we can still try to solve it as efficiently as possible since it has many applications in e.g. SAT solvers. We start by examining several potential algorithms.

Algorithm: Naive 3-SAT

- Enumerate over all possible $\overline{x} = (x_1, \dots, x_n) \in \{0,1\}^n$.
- For each $\overline{x}$, check if $\phi(\overline{x}) = 1$. If so, output `sat`, otherwise output `unsat`.

This algorithm runs in time $O(2^n \cdot m) = O(2^n \cdot n^{O(1)})$, which is exponential in the number of variables. But can we improve this somehow? Randomness can help us here.

Algorithm: Random 3-SAT

- Repeat the following procedure T times:
  - Choose a random assignment $\overline{x} \in \{0,1\}^n$.
  - Check if $\phi(\overline{x}) = 1$. If so, output `sat`, otherwise continue.
- If no satisfying assignment is found after T iterations, output `unsat`.

Observe that

$$\Pr_{\overline{x}}[\phi(\overline{x}) = 1] = \frac{\#\text{ satisfying assignments}}{2^n} \geq 2^{-n},$$

if there exists a satisfying assignment. Hence if we set $T = O(2^n)$, the probability of not finding a satisfying assignment is at most $(1 - 2^{-n})^{2^n} \leq e^{-1}$. Yet we can still do better, thanks to an algorithm due to Schöning which combines a random walk with a local search.

```
Algorithm: SCHOENING

• Repeat the following procedure $T = 99(n+1)(4/3)^n$ times:
    – Choose a random assignment $\overline{x} \in \{0,1\}^n$.
    – Repeat the following procedure $3n$ times:
        ∗ If $\phi(\overline{x}) = 1$, output sat.
        ∗ If $\phi(\overline{x}) = 0$:
            · Let $C$ be a falsified clause in $\phi$.
            · Choose a random literal $l$ in $C$, and flip the value of the corresponding
              variable in $\overline{x}$.       // local step, 1/3 chance of flipping the correct value
• If no satisfying assignment is found after $T$ iterations, output unsat.
```

Theorem 27.1. *Suppose that ϕ is satisfiable. Then* SCHOENING *finds a satisfying assignment with probability at least $1 - e^{-99}$.*

Proof. We will prove this by proving several claims. Throughout, x^* will be a satisfying assignment to ϕ, and $h(x, x^*)$ will be the Hamming distance between x and x^*.

Claim 27.2. *Suppose ϕ is satisfiable. Then,*

$$\Pr[\textit{the inner loop returns } x^* \textit{ starting from } x] \geq \frac{\left(\frac{1}{2}\right)^{h(x,x^*)}}{n+1}.$$

Proof. Let $t = h(x, x^*)$, and consider the event E denoting that we make t "bad" choices of which variable to flip and $2t$ "good" choices in the inner loop over a $3t$ choice sequence of flips. Each bad step costs probability $2/3$, each good step costs probability $1/3$, and there are $\binom{3t}{t}$ ways to step t-incorrectly and $2t$-correctly. Hence

$$\Pr[E] \geq \left(\frac{1}{3}\right)^{2t}\left(\frac{2}{3}\right)^{t}\binom{3t}{t} \geq \frac{1}{2^t}\left(\frac{1}{3}\right)^{t}\left(\frac{2}{3}\right)^{2t} \cdot \binom{3t}{t}.$$

Now interpret $P = \binom{3t}{t}(\frac{1}{3})^t(\frac{2}{3})^{2t}$ as the probability that in $3t$ independent trials with probability $1/3$ of success, we get t successes. Decorate each success in a given sequence with one of two labels (giving a total of 2^t possible decorations), so that every sequence is counted with weight 2^t. From the cycle lemma, we know that given all the cyclic shifts of a sequence of $3t$ markers (with t successes and $2t$ failures), exacly one of the $3t+1$ (including the original) rotations has the "balanced" property that a related weighted walk never goes below zero; in particular, among the 2^t decorated copies of a sequence, exactly a $1/(3t+1)$ fraction of them are good. Since the total weighted probability is 2^tP, we have

$$\Pr[E] \geq \frac{1}{2^t} \cdot \frac{1}{3t+1} = \frac{1}{2^t(3t+1)} \geq \frac{1}{(n+1)2^t} \qquad \square.$$

Claim 27.3. $\Pr_x[h(x, x^*) = k] = \binom{n}{k}2^{-n}$.

Proof. There are $\binom{n}{k}$ ways to choose the k variables that differ between x and x^*, and each of these variables can be flipped in x with probability $1/2$. Hence the probability of flipping the correct value for each of the k variables is 2^{-k}, and the probability of flipping the incorrect value for each of the $n-k$ variables is $2^{-(n-k)}$. The result follows by multiplying these probabilities together. $\square$

Now observe that

$$\Pr[\text{one repetition returns } \texttt{sat}] \geq \frac{1}{2^n} \cdot \sum_{x \in \{0,1\}^n} \Pr[\text{the inner loop returns } x^* \text{ starting from } x].$$

From both of these results, we see that this quantity is

$$\geq \frac{1}{2^n} \sum_{k=0}^{n} \binom{n}{k} \cdot \frac{1}{2^k(n+1)} = \frac{1}{2^n(n+1)} \sum_{k=0}^{n} \binom{n}{k} 2^{-k} = \frac{1}{2^n(n+1)} \left(1 + \frac{1}{2}\right)^n = \left(\frac{3}{4}\right)^n \cdot \frac{1}{n+1}.$$

Hence, since $1 - c/x \leq e^{-c/x}$ for a constant c, the probability of not finding a satisfying assignment after $T = 99(n+1)(4/3)^n$ repetitions is at most

$$\left(1 - \left(\frac{3}{4}\right)^n \cdot \frac{1}{n+1}\right)^{99(n+1)(4/3)^n} \leq e^{-99}. \qquad \square$$

This problem is connected to more modern discussions about the intractability of algorithms, some of which we will now discuss.

Conjecture 27.4 (Exponential time hypothesis (ETH))**.** *3-SAT takes $2^{\Omega(n)}$ time to solve on a deterministic Turing machine.*

Note that this is stronger than $\mathsf{P} \neq \mathsf{NP}$, and it is widely believed to be true. There's an even stronger hypothesis:

Conjecture 27.5 (Strong exponential time hypothesis (SETH))**.** *For any $\varepsilon > 0$, there exists some $k = k(\varepsilon)$ such that solving k-SAT on n variables takes at least $2^{(1-\varepsilon)n}$ time on a deterministic Turing machine.*

Both of these results are a big part of the foundation of modern hardness results, especially in what is known as *fine-grained complexity theory.* Here's an example of a problem for which these notions of complexity describe its hardness.

Problem 27.6 ($\mathbb{R}^d$*-closest-pair*)**.** Given a point set $P \subset \mathbb{R}^d$ where $d > \Omega(\log n)$ where n is the size of the data set, find a $(1+\varepsilon)$-approximation to the closest pair of points in P (according to the Euclidean metric).

Theorem 27.2 ([DL+16, KM20])**.** *Under SETH, $\mathbb{R}^d$-closest-pair cannot be solved by any algorithm in time $O(n^{2-\delta})$ for any $\delta = \delta(\varepsilon) > 0$.*

§27.2 Approximate matrix multiplication via dimension reduction

Matrix multiplication is one of the most fundamental problems in computer science, and it is in P.

Problem 27.7 (*Exact-MM*). *Given $A, B \in \mathbb{R}^{n\times d}$, compute $C = A^\top B \in \mathbb{R}^{d\times d}$.*

In general, one may consider the problem over any field $\mathbb{F}$, but we will focus on the reals for simplicity. Naively, we can solve *Exact-MM* in $O(n^2 d)$ time by fixing a row of A and computing the dot product with each row of B. Using the fact that the time complexity of matrix multiplication is $O(n^\omega)$, we can solve this problem in time $O(d^2 n^{\omega-2})$. However, this isn't even quadratic; we are interested in a near-linear time algorithm. Doing this exactly is hard, so we relax the problem to an approximate version. First, we define the following matrix norm to characterise our approximation guarantee.

Definition 27.8 (Frobenius norm). *The* Frobenius norm *of $A \in \mathbb{R}^{n\times d}$ is $\|A\|_F = \sqrt{\sum_{i=1}^n \sum_{j=1}^d A_{ij}^2}$.*

Problem 27.9 (*Approx-MM*). *Given $A, B \in \mathbb{R}^{n\times d}$ and $\varepsilon > 0$, compute $C \in \mathbb{R}^{d\times d}$ such that with high probability,*

$$\|A^\top B - C\|_F \leq \varepsilon \|A\|_F \|B\|_F.$$

For the rest of the discussion, we will take $A = \begin{bmatrix} x_1^\top & \cdots & x_n^\top \end{bmatrix}^\top$ and $B = \begin{bmatrix} y_1^\top & \cdots & y_n^\top \end{bmatrix}^\top$ where $x_i, y_i \in \mathbb{R}^d$ are the rows of A and B respectively.

Sampling via a Horovitz-Thompson estimator The idea here is to sample a subset of the rows of A and B and compute the exact product of the sampled rows. We then use this to estimate the product of the entire matrix.

Claim 27.10. $A^\top B = \sum_{k=1}^n x_k y_k^\top$.

Proof. We have $C_{ij} = \left(\sum_{k=1}^n x_k y_k^\top\right)_{ij} = \sum_{k=1}^n x_{ki} y_{kj}$. □

Simple as it is, this result helps us come up with the following algorithm.

Algorithm: Horovitz-Thompson

- For each $k = 1, \ldots, n$, define $p_k = \dfrac{\|x_k\|_2 \|y_k\|_2}{\sum_{i=1}^n \|x_i\|_2 \|y_i\|_2}$.
- Sample m indices $k_1, \ldots, k_m$ independently from $\{1, \ldots, n\}$, with each k_i chosen with probability p_{k_i}.
- Construct the estimator $\widehat{C} = \dfrac{1}{m} \displaystyle\sum_{t=1}^m \frac{x_{k_t} y_{k_t}^\top}{p_{k_t}}$.

Computing all $\|x_k\|_2$ and $\|y_k\|_2$ normalising to get p_k takes $O(nd)$ time. Sampling m indices can be done in time $O(n + m\log n)$ with a prefix-sum and binary search, and constructing $\widehat{C}$ takes $O(md^2)$ time. Hence the total time complexity is $O(nd + md^2 + m\log n) = \widetilde{O}(nd + md^2)$. We now show that this algorithm gives us the desired approximation.

Theorem 27.3. *$\widehat{C}$ is an unbiased estimator of $C = A^\top B$. Furthermore,*

$$\Pr_{k_1,\dots,k_m}\left[\|\widehat{C} - C\|_F \le \varepsilon\|A\|_F\|B\|_F\right] < \frac{1}{\varepsilon^2 m},$$

and so we can achieve the desired approximation by setting $m = \Omega(1/\varepsilon^2)$.

Proof. Since each sample is drawn i.i.d. from the same distribution, it suffices to check the expectation of one sample.

$$\mathbb{E}[\widehat{C}] = \frac{1}{m}\mathbb{E}\left[\sum_{t=1}^m \frac{x_{k_t}y_{k_t}^\top}{p_{k_t}}\right] = \frac{1}{m}\sum_{t=1}^m\sum_{k=1}^n \frac{x_k y_k^\top}{p_k}\cdot\Pr[k_t = k] = \sum_{k=1}^n \frac{p_k x_k y_k^\top}{p_k} = C.$$

Now we compute the variance V as follows:

$$\begin{aligned}
V = \mathbb{E}[\|\widehat{C} - C\|_F^2] &= \mathbb{E}\left[\sum_{i=1}^d\sum_{j=1}^d(\widehat{C}_{ij} - C_{ij})^2\right] = \sum_{i=1}^d\sum_{j=1}^d \operatorname{Var}[\widehat{C}_{ij}] \le \sum_{i=1}^d\sum_{j=1}^d \operatorname{Var}\left[\frac{1}{m}\sum_{t=1}^m \frac{x_{k_t}y_{k_t}^\top}{p_{k_t}}\right] \\
&= \sum_{i=1}^d\sum_{j=1}^d \frac{1}{m^2}\sum_{t=1}^m \operatorname{Var}\left[\frac{x_{k_t}y_{k_t}^\top}{p_{k_t}}\right] \le \frac{1}{m}\sum_{i=1}^d\sum_{j=1}^d \mathbb{E}\left[\left(\frac{x_{ki}y_{kj}^\top}{p_k}\right)^2\right] \\
&= \frac{1}{m}\sum_{i=1}^d\sum_{j=1}^d\sum_{k=1}^n p_k\left(\frac{x_{ki}y_{kj}^\top}{p_k}\right)^2 = \frac{1}{m}\sum_{k=1}^n\frac{1}{p_k}\sum_{i=1}^d\sum_{j=1}^d x_{ki}^2 y_{kj}^2 = \frac{1}{m}\sum_{k=1}^n \frac{1}{p_k}\|x_k\|_F^2\|y_k\|_F^2 \\
&= \frac{1}{m}\sum_{k=1}^n \frac{\sum_{i=1}^n \|x_i\|_F\|y_i\|_F}{\|x_k\|_F\|y_k\|_F}\|x_k\|_F^2\|y_k\|_F^2 \le \frac{\left(\sum_{k=1}^n\|x_k\|_F^2\right)\left(\sum_{k=1}^n\|y_k\|_F^2\right)}{m} \\
&= \frac{\|A\|_F^2\|B\|_F^2}{m},
\end{aligned}$$

where we have applied the Cauchy-Schwarz inequality in the penultimate line. Applying Chebyshev's inequality to this variance, we get

$$\Pr\left[\|\widehat{C} - C\|_F \ge \varepsilon\|A\|_F\|B\|_F\right] \le \frac{V}{\varepsilon^2\|A\|_F^2\|B\|_F^2} = \frac{1}{\varepsilon^2 m}. \qquad \square$$

Using randomised projections Now with some thought, we can see that the above algorithm is essentially a randomised projection of the rows of A and B onto a lower-dimensional space. We can use this idea to come up with the following algorithm.

Algorithm: DR-MatMul

- Choose a random $\Pi \in \mathbb{R}^{m\times n}$, where

$$\Pi_{i,j} = \begin{cases} 1/\sqrt{mp_k} & \text{if } (i,j) = (t,k_t) \text{ for some } t = 1,\dots,m, \\ 0 & \text{otherwise.}\end{cases}$$

- Compute $\widehat{C} = (\Pi A)^\top(\Pi B)$.

Clearly, these algorithms do the same thing; DR-MatMul just returns the same estimator in a different way:

$$(\Pi A)^\top (\Pi B) = \sum_{t=1}^{m} (\Pi A)_{t,\cdot} (\Pi B)_{t,\cdot}^\top = \sum_{t=1}^{m} \left(\frac{1}{\sqrt{m p_{k_t}}} x_{k_t} \right) \left(\frac{1}{\sqrt{m p_{k_t}}} y_{k_t} \right)^\top = \sum_{t=1}^{m} \frac{x_{k_t} y_{k_t}^\top}{m p_{k_t}} = \frac{1}{m} \sum_{t=1}^{m} \frac{x_{k_t} y_{k_t}^\top}{p_{k_t}}.$$

However, we can make gains from this algorithm. Note that the algorithm requires two passes over the data, one to sample Π (to compute the p_ks), and one to compute the "reduced" matrices ΠA and ΠB. Given this randomised embedding formulation of the algorithm, Johnson-Lindenstrauss-type results can be used here.

Definition 27.11 (Dimension reducing matrix)**.** *Say that $\Pi \in \mathbb{R}^{m\times n}$ is an (ε, δ)-dimension reducing matrix (or an (ε, δ)-DR matrix) if, for all $x \in \mathbb{R}^n$, we have $\Pr_\Pi \left[\left| \|\Pi x\|_2^2 - \|x\|_2^2 \right| \geq \varepsilon \|x\|_2^2 \right] \leq \delta$.*

Theorem 27.4. *If Π is an (ε, δ)-DR matrix, then $\widehat{C}$ computed by* DR-MatMul *satisfies*

$$\Pr\left[\|\widehat{C} - C\|_F \geq 3\varepsilon \|A\|_F \|B\|_F \right] \leq 3d^2\delta.$$

With a more precise formulation of the Johnson-Lindenstrauss lemma, we can remove the factor of d^2 in the bound. A corollary of this result is the following:

Corollary 27.12. *If we choose $m = O(\varepsilon^{-2}\log(\delta^{-1}))$ and $\delta = 1/(10d^2)$, then we can naively compute C' in time $O(mnd) + O(dmd) = O(\varepsilon^{-2}(nd + d^2)\log d) = \widetilde{O}(\varepsilon^{-2}(nd+d^2))$ such that with high probability, $\|C' - C\|_F \leq 3\varepsilon\|A\|_F\|B\|_F$.*

§28 Lecture 28—29th April, 2024

We begin by finishing off the result from last time.

Proof of Theorem 27.4. Write $A = \begin{bmatrix} A_1 & \cdots & A_d \end{bmatrix}$ and $B = \begin{bmatrix} B_1 & \cdots & B_d \end{bmatrix}$, where $A_i, B_i \in \mathbb{R}^{n\times d}$ are the i-th blocks of A and B respectively, and define $a_i := A_i/\|A_i\|_2$ and $b_i := B_i/\|B_i\|_2$, so that $C_{i,j} = A_i^\top B_j = \|A_i\|_2\|B_j\|_2 a_i^\top b_j$. Then with probability $\geq 1 - 3\delta$,

$$\begin{aligned}
\widehat{C}_{ij} = (\Pi A_i)^\top (\Pi B_j) &= \|A_i\|_2 \|B_j\|_2 (\Pi a_i)^\top (\Pi b_j) \\
&= \|A_i\|_2 \|B_j\|_2 \left(\|\Pi a_i\|_2^2 + \|\Pi b_j\|_2^2 - \frac{1}{2}\|\Pi a_i - \Pi b_j\|_2^2 \right) \\
&= \|A_i\|_2 \|B_j\|_2 \left(\|a_i\|_2^2 + \|b_j\|_2^2 - \|a_i - b_j\|_2^2 \pm 3\varepsilon \right) \\
&= \|A_i\|_2 \|B_j\|_2 \left(a_i b_j \pm 3\varepsilon \right);
\end{aligned}$$

that is, with probability $\geq 1 - 3\delta$, $(\widehat{C}_{i,j} - C_{i,j})^2 \leq \|A_i\|_2^2 \|B_j\|_2^2 (3\varepsilon)^2$, and a union bound gives that with probability $\geq 1 - 3d^2\delta$,

$$\|\widehat{C} - C\|_F^2 \leq \sum_{i=1}^{d}\sum_{j=1}^{d} \|A_i\|_2^2 \|B_j\|_2^2 (3\varepsilon)^2 = 9\varepsilon^2 \|A\|_F^2 \|B\|_F^2. \qquad \square$$

§28.1 Compressed sensing

Now we switch gears to discuss the final topic, compressed sensing. This area was borne out of problems in signal processing, where one is interested in recovering a signal from a small number of linear measurements.

Problem 28.1 (*Compressed sensing*, informal)**.** Suppose we are given $x \in \mathbb{R}^n$—which we interpret as the "signal" to be recovered—and which is very large. The goal is to perform m measurements of a linear transformation y of x (i.e. to compute $y = Ax$ for some $m \ll n$), and from these measurements, recover the signal x.

Often, the signals have a particular structure; in our particular case, we focus on a structured signal which is well approximated by a k-sparse vector.

Definition 28.2 (k-sparseness)**.** *A vector $x \in \mathbb{R}^n$ is k-sparse if it has at most k non-zero entries. Similarly, a matrix $X \in \mathbb{R}^{n\times m}$ is k-sparse the total number of non-zero entries in X is at most k, i.e. $|\{(i,j) : X_{ij} \neq 0\}| \leq k$.*

For example, we can consider an image (represented as a matrix) where x is the signal in the wavelet basis for the Fourier domain; we know that images are sparse in a suitable basis, so we can indeed compress images. We will try to show how to recover the signal with much fewer measurements through compressed sensing.

Problem 28.3 (*Compressed sensing*, formal)**.** *Solve the following optimization problem:*

$$\begin{aligned} \min \quad & \|x^*\|_0 \\ \textit{s.t.} \quad & Ax^* = y, x^* \in \mathbb{R}^n. \end{aligned}$$

Unfortunately, the ℓ_0-minimisation problem is NP-hard, and getting the absolute minimum would require some sort of brute-force search. So instead of solving the ℓ_0-minimisation problem, we solve the ℓ_1-minimisation problem:

$$\begin{aligned} \min \quad & \|x^*\|_1 \\ \text{s.t.} \quad & Ax^* = y, x^* \in \mathbb{R}^n. \end{aligned} \tag{$\star\star$}$$

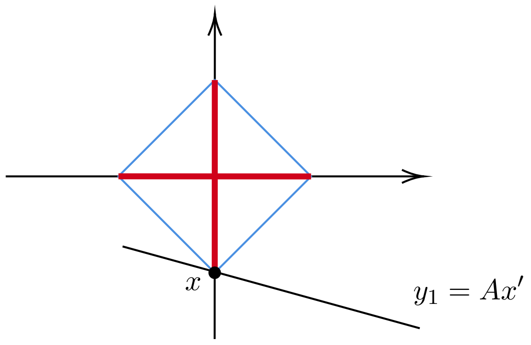
This convex relaxation is rather common in the field of machine learning. We will search for k-approximate solutions by trying to recover x^*, the best approximation to the original signal x, from $y = Ax$, the measurements of x. Ideally, we want:

$$x^* = \arg \min_{\substack{x' \in \mathbb{R}^n \\ x' \text{ is } k\text{-sparse}}} \|x - x'\|_1.$$

The solution for this problem will be the same as the k largest coordinates of x, and zero otherwise.

We now discuss how the ℓ_1-minimisation problem might yield solutions similar to the ℓ_0-minimisation problem. Suppose we're working in $\mathbb{R}^2$, we have a single measurement, and the signal is 1-sparse. The set of possible solutions we're looking for is the intersection of the line $y = Ax$ and the ℓ_1-ball.

The solution to the ℓ_1-minimisation problem is the point on the line that is closest to the origin. The solution to the ℓ_0-minimisation problem is the point on the line that is closest to the origin and is also 1-sparse. The solution to the ℓ_1-minimisation problem is the same as the solution to the ℓ_0-minimisation problem modulo some scaling factor. This is illustrated in the figure below.



Next we discuss the restricted isometry property, which helps show that the solution to ℓ_1-minimisation is the same as the solution to ℓ_0-minimisation under some nice conditions on the matrix A.

§28.1.1 Restricted isometry property and iterative hard thresholding

The restricted isometry property (RIP) is a property of the matrix A that ensures that the matrix A preserves the distances between vectors.

Theorem 28.1. *Fix k, and pick some $\varepsilon \in (0, 1/3)$. If the entries of A are i.i.d. $\mathcal{N}(0,1)$ and $m = O(k \log(n/k))$, then with high probability, for all $x \in \mathbb{R}^n$, if x^* is the solution to the ℓ_1-minimisation problem ($\star\star$), then $\|x^* - x\|_1 \leq C \cdot \mathsf{Err}_1^k(x)$, where*

$$\mathsf{Err}_1^k(x) = \min_{\substack{x' \in \mathbb{R}^n \\ x' \text{ is } k\text{-sparse}}} \|x - x'\|_1,$$

and C is a constant that depends on ε.

Note the following:

- For $C = 1 + \varepsilon$, $m = \Theta(1/\varepsilon)$.
- If x is k-sparse, then $\mathsf{Err}_1^k(x) = 0$, and $x^* = x$.
- The x^* recovered from the ℓ_1-minimisation problem, while it may not be k-sparse, is the best k-sparse approximation to x.

An analogy to *Heavy Hitters* Recall that we used the COUNT-SKETCH to find the heavy hitters, the items that occur with a high enough frequency in a stream. We can think of the ℓ_1-minimisation problem as a way to find the heavy hitters in a vector, where the heavy hitters are the k largest coordinates of the vector; in fact, we can draw several parallels between the two problems.

Algorithmic properties for...	...Count-Sketch	...ℓ_1-min. compressed sensing
m linear measurements	$O(k \log n)$	$O(k \log(n/k))$
recovery time	$O(n \log n)$	a linear program, so $n^{O(1)}$ time
probability of success	$\geq 1 - 1/n$	N/A (deterministic)
guarantee comparison	x s.t. $\lvert x_i^* - x_i \rvert \leq C \cdot \mathsf{Err}_1^k(x)/k$	x s.t. $\lVert x^* - x \rVert_1 \leq C \cdot \mathsf{Err}_1^k(x)$

Definition 28.4 (Restricted isometry property). *A matrix $A \in \mathbb{R}^{m \times n}$ satisfies the* restricted isometry property *(RIP) of order k if there exists a constant $\delta_k \in (0,1)$ such that for all k-sparse vectors $x \in \mathbb{R}^n$,*

$$(1-\delta_k)\|x\|_2^2 \leq \|Ax\|_2^2 \leq (1+\delta_k)\|x\|_2^2.$$

We say that A is (k, δ_k)-RIP if it satisfies the above property.

Matrices with this property behave like an isometry (or orthogonal matrix) when restricted to k-sparse vectors. If A was purely orthogonal it would have size $n \times n$, but we want it to be of size $m \times n$ where $m \ll n$. We can show that such matrices exist with high probability.

Definition 28.5 (Oblivious subspace embedding). *An* oblivious subspace embedding *(OSE) is a random linear map $S \in \mathbb{R}^{m \times n}$ (drawn from some distribution independent of any input subspace) such that for any fixed d-dimensional subspace $U \subseteq \mathbb{R}^n$, with probability $1-\delta$ over the random choice of S, for all $x \in U$,*

$$(1-\varepsilon)\|x\|_2^2 \leq \|Sx\|_2^2 \leq (1+\varepsilon)\|x\|_2^2.$$

Theorem 28.2. *If A is an oblivious subspace embedding for subspace dimension k, with probability $1-\delta$ where $\delta < 0.1/\binom{n}{k}$, then A is (k, δ_k)-RIP.*

Proof. Consider all possible sets $I \subseteq \{1, \ldots, n\}$ of size k. Each set I defines a k-dimensional subspace $U_I \subseteq \mathbb{R}^n$ spanned by the standard basis vectors e_j for $j \in I$,and there are $\binom{n}{k}$ such subspaces. Every k-sparse vector x lies in exactly one of those subspaces U_I. Now for each subspace U_I of dimension k, the OSE property guarantees that A is not (k, δ_k)-RIP with probability δ; union-bounding over all $\binom{n}{k}$ subspaces, the probability that A is not (k, δ_k)-RIP is at most $\delta \cdot \binom{n}{k} < 0.1$. $\square$

The following two results are easy exercises.

Claim 28.6. *We can construct a random matrix A with $m = O_\varepsilon(k + \log \delta^{-1}) = O(k \log(n/k))$ rows such that with probability $1-\delta$, A is (k, δ_k)-RIP.*

Theorem 28.3. *If A is a $(2k, \varepsilon)$-RIP matrix, then x^*, the solution to the ℓ_1-minimisation problem satisfies $\|x^* - x\|_1 \leq C \cdot \mathsf{Err}_1^k(x)$, where $C = 1 + O(\varepsilon)$.*

Definition 28.7 (Null-space property). *An $m \times n$ matrix A satisfies a* null-space property *of order k with constant C if for all $\eta \in \mathbb{R}^n$ such that $A\eta = 0$, and any set $T \subset \{1, \ldots, n\}$ of size k, we have*

$$\|\eta\|_1 \leq C\|\eta_{-T}\|_1 \implies \|\eta_T\|_1 \leq (C-1) \cdot \|\eta_{-T}\|_1,$$

where η_T is the subvector of η indexed by T, and η_{-T} is the subvector of η indexed by $\{1, \ldots, n\} \setminus T$.

Lemma 28.8. *If A is a $(k(2+r), \varepsilon)$-RIP matrix, then A satisfies the null-space property of order $2k$ with constant $C = \sqrt{2/r}(1+\varepsilon)/(1-\varepsilon)$, where $r \geq 1$.*

Lemma 28.9. *If A satisfies the $(2k, \varepsilon)$-null-space property with constant C for $\varepsilon < 1/2$, then*

$$\|x - x^*\|_1 \leq \frac{2\varepsilon}{1-\varepsilon}\mathsf{Err}_1^k(x),$$

where x^* is the solution to the ℓ_1-minimisation problem.

With these results (all exercises), we can now complete our basis pursuit strategy via iterative hard thresholding.

Iterative hard thresholding The basic idea of the algorithm we will discuss is to iteratively prioritise the relevant coordinates for producing a k-sparse vector. Observe that the function

$$P_k(z) = \arg \min_{\substack{z' \in \mathbb{R}^n \\ z \text{ is } k\text{-sparse}}} \|z - z'\|_1$$

is the function that keeps the k largest coordinates of z and sets the rest to zero. We can use this function to define the iterative hard thresholding algorithm, which takes in as inputs the observation vector $y = Ax$ and the number of iterations T, and returns a k-sparse approximation $x^{(T+1)}$ to the original signal x.

Algorithm: Iterative-Hard-Thresholding (IHT)

```
• Initialise the n-dimensional vector $x^{(0)} = (0, \ldots, 0)$.
• For $t = 1, 2, \ldots, T$:
    – $x^{(t+1)} = P_k(x^{(t)} + A^\top(y - Ax^{(t)}))$.
• Output $x^{(T+1)}$.
```

Intuitively, the function P_k works for the following reason. If we define $a^{(t+1)} = x^{(t)} + A^\top(y - Ax^{(t)})$, then we can see that although $x^{(t)}$ is inductively guaranteed to be k-sparse, $A^\top(y - Ax^{(t)})$ could introduce nonzero entries into other coordinates. However, the function P_k will zero out all but the k largest coordinates of $a^{(t+1)}$, restoring the k-sparsity of $x^{(t+1)}$, and the algorithm will converge to a k-sparse vector. In this sense, there's a similarity between this algorithm and the typical projection methods in constrained optimisation.

Although we will not prove it, the algorithm has the following guarantee.

Theorem 28.4 ([BD08]). *Suppose A is a $(3k, \varepsilon)$-RIP matrix, where $\varepsilon < 1/8$. Let $y = Ax + \widehat{\varepsilon}$, where $\widehat{\varepsilon}$ is the error term, then for all $T \geq 1$, the iterative hard thresholding algorithm* IHT *iterate $x^{(T+1)}$ satisfies*

$$\|x^{(T+1)} - x\|_2 \leq O(1) \cdot \left(2^{-T} \cdot \|x\|_2 + \frac{\mathsf{Err}_1^k(x)}{\sqrt{k}} + \|\widehat{\varepsilon}\|_2\right).$$

An easier version of this is below, and left as an exercise.

Theorem 28.5. *Suppose $x \in \mathbb{R}^n$ is k-sparse, and $\widehat{\varepsilon} = 0$, then $\|x^{(T+1)} - x\|_2 \leq O(2^{-T}) \cdot \|x\|_2$.*

References

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, 1996. 36, 37

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 574–583, 2014. 117

[AV88] Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. 112

[BC05] Bo Brinkman and Moses Charikar. On the impossibility of dimension reduction in l1. *Journal of the ACM (JACM)*, 52(5):766–788, 2005. 42

[BD08] Thomas Blumensath and Mike E Davies. Iterative thresholding for sparse approximations. *Journal of Fourier analysis and Applications*, 14:629–654, 2008. 129

[BH89] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the crcw pram. *Journal of the ACM (JACM)*, 36(3):643–670, 1989. 116

[BKS17] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):1–58, 2017. 117

[Bri21] Karl Bringmann. Fine-grained complexity theory: Conditional lower bounds for computational geometry. In *Connecting with Computability: 17th Conference on Computability in Europe, CiE 2021, Virtual Event, Ghent, July 5–9, 2021, Proceedings 17*, pages 60–70. Springer, 2021. 45

[Dan16] George B Dantzig. Linear programming and extensions. In *Linear programming and extensions*. Princeton university press, 2016. 80

[DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. 18

[DL$^+$16] Roee David, Bundit Laekhanukit, et al. On the complexity of closest pair via polar-pair of point-sets. *arXiv preprint arXiv:1608.03245*, 2016. 122

[FM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985. 23

[GSZ11] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer, 2011. 117

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998. 49, 53

[JL84] William B Johnson and Joram Lindenstrauss. Extensions of lipshitz mappings into hilbert spaces. In *Conference modern analysis and probability, 1984*, pages 189–206,

1984. 40

[JN10] William B Johnson and Assaf Naor. The johnson–lindenstrauss lemma almost characterizes hilbert space, but not quite. *Discrete & Computational Geometry*, 43(3):542–553, 2010. 43

[Kha79] Leonid Genrikhovich Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, volume 244, pages 1093–1096. Russian Academy of Sciences, 1979. 83

[KM20] CS Karthik and Pasin Manurangsi. On closest pair in euclidean metric: Monochromatic is as hard as bichromatic. *Combinatorica*, 40(4):539–573, 2020. 122

[KOR98] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 614–623, 1998. 48

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010. 117

[LN17] Kasper Green Larsen and Jelani Nelson. Optimality of the johnson-lindenstrauss lemma. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 633–638. IEEE, 2017. 43

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978. 7

[ST03] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of termination of linear programming algorithms. *Mathematical Programming*, 97:375–404, 2003. 80

[Tho13] Mikkel Thorup. Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 371–380, 2013. 27