CS 121.5 2019 Notes

Scribe: Noah Singer

Fall 2019

Contents

Introduction

These are notes from the 2019 iteration of CS 121.5, the weekly seminar-style discussion section for advanced topics in CS 121: Introduction to Theoretical Computer Science at Harvard University (see the course website https://cs121.boazbarak.org/index.html). The course was taught by Profs. Boaz Barak and Madhu Sudan.

This semester, we have been extremely fortunate to have some excellent guest talks in a diverse set of areas, as well as robust interest from our students. I am thankful to Profs. Barak and Sudan for helping me enlist many of the speakers, attending our weekly talks, and offering their insights and time in helping edit these notes. I am also thankful to Brian Ramirez for helping with logistics and with ordering our weekly coffee and cookies.

The course textbook [Bar19] uses a number of conventions that differ from standard introductory texts such as [Sip12]; a few of these notational differences have affected my notation in these notes. One primary difference is that we speak of *computing functions* $f : \{0,1\}^* \to \{0,1\}$ instead of *deciding languages* $\mathcal{L} \subset \{0,1\}^*$. CS 121 students are also familiar with circuits as a nonuniform model, since this is used as the primary "weak" model before introducing Turing machines; familiarity with circuits and circuit complexity is essential in Sections 2 and 7 and might be mentioned occasionally in other sections. Aside from that, these notes assume some level of mathematical background exceeding the baseline for students in the class; for instance, familiarity with eigendecompositions is crucial for the main result in Section 7, and notions of inference and learning are useful in Section 4.

After several of the talks, I consulted with the speakers and edited the notes to fill in background, details, or clarifications. I hope you enjoy!

— Noah Singer, *December 2019*

1 Madhu Sudan: *Error-Correcting Codes* (9/12/2019)

Biography

Prof. Sudan got his Ph.D. at Berkeley in the 90s. He has worked in industry at places like IBM and Microsoft, and he was also previously a professor at MIT. His interests include error-correcting codes and computational complexity; in particular, he was involved in proving major results about *probabilistically checkable proofs (PCPs)*[1]. An example application of a PCP is as follows: Say a professor wants to grade lots of homework in very little time; the professor's goal is to quickly scan a given proof (i.e., only look at a small set of steps throughout the proof), while being able to detect mistakes with high probability. While these devices are mostly theoretical because the resulting proofs are very large, work on PCPs has had lots of impact on things like digital contracts and cryptocurrencies. Prof. Sudan is also interested in communication and information in general. He will teach a course in error-correcting codes (CS 229R) in Spring 2019.

1.1 Motivations and a toy example

Any storage medium (hard drives, flash drives, etc.) is *noisy* — over long period of time, the introduction of errors is almost guaranteed (due to various internal and environmental factors, e.g., say, fluctuations in outside magnetic fields). Eventually, we expect that some random subset of zero bits will become ones and vice versa. Noisiness is also a huge challenge for communication channels, like Ethernet cables or wireless networks. And these errors are a really big deal; we don't want to have to worry about not being able to retrieve the information we want to store or transmit!

Error-correcting codes (ECCs) are used to recover from errors (and not just in theory, in the real world, too). Let's look at a simple but illustrative example of an ECC. Say we want to transmit a message such as

```
ARE YOU OK?
```

Before transmitting the message, we will duplicate each character a number of times:

```
AAA RRR EEE YYY OOO UUU OOO KKK ???
```

Then when we receive the message, we'll have a noisy sequence such as

```
ABA RRR EEF YYZ POR UVU OOO KLL ???
```

We can reconstruct the original message by taking the most common letter in each group of three:

```
ARE YPU OL?
```

As you can see, we've been able to recover most of the information in the original message; by using a more sophisticated code (or a code that produced longer encoded messages), we might have been able to recover the information entirely.

This ECC has a lot in common with the other ECCs we'll consider. In particular, it has an *encoding step* and a *decoding step*; the encoding step increases the length of the transmission to (hopefully) improve reliability, and the decoding step attempts to recover the original transmission. Nothing will be perfect, but we can ask questions like, if we're willing to put up with an encoded message that's n times longer than the original, how many errors might our code be able to correct?

These problems were first studied by Claude Shannon in a groundbreaking paper [Sha48], and many of these ideas were further studied by Hammond [Ham50].

[1] We might talk about these later in the semester.

1.2 Formal analysis of ECCs

An *alphabet* Σ is a set of allowed characters in a communication system (such as the binary alphabet $\Sigma = \{0, 1\}$ or the alphabet of letters $\Sigma = \{A, \ldots, Z\}$). An error-correcting code will be built from an *encoding function* $E : \Sigma^k \to \Sigma^n$ (i.e., mapping strings of length k to strings of length n in our alphabet). For instance, in the threefold repetition example we considered, we use an encoding function

$$E : \Sigma^k \to \Sigma^{3k} : (\sigma_1 \cdots \sigma_k) \mapsto (\sigma_1\sigma_1\sigma_1\sigma_2\sigma_2\sigma_2 \cdots \sigma_k\sigma_k\sigma_k).$$

There is also a complementary *decoding function* $D : \Sigma^n \to \Sigma^k$; these are complementary in the sense that they must satisfy the axiom that for all strings $x \in \Sigma^n$, $D(E(x)) = x$. But we really want a stronger error-correction property that formally captures a property like

$$D(E(x) + \text{error}) = x$$

for "small enough errors."[2] First, let's make a preliminary definition:

Definition 1 (Hamming distance)**.** *Given two strings $x = (x_0 \cdots x_{n-1})$ and $y = (y_0 \cdots y_{n-1})$ in Σ^n, the Hamming distance $\Delta(x, y)$ between x and y counts the number of positions at which the characters of x and y differ, i.e.,*

$$\Delta(x, y) = |\{i \in 0, \ldots, n-1 : x_i \neq y_i\}|.$$

For instance, $\Delta(abcd, bbdd) = 2$ since the strings differ in the second and fourth positions. The Hamming distance is a very useful technical tool for defining and studying error-correcting codes. It has the following properties:

- $\Delta(x, y) = 0$ if and only if $x = y$ for all strings $x, y \in \Sigma^n$
- $\Delta(x, y) = \Delta(x, y)$ for all strings $x, y \in \Sigma^n$
- $\Delta(x, z) \leq \Delta(x, y) + \Delta(y, z)$ for all strings $x, y, z \in \Sigma^n$

This last inequality might look familiar to you; it's the *triangle inequality*, and it, together with the other facts, means that Δ is a metric[3] on the space of strings Σ^n.

Definition 2 (t-error-correcting code)**.** *Given an encoding function $E : \Sigma^k \to \Sigma^n$ and $D : \Sigma^n \to \Sigma^k$, the pair (E, D) is a t-error-correcting code if for all $x \in \Sigma^k$, and $y \in \Sigma^n$,*

$$\Delta(E(x), y) \leq t \implies D(y) = x.$$

The concept of a t-ECC captures exactly the intuitive notion of a code that is able to correctly recover its input if at most t characters are corrected.

We call the possible values of $E(x)$ for $x \in \Sigma^k$ the *codewords* of our error-correcting code. We can visualize the universe of all possible encoded messages Σ^n as containing a bunch of "balls" around each codeword of radius t; if the code is a t-ECC, these balls will not intersect, and so within each of these balls, all messages are guaranteed to be decoded to the (decoding of the) central codeword.

From now on, let's make the assumption that $D(y)$ simply returns the decoding of the closest codeword (regardless of whether it's within some fixed t or not of the codeword). Mathematically, we can write

$$D(y) = \arg\min_{x \in \Sigma^k} \Delta(E(x), y).$$

This is a reasonable assumption, if, for instance, our underlying model is that all bits will be corrupted with identical and independent probability.

At this point, we can define two more important quantities that measure the "goodness" of ECCs:

[2] Note that this is actually a pretty strong condition — it implies that we want perfect recovery from errors! We could also study "approximate recovery" of x, but that would be a whole other problem.

[3] A *metric* is a distance-measuring function on a set. For instance, on the Cartesian plane $\mathbb{R}^2$, the typical metric is the Euclidean metric $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, but another valid metric is the taxicab metric $d((x_1, y_1), (x_2, y_2)) = |x_2 - x_1| + |y_2 - y_1|$. You can visualize a metric as telling us how "close" two points are.

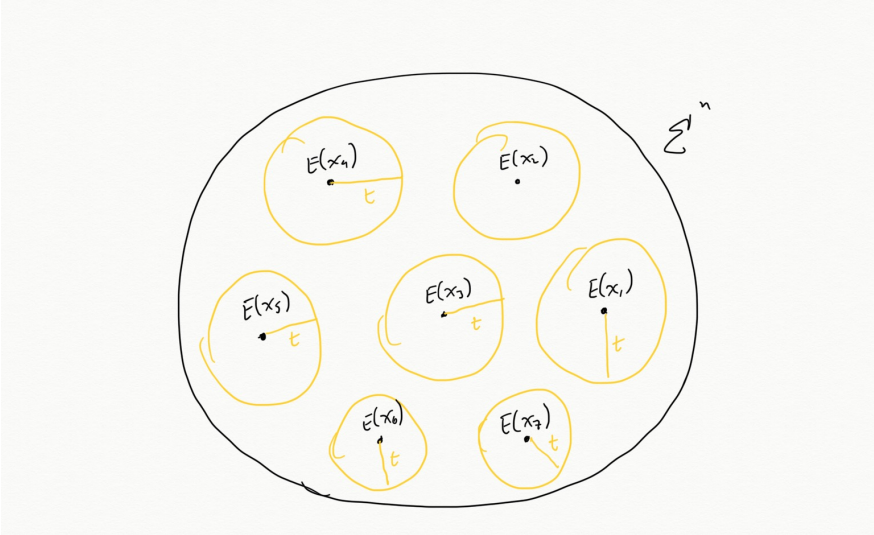

Figure 1: The "universe" Σ^n of all possible encoded messages. Seven codewords are depicted, and around each is a ball of radius t containing all messages that differ by t or fewer characters. Any corrupted message within one of these balls can be decoded unambiguously.

Definition 3 (Rate)**.** *The rate of an error-correcting code $E : \Sigma^k \to \Sigma^n$ is*

$$R(E) = \frac{k}{n}.$$

The rate measures the extra length of the encoded message relative to the original message; note that $R \leq 1$, and we want R to be as close to 1 as possible.

Definition 4 (Distance)**.** *The distance of an error-correcting code $E : \Sigma^k \to \Sigma^n$ is*

$$D(E) = \min_{x,x' \in \Sigma^k, x \neq x'} \{\Delta(E(x), E(x'))\}.$$

Having a higher distance is desirable because it means that there's more "room for error"; we can corrupt more characters of $E(x)$ without confusing the resulting message for some other x'.

So our threefold repetition code has rate $R = 1/3$, distance $D = 3$ (changing a single character in the unencoded message results in a 3-character difference in the encoded message), and is a 1-ECC (if we change both of the first two characters to the same new character, our decoding will be wrong). If we use $(2\ell + 1)$-repetition instead of 3-repetition, we'll get a rate $R = 1/(2\ell + 1)$, a distance $D = 2\ell + 1$, and the ability to correct ℓ errors.

This isn't a very efficient code. Say we want to correct 10 errors; we'd need a 21-fold repetition code and our encoded message would become 21 times longer than the original! Can we do better? Let's establish some basic facts about ECCs first!

Proposition. *In order for E to be a t-ECC, we must have $D(E) \geq 2t + 1$, and vice versa.*

This follows from the above picture; having a t-ECC means that the balls of radius t around the codewords do not intersect, and having $D(E) \geq 2t+1$ means that there is distance at least $2t+1$ between each codeword. We leave the formal details to the reader.

Theorem 5. *For every encoding function $E : \Sigma^k \to \Sigma^n$, if the rate $R(E) = k/n$, then $D(E) + k \leq n + 1$.*

Proof. Project all the $|\Sigma|^k$ codewords onto the first $k-1$ coordinates (i.e., take the first $k-1$ characters). Since this projection only has $|\Sigma|^{k-1}$ possible results, by the pigeonhole principle, we must have two codewords $E(x)$ and $E(x')$ that have the same first $k-1$ characters. But then they differ in at most $n-(k-1)$ characters, and so

$$D(E) + k \leq n.$$

□

We might expect that this bound is not very tight, but it actually is!

1.3 Reed-Solomon codes

Theorem 6. *For all k, n, there exists some alphabet Σ and an encoding function $E : \Sigma^k \to \Sigma^n$ such that $D(E) + k \geq n + 1$.*

We will actually prove this by constructing a particular set of codes called the *Reed-Solomon codes*.

Proof. Define $\Sigma = \mathbb{Z}/\mathbb{Z}p = \{0, 1, \ldots, p-2, p-1\}$ for some prime $p \geq n$ (essentially, we're doing all of our work modulo p). Given a message $m = (m_0, \ldots, m_{k-1}) \in \Sigma^k$, define the polynomial

$$M(x) = \sum_{i=0}^{k-1} m_i x^i.$$

But instead of sending the coefficients $m_0, \ldots, m_{k-1}$, we will send the sequence

$$E(m) = (M(0), M(1), \ldots, M(n-1)).$$

Since we're working in $\mathbb{Z}/\mathbb{Z}p$, we take all of these values modulo p; thus, they're not very large (at least compared to what they might be without taking the modulo).

What do we want to prove? For all *distinct* degree-$(k-1)$ polynomials M, P, we want to show that

$$\Delta((M(0), \ldots, M(n-1)), (P(0), \ldots, P(n-1))) \geq n - (k-1).$$

In other words, we want to prove that if $M \neq P$, then

$$|\{i \in 0, \ldots, k-1 : M(i) = P(i)\}| \leq k - 1.$$

An even nicer way to use the contrapositive: If we define $Q(x) = P(x) - M(X)$, we want to show that if Q is zero at least k values (in particular $0, \ldots, k-1$), then Q is the zero polynomial. But since Σ is a finite and algebraically closed field, the fundamental theorem of algebra applies; Q has degree $k-1$ and k zeros, so it must be the zero polynomial! □

These codes, aside from being theoretically important, also are used very commonly in storage media. But note that we rely on having a very large alphabet. What if we want a *binary* error-correcting code that we could actually use on a digital communication medium? (We can't just express each element of Σ in binary, since then we'd account for errors differently – in particular, in the above bound, multiple bit errors within a block corresponding to a particular element of Σ would only account for a single error.)

Shannon provides us with the following result:

Theorem 7. *For any n, there is some binary error-correcting code with encoding function $E : \{0,1\}^k \to \{0,1\}^n$ that has distance at most $n/10$ and rate at least $1/2$.*

We won't prove this theorem rigorously, but informally, he basically argues that if we let $n = 2k$, the number of balls of radius $n/10$ that can fit into the space of 2^n possible codewords is at least 2^k, and thus, that we can choose codewords for all Σ^k messages. While this argument is theoretically interesting, note that it gives us no guarantees about how long E and D would actually take to run on a practical computer; in fact, since we constructed them in such a general way, we can't guarantee better than an exponential runtime in k (i.e., iterating through all 2^k selected codewords and seeing which is closest). Thus, a very major area of interest in error-correcting codes has been constructing both good and efficiently computable binary codes.

If you want to explore error-correcting codes further, see [Sud08], or take CS 229R in the spring!

2 Sasha Golovnev: *Circuit Lower Bounds* (9/19/2019)

Biography

Sasha Golovnev is currently a postdoctoral researcher here at Harvard studying theoretical computer science, and he has lots of interests. For instance, he's interested in finding better algorithms for NP-hard problems[4]. For instance, he likes the *traveling salesman problem* (TSP), which is the problem of finding a so-called *Hamiltonian* cycle — a cycle through a given graph that visits every vertex exactly once. He's also interested in *data structure* problems where you get some data, get to do some preprocessing on it, and then have to answer a bunch of questions about it. For instance, he's studied the *nearest neigbhor* problem, where you get a collection of points (say $x_1, \ldots, x_n \in \mathbb{R}^n$), and then after some preprocessing (i.e., creating your data structure), have to be able to quickly determine which of the points x_i is closest to some new point $y \in \mathbb{R}^n$; there are approximate versions of this problem (e.g. find one of the k closest neighbors), as well as generalizations to different distance metrics on $\mathbb{R}^n$ or even spaces aside from $\mathbb{R}^n$. At Apple, he studied a version of the nearest neighbor problem where the goal is to find the word in a set of words most similar to a given word. He's also interested in other areas like cryptography and machine learning. But his favorite area is Boolean circuits, which is what we'll discuss today.

2.1 Setup

As we discussed in class, circuits are equivalent to *straight-line programs*, which are programs in which each line corresponds to a gate. We like studying straight-line programs exactly because they are straight lines; we have no complicated behavior stemming from branches or loops.

We'll study circuits over the *full binary basis*; this just means that we'll allow all 16 possible 2-input gates (as opposed to just considering NAND or AND, OR, and NOT). We might call this language FULL-CIRC, as opposed to NAND-CIRC or AON-CIRC. Note that since these are all universal bases of gates, they have equivalent power; however, e.g., it might take up to C times as many gates to represent a function in NAND-CIRC as opposed to FULL-CIRC for some constant factor C, and since these constants will be extremely important in the results of this talk, we'll stick with FULL-CIRC.

Any n-input circuit C computes some $f : \{0,1\}^n \to \{0,1\}$. We proved in class that every such function can be computed in at most 2^n lines; this can be strengthened using a dynamic-programming technique to $2^n/n(1 + o(1))$ lines [Lup58]. Furthermore, we showed that there exist functions that require $2^n/n$ lines; in fact, we could show that in a certain sense, "most" Boolean functions $f : \{0,1\}^n \to \{0,1\}$ require $c2^n/n$ lines for some constant c [Sha49]! And in general, we know that there are circuits with all kinds of size complexities:

Theorem 8 (Size hierarchy theorem)**.** *For any function $T(n) < 2^n/n$, there is some function $f : \{0,1\}^n \to \{0,1\}$ such that $T(n) - n \leq \text{Size}(f) \leq T(n)$.*

[4] We'll learn what it means to be NP-hard, and consider examples of NP-hard problems, later in the course — for now, suffice it to say that these are "reasonable" problems that we don't have polynomial-time algorithms for and believe are in general hard to solve.

Proof. Suppose that $T(n) < 2^n/n$. Then consider the constant zero function z (i.e., $\text{Size}(z) = 1$, if we incorporate a constant zero gate), and the "hard" function f that we mentioned above with $\text{Size}(f) = 2^n/n$. We want to "interpolate" between these two functions to come up with a series of increasingly difficult functions, including our desired function.

First, we claim as a lemma that for any $g_1, g_2 : \{0,1\}^n \to \{0,1\}$ that only differ on a single input $y \in \{0,1\}^n$,

$$|\text{Size}(g_1) - \text{Size}(g_2)| \leq n.$$

See the proof below.

Now, suppose that all the inputs producing nonzero outputs of f are $X = \{x_1, \ldots, x_k\} \subset \{0,1\}^n$.[5] We can define a sequence of functions $g_0, g_1, \ldots, g_{k-1}, g_k : \{0,1\}^n \to \{0,1\}$ where g_{i+1} is 1 exactly on the values $x_1, \ldots, x_i$. Thus, $g_0 = z$ and $g_k = f$. Furthermore, each pair (g_i, g_{i+1}) differ on exactly one input value, so we know that for each i, $|\text{Size}(g_i) - \text{Size}(g_{i+1}) \leq n$ by the lemma.

At this point, we have broken up the interval of all possible size complexities $[1, 2^n/n]$ into subintervals

$$[\text{Size}(g_i), \text{Size}(g_{i+1})],$$

each of size at most n; this means that if we take any $T(n) < 2^n/n$, it will fall into some subinterval that starts at $\text{Size}(g_i)$ for some i. $T(n) - n$ will fall into the previous subinterval, and so we have exactly that

$$T(n) - n \leq \text{Size}(g_i) \leq n,$$

as desired. See 2 for a helpful picture. □

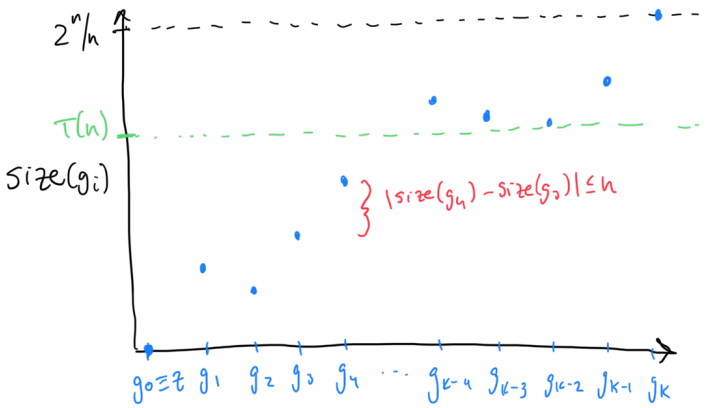

Figure 2: What a graph of successive values of $\text{Size}(g_i)$ might look like. $\text{Size}(g_i)$ and $\text{Size}(g_{i+1})$ never differ by move than n. $\text{Size}(g_k) = \text{Size}(f) = 2^n/n$, while $\text{Size}(g_0) = \text{Size}(z) = 0$. Any value $T(n) < 2^n/n$ must fall into the interval between some $\text{Size}(g_i)$ and $\text{Size}(g_{i+1})$, which gives us the desired result.

Proof of lemma. Let $\mathcal{C}(C)$ denote the number of gates in any circuit circuit C. Let $g_1, g_2 : \{0,1\}^n \to \{0,1\}$ be function that differ on only one input $y \in \{0,1\}^n$. Suppose WLOG that $g_1(y) = 0$ while $g_2(y) = 1$. Furthermore, suppose a circuit C_1 computes g_1. Consider the circuit C_2 that computes

$$C_1(x) \vee (x_1 = y_1 \wedge x_2 = y_2 \wedge \cdots \wedge x_n = y_n).$$

[5] This is often called the *support* of f.

This uses at most $\mathcal{C}(C_2)+n$ gates (since the y_i's are fixed, we're essentially just using a series of $n-1$ AND's with some of the inputs possibly negated, along with the one OR). Thus, $\text{Size}(g_1) \leq \text{Size}(g_2)+n$. Similarly, given any circuit C_2 computing g_2, the circuit

$$C_2(x) \vee \overline{x_1 = y_1 \wedge x_2 = y_2 \wedge \cdots \wedge x_n = y_n}$$

computes g_1 in at most $\mathcal{C}(C_2)+n$ gates. We conclude that since $\text{Size}(g_2) \leq \text{Size}(g_1)+n$, $|\text{Size}(g_1)-\text{Size}(g_2)| \leq n$. □

2.2 Applications

So why are we interested in studying circuits? As we'll see later in the course, *the main goal of theoretical computer science is to establish lower bounds for the complexity of certain problems*; that is, to prove that *no algorithm* exists that solves a given problem in time less than $T(n)$.[6] There's a general theorem that says given an algorithm that computes a general function $f : \{0,1\}^* \to \{0,1\}$ in time $T(n)$, we can produce a *sequence of circuits* $C_1, C_2, \ldots$ such that

$$\mathcal{C}(C_n) = O(T(n) \log T(n)).$$

(This should make sense because we can "unroll the loop" or loops in the general algorithm to produce a single straightline program; details will come later in the book.) So in order to prove lower bounds for general algorithms, we could maybe instead prove some lower bounds for circuits. This might be a lot easier — after all, circuits are just combinatorial objects (essentially DAGs with labelled vertices); we don't have to worry about branching, recursion, etc.

Advanced note: In terms of the complexity classes we'll see later in the course, we say that

$$\mathbf{P} \subset \mathbf{P/poly}.$$

To prove that $\mathbf{P} \neq \mathbf{NP}$, it would suffice to prove that

$$\mathbf{NP} \not\subset \mathbf{P/poly}.$$

This is conjectured to be true, even though it is a strictly stronger assertion than $\mathbf{P} \neq \mathbf{NP}$, and it is often considered as one of the more viable ways to approach a proof of $\mathbf{P} \neq \mathbf{NP}$ since circuits are so much easier to work with than general algorithms. And to show that $\mathbf{NP} \not\subset \mathbf{P/poly}$, it would suffice to find an **NP** problem for which we could prove a superpolynomial circuit lower bound. Also, note that circuits do give us extra power as compared to polynomial-time logarithms; for instance, there is a family of trivial circuits that solve the *unary halting problem* (on input x, decide whether the $|x|$-th Turing machine halts on any input), but no general algorithm exists that can solve this problem. This means that $\mathbf{P} \subsetneq \mathbf{P/poly}$. In general, such *nonuniform* models of computation (roughly, models that are allowed to have different behavior on each input size) are strictly stronger than uniform models (such as Turing machines); if we wanted, we could impose the condition on a family of circuits $C_1, C_2, \ldots$ that an algorithm must exist that outputs each C_n in logarithmic space in n, and this would be equivalent to a uniform algorithm model.

2.3 A simple circuit lower bound

So let's try and establish some basic circuit lower bounds. What's a good baseline? Well, consider the function $\textsc{Xor} : \{0,1\}^n \to \{0,1\}$ that computes the exclusive-or of all its inputs:

$$\textsc{Xor}(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

It's easy to see that $\text{Size}(\textsc{Xor}) \geq n-1$ since the output of Xor changes if any one of its inputs is changed. But can we do better than this trivial bound?

[6] We'll formalize what these "general algorithms" mean later in the course, but your intuitive understanding should be enough for now, since we're just motivating why we care about them.

Let's define the three functions $\text{MOD}^3_{n,0}$, $\text{MOD}^3_{n,1}$, and $\text{MOD}^3_{n,2}$ where

$$MOD^3_{n,r} = \begin{cases} 1 & \sum_{i=1}^n x_i \equiv r \pmod 3 \\ 0 & \text{otherwise.} \end{cases}$$

So, for instance, $\text{MOD}^3_{n,0}$ is 1 iff n is divisible by 3.

Theorem 9. *For all $n \in \{1, 2, \ldots\}$ and $r \in \{1, 2, 3\}$,*

$$\text{Size}(\text{MOD}^3_{n,r}) \geq 2n - 6.$$

The proof of this theorem follows from a general technique called *gate elimination*; the basic idea is that we can reduce one circuit to a smaller circuit by arguing that we can get rid of some gates. The theorem was first proved in [Sch74].

Proof. We will proceed by induction on n. The base case is trivial for $n \leq 3$ since the lower bound is negative. Now consider any $n \in \{1, 2, \ldots\}, r \in \{1, 2, 3\}$. Consider any circuit C computing $\text{MOD}^3_{n,r}$. Pick some gate f in C; WLOG let f be fed by inputs x_1 and x_2. See 3 for a graphical explanation.

Case 1: x_1 and x_2 both feed only f. Consider the $(n-2)$-input circuits $C_{00}, C_{01}, C_{10}, C_{11}$ which are identical to C except that in C_{ij}, we set $x_1 = i$ and $x_2 = j$. More formally, C_{ij} is a circuit where we eliminate x_1, x_2, and f, and now feed $f(x_1, x_2)$ directly into whatever gates f previously fed — this will cause them to change, but that's okay because we allow all 16 possible gates in our circuit.

Now, we observe that since f only has two possible outputs, $C_{00}, C_{01}, C_{10}, C_{11}$ can only compute two distinct functions among them. However, by the definition of modulus, $C_0 0$ should compute $\text{MOD}^3_{n-2,r}$; C_{01} and C_{10} should compute $\text{MOD}^3_{n-2,r-1}$, and C_{11} should compute $\text{MOD}^3_{n-2,r-2}$. But these three functions are distinct, and so this case is impossible. We conclude that at least one of the inputs x_1 or x_2 must feed a gate other than f.

Case 2: Suppose WLOG that x_1 feeds f and some other gate g (which has some other input x_3). Furthermore, suppose for the sake of contradiction that C has fewer than $2n - 6$ gates. Then we could just consider the circuit C_0 where we fix $x_1 = 0$ and eliminate f and g, absorbing the behavior of $f(0, x_2)$ and $g(0, x_3)$ into the gates that they feed. (If x_1 feeds more gates, we cut them out, too.) Thus, C_0 computes $\text{MOD}^3_{n-1,r}$ with fewer than $2n - 6 - 2 = 2n - 8$ gates; however, by the inductive hypothesis, every circuit computing $\text{MOD}^3_{n-1,r}$ requires at least $2n - 8$ gates, and so we have a contradiction.

Thus, C has at least $2n - 6$ gates, as desired. □

This might seem like a "cute" argument of little importance, but after 40 years of research, the best known lower bound is only about $3n$. Furthermore, we have actually proven that none of our current techniques for proving circuit bounds (i.e., the gate elimination technique we just used) do much better. (Also, this is a tight bound for $\text{MOD}^3_{n,r}$ — we can actually construct circuits for $\text{MOD}^3_{n,r}$ requiring exactly $2n - 6$ gates).

2.4 Upper-bounding circuit complexity of symmetric functions

So could we hope to prove much stronger lower bounds for functions that are similar to $\text{MOD}^3_{n,r}$? As it turns out, the answer is no. From now on, for ease of discussion, let's assume that n is a power of 2.

Definition 10 (Symmetric Boolean function)**.** *A $f : \{0,1\}^n \to \{0,1\}$ is symmetric if it can be written as a function of the sum of its bits, i.e., there is some function $g : \{0,1\}^{\log_2 n+1} \to \{0,1\}$ such that*

$$f(x_1, \ldots, x_n) = g\left(\sum_{i=1}^n x_i\right).$$

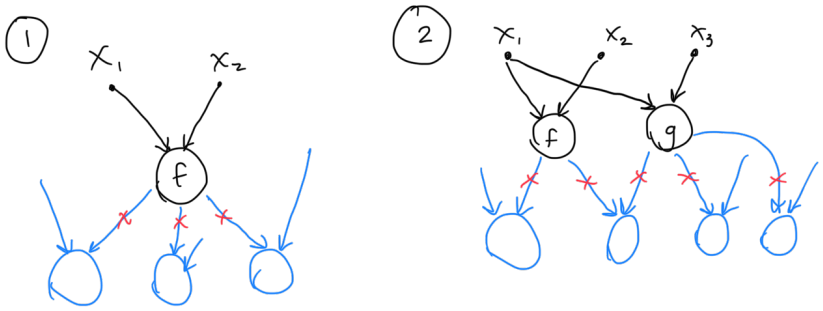

Figure 3: *Case 1:* x_1 and x_2 only feed f. In this case, we could cut x_1, x_2, and f out of the circuit; this case is impossible because the circuits we get could only compute two distinct functions among them, while restricting x_1 and x_2 should enable us to compute the functions $\textsc{Mod}^3_{n-2,0}$, $\textsc{Mod}^3_{n-2,1}$, and $\textsc{Mod}^3_{n-2,2}$, which are all distinct. *Case 2:* x_1 feeds at least two gates, f and g. In this case, we could set $x_1 = 0$ and cut f and g out of the circuit (along with all the other gates that x_1 feeds), leading two a circuit for computing $\textsc{Mod}^3_{n-1,r}$ with at least two fewer gates.

An equivalent definition of a symmetric function is that the value of f does not change if you permute its inputs; in other words, it does not depend on the order of its inputs, only upon the number of its inputs that are 1's. Examples of symmetric functions on n inputs include the n-bit XOR, AND, and OR, as well as our friends $\textsc{Mod}^3_{n,r}$ for each r. And it turns out that we can't hope to prove much stronger lower bounds for symmetric functions:

Theorem 11. *For any symmetric function* $f : \{0,1\}^n \to \{0,1\}$,

$$\text{Size}(f) \leq 4.5n + o(n).$$

This was proved in [Dem+10]; we will prove the slightly weaker bound that $\text{Size}(f) \leq 5n + o(n)$.

Proof. Since g has only $\log_2 n+1$ inputs, there is a circuit that computes it in at most $2^{\log_2 n+1}/(\log_2 n+1) \approx n/\log_2 n = o(n)$ gates. Thus, if we define the function

$$SUM : \{0,1\}^n \to \{0,1\}^{\log_2 n+1} : (x_1, \ldots, x_n) \mapsto \sum_{i=1}^{n} x_n,$$

if we can show that $\text{Size}(SUM) \leq 5n$, then since $f(x) = g(SUM(x))$, we have that $\text{Size}(f) \leq 5n + o(n)$.

We will do this by explicitly constructing an *adder* circuit that computes SUM. First, we define an *adder gadget* that uses five gates as in Figure 4. The gadget has three one-bit inputs, x_1, x_2, and x_3, and two outputs, the *parity bit* y_1, and the *carry bit* y_2. y_1 will be 1 iff there is an odd number of 1's among x_1, x_2, and x_3; y_2 will be 1 iff at least two of x_1, x_2, and x_3 are 1. In other words, this circuit performs addition on three 1-bit binary numbers; $x_1 + x_2 + x_3 = 2 \cdot y_2 + y_1$.

Then we can chain a bunch of gadgets together in layers to actually compute the sum of $x_1, \ldots, x_n$, as in Figure 5. There are $\log_2 n$ layers. The first layer, which contains roughly $n/2$ gadgets, will compute the first bit y_1 of the output by just XORing $x_1, \ldots, x_n$ together (we say "roughly" because we might have slight issues with divisibility, but this isn't a big deal as it adds at most constant overhead per layer, which can be absorbed into the $o(n)$). This process will also produce roughly $n/2$ carry bits, which will then be fed into the second layer. In the second layer, $n/4$ adder gadgets will XOR together the $n/2$ carries from the first layer to produce the second bit y_2 of the output; $n/4$ carry bits will also be produced. In general, the i-th

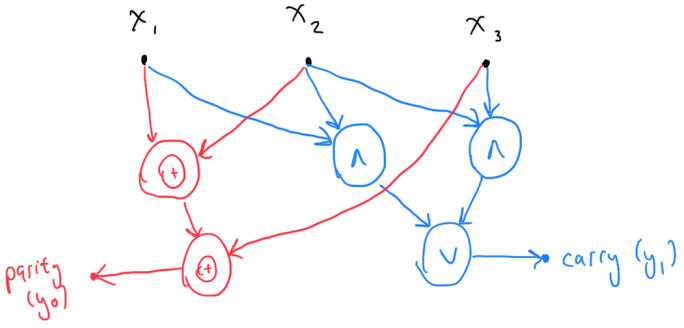

Figure 4: The 5-gate adder gadget. There are three inputs x_1, x_2, and x_3, and the adder produces two bits y_1 (the *parity*) and y_2 (the *carry*) such that $x_1 + x_2 + x_3 = 2 \cdot y_2 + y_1$.

layer will have roughly $n/2^i$ gadgets, XORing the $n/2^{i-1}$ carry bits from the previous layer to produce y_i, and creating $n/2^i$ carry bits for the next layer in the process. The final layer, $y_{\log_2 n}$, will produce only a single carry bit, which then simply becomes $y_{\log_2 n+1}$. In the end, we use a total of

$$\frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1 = n$$

gadgets to produce the output $y_1 y_2 \cdots y_{\log_2 n} y_{\log_2 n+1}$, where

$$\sum_{i=1}^{n} x_i = \sum_{j=1}^{\log_2 n+1} 2^{j-1} y_j.$$

We use a total of 5 gates per gadget, and so the total number of gates used is $5n$, as desired.

For more background on this subject, see [Juk12, Chapter 1] and [AB12, Chapter 6]. □

3 Nada Amin: *The λ-Calculus* (9/27/2019)

Biography

Prof. Nada Amin grew up in Geneva, Switzerland; she did her Bachelors and Masters at MIT before working as a software engineer at Google Zürich and getting her Ph.D. at EPFL. She was a lecturer at Cambridge University before joining the faculty at Harvard. In general, she studies programming languages; specific things that she's interested in include:

- The mathematical theory of types that underlies programming languages
- *Generative programming*: reasoning about programs that themselves write other programs, and constructing "hierarchies" of increasingly complex programs in this manner
- Generating representations of non-traditional means of computation via programming languages (e.g. *neuro-symbolic programming*)
- *Reflection*: Reasoning about programs that can access or modify their own code or constructs
- *Verification*: Building or reasoning about systems that can prove that programs have certain semantic properties

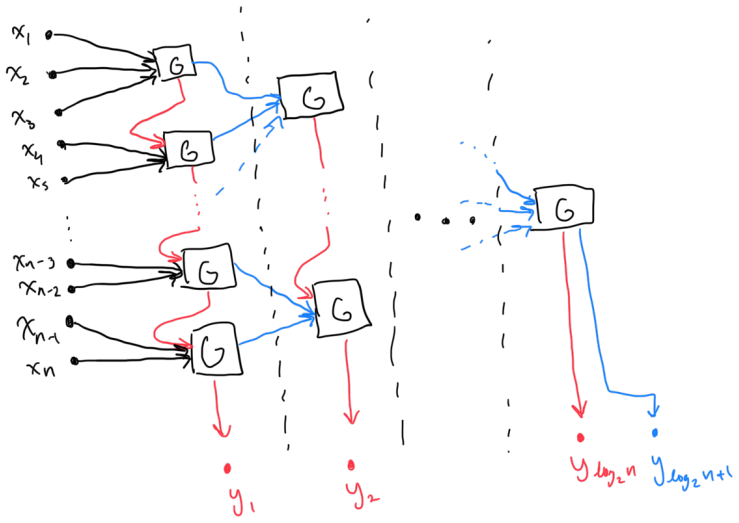

Figure 5: The adder for n bits $x_1, \ldots, x_n$. There are a series of layers, each of which uses half as many gadgets as the previous, and XORs its input to produce a parity bit for the output, as well as producing a series of carry bits that are used by the next layer. In the end, a string of $\log_2 n+1$ bits $y_1 y_2 \cdots y_{\log_2 n} y_{\log_2 n+1}$ is produced that represents the sum of the bits $x_1, \ldots, x_n$.

In her daily life, Prof. Amin uses programming languages that include Common LISP, Clojure, Racket/Scheme, OCaml, Coq (a proof assistant), and Scala. All of these are extensions or applications of the λ-calculus formalism that we'll discuss today.

3.1 Intuition and formalism

The *λ-calculus* is a universal model of computation that is radically different than, for example, Turing machines. The *syntax* of λ-calculus (i.e., the so-called *λ-expressions* that are valid strings in the language of λ-calculus) is very simple; it consists of strings that can be built up in three ways:

1. *Variables*: A variable x is a λ-expression.
2. *Function abstractions*: Given a variable x and a λ-expression e, $\lambda x.e$ is a valid expression.
3. *Function application*: Given two λ-expressions e_1 and e_2, $e_1\ e_2$ is a valid λ-expression.

So, for instance, $\lambda x.\lambda y.x\ (y\ x)$ is a valid λ-expression.

At this point, these are just some peculiar-looking strings. First, let's give a high-level interpretation of what these strings mean; then we can formally define them as a model at a more technical level.

From an intuitive standpoint, each of these strings represents a function. The string $\lambda x.e_1$ represents the function mapping the variable x to the expression e_1 (which might involve x). We might think of x as the function's *parameter* and e_1 as the function's *body*. For instance, $\lambda x.x$ is the identity function. All these functions are *anonymous* — they don't have names. For comparison, traditionally, we might define a function $f(x) = x^2 + 1$ and then refer to the function f. In λ-calculus, we'd instead just refer to the expression $\lambda x.(x^2 + 1)$; this represents the anonymous function we might write mathematically as

$$x \mapsto x^2 + 1$$

(read: "x maps to $x^2 + 1$"). (We'll eventually see how to represent things like arithmetic in λ-calculus). In effect, we are separating the creation and naming of a function.[7] Correspondingly, we can "apply" a function by writing it and then its argument; for instance, $(\lambda x.x^2)\ 7$ is 49.

What if we want to represent a function that takes multiple arguments? We interpret a function of two variables as a function mapping one variable to another function that maps the second variable to the output! For example, $\lambda x.\lambda y.(x^2 + y^2)$ "represents" a function that maps x to a second function f_x, which maps y to $x^2 + y^2$. In math terms,

$$x \mapsto (y \mapsto x^2 + y^2).$$

This is often called *currying*.

All of this talk about λ-expressions representing function is useful for building intuition, but it is not a formal definition of λ-calculus as a model of computation. Let's return to just thinking about λ-expression as strings. We introduce three types of rules that let us relate λ-expressions:

- The most important rule is called *β-reduction*. We say that a λ-expression of the form $(\lambda x.e_1)\ e_2$ can be *β-reduced* to the substituted expression $e_1[x := e_2]$ (this means to take e_1 and then replace any instances of the variable x with e_2).[8] Expressions of the form $(\lambda x.e_1)\ e_2$ are called *β-redexes*; they are the locations at which β-reduction could take place. For instance, the following λ-expression contains three β-redexes:

  $$\underline{(\lambda x.x)\ (\underline{(\lambda x.x)}\ (\lambda z.\underline{(\lambda x.x)}\ z))}$$

  Thus, a single β-reduction could result in $(\lambda x.x)\ (\lambda z.(\lambda x.x)\ z)$ (either the outer or middle β-redex) or $(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.z))$ (the inner β-redex).

- A λ-expression $\lambda x.e_1$ is *α-equivalent* to the λ-expression $\lambda y.e_1[x := y]$. This simply reflects the fact that renaming the parameter doesn't change the underlying function.

- A λ-expression $\lambda x.(f\ x)$ can be *η-converted* into f and vice versa, as long as x doesn't appear in f. This should make sense, since it captures the fact that $\lambda x.(f\ x)\ y$ will always be equal to $f\ y$ and so $\lambda x.(f\ x)$ and f represent the same underlying function.

3.2 Encodings

In λ-calculus, everything is a function; thus, to start building up notions of computation, we'll want some way to represent things like numbers and Boolean values as functions. Essentially, we'll be building up syntactic sugar; we'll define some expression, and then whenever we use it later, we really mean that we're substituting all of our definitions in to make one long string.

We'll start by defining λ-calculus encodings of Booleans. We'll just define true and false as

$$\texttt{true} := \lambda x.\lambda y.x \quad \text{and} \quad \texttt{false} := \lambda x.\lambda y.y.$$

In other words, we interpret `true` as taking two inputs and returning the first input, while `false` takes two inputs and returns the second input. Then we can express "if" as

$$\texttt{if} := \lambda b.\lambda x.\lambda y.b\ x\ y.$$

Note that if the first input b is `true`, then by applying b to the last two inputs x and y, we will get x; similarly, if b is `false`, we will get y, as desired. For instance, `if true false true` results in `false`,

[7] Some extensions of λ-calculus have expressions of the form "let x be e_1 in e_2", where x is a variable and e_1 and e_2 are expressions. These *let-forms* allow naming functions. However, this is just "syntactic sugar" for the λ-expression $(\lambda x.e_2)\ e_1$, as we'll see.

[8] Actually, there are certain technical conditions we must add to ensure that this substitution is *capture-avoiding*; for instance, if e_1 itself contains an expression of the form $\lambda x.e_3$, we don't want to substitute e_2 for x in e_3. However, this detail isn't very important for understanding how λ-calculus works.

while `if false false true` results in `true`.[9] And by "results in," we mean that there is some sequence of β-reductions — function applications — along with renamings and η-conversions that makes the more complicated expression become the less complicated expression. A good exercise at this point is to define `and`, `or`, and `not`.

We'll continue by defining λ-calculus encodings of the natural numbers. Intuitively, the λ-calculus expression representing some natural number n will be a function taking a function s and a value z that applies s to z exactly n times. For instance, we'll define

$$\texttt{zero} := \lambda s.\lambda z.z;$$

so `zero` "takes in" a function s and a value z and just returns z, thereby applying s zero times. Next,

$$\texttt{one} := \lambda s.\lambda z.s\ z;$$

this applies s once to z. And

$$\texttt{two} := \lambda s.\lambda z.s\ (s\ z).$$

`six` will be $\lambda s.\lambda z.s\ (s\ (s\ (s\ (s\ (s\ z)))))$.

Okay, so we have numbers. But how can we do arithmetic? Let's start by defining the *successor function* (i.e., the function mapping n to $n+1$). There are two equivalent ways to do this. First, we could write

$$\texttt{succ} := \lambda n.\lambda s.\lambda z.s\ (n\ s\ z).$$

The interpretation is as follows: `succ` n is a number; it takes some function s and value z, and then applies s n times to z (this is the sub-expression $n\ s\ z$), and then applies s one more time to z. Alternatively, we could have written

$$\texttt{succ} := \lambda n.\lambda s.\lambda z.n\ s\ (s\ z);$$

here, we apply s to z once and then apply it n times. Next, we can do addition. We can do it directly:

$$\texttt{plus} := \lambda n.\lambda m.\lambda s.\lambda z.n\ s\ (m\ s\ z).$$

Here, we just apply s m times to z, and then apply it n more times. Alternatively, we could write

$$\texttt{plus} := \lambda n.\lambda m.n\ \texttt{succ}\ m;$$

this means that we'd apply the successor function n times to m. We can also do multiplication:

$$\texttt{times} := \lambda n.\lambda m.\lambda s.\lambda z.n\ (m\ s)\ z.$$

Here, we *partially apply* m to s, so that $m\ s$ is a function that applies s m times to its argument; then, we apply the function $m\ s$ itself n times to z, resulting in a total of $n \times m$ applications of s to z. Alternatively, we could write

$$\texttt{times} := \lambda n.\lambda m.n\ (\texttt{plus}\ m)\ \texttt{zero}.$$

This would correspond to defining the partially applied `plus` m function, which adds m to its input, and then repeating this function n times on zero.

At this point, we have addition and multiplication. The next logical step is to try and implement some type of decreasing operation, like subtraction. We might even want to do something easier — like the *predecessor function* (mapping n to $n-1$). It turns out, though, that this is pretty difficult.

First, we'll need a way to represent *pairs* (ordered tuples) of expressions. We'll say

$$\texttt{pair} := \lambda x.\lambda y.\lambda b.b\ x\ y.$$

[9] Note that the implied parentheses associate from left-to-right; that is, `if true false true` means ((`if true`) `false`) `true`.

In effect, we'll partially apply `pair` to two values x and y; the pair is then stored as a function of some input b that returns x if b is `true` and y if b is `false`. Conversely, given a pair, we might want to extract its first and second elements:

$$\texttt{fst} := \lambda p.p\ \texttt{true} \quad \text{and} \quad \texttt{snd} := \lambda p.p\ \texttt{false}.$$

Thus, we should have that, e.g., `fst` (`pair one two`) is `one`, while `snd` (`pair one two`) is `two`.

How does this help us represent the predecessor? We claim that we can write

$$\texttt{pred} := \lambda n.\texttt{fst}\ (n\ (\lambda p.(\texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{succ}\ (\texttt{snd}\ p))))\ (\texttt{pair zero zero})).$$

Let's break this down. `pred` takes an input n. First, it applies the complicated-looking function

$$\lambda p.(\texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{succ}\ (\texttt{snd}\ p)))$$

n times to the base value (`pair zero zero`). We might call the above function a "helper function" — it takes a pair (a, b) and returns $(b, b+1)$, ignoring the first value a. Since this is applied n times to $(0, 0)$, we end up with $(n-1, n)$. Then `pred` simply takes the first element of this pair.

Woohoo! We can basically do all arithmetic now. We could define subtraction by taking repeated predecessors, or exponentiation by taking repeated products. But now, we'll move on to what makes λ-calculus truly powerful: *recursion*.

Typically, when we write recursive functions in a normal programming language, we name a function and then let it refer to its own name in its code. Part of the challenge with recursion in λ-calculus is that since all of our functions are anonymous, they can't refer to their own names! Our approach will be to try and get around this by passing a function into itself as an argument. For instance, we might try and compute the factorial function as

$$\texttt{fct} := \lambda f.\texttt{if}\ (\texttt{zero?}\ n)\ \texttt{one}\ (\texttt{times}\ n\ (f\ (\texttt{pred}\ n))),$$

where `zero?` is the predicate

$$\texttt{zero?} := \lambda n.n\ (\lambda x.\texttt{false})\ \texttt{true}$$

that returns `true` iff its input n is `zero`. Then if we could pass `fct` in for f, we would be all set! Unfortunately, we can't simply pass `fct` into itself, because the argument `fct` itself would have to have its own copy of `fct` passed into it, and so on and so on. We need some way to get at the concept of an infinite loop — that is, we want to build the ability to "pass something into itself an infinite number of times."

First off, how can we make a normal infinite loop? Consider the λ-expression

$$\texttt{inf} := (\lambda x.x\ x)\ (\lambda x.x\ x).$$

If we try and β-reduce `inf`, we'll just get `inf` again, forming an infinite loop of β-reductions!

Now, let's try and harness the "infinite expansion" idea behind `inf` to allow us to perform useful recursion — that is, we want to apply f to an input an infinite number of times (note that if f has a proper base case, like `fct` above, it will terminate after a finite number of times). Our goal is to produce an expression called `Y` (for historical reasons) that has the property that $\texttt{Y}\ f = f\ (\texttt{Y}\ f) = f\ (f\ (\cdots f(\texttt{Y}\ f)))$; this expression is generally called the *fixed-point combinator* or *Y-combinator*. Its form is very similar to `inf`:

$$\texttt{Y} := (\lambda f.(\lambda x.f\ (x\ x)))\ (\lambda x.f\ (x\ x)).$$

Note that

$$\begin{aligned}\texttt{Y}\ f = (\lambda f.((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))))f &\to (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\ &\to f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) = f\ (\texttt{Y}\ f).\end{aligned}$$

where each arrow is a β-reduction. Thus, for instance, via a chain of reductions, we will have

$$\begin{aligned}
\texttt{Y fct three} &\to \texttt{fct (Y fct) three} \\
&\to \texttt{times three (Y fct two)} \\
&\to \texttt{times three (fct (Y fct) two)} \\
&\to \texttt{times three (times two (Y fct one))} \\
&\to \texttt{times three (times two (fct (Y fct) one))} \\
&\to \texttt{times three (times two (times one (Y fct zero)))} \\
&\to \texttt{times three (times two (times one (fct (Y fct) zero)))}.
\end{aligned}$$

But this final `fct (Y fct) zero` can be immediately reduced to `one`, since `zero? one` is `true`.

3.3 λ-calculus as a model of computation

From a computational standpoint, λ-calculus as we've seen it hasn't been fully specified. In particular, for certain expressions, there might be many possible reduced forms of that expression (for instance, if there are many β-redexes present, not to mention possible α-conversions). This is especially important in the context of recursion: For instance, in the factorial example above, we could have kept expanding `Y fct` infinitely many times. Clearly, that is a "bad order" for the reductions; if we had done the reductions in another order, we would have gotten the correct answer. We should only say that we have an infinite loop if all possible paths of reductions continue forever. Otherwise, the process will terminate at what's called a β-normal form: A λ-expression that contains no β-redexes (i.e., where further β-reduction is possible). How can we attempt to find such a β-normal form? Simply brute force searching is not a satisfactory answer, because simple programs shouldn't have to enumerate over some exponentially large space of paths! So how can we think of λ-calculus as specifying the kind of "unambiguous instructions" we think that computations should represent.

We need some way to actually decide which β-reductions to make for a given λ-expression. You might expect that the correct approach is: Given $(\lambda x.e_1)\ e_2$, first reduce e_2 to get some base value v_2, and then simply substitute v_2 for x in e_1. This approach is called the *applicative order* of evaluation, since we first evaluate a function's arguments and then apply the function. However, it turns out the applicative order can often result in infinite loops, even for expressions for which there exists a finite, terminating sequence of reductions! For instance, evaluating `Y fct three` as above would immediately lead to an infinite loop as `Y fct` would keep getting expanded to `fct (Y fct)` before any other evaluations would take place.

A better order is called the *normal order*, in which we always target the *outermost* possible λ-extraction (i.e., when evaluating $(\lambda x.e_1)\ e_2$, we just replace x with e_2 in e_1 and then continue reducing). This might seem more inefficient, as we might be duplicating e_2 many times inside of e_1; however, it turns out that if there exists a finite sequence of β-reductions for the original expression, the normal order is guaranteed to find it. In other words, given an initial λ-expression, reducing the expression repeatedly in the normal order is guaranteed to produce a β-normal form if possible.

Another reassuring fact is that the *Church-Rosser theorem* for λ-calculus that says that β-reductions are *confluent*: Say e is a λ-expression that, β-reduced in one way, yields e_1, while β-reduced in another way yields e_2. Confluence states that there is some λ-expression e^* to which both e_1 and e_2 can β-reduce. So while we could mess up and pick a path of β-reductions that is infinitely long, it can't be counterproductive in the sense that we can reduce to an expression that then can never be reduced to the form we want. This theorem also implies that β-normal forms are unique (because if we derived two distinct β-normal forms from the same initial λ-expression, they would be confluent to some mutual downstream expression, but by definition neither could be reduced further)!

So, given a λ-expression, we can define "evaluating" it as repeatedly executing β-reductions in the normal order. This process is guaranteed to either halt at a β-normal form, or continue forever. This should remind you of Turing machines. In fact, we have all we need to know that Turing machines and λ-calculus are in

fact equivalent![10] Then the above discussion implies that Turing machines can simulate λ-calculus: We just start with a given λ-expression and repeatedly apply the next normal-order reduction. Conversely, we know that λ-calculus can model Booleans, numbers, and pairs. By extension, it can model lists (which are just pairs that themselves contain pairs, as in the LISP programming language). And we can encode the entire *configuration* of a Turing machine (all the information that captures its current status, including its state and all the information on the tape) as a λ-expression made of some pairs and lists of numbers! Then we can define a λ-expression that checks whether a given configuration represents an accept state, and another λ-expression that, given a Turing configuration, produces the next Turing configuration. Thus, the entire Turing machine can be simulated by recursively applying this "next" function to the initial configuration until the "accept" function returns `true`.

λ-calculus, as an equivalent model of computation, makes for an interesting comparison with the Turing machine. While they are both equivalent in computation power, Turing machines are much more complicated to define upfront, while λ-calculus only has a basic syntax and a couple rules. However, Turing machines are more naturally suited to how we think of computation, while in λ-calculus we must do significant work to develop basic computational concepts like numbers and lists. And since λ-calculus and Turing machines are equivalent, many of the essential characteristics of Turing machines we've seen have analogues in the λ-calculus world. For instance, there are several major analogues to the halting problem for Turing machines: The problem of deciding whether a given λ-expression will eventually produce a β-normal form during normal order reductions is uncomputable; and the problem of deciding whether two λ-calculus expressions are equivalent is also uncomputable.

3.4 Applications

λ-calculus also has a lot of important applications in its own right that go beyond being an equivalent model to Turing machines. So far, we've been talking about *untyped λ-calculus*: We allow ourselves to apply any λ-expression to any other λ-expression. On the other hand, in *typed λ-calculus*, every λ-expression is given some type T:

- Variables have types that are assigned based on some underlying context (e.g. x might be an integer, while y might be a Boolean).
- If x has type A and e has type B, then $\lambda x.e$ has type $A \to B$.
- If e_1 has type $A \to B$ and e_2 has type A, then $e_1\ e_2$ has type B.

The key observation to make is that this type system prevents us from ever applying an expression to itself: If we have $e\ e$, then e would have to have both types $A \to B$ and A, but these are always distinct. This immediately rules out the infinite loop `inf` that we discussed above, since we apply an expression to itself. In fact, we can prove that these typing rules suffice to guarantee that all programs terminate. Typed λ-calculus is therefore not as powerful as Turing machines and untyped λ-calculus; however, it's a very useful restricted model to consider. The situation is similar to how finite and pushdown automata are useful restricted models of Turing machines. You can imagine that by relaxing these rules on types, we will allow our expressions to have more computational power, but we will pay a price in terms of being able to reason about them.

An extremely important application of type systems in λ-calculus is in *automated theorem proving*. Each λ-expression corresponds to a proof of some statement. For instance, consider the expressions

$$\lambda x.x : A \to A,$$

while

$$\lambda f.\lambda x.fx : (A \to B) \to A \to B$$

[10] To be fully formal, we'll have to specify what it means for a λ-expression to compute a function. For instance, we might say that a λ-expression e (a "program") computes a partial function $f : \mathbb{N} \to \{0, 1\}$ if, for each input n, $e\ \hat{n}$ either reduces to `true` (and $f(n) = 1$) or `false` (and $f(n) = 0$), or neither (and $f(n)$ is undefined). (Here $\hat{n}$ is the λ-calculus expression representing n.)

(the colon means "has type"). We can interpret the first expression as representing the (trivial) fact that, assuming A, we can derive A; similarly, the second expression represents the fact that, assuming $A \to B$ and A, we can prove B (the so-called *modus ponens* rule for deduction). While these examples might not seem that powerful, it turns out that by considering more interesting type systems or primitive types, we can actually express all proofs of mathematical theorems as λ-expressions. This is the foundation of the famous *Curry-Howard correspondence*, which equates *programs* (in the form of λ-expressions) with *proofs* (in the form of types). Under this model, β-reductions of λ-expressions correspond to simplifying a proof (e.g. by eliminating redundant hypotheses). Type checking a program is equated with verifying a proof of a theorem. This fundamental correspondence underlies modern automated theorem proving and proof checking software, such as the widely-used *Coq proof assistant*.

An interactive version of many of the formulae expressed in these notes is available online at `http://io.livecode.ch/learn/namin/lambda-calculus`. For more resources on λ-calculus and its relationship to type theory, see the books [Pie02] and [HS08], as well as the article [Wad15]. Also, consider taking CS 152 in the spring!

4 Preetum Nakkiran: *New Frontiers in the Theory of Machine Learning* (10/4/2019)

Biography

Preetum Nakkiran is a graduate student here at Harvard. He got his undergraduate degree at Berkeley (EECS, '16). As an undergrad, he worked with Sanjam Garg in cryptography, trying to understand the theoretical underpinnings of program obfuscation; he also worked with Anant Sahai on some distributed coding problems (à la error-correcting code idea that we discussed earlier). He was also interested in computational hardness of approximation. Outside of academia, he did an internship in Summer 2014 on a team at Google that was building the "OK Google" activation feature for Google's smart assistant; this was an interesting problem because you wanted a robust system (i.e., neural network) that was compact/efficient enough to be "online" (running all the time and continuously integrating data). Here at Harvard, he's worked with Madhu Sudan on *polar codes*, which are a type of error-correcting code that has some particularly nice information-theoretic interpretations. Now, he's really interested in the theory of machine learning, and in the wide gap between classical understandings of machine learning algorithms and their actual performance in practice.

4.1 Foundations of classical learning

In general, the goal of a learning problem is to find some underlying unknown pattern that describes the world. In the *supervised learning* paradigm, we want to learn some *classifier*, which is a function $f : \mathcal{X} \to \mathcal{Y}$ from the *input space* $\mathcal{X}$ to the *label space* $\mathcal{Y}$. For instance, $\mathcal{X}$ could be the set of 100x100 images, and $\mathcal{Y}$ could be the set of animals $\{\text{dog}, \text{cat}, \text{bear}\}$; f would be a function that tells us whether a given image represents a cat, dog, or a bear.

To learn such a classifier f, we'll get some labeled examples $\mathcal{S} = (x_1, y_1), \ldots, (x_n, y_n)$, which we call the *training data*. The goal of a *learning algorithm* $\mathcal{A}$ is to take in $\mathcal{S}$ and output a "good" classifier $\hat{f} = \mathcal{A}(\mathcal{S})$ (and determining what "good" means is very difficult!).

We'll assume the samples are drawn i.i.d.[11] from an unknown distribution $\mathcal{D}$. So we can formulate the supervised machine learning problem as follows:

[11] *Independently and identically distributed*; this just means we get a bunch of random data points that are each "uncorrelated" with the others.

Supervised learning problem.
Given: n i.i.d. samples $(x_1, y_1), \ldots, (x_n, y_n) \sim \mathcal{D}^n$.
Goal: Output a classifier $\hat{f} : \mathcal{X} \to \mathcal{Y}$ with low *generalization error*

$$\mathcal{L}_{\mathcal{D}}(\hat{f}) = \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}}[\hat{f}(x) \neq y].$$

To continue the analogy above, we might get $n = 500$ images of animals from some underlying "database" of animals $\mathcal{D}$, each labeled as a cat, dog, or bear, and we want to produce a classifier $\hat{f}$. The generalization error $\mathcal{L}_{\mathcal{D}}(\hat{f})$ represents how likely, if we were to pick a random image from $\mathcal{D}$, it would be that $\hat{f}$ would incorrectly classify the image. The ultimate goal of a supervised learning is to make this error as low as possible.

But as we've formulated it, this might be impossible. For instance, what if $\mathcal{D}$ consisted entirely of random images labeled with random animals? We wouldn't be able to come up with a classifier $\hat{f}$ that did any better than randomly guessing which animal a given image was!

To make supervised learning feasible, we'll introduce an additional assumption about the problem: We'll say there's a *ground truth* classifier $f : \mathcal{X} \to \mathcal{Y}$, and assume that the underlying distribution $\mathcal{D}$ has a special property: For each given $x \in \mathcal{X}$, we will only ever see (x, y) if $y = f(x)$. In other words,

$$\mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}}[y \neq f(x)] = 0.$$

The next definition attempts to formalize what it means for a certain type of model to be "learnable":

Definition 12 (PAC-learnability)**.** *A family of classifiers $\mathcal{F}$ is PAC-learnable iff there is some learning algorithm $\mathcal{A}$ such that for all possible ground truths $f \in \mathcal{F}$, for any desired error ϵ and failure probability δ, there is some number of samples n such that given an n-size sample $\mathcal{S} = \{(x_1, y_1), \ldots, (x_n, y_n)\} \sim \mathcal{D}^n$, the learned classifier $\hat{f} = \mathcal{A}(\mathcal{S})$ has high probability of low error:*

$$\mathop{\mathbb{P}}_{\mathcal{S}\sim\mathcal{D}^n}[\mathcal{L}_{\mathcal{D}}(\hat{f}) \leq \epsilon] \geq 1 - \delta.$$

There are many quantifiers in this definition, so let's try and break it down. First off, it's important to note that PAC-learnability is a property of a *family* $\mathcal{F}$ of classifiers. (Oftentimes $\mathcal{F}$ is called a *hypothesis class*, and its elements *hypotheses*.) In an example where $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{0, 1\}$ (i.e.,, we're classifying points in $\mathbb{R}^2$ as either 1 or 0), one reasonable hypothesis class would be the *half-plane classifiers*: Each classifier $f \in \mathcal{F}$ corresponds to a half-plane, and for any point $x \in \mathbb{R}^2$, $f(x) = 1$ iff x is in the half-plane. We could similarly define *rectangle classifiers* or *circle classifiers*.

If a family $\mathcal{F}$ is PAC-learnable, there is some algorithm $\mathcal{A}$ that can come up with a "pretty good" classifier $\hat{f}$ after seeing some number of samples n. n depends on exactly how good we want our classifier $\hat{f}$ to be — the better we want it to be, the more samples we need to feed into $\mathcal{A}$. And "pretty good" is quantified by two parameters, ϵ and δ. ϵ captures the fact that we want the generalization error of $\hat{f}$ to be low; δ allows for the reality that we might be unlucky and get a bad sample from $\mathcal{D}$, which would result in a poorly generalizing ϵ (e.g. in the animal example, there's some low probability that we might only see cats and dogs in our sample $\mathcal{S}$, so the learned classifier $\hat{f}$ would have no clue how to recognize a bear). And all of this applies for any possible ground truth classifier f.

PAC-learning stands for *probably approximately correct* learning; it was introduced by Les Valiant (at Harvard!) [Val84]. δ corresponds to the "probably" and ϵ corresponds to the "approximately". Again, you should think of PAC-learnability as making a statement about how powerful a class of classifiers is: It means that, given enough data, we can always "learn" a classifer that will be mostly accurate, most of the time.

One important caveat is that PAC-learning is a *statistical* notion: It expresses how many samples we need in order to learn a function. In practice, we also care greatly about the *computational* aspects of learning. That is, we want learning to be *efficient*: the number of samples n should not be too large to be practical, and our learning algorithm $\mathcal{A}$ should not to be too slow in terms of n.

Theorem 13. *If $\mathcal{F}$ is a finite set of classifiers, then $\mathcal{F}$ is PAC-learnable with*

$$n = \frac{\log(2|\mathcal{F}|/\delta)}{2\epsilon^2}$$

samples.

Again, this just implies that it is statistically possible to learn a classifier, not that it is computationally tractable. We'll be able to prove this theorem aftern, not that discussing a particular type of learning algorithm. First, an important definition:

Definition 14 (Empirical loss)**.** *Given a sample $\mathcal{S}$, define the empirical loss of a classifier f as*

$$\mathcal{L}_{\mathcal{S}}(f) = \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{S}} \ell(f(x_i), y_i).$$

where ℓ is the loss function

$$\ell(y_1, y_2) = \begin{cases} 1 & y_1 = y_2 \\ 0 & y_1 \neq y_2 \end{cases}$$

(for classification problems).

The empirical loss represents how well a classifier f does on the training data $\mathcal{S}$. Compare the empirical loss to the *true/population loss*

$$\hat{\mathcal{L}}_{\mathcal{D}}(f) = \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \ell(f(x), y),$$

which is the same as the generalization error discussed above. The optimal classifier is the classifier that minimizes the population loss. But the trouble is, we in general have no clue how to compute the population loss, because we know neither the ground truth function f nor the underlying distribution $\mathcal{D}$! A natural idea, instead, is to *choose the classifier that does the best on the training data, and then hope that it does well for future test data.* We call such a classifier an *empirical risk minimizer (ERM)*:

Definition 15 (Empirical risk minimizer)**.** *A classifier f is an empirical risk minimizer for a given sample $\mathcal{S}$ if it minimizes the empirical risk among all possible classifiers in $\mathcal{F}$:*

$$\hat{\mathcal{L}}_{\mathcal{S}}(f) = \min_{f' \in \mathcal{F}} \hat{\mathcal{L}}_{\mathcal{S}}(f').$$

An *empirical risk minimization (ERM) algorithm* is a learning algorithm that finds an ERM. A critical observation for later in the talk is that there might be multiple such optimal classifiers; an ERM algorithm only needs to find one such classifier.

We want all ERM algorithms to be PAC-learning algorithms (i.e., algorithms that satisfy the conditions to $\mathcal{F}$ a PAC-learnable hypothesis class). The condition that's necessary to make this true is the following:

Definition 16 (Uniform convergence)**.** *A hypothesis class $\mathcal{F}$, along with an underlying distribution $\mathcal{D}$, is (ϵ, δ)-uniformly convergent there is some n such that*

$$\mathop{\mathbb{P}}_{\mathcal{S}\sim\mathcal{D}^n}[\forall f \in \mathcal{F},\ |\mathcal{L}_{\mathcal{D}}(f) - \hat{\mathcal{L}}_{\mathcal{S}}(f)| \leq \epsilon] \geq 1 - \delta.$$

This captures intuitive notion of, "it is very likely that a classifier's performance on the training data will not diverge wildly from its performance on the testing data".

To make this more concrete, it's useful to think of a class of functions that *isn't* nontrivially uniformly convergent. Consider the hypothesis class $\mathcal{F}$ of all possible classifiers $f : \mathcal{X} \to \mathcal{Y}$. For any finite sample $\mathcal{S}$, there exist classifiers $f \in \mathcal{F}$ that have zero empirical loss (i.e., they fit every point in $\mathcal{S}$ correctly) but have any desired output on all other points. This hypothesis class is too broad — it is not uniformly convergent (since we shouldn't be able to do any better than $\epsilon = 1$, assuming that the probability of getting sample $\mathcal{S}$ is

negligible), and ERM won't work (it makes no sense to think of "choosing the best classifier from among all possible classifiers of any type"). The uniform convergence property captures some notion of the hypothesis class not being "too big to learn".

On the other hand, there are certainly some hypothesis classes that are uniformly convergent and for which ERM will succeed and that are PAC-learnable! (All of these things turn out to be equivalent under some mild regularity conditions.) For instance, any finite hypothesis class $\mathcal{F}$ is uniformly convergent (which means that an ERM algorithm for a finite hypothesis class is a PAC-learner):

Lemma 17. *If $\mathcal{F}$ is finite, then $\mathcal{F}$ is uniformly convergent: That is, for any ϵ and δ,*

$$\mathop{\mathbb{P}}_{\mathcal{S}\sim\mathcal{D}^n}[\exists f \in \mathcal{F} : |\mathcal{L}_{\mathcal{D}}(f) - \hat{\mathcal{L}}_{\mathcal{S}}(f)| > \epsilon] \leq |\mathcal{F}|e^{-\epsilon^2 n} = \delta$$

where $n = \frac{\log(2|\mathcal{F}|/\delta)}{2\epsilon^2}$.

Proof. This comes from [SB14, Chapter 4.2]. By the union bound,

$$\mathop{\mathbb{P}}_{\mathcal{S}\sim\mathcal{D}^n}[\exists f \in \mathcal{F} : |\mathcal{L}_{\mathcal{D}}(f) - \hat{\mathcal{L}}_{\mathcal{S}}(f)| > \epsilon] \leq \sum_{f\in\mathcal{F}} \mathop{\mathbb{P}}_{\mathcal{S}\sim\mathcal{D}^n}[|\mathcal{L}_{\mathcal{D}}(f) - \hat{\mathcal{L}}_{\mathcal{S}}(f)| > \epsilon].$$

Then we can use *Hoeffding's inequality*, which states that if $X_1, \ldots, X_n$ are i.i.d. random variables with mean μ, and $\mathbb{P}[a \leq X_i \leq b] = 1$,

$$\mathbb{P}\left[\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| > \epsilon\right] \leq 2\exp(-2n\epsilon^2/(b-a)^2).$$

In this case, each $X_i = \ell(f(x_i), y_i)$, so that $\hat{\mathcal{L}}_{\mathcal{S}}(f) = \frac{1}{n}\sum_{i=1}^{n} X_i$ and $\mathcal{L}_{\mathcal{D}}(f) = \mu$. Thus,

$$\mathbb{P}[|\mathcal{L}_{\mathcal{D}}(f) - \hat{\mathcal{L}}_{\mathcal{S}}(f)| > \epsilon] \leq 2\exp(-2n\epsilon^2),$$

since each $X_i \in [0, 1]$. Thus, since we picked $n = \frac{\log(2|\mathcal{F}|/\delta)}{\epsilon^2}$, we have

$$\mathop{\mathbb{P}}_{\mathcal{S}\sim\mathcal{D}^n}[\exists f \in \mathcal{F} : |\mathcal{L}_{\mathcal{D}}(f) - \hat{\mathcal{L}}_{\mathcal{S}}(f)| > \epsilon] \leq 2|\mathcal{F}|\exp(-2n\epsilon^2) = \delta,$$

as desired. □

(Don't worry if you're not familiar with the above; they're some standard techniques in probability theory that aren't necessarily for understanding the rest of the talk.)

4.2 The bias-complexity tradeoff

At this point, we want to discuss how the generalization error changes as a hypothesis class becomes more complex (for a fixed number of samples n, underlying distribution $\mathcal{D}$, and ground truth f). For instance, we might have several points in the plane, and we could consider fitting a k-th degree polynomial to them. Each time we increase k, we're considering an increasingly large set of possible models. How do we expect the generalization error (the performance of our model on test data) to change as a function of k?

Classical learning theory calls this situation a *bias-complexity* (or *bias-variance*) *tradeoff*. In situations where our hypothesis class is "too small", we won't be able to generalize well — for instance, if the positive examples in the plane are in a curved shape, but we're only considering linear models, we will likely *underfit* the data. We say that the class is too *biased*, in the sense that it makes some really strong and unrealistic assumptions about what the true classifier looks like. However, we can also have a problem with *overfitting* if our class is too "large" — there will be some classifiers that have extremely low (maybe even zero) empirical loss but don't generalize well! For instance, if we have 10 points in the plane, and we try and fit a 20-th degree polynomial, we'll almost certainly get an overcomplicated model that will generalize terribly.

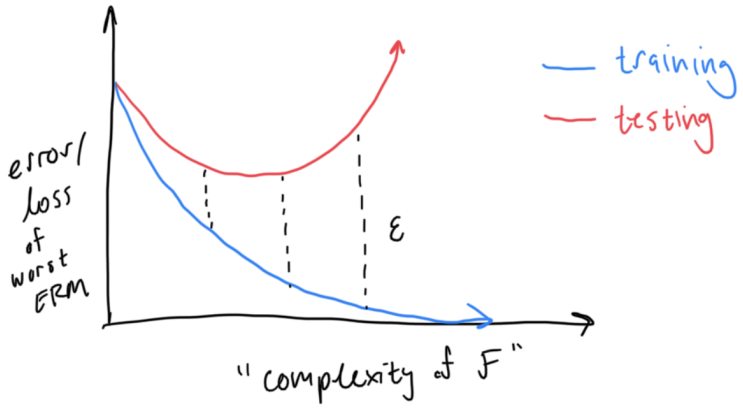

Figure 6: As the hypothesis class becomes more complex, empirical risk on a given sample goes to zero; however, generalization error should get worse, since we can't guarantee that we'll find an ERM that generalizes well. This disparity between training and testing performance for increasingly complex hypothesis classes is called the *generalization gap*.

It's important to recognize that there might be some good ERMs (ERMs that generalize well), but if our hypothesis class is too complex, we'll have no way to distinguish between the good ERMs and the (presumably more numerous) bad ERMs, so we're interested in the worst case scenario. In the polynomial example, we are aware that there will be *some* 20-th degree polynomials that generalize well, but we have no way to tell whether a 20-th degree polynomial that fits the 10 data points perfectly will generalize well or not.

The classical understanding of this tradeoff is encapsulated in Figure 6. On the x-axis, we have some suitable measure of how "complex" our hypothesis class $\mathcal{F}$ is[12], and on the y-axis, we have the worst-case generalization error of an ERM. As $\mathcal{F}$ becomes more complicated, the empirical loss on the training data will become really low; however, the population loss of the worst ERM will start increasing.

According to this theory, there is some "sweet spot" in the tradeoff that should be the goal of machine learning in practice. Correspondingly, the classical approach to a machine learning problem might be summed up as follows:

1. Choose some collection of hypothesis classes parametrized by one or more "complexity parameters" (e.g. k-th degree polynomials, where k controls the complexity).
2. Choose an ERM algorithm to find a good classifier in each hypothesis class (often using some numerical optimization algorithm).
3. Figure out which specific hypothesis class to use by balancing bias and complexity.

4.3 The breakdown of the classical approach: Deep learning

Deep learning is the burgeoning subfield of machine learning that uses *neural networks* for classifiers. At their simplest, neural networks are large networks of layered, interconnected "neurons" (or equivalently, vertices in a graph). Each neuron gets several signals as its input, combines them (usually via a linear combination), applies some nonlinearity (such as the *sigmoid function* $\sigma(z) = 1/(1 + e^{-z})$), and then outputs this value or sends it on as a signal to other neurons. More sophisticated networks might have loops or the ability to

[12]One commonly used measure is the *Rademacher complexity*, which roughly measures, given a hypothesis class $\mathcal{F}$, along with an underlying distribution $\mathcal{D}$, the empirical loss for the family on n randomly labelled training samples.

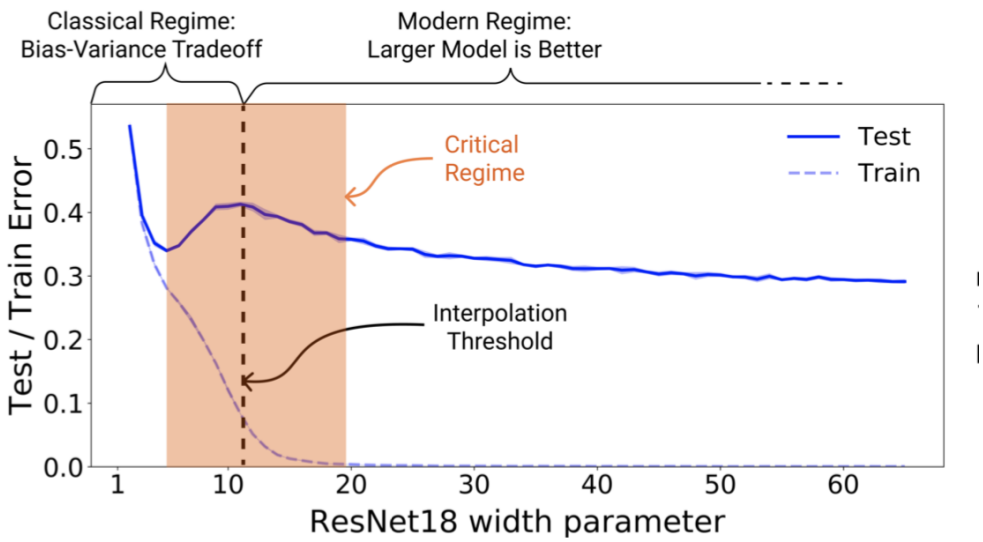

Figure 7: Training and testing error on a standard deep learning task as a function of the network's complexity. In the "classical regime", testing error decreases and then increases as a function of model complexity, representing the bias-complexity tradeoff. However, after the "interpolation threshold", around where the training error goes to zero, the test error of networks again begins decreasing as a function of complexity. Image from [Nak+19].

observe inputs over time. These models are supposed to imitate the architecture of the animal brain, and have essentially surpassed all other types of models on most "cognition" tasks such as image recognition, speech recognition, and natural language processing.

Large neural networks might contain thousands of neurons and millions of connections; since each connection has some weight that controls how much one neuron influences another, and the class $\mathcal{F}$ of all possible neural models with a given neuron architecture, parametrized by the weights on the connections, is gigantic. Neural network optimization is an extremely hard problem in its own right, because finding good settings for these parameters is difficult (the output function of the network is an extremely *non-convex* function of its parameters, while most optimization algorithms focus on convex functions). Typically, practioners set network parameters using variations of an algorithm called *stochastic gradient descent (SGD)* that, roughly, iteratively updates the parameters in the direction of steepest descent for a random data point(s). In other words, SGD finds an (approximate) ERM in the space of all neural networks $\mathcal{F}$ of a given architecture, parametrized by weights. And here's the kicker: With SGD for neural networks, we don't actually observe the classically predicted bias-variance tradeoff.

The amazing thing about SGD is that not only is it good at minimizing the empirical risk (which, since the networks are so complicated and non-convex, is extremely difficult in it of itself), the networks actually generalize pretty well. Generally, the observed behavior for learning classifiers with neural networks is that as the networks become larger, there is a finite *classical regime* in which the generalization error increases with network size, and then an *overparametrized regime* in which the generalization error keeps falling (asymptotically towards some nonzero number). The point at which this crossover happens is called the *interpolation threshold*, which seems to happen roughly when the complexity is enough so that the empirical loss can hit zero (see Figure 7). This phenomenon is called the *double descent* of the empirical loss [Bel+19]. It seems to be a relatively robust effect, occurring even if e.g. random noise is added to all of the labels.

The issue is, nobody understands why the ERM networks that SGD finds generalize well. There is nothing in classical learning theory to explain it. Perhaps this means that ERM is the wrong way to think about deep learning generalization. There is some evidence for this hypothesis in that some optimization algorithms

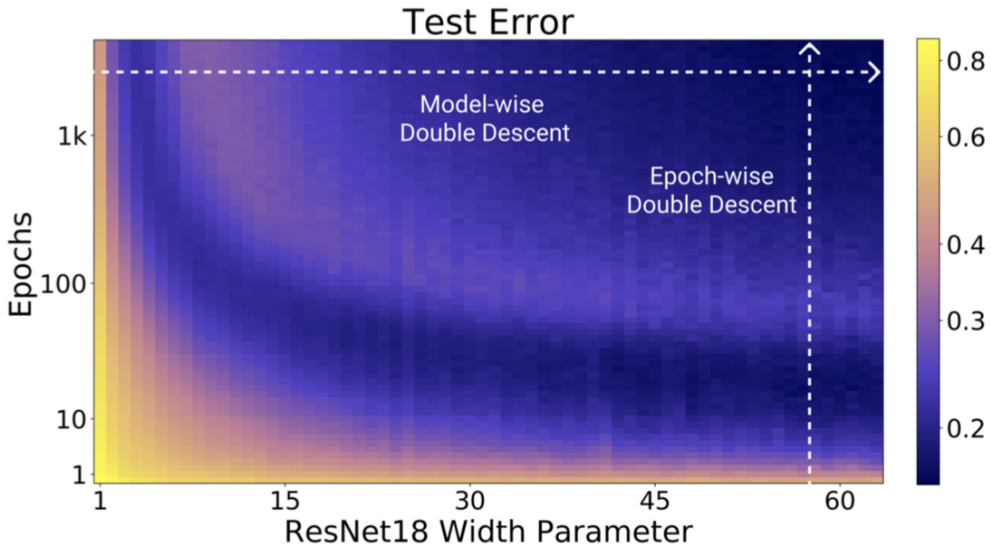

Figure 8: Double-descent in test error as a function of two variables: The model complexity of a network and the number of epochs it has been trained for. There is a "ridge" at which test error is maximized as the complexity and/or number of epochs increases, and then the error decreases on the other side of this ridge. Image from [Nak+19].

which converge faster or give better results (lower empirical loss) as compared to SGD still generalize more poorly than SGD-constructed models. Being able to understand why SGD performs well is a huge open question underlying all of modern machine learning theory.

The generalization performance of SGD is also extremely sensitive to setting various *hyperparameters* that control the SGD algorithm and network architecture themselves; typically ML practitioners will just try many different settings for these parameters. A stronger theoretical understanding of generalization in deep learning would hopefully enable a much more focused and efficient process for training neural networks. All we seem to know right now is that while classically, the particular ERM algorithm that is selected is seen as irrelevant, there are some very special properties of the particular ERM (SGD) that we use in deep learning.

Recent work suggests that we consider the *model complexity* $M(\mathcal{A}, \mathcal{D})$ of a given ERM algorithm $\mathcal{A}$ for a distribution $\mathcal{D}$, which is defined to be the largest number of samples n such that

$$\mathop{\mathbb{E}}_{\mathcal{S}\sim\mathcal{D}^n}[\hat{L}_{\mathcal{S}}(\mathcal{A}(\mathcal{S}))] \approx 0.$$

Then we say that a network is *overparametrized*, *critically parametrized*, or *underparametrized* for a given sample $\mathcal{S}$ if $|\mathcal{S}| \ll M(\mathcal{A}, \mathcal{D})$, $|\mathcal{S}| \approx M(\mathcal{A}, \mathcal{D})$, or $|\mathcal{S}| \gg M(\mathcal{A}, \mathcal{D})$, respectively [Nak+19]. Interestingly, this work also finds experimental evidence for "double descent" behavior for other hyperparameters, such as the number of *epochs* ($\approx$ iterations) that SGD runs.

Theoretical computer scientists and machine learning engineers aren't the only people studying these issues; there are a large number of disciplines that have something to add to the conversation. For instance, an increasing number of physicists are applying the mathematical and statistical techniques used in physics for modeling the behavior of extremely complex physical systems to neural networks. And there is a huge need for innovative experimentation to guide theoretical advances.

A very good reference for the material on learning theory is [SB14].

5 Chi-Ning Chou: *The MRDP Theorem* (10/11/2019)

Biography

Chi-Ning Chou is a third-year graduate student; Boaz Barak is his adviser. He was an undergraduate at National Taiwan University; he has lots of interests, but around the end of his third year, decided that he wants to do math and prove theorems. He's very interested in *algebraic complexity*. Traditionally, computer scientists have studied *Boolean complexity*: For functions $f : \{0,1\}^n \to \{0,1\}$, what is the size of the smallest circuit computing f? Can we establish lower bounds (as in Sasha Golovnev's discussion in Section 2)? It turns out that we don't know how to make progress on boolean circuit lower bounds: We don't know any explicit functions that lack small circuits (even though by a counting argument, almost all of them do). Algebraic complexity instead asks us to look at more general functions $f : \mathbb{K} \to \mathbb{K}$, where $\mathbb{K}$ is some *field* (either a finite field like $\mathbb{K}_7$ or possibly a number field like $\mathbb{C}$). *Algebraic circuits* have gates that are the usual arithmetic operators, such as addition, multiplication, and scalar multiplication. For algebraic complexity problems, we know how to prove $\Omega(n \log n)$ lower bounds (as opposed to just $\Omega(n)$ for Boolean complexity). It's also regarded as a more manageable research area because there's a lot of potential to apply pure mathematics, such as algebraic geometry, to what is otherwise a combinatorial problem.[13]

5.1 Hilbert's tenth problem

David Hilbert was one of the greatest mathematicians of the 20th century. In 1900, he famously proposed a list of 23 mathematics problems, now known as *Hilbert's problems*. These spanned diverse areas in mathematics and many of them were pursued vigorously in the coming decades, leading to huge breakthroughs in various areas.[14]

We'll discuss *Hilbert's tenth problem*, which is the main Hilbert problem that has to do with computer science (although CS wasn't actually a field yet). This problem concerns *Diophantine equations*, which are polynomial equations with integer coefficients where we consider integer solutions. Hilbert's original goal was as follows [Hil02]:

> "Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: *To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.*"

For instance, we know that there are solutions to the equation

$$x^2 + y^2 = z^2$$

over the integers (the so-called *Pythagorean triples* (x, y, z), such as $(3, 4, 5)$ or $(5, 12, 13)$). But is there an algorithm that can determine whether a given Diophantine equation is solvable in general? It turns out that the answer is a resounding *no*, and this result is known as the *MRDP theorem*, which was the product of decades of work by several mathematicians. While there are many technical details involved, we will at least give an streamlined sketch of the progression of the proof.

5.2 Preliminaries

Hilbert posed this problem decades before the publication of notions such as Turing machines and computability. In a more modern setting, we'd define the function Dio that takes as input the description

[13] These types of analyses underlie *Geometric Complexity Theory (GCT)*, which conjecturally links open problems in theoretical computer science, including the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, to open problems in algebraic geometry. GCT is often considered the only currently known avenue of research that could possibly achieve a proof that $\mathbf{P} \neq \mathbf{NP}$, although it's a long way off.

[14] Among the other problems on Hilbert's list include the resolving the Riemann hypothesis (the eighth problem, unresolved); proving the continuum hypothesis (the first problem, proven impossible since it is independent of ZFC); proving the axiom of arithmetic are consistent (the second problem, generally considered to be proven impossible by Gödel; and providing an axiomatic foundation for probability theory (completed by Kolgomorov), as well as many more quantitative problems.

of Diophantine equation and outputs 1 iff that equation has a solution; then Hilbert's tenth problem asks whether DIO is computable.

To resolve this problem, we'll actually rephrase it into a slightly different language.

Definition 18 (Diophantine set)**.** *Given $n \in \mathbb{N}$, a set $S \subset \mathbb{N}^n$ is Diophantine iff it is the set of zeros of some polynomial $p \in \mathbb{Z}[x_1, \ldots, x_n]$;*[15] *that is,*

$$(a_1, \ldots, a_n) \in S \iff p(a_1, \ldots, a_n) = 0.$$

For instance, $S = \{1\}$ is Diophantine, since it is the set of zeros of the polynomial $p(x_1) = x_1 - 1$. Similarly, $S = \{5, 7\}$ is Diophantine, via the polynomial $p(x_1) = (x_1 - 5)(x_1 - 7)$. But Diophantine sets can be much more powerful than this. For instance, the set of zeros of the polynomial $p(a, d, r, q) = a - r - dq$ is the set of all tuples (a, b, c, k) where a, when divided by d, has quotient q and remainder r.

Thus we can equivalently think of DIO as representing whether the Diophantine set corresponding to a polynomial p is nonempty. It turns out that it actually makes more sense to think about these sets than to think about the original computational problem, as we'll see later. Now, note that with our current definition of a Diophantine set, it is true that e.g. a Diophantine set corresponding to only one variable is always finite (unless the polynomial is the zero polynomial). To enable us to conveniently work with a broader class of sets, we make the following slight change to the definition:

Definition 19 (Diophantine set (in a subset of variables))**.** *Given $n \in \mathbb{N}$, a set $S \subset \mathbb{N}^n$ is Diophantine iff there is some polynomial $p \in \mathbb{Z}[x_1, \ldots, x_n, y_1, \ldots, y_m\}$ such that,*

$$(a_1, \ldots, a_n) \in S \iff \exists b_1, \ldots, b_m : p(a_1, \ldots, a_n, b_1, \ldots, b_m) = 0.$$

Note that from a computational standpoint, telling whether such a set corresponding to a subset of variables of a polynomial p is nonempty is at most as hard as telling whether the entire set is empty. And now we can have infinite Diophantine sets in one variable. (Mathematically, we might think of these new sets as *projections* of our original sets onto subsets of coordinates.)

Diophantine sets have several important closure properties. If S_1 and S_2 are both Diophantine, we claim that $S_1 \cup S_2$ is Diophantine: If S_1 is defined as

$$(a_1, \ldots, a_n) \in S_1 \iff \exists b_1, \ldots, b_m : p(a_1, \ldots, a_n, b_1, \ldots, b_m) = 0,$$

and similarly S_2 is defined as

$$(a_1, \ldots, a_n) \in S_2 \iff \exists b_1, \ldots, b_m : q(a_1, \ldots, a_n, b_1, \ldots, b_m) = 0,$$

then $S_1 \cup S_2$ is the Diophantine set defined as

$$(a_1, \ldots, a_n) \in S_1 \cup S_2 \iff \exists b_1, \ldots, b_m : p(a_1, \ldots, a_n, b_1, \ldots, b_m) q(a_1, \ldots, a_n, b_1, \ldots, b_m) = 0.$$

Similarly, we could define $S_1 \cap S_2$ via the equation

$$(a_1, \ldots, a_n) \in S_1 \cap S_2 \iff \exists b_1, \ldots, b_m : (p(a_1, \ldots, a_n, b_1, \ldots, b_m))^2 + (q(a_1, \ldots, a_n, b_1, \ldots, b_m))^2 = 0,$$

since the only way this can be zero is if both are zero.

5.3 Outline of proof

The MRDP theorem was ultimately proved via a sequence of reductions that considers successively more restricted classes of statements, and finally ends up at Diophantine equations (DIO). We will define each type of statement, but the overall sequence is as follows:[16]

$$\text{HALT} \leq \text{QIS} \leq \text{DAVISQIS} \leq \text{EXPDIO} \leq \text{DIO}.$$

Let's discuss the actual types of statements, and the corresponding "set" forms.

[15] This notation simply means a polynomial with integer coefficients in the variables $x_1, \ldots, x_n$.

[16] Here, we use the notation where $X \leq Y$ means that X reduces to Y (i.e., the problem X is at most as hard as the problem Y). For instance, for any computable function f, $f \leq \text{HALT}$, while $\text{HALT} \not\leq f$.

- *Quantified integer statements* are general statements that consist of an arbitrary sequence of universal or existential quantifiers ($\forall$ or $\exists$) that quantify over some variables for a polynomial. An example of a QIS-*set* S might be, say,

$$(x_1, x_2) \in S \iff \exists x_3 \forall x_4 \exists x_5 : x_1^2 + x_1 x_2 - x_3^3 - x_4^2 - x_5 + 3 = 0.$$

  The problem QIS is, given a quantified integer statement, to determine whether the corresponding QIS-set is nonempty.

- There is a particular modified and restricted form of quantified integer statements called *Davis normal form*; these are statements of the form

$$\exists y, u \in \mathbb{N}\ \forall k \leq y\ \exists z_1, \ldots, z_m \leq u : p(x, y, k, z_1, \ldots, z_m) = 0$$

  where p is some polynomial. We could also consider *Davis-sets* and the problem DAVISQIS of determining whether a statement in Davis normal form yields a nonempty Davis-set.

- *Exponential Diophantine equations* are just like Diophantine equations except that they may also involve taking constants or variables to the power of variables. An example of an *exponential-Diophantine set* S would be

$$(x_1, x_2, x_3) \in S \iff \exists x_4, x_5 : x_1^2 + 2^{x_2} - x_3^{x_4} + x_5 = 0.$$

  Again, we have the problem EXPDIO of determining whether a given exponential Diophantine equation has solutions.

In class, we saw the reduction from HALT to QIS, which is a consequence of Gödel's incompleteness theorem. The second reduction is due to Davis [Dav53]; the third to Davis-Putnam-Robinson [DPR61]; and the fourth to Robinson-Matiyasevich [Rob52; Mat70a].

The sequence of reductions actually shows something *stronger* than just the problems being equivalent. First off, we'll make the following definition:

Definition 20. *A set $S \subset \mathbb{N}^k$ is computably enumerable iff there is some algorithm A such that*

$$A(n_1, \ldots, n_k) = \begin{cases} 1 & (n_1, \ldots, n_k) \in S \\ \bot & (n_1, \ldots, n_k) \notin S \end{cases}$$

where $A(n_1, \ldots, n_k) = \bot$ means that A does not halt on input $n_1, \ldots, n_k$.

In other words, a set S of tuples of numbers is computably enumerable iff there is an algorithm A computes the partial function that defined only on $S \subset \mathbb{N}^k$ and always equal to 1. Gödel's incompleteness theorem implies that every computably enumerable set is a QIS-set (i.e., it is the set of solutions to some QIS). The series of theorems that we'll discuss imply that if a given set S is a QIS-set, it is also a Davis-set; if S is a Davis-set, it is also an exponential Diophantine set; and if S is an exponential Diophantine set, it is also Diophantine.

Note that this is stronger than just the reductions in the sense that the above facts could conceivably be false while the computational reduction of determining emptiness of such sets might still be possible. We can write the "strengthened" version of the MRDP theorem as follows:

Theorem 21 (MRDP theorem)**.** *All computably enumerable sets are Diophantine.*

The reason we like to think about the sets this way is that, after Gödel's result, each of these the reductions becomes a *mathematical* statement about solution sets of equations that have (seemingly) nothing to do with Turing machines or computation.

One final important way to think about these sets is as *relations*. The fact that e.g.

$$S_{remainder} = \{(a, d, r) : \exists q\ a = r + dq\}$$

is a Diophantine set, along with the closure of Diophantine sets under intersections, means that we have "syntactic sugar" for the remainder operation; we can add it as a *predicate* (constraint on variables) in a Diophantine equation and this equation is still "Diophantine", in that its solutions form a Diophantine set (we can "unsugar" the predicate to get a single Diophantine equation). For instance, we can add conditions such as "x_1 divided by x_2 has remainder x_3" among the variables we're quantifying over when defining the set and the resulting set is still Diophantine. (This interpretation of sets as relations that we can apply among variables also makes sense in the setting of e.g. exponential Diophantine sets, since these are also closed under intersections.)

We will not go into significant detail about the first two reductions (Qis to DavisQis and DavisQis to ExpDio); the first reduction is due to some nice tricks to eliminate quantifiers that come from number theory, and the second uses further number-theoretic tricks (in particular the Chinese remainder theorem) to get eliminate the bounded universal quantifier $\forall k \leq y$ of Davis normal form and produce a statement containing only existential quantifiers.

5.4 Exponential Diophantine sets and the final step

Define the *exponential relation*

$$S_{exp} = \{(a, b, c) : a = b^c\}.$$

An equivalent way to think about exponential Diophantine sets is as Diophantine sets with the addition of S_{exp} (i.e., we can add conditions such as $x_1 = x_2^{x_3}$ among the variables we're quantifying over in the definition of the set). Thus, to show that every exponential Diophantine set is Diophantine, it suffices to show that the exponential relation is a Diophantine set; this is essentially what Matiyasevich did.

Before outlining the proof, it's good to have some general intuition about why exponential Diophantine sets are powerful. It should not be surprising that we can represent, say, the *binomial relation* $S_{binom} = \{(a, b, c) : a = \binom{b}{c}\}$, as an exponential Diophantine set, via some clever algebra with the binomial formula

$$(1 + x)^n = \sum_{i=0}^{n} \binom{n}{i} x^i.$$

And once we have binomials, we can show that the *factorial relation* $S_! = \{(a, b) : a = b!\}$ is also an exponential Diophantine set by manipulating the definition of binomials in terms of factorials. (Both of these derivations require some careful arguments about bounding differences between different expresssions, etc., and are not very profound.) Finally, we can even represent the *primality relation* $S_{prime} = \{a : a \text{ is a prime}\}$, since we should be able to represent coprimality, and then n is prime iff it is coprime with $(n - 1)!$. (See [Rob52] for the details of these derivations.)

We need one more definition:

Definition 22 (Roughly exponential growth relation)**.** *$S \subset \mathbb{N}^2$ is a roughly exponential growth relation iff*

1. *For all $(a, b) \in S$, $b < a^a$.*
2. *For all $n \in \mathbb{N}$, $\exists (a, b) \in S$, $b \geq a^n$.*

Intuitively, condition (1) implies that S cannot grow "much faster" than the exponential function — in particular, it cannot outpace the superexponential function a^a. Condition (2) implies that S cannot grow "much slower" than the exponential function — it cannot grow more slowly than polynomials. Examples of such relations would include the relations $(a, f(a))$ where the function f is $a^{\log a}$, a^a, 2^a, or $a^{\log_* a}$; at the same time, the relation corresponding to $f(a) = 2^{2^a}$ grows too quickly, while the relation corresponding to $f(a) = a^8$ grows too slowly.

The first major step of the reduction from ExpDio to Dio was due to Julia Robinson [Rob52]: She said that *any* roughly exponential growth relation R is Diophantine, then the exponential relation S_{exp} is

Diophantine (so that all exponential Diophantine sets are Diophantine). Robinson's proof relied on clever usage of the *Pell equation*

$$x^2 - (a^2 - 1)y^2 = 1,$$

a well-studied Diophantine equation whose solutions grow very quickly. This represents a great strategy for tackling a hard math problem in general: We have *relaxed* the class of objects we're looking at, so that we now have more candidate objects to try and prove something about. After Robinson's result, Davis remarked,

> "I think that Julia Robinson's hypothesis is true, and it will be proved by a clever young Russian."

The final step in the proof of the MRDP theorem was provided by such a mathematician, the twenty-two year old Yuri Matiyasevich, who considered the Fibonacci sequence defined by $F(0) = 0$, $F(1) = 1$, and $F(n+2) = F(n) + F(n+1)$ [Mat70a]. The heart of his proof is to prove that the relation

$$S_{Fib} = \{(a, b) : b = F(2a)\}$$

is Diophantine: He first proves the identity

$$F(n+1) - F(n+1) \cdot F(n) - F(n) = (-1)^n$$

via induction, and then shows that S_{Fib} is actually *equal* to the Diophantine set defined by $(y^2 - xy - x^2)^2 = 1$ (for the naturals). Then he concludes that since S_{Fib} is a roughly exponential growth relation (this can be shown using the explicit formula for the Fibonacci sequence), all exponential Diophantine sets are Diophantine.

First off, note that there are some pretty fascinating additional consequences of this result. For instance, since S_{prime} is an exponential Diophantine set, it must also be a Diophantine set — that is, there are multivariate *prime-producing polynomials* that take on exactly the prime numbers (when their inputs are positive).

Research also focuses on proving similar results for "Diophantine" sets with coefficients in and quantifiers over other rings, such as $\mathbb{Z}[\sqrt{2}]$ which consists of all expressions of the form $a + b\sqrt{2}$ for $a, b \in \mathbb{Z}$. Whether an analogue of the MRDP theorem for rationals can be proven remains open.

For more background and further reading, see Matiyasevich's reflections on the proof [Mat70b], as well as the surveys [Pas07] and [Cab04].

6 Alec Sun: *Average-Case Complexity* (10/18/2019)

Biography

Alec Sun is a junior in Winthrop House. He's mostly interested in number theory, combinatorics, and theoretical computer science. He participated in the Research Experience for Undergraduates (REU) program at the University of Minnesota–Duluth in combinatorics last summer. His favorite types of problems in theoretical CS involve the *probabilistic method*, which is a nonconstructive way to show that an object exists with property P without actually constructing such an object — by for example showing that the probability of a *random* object having probability P is strictly greater than zero.

6.1 Introduction

When defining complexity classes such as **P**, we have generally considered the *worst-case scenario* of an algorithm's runtime. In particular, we'd define the runtime $T(n)$ of an algorithm on inputs of length n as the maximum of the algorithm's runtime on all input strings of length n. However, in the real world, we might be interested not in how long the algorithm takes in the absolute worst case, but in how long the algorithm takes on "average" inputs for some suitable definition of average.

For instance, it's possible to show that the worst-case runtime of any comparison-based sorting algorithm is $\Omega(n \log n)$. However, maybe our algorithm will mostly only see lists that are "moderately unsorted" — for instance, we might have a high probability that element is at most k indices away from its correct position for some fixed k. Then we might expect that even simple algorithms such as *insertion sort* (where we iterate through the list an element at a time and add them to a growing correctly sorted sublist at the beginning) should perform in $\Theta(n)$ time on average (since no element must be swapped backwards more than k positions, the runtime is $\Theta(kn)$). Overall, the moral of the story is that *under certain distributions of instances, the problem can take less time on average than it does in the worst case.*

A more interesting example is the problem 3-COLORING, which asks: Given a graph $G = (V, E)$, can I color the vertices either red, green, or blue such that each edge has endpoints of different colors. Right now, nobody knows a way to check whether a graph is 3-colorable in polynomial time (in fact, as we'll see, 3-COLORING is an **NP**-complete problem, which implies that we believe it has no polynomial time algorithm). But could we do better with the *average performance* of some algorithm on a *random graph*?

Definition 23 (Erdös-Renyi graph)**.** *The Erdös-Renyi graph $G(n, p)$ is the graph on n vertices where each edge is included independently with probability p.*

It turns out that there exists an algorithm for 3-COLORING that runs in expected linear time $O(n)$ if its input is an Erdös-Renyi graph $G(n, \frac{1}{2})$. Note that there are $2^{\binom{n}{2}}$ equally likely graphs (since there are $\binom{n}{2}$ possible edges, and each is included with probability $\frac{1}{2}$ independently of the others).

The basic idea behind this algorithm is a pair of clever observations:

1. K_4 (the *complete graph on 4 vertices*, which consists of 4 vertices with all 6 possible interconnecting edges, is not 3-colorable; any graph which contains K_4 as a subgraph is also not 3-colorable.

2. The vast majority of all of the $2^{\binom{n}{2}}$ graphs on n vertices contain K_4 as a subgraph.

So it turns out that with some careful combinatorial analysis, we can show that the following algorithm runs in expected linear time:

Algorithm 1 3-COLORING algorithm

Input: A graph $G = (V, E)$

1: Look for K_4 in G: Iterate through all subsets of 4 vertices in V, and check whether E contains all 6 edges between them. If K_4 is in G, return true.

2: Check all possible 3-colorings of G by brute force. If one of them works, return true; otherwise, return false.

This follows since the algorithm requires exponential time on only a tiny fraction (i.e., subexponentially small number) of the inputs (those not containing K_4).

6.2 The permanent problem

Consider the following general definition of the determinant of an $n \times n$ matrix:

Definition 24 (Determinant)**.** *Given a matrix A, we define its determinant*

$$\det(A) = \sum_{\pi \in S_n} (-1)^{\text{sgn}(\pi)} \prod_{i=1}^{n} A_{i,\pi(i)}.$$

Here S_n is the set of all possible permutations of $\{1, \ldots, n\}$, and $\text{sgn}(\pi)$ is the *sign* of a given permutation π; the sign is $+1$ or -1 depending on whether the total number of pairs of elements swapped by the permutation is even or odd, respectively. While this definition might not be familiar to you, you should be

able to verify that it matches the formulas you've seen for the determinant of a 2×2 or 3×3 matrix (e.g. in the 3×3 case, note that the sum is over $3! = 6$ terms, 3 of which are positive and 3 of which are negative).

Computing the determinant of a matrix is really important for a lot of linear algebra applications. But if we tried to apply this formula naïvely to compute the determinant of a matrix, it wouldn't be very efficient — since we'd have to iterate over all possible $n!$ permutations. Is there a more efficient algorithm?

It turns out that via the classic technique of *Gaussian elimination*, we can do this in $O(n^3)$ time. We'll attempt to describe this roughly; the details are not important for us and can be found in any standard linear algebra textbook. Gaussian elimination involves manipulating the rows of a matrix until all entries below the diagonal are zero (this is called *upper triangular form*). The three manipulations that are allowed are:

1. Switch row R_1 with row R_2
2. Add row R_1 to row R_2
3. Multiply row R by a nonzero constant c

And you can verify from the definition that these operations have the following effect on the determinant of a given matrix:

1. Switching two rows negates the determinant
2. Adding one row to another doesn't change the determinant
3. Multiplying a row by c multiplies the determinant by c

Gaussian eliminations uses these three basic operations to systematically zero out entries below the diagonal one by one: Each entry is zeroed out by adding an appropriate multiple of a row further up in the matrix, and switching rows might be necessary if e.g. one of the higher rows has too many zeros. This whole process requires $O(n^3)$ time. Thus, we can convert any original matrix into an upper triangular matrix, and knowing the determinant of that upper triangular matrix, we can know the determinant of the original matrix. And, furthermore, the determinant of an upper triangular matrix is the product of its diagonal entries; this term corresponds to the identity permutation (any other permutation will necessarily include some zero entry below the diagonal).

Now we can also define the following quantity which is closely related to the determinant:

Definition 25 (Permanent)**.** *Given a matrix A, we define its permanent*

$$\text{per}(A) = \sum_{\pi \in S_n} \prod_{i=1}^{n} A_{i,\pi(i)}.$$

The *only* difference between the determinant and permanent of A is that we omit the $(-1)^{\text{sgn}(\pi)}$ term. Gaussian elimination no longer helps us with our computation; in fact, this small difference seems to make the permanent much harder to compute than the determinant.

Both the determinant and permanent are *counting problems*: Their output is not a yes/no value but rather a number. Typically, we will work over a finite field $\mathbb{F}_m$ (in essence, the matrix entries will be in $\{0, \ldots, m-1\}$, and we want the permanent modulo m, where m is chosen so that there are multiplicative inverses for all nonzero numbers $1, \ldots, m-1$); otherwise, our answers could be exponentially large in the input size (e.g. for a matrix of all 1's, the permanent is $n!$). Note, however, that we can make both problems into decision problems by asking questions of the form "Is the determinant/permanent of this matrix equal to k?"[17] In what follows, we will refer to the underlying field as $\mathbb{F}$.

[17] *Advanced note* (that should make sense after we see **NP** and **NP**-completeness): The determinant and permanent problems lie in the complexity class #**P** (pronounced "sharp P"), which is roughly defined as follows: A counting problem $f : \{0,1\}^* \to \mathbb{N}$ is in #**P** iff there is a polynomial-time algorithm $A(x, w)$, where x is a problem instance and w is a "witness" string, such that $f(x)$ is the number of unique values w such that $A(x, w) = 1$. In other words, $f(x)$ counts the number of witnesses accepted by A for x. Furthermore, the permanent problem (even over the binary field $\mathbb{F} = \{0, 1\}$) is #**P**-complete, in the sense that all other #**P** problems reduce to it. In some sense, every problem in **NP** reduces to a problem in #**P**, because **NP** problems ask whether there is "at least one" witness, while #**P** problems ask exactly how many witnesses there are.

What makes the permanent interesting in terms of average case complexity? It turns out that *computing the permanent in the "average case" is just as hard as computing it in the worst case*; this is called the *random self-reducibility* of the permanent. The reason this reducibility is called random is that it applies to *randomized algorithms* for computing the permanent — algorithms $A(x; r)$ that get an additional random input r and must only be correct on input x with probability $\frac{2}{3}$ over the choice of r. (We'll hear much more about randomized algorithms in class in the coming weeks.) In what follows, we will take two types of probabilities:

1. Probabilities over the (uniformly chosen) internal randomness of the algorithm on input X; e.g.

$$\mathbb{P}[A(X) = \text{per}(X)] \geq \frac{2}{3}$$

   means "$A(X)$ is correct with probability at least $\frac{2}{3}$".

2. Probabilities over all (uniformly chosen) inputs to the algorithm; e.g.

$$\mathop{\mathbb{P}}_{X \sim \mathbb{F}^{n \times n}} \left[\mathbb{P}[A(X) = \text{per}(X)] \geq \frac{2}{3} \right] \geq \frac{2}{3}$$

   means "A is correct with probability $\frac{2}{3}$ for at least $\frac{2}{3}$ of the possible $n \times n$ matrices X").

Theorem 26. *Suppose there is a polynomial-time randomized algorithm A such that*

$$\mathop{\mathbb{P}}_{X \sim \mathbb{F}^{n \times n}} \left[\mathbb{P}[A(X) = \text{per}(X)] \geq \frac{2}{3} \right] \geq \frac{1}{3n+3}.$$

Then there exists a randomized algorithm B such that for all $n \times n$ matrices X,

$$\mathbb{P}[B(X) = \text{per}(X)] \geq \frac{2}{3},$$

and B runs in expected polynomial time.

In other words, if we have an randomized algorithm A that correctly computes the permanent with $\frac{2}{3}$ probability for at least $\frac{1}{3n+3}$ of all $n \times n$ matrices X, then we have a randomized algorithm B that correctly computes the permanent for *all* $n \times n$ matrices X.

First, observe that $\text{per}(X)$ is a degree-n polynomial over the n^2 variables $x_1 1, x_1 2, \ldots, x_{nn}$ that are entries of the matrix X. Thus, it actually suffices to prove this broader claim:

Claim. *Let p be a degree-d polynomial in n variables $x_1, \ldots, x_n$ over the finite field $\mathbb{F}$. Suppose there is a polynomial-time randomized algorithm A such that*

$$\mathop{\mathbb{P}}_{x \in \mathbb{F}^n} \left[\mathbb{F}[A(x) = p(x)] \geq \frac{2}{3} \right] \geq \frac{1}{3d+3}.$$

Then there exists a randomized algorithm B such that for all $x \in \mathbb{F}^n$,

$$\mathbb{P}[B(x) = p(x)] \geq \frac{2}{3},$$

and B runs in expected polynomial time.

Proof. Suppose that the algorithm A exists. Consider Algorithm 2 as B.

First off, we recall that in a field, addition and multiply by nonzero constants are bijections. Then for fixed x, the distribution of $x + ty$ for random y is uniformly random over $\mathbb{F}^n$, since ty is itself uniformly random over $\mathbb{F}^n$. (Recall that a discrete distribution being uniformly random means that all outcomes

Algorithm 2 Polynomial computation algorithm B

Input: An input $x = (x_1, \ldots, x_n) \in \mathbb{F}^n$

Choose $y = (y_1, \ldots, y_n) \sim \mathbb{F}^n$ uniformly at random.
Let T be any subset of $\mathbb{F}_n \setminus \{0\}$ of size $d+1$ (e.g. could choose $T = \{1, 2, \ldots, d+1\}$).
for all $t \in T$ **do**
    Set $a_t := A(x + ty)$.
end for
Let $q(t)$ be the unique polynomial of degree d such that for all t, $q(t) = a_t$.
return $q(0)$

are equally likely — thus knowing that multiplication by t and addition of x are bijections means that all outcomes of $x + ty$ are equally likely.)

Then for each t,

$$\mathbb{P}_{y \sim \mathbb{F}^n}[A(x+ty) \neq p(x+ty)] < \frac{1}{3d+3}$$

by assumption. Then by the union bound,

$$\mathbb{P}_{y \sim \mathbb{F}^n}[\forall t, A(x+ty) \neq p(x+ty)] < \frac{1}{3}$$

since there are $d+1$ values of 1. Thus, for any $x \in \mathbb{F}_n$,

$$\mathbb{P}_{y \sim \mathbb{F}_n}[\forall t, a_t = p(x+ty)]$$

is at least $\frac{2}{3}$ since we set each a_t to $A(x+ty)$.

Now since p is degree-d polynomial, and q is a degree-d polynomial, and they agree at the $d+1$ points (a_t) with $\geq \frac{2}{3}$ probability, they must in fact be the same polynomial with $\geq \frac{2}{3}$ probability; thus, by evaluating $q(0)$, we get the true value of $p(0)$ with $\geq \frac{2}{3}$ probability, as desired. □

7 Ben Edelman: *The Sensitivity Theorem* (10/25/2019)

Biography

Ben Edelman is a second year graduate student at Harvard studying theoretical computer science; his adviser is Les Valiant. He studied at Princeton as an undergraduate. He is interested in lots of areas of theoretical computer science; recently, he's focused on trying to understand the theoretical properties of deep learning algorithms (see Preetum Nakkiran's talk in Section 4). He's also perennially interested in questions in computational complexity and game theory.

7.1 Boolean function complexity

As we've discussed before, one important area of research in theoretical computer science is *Boolean function complexity*. The goal of this field is to quantify the "hardness" of functions $f : \{0,1\}^n \to \{0,1\}$. The main measure of complexity that we've seen so far has been *circuit complexity*: What is the size of the smallest NAND circuit that computes f? We care about understanding circuit complexity, and in particular proving circuit complexity lower bounds, since they could someday lead to a proof that $\mathbf{P} \neq \mathbf{NP}$.

However, as we saw in Sasha Golovnev's talk in Section 2, lower bounds in circuit complexity have turned out to be very difficult to prove; in particular, while by counting arguments, we know that almost all functions $f : \{0,1\}^n \to \{0,1\}$ require circuits of size $O(2^n/n)$, we haven't come up with explicit bounds for any $f : \{0,1\}^* \to \{0,1\}$ on inputs of length n better than $O(cn)$ for some small, fixed c!

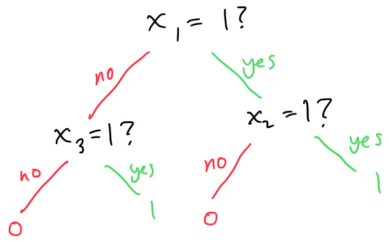

Figure 9: A decision tree for the function $IF(x_1, x_2, x_3)$.

One natural way we might start to better understand the "hardness" and structure of Boolean functions is to define other measures of function complexity.

A *decision tree* is an adaptive strategy for computing a Boolean function where one can start by checking a single bit x_{i_1} of the input x; based on the value of x_{i_1}, one then chooses a second bit x_{i_2} to check, and continues choosing a new bit to check based on the previous bits until at some point one can deduce the value of f.

The *decision tree complexity* or *query complexity* of a Boolean function f is depth of the shallowest decision tree that computes f.[18] For instance, Figure 9 depicts a decision tree for the IF function of depth 2; this means that the query complexity for IF is at most 2. In fact, it is equal to 2.[19] The query complexity of $f(x) : \{0,1\}^{10} \to \{0,1\} : x \mapsto x_5 \vee x_8$ is 2, since we know we must look at both x_5 and x_8 in the worst case. The query complexity of a constant function is 0 since no queries must be made to determine its value; on the other hand, the query complexity of the n-bit Xor function is n, since no matter what strategy we pick, we will have to know all bits. In general, the query complexity of any f is between 0 and n, inclusive.

This property of adaptivity (i.e., that we are allowed to query different bits depending on earlier query results) is very important. For instance, consider the function $f(x, i) = x_i$ where x is 2^k bits and i is k bits. The decision tree complexity of this function is $k+1$: An optimal decision tree for f will inspect the k bits of i and then adaptively select the correct bit of x. On the other hand, any non-adaptive strategy must make $2^k + k$ queries in the worst case.

The best way to think of query complexity is as a restricted model of circuit computation: We can directly convert any decision tree of depth k to a circuit with $c2^k$ gates computing the same function. However, the query complexity of a function does not necessarily correspond to the computational hardness/circuit complexity of the function in general; for instance, Xor is a "hardest" query-complexity problem, but the corresponding $O(2^n)$-circuit is not the best way to compute Xor — it can be of course computed in $O(n)$ gates.

A related notion is called the *certificate complexity* of f. On each input x, a *certificate* is a subset $S \subset [n]$ such that by looking at x_i for each $i \in S$, the value of f is completely determined. (Formally, if y is a string agreeing with x on the indices in S, then $f(y) = f(x)$.) The certificate complexity of f is the maximum over inputs $x \in \{0,1\}^n$ of the smallest certificate for x. Note that the certificate complexity is always at most the decision tree complexity, since corresponding subsets can be created for each x by tracing down the decision tree.

However, there are functions that have lower certificate complexity than query complexity. Consider the function $f : \{0,1\}^n \to \{0,1\}$ that takes in a $\sqrt{n} \times \sqrt{n}$ matrix X of 0's and 1's and outputs 1 iff there is a column of all 1's in X. The certificate complexity is $\sqrt{n}$ (if X has a column of all 1's, then a certificate is the indices of the entries in the column; if X does not have a column of all 1's, then a certificate is indices of 0 entries in each column). However, the query complexity is something like $n - \sqrt{n}$, since to conclude that

[18] Decision tree complexity is important in applications. For instance, in *differential diagnosis* procedures, the goal might be to find the smallest decision tree of symptoms to ask about before determining the patient's disease. Finding shallow decision trees for binary classification problems is also an important problem in machine learning.

[19] This and the other bounds mentioned in this paragraph should not be difficult to prove rigorously, but we just present them intuitively.

$f(X) = 0$, we will have to see at least $n - \sqrt{n}$ entries in the worst case.

In all of the examples we've seen so far, while the query complexity and certificate complexity might not have been the same, they have been *polynomially close*, in the sense that there exist polynomials p and q such that if $C_{query}(f)$ is the query complexity of f and $C_{cert}(f)$ is the certificate complexity of f, then $C_{query}(f) \leq p(C_{cert}(f))$ and $C_{cert}(f) \leq q(C_{query}(f))$. Crucially, these polynomials are independent of the particular choice of f. It turns out we can construct polynomials p and q such that the bounds *always* hold, for all choices of functions f.

We can define more types of query-related complexity. For instance, *randomized query complexity*, which are decision trees that also have random "coin flip" inputs and only need to be correct with probability $\frac{2}{3}$. There is a similar notion of *quantum query complexity* (we will discuss quantum algorithms and circuits at the end of the course). (By comparison, the plain decision tree complexity discussed above is often called *deterministic query complexity*, and the certificate complexity is called *nondeterministic query complexity*.) These still turn out to be polynomially close.[20]

There are also non-query-based measures of complexity. One important example is called the *polynomial degree* $d(f)$ of f; this is the minimum degree of a polynomial in $x_1, \ldots, x_n$ with coefficients in $\mathbb{R}$ that is equal to f.[21] Note that since $x_i^n = x_i$ for any $n > 1$, it is equivalent to consider only *multilinear* polynomials in $x_1, \ldots, x_n$ (i.e., where no variable is raised to a power above 1). These polynomials take the form

$$p(x_1, \ldots, x_n) = \sum_{S \in [n]} c_S \prod_{i \in S} x_i$$

with the coefficients $c_S \in \mathbb{R}$; the degree of p is the cardinality of the largest S such that c_S is nonzero.

Observe that at least one polynomial always exists that agrees with f. There are two good ways to see this:

1. Consider any $y \in \{0,1\}^n$; define $q(x_i) = x_i$ if $y_i = 1$ and $q(x_i) = 1 - x_i$ if $y_i = 0$. Then write $p_y(x_1, \ldots, x_n) = q(x_1) \cdots q(x_n)$ (i.e., $p_y(x)$ is the indicator function for $x = y$), and write the overall polynomial
$$p(x_1, \ldots, x_n) = 1 - \sum_{y \in \{0,1\}^n} f(y) p_y(x_1, \ldots, x_n).$$

2. Simply observe that we can write polynomials $p_{NOT}(x) = 1 - x$ and $p_{AND}(x_1, x_2) = x_1 x_2$ that agree with the NOT and AND functions everywhere; then since NOT and AND are a universal set of gates, we can write a polynomial matching any function.

It turns out that the polynomial degree of f is polynomially close to all the query complexity measures. (In fact, the way that we prove that some of the query complexity measures are polynomially close to each other is that we show that they are both polynomially close to the polynomial degree.) This is especially surprising since this is a very algebraic measure of complexity and seems unrelated to the relatively combinatorial notion of decision trees.

7.2 Sensitivity and the conjecture

Definition 27 (Sensitivity)**.** *The sensitivity $S(f, x)$ of a function $f : \{0,1\}^n \to \{0,1\}$ on input $x \in \{0,1\}^n$ is the number of bits of x that, if flipped, flip the value of f:*

$$S(f, x) = \left| \{ i \in [n] : f(x \oplus e_i) \neq f(x) \} \right|,$$

[20] However, note that the particular polynomial relationship actually matters a lot. Computing the n-bit OR function, for instance, has n deterministic or randomized query complexity, but as we'll see at the end of the semester, it has $\sqrt{n}$ quantum query complexity by *Grover's algorithm*.

[21] It can also be useful to think about polynomials where we plug in -1 and 1 instead of 0 and 1 (although we can convert between the two by composing with linear maps). One good reason to do this is that the coefficients of the resulting polynomial end up being related to the *Fourier transform* of the Boolean function F.

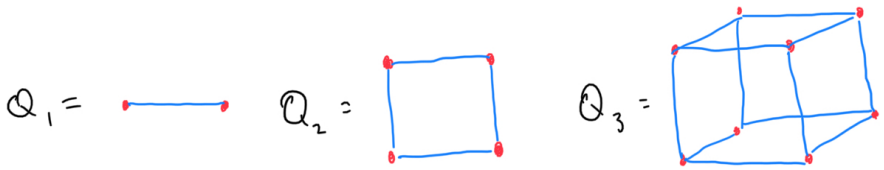

Figure 10: Visual representations of the first three hypercube graphs.

where e_i is the string of all 0's except for a 1 at position i. The sensitivity $S(f)$ of a function $f : \{0,1\}^n \to \{0,1\}$ is the maximum of its sensitivities on each input:

$$S(f) = \max_{x \in \{0,1\}^n} S(f, x).$$

Let's consider examine simple sensitivities. The sensitivity of n-bit AND and OR are both n (consider the string of all 1's in the case of AND, and of all 0's in the case of OR — in both cases, flipping any bit will flip the output). We also claim that sensitivity of the matrix 1-column problem as discussed above is $\sqrt{n}$:

- If $f(X) = 1$, note that flipping a 0 to a 1 will never make f become 0. Then consider two cases. If there is only one column of 1's, then flipping any of those 1's to a 0 will flip the output, so the sensitivity is $\sqrt{n}$; if there are two columns of 1's, then flipping a single 1 to a 0 can never flip the output, so the sensitivity is 0.
- If $f(X) = 0$, similarly, flipping a 1 to a 0 will never make f become 1. And the only way that flipping a 0 to a 1 flips f is if that 0 is the only 0 in its column; thus, the sensitivity is at most $\sqrt{n}$.

Since we've seen many polynomially close forms of complexity, there is a natural conjecture:

Conjecture 28 (Sensitivity conjecture)**.** *The sensitivity of f is polynomially close to the polynomial degree of f (or, equivalently, to e.g. deterministic decision tree complexity).*

This was first raised in [NS94].[22] Furthermore, their work implied that $S(f) \leq d(f)^2$ (where $S(f)$ is f's sensitivity and $d(f)$ is f's polynomial degree), which is one direction of the relationship. Thus, to prove the sensitivity conjecture, it remained to show that $S(f)$ is bounded below by a polynomial function of $d(f)$. This past summer, [Hua19] proved the conjecture, and his proof only takes two pages. This was exciting because it's very rare to see a longstanding open problem resolved by such a short and sweet proof. In what follows, we will present some of the key aspects of Huang's result, and attempt to motivate them. For more background on the conjecture, see [HKP10].

7.3 From functions to graphs

The first important step towards proving the sensitivity conjecture was a result in [GL92], which showed that the missing direction of the conjecture would follow from a conjecture about a particular graph. To pose this conjecture, we'll first need some definitions.

Definition 29 (Hypercube graph Q_n)**.** *The hypercube graph Q_n is a graph on vertices $\{0,1\}^n$ with an edge between u and v iff $u = v \oplus e_i$ for some i.*

See Figure 10 for visual representation of the first several hypercube graphs.

To give some intuition as to why it makes sense to think about these graphs; we draw a correspondence between a function $f : \{0,1\}^n \to \{0,1\}$ and the set $U = \{v \in V : f(v) = 1\}$. $(U, \overline{U})$ is a cut on the graph

[22] This initial form of the conjecture actually posited that sensitivity is polynomially close to a measure of complexity called *block sensitivity*, which is not of interest to us, since block sensitivity is polynomially close to the other measures we've seen (this was shown in the original paper).

Q_n, and $S(f, v)$ is the number of edges incident to v that cross the cut. Equivalently, since every vertex in Q_n has degree n,

$$S(f, v) = n - \deg_U v,$$

where $\deg_U v$ represents the number of edges from v to another vertex in U.

And now for a few more definitions:

Definition 30 (Induced subgraph)**.** *Given a graph $G = (V, E)$, the subgraph induced by a subset of vertices $U \subset V$ is the graph $G_U = (U, E \cap (U \times U))$.*

In other words, this is graph we get when we restrict our attention to the vertices U and all edges that go between vertices in U.

Definition 31 (Maximum degree)**.** *Given a graph $G = (V, E)$, the maximum degree $\Delta(G)$ of G is*

$$\Delta(G) = \max_{v \in V} \deg(v).$$

Now we are in a position to state the conjecture of [GL92], which implies the sensitivity conjecture:

Conjecture 32. *For any induced subgraph G of Q_n with more than 2^{n-1} vertices, the maximum degree $\Delta(G) \geq \sqrt{n}$.*

This is the conjecture that was proved in [Hua19]. The reason that this conjecture is nice is that we no longer have to worry about the polynomial degree of f; it is purely about graphs. Observe also that to prove it for all induced subgraphs, it suffices to prove it in the limited case of induced subgraphs with $2^{n-1} + 1$ vertices, since any subgraph with more vertices can be further induced to a subgraph with $2^{n-1} + 1$ vertices while the maximum degree can only decrease.

One more important observation about the hypercube Q_n is that it is bipartite. In particular, we partition the vertices based on the parity of the corresponding strings — there are no edges between strings of the same parity. This partition shows us why it's essential that we consider induced subgraphs with strictly more than 2^{n-1} vertices in the conjecture: Otherwise, the subgraph induced by the subset of vertices of one particular parity (there are 2^{n-1} such vertices), the maximum degree is zero (since there are no edges at all in the induced subgraph). But adding one more vertex to this particular set causes the maximum degree of the induced subgraph to jumps to at least $\sqrt{n}$ (it actually becomes n). We wish to show that the conjecture holds in all possible induced subgraphs with $2^{n-1} + 1$ vertices.

7.4 From graphs to matrices

So how can we prove this conjecture about graphs? We will employ tools from *spectral graph theory*, which is a branch of discrete mathematics that proves properties about graphs based on the eigenvalues of some matrices related to the graphs. (As a matter of vocabulary, the set of eigenvalues of a matrix is referred to as its *spectrum*.) While this probably makes no sense if you haven't seen such methods before, we will soon see why it is a fruitful approach.

Definition 33 (Adjacency matrix)**.** *The adjacency matrix $A(G) \in \{0, 1\}^{n \times n}$ for a graph $G = ([n], E)$ is given by the entries*

$$(A(G))_{ij} = \begin{cases} 1 & (i, j) \in E \\ 0 & (i, j) \notin E. \end{cases}$$

Note that the adjacency matrix is always symmetric in an undirected graph. This implies, by the *spectral theorem*, a standard result of linear algebra, that the adjacency matrix is diagonalizable.

We will, following the method of [Hua19], use spectral graph theory techniques to prove a lower bound on the maximum degree of induced subgraphs of Q_n by considering the eigenvalues of the adjacency matrix.[23] This approach stems from the following essential proposition:

[23] Another important matrix in spectral graph theory is called the *Laplacian* $L(G)$; this is formed by writing the diagonal matrix $D(G)$ where $(D(G))_{ii} = \deg(i)$, and then $L(G) = D(G) - A(G)$.

Proposition. *For a graph G, the maximum eigenvalue $\lambda_1(A(G))$ of $A(G)$ is at most the maximum degree $\Delta(G)$.*

Proof. To prove the theorem, we will show that $\Delta(G)$ is greater than or equal to all eigenvalues of $A(G)$. So let λ be an arbitrary eigenvalue of $A(G)$ with eigenvector v; we want to show that $\Delta(G) \geq \lambda$.

Note that the maximum degree of G is, by definition, the same as the maximum number of 1's in any row of the adjacency matrix (or column, since it's symmetric). WLOG rescale v so that its greatest entry is 1 in position i and its other entries are at most 1. Since $Av = \lambda v$, we have in particular

$$\lambda = \lambda v_i = \sum_{j=1}^{n} a_{ij} v_j.$$

But we picked v so that all entries are at most 1, so

$$\sum_{j=1}^{n} a_{ij} v_j \leq \sum_{j=1}^{n} a_{ij} = \deg(i),$$

the degree of the i-th vertex of G. Thus, $\lambda \leq \deg(i) \leq \Delta(G)$. □

When we restrict a matrix A to a subset of the coordinates (i.e., the same subset of rows and columns), we call the resulting matrix a *principal submatrix*; thus, the adjacency matrix of an induced subgraph of G is a principal submatrix of the adjacency matrix of G. So at this point, via the proposition, to prove the sensitivity conjecture, it suffices to prove a lower bound on the largest eigenvalues of principle submatrices in more than half of the coordinates.[24]

The tool we want to use to bound the eigenvalue is the following classic theorem of linear algebra:

Theorem 34 (Cauchy's interlace theorem)**.** *Let X be a symmetric $n \times n$ matrix, and Y be a principal $m \times m$ submatrix. If X has eigenvalues $\lambda_1 \geq \ldots \geq \lambda_m$, and Y has eigenvalues $\mu_1 \geq \ldots \geq \mu_n$, then for each i,*

$$\lambda_{i+n-m} \leq \mu_i \leq \lambda_i.$$

For the particular induced subgraphs we're considering, $X = A(Q_n)$ is a $2^n \times 2^n$ matrix, and $Y = A(G)$ is a $(2^{n-1}+1) \times (2^{n-1}+1)$ principal submatrix. Our goal is to bound μ_1, the largest eigenvalue of the adjacency matrix of the subgraph induced by some subset of $2^{n-1}+1$ vertices. Cauchy's interlace theorem gives

$$\lambda_{2^{n-1}} = \lambda_{1+2^n-(2^{n-1}+1)} \leq \mu_1 \leq \lambda_1.$$

Unfortunately, the lower bound of $\lambda_{2^{n-1}}$ for μ_1 is useless, since for $A(Q_n)$, it turns out to be a small constant, while we want a bound of $\sqrt{n}$.

However, we shouldn't give up hope — the proof is salvageable! A clever insight of Huang is to consider *signings* of $A(Q_n)$, which are matrices $A'(Q_n)$ that agree with $A(Q_n)$ except for the signs of the entries, i.e., $A'(Q_n)$ can have -1's in position where $A(Q_n)$ has a 1. This is a whole family of matrices related to $A(Q_n)$. And if we carefully inspect the proof of Proposition 7.4, we see that it extends to a stronger statement: We can lower bound the maximum degree of G by the largest eigenvalue of any *signing* of $A(G)$.

At this point, we just have to choose a good signing of $A(Q_n)$ (i.e., one for which $\lambda_{2^{n-1}} = \sqrt{n}$); Cauchy's interlace theorem implies a lower bound for the largest eigenvalue of the signed adjacency matrix of any induced subgraph with 2^{n-1} vertices; then this lower bound in turn implies a lower bound the maximum degree of any such induced subgraph!

What signed version $A'(Q_n)$ of $A(Q_n)$ should we choose? We will attempt to do a bit of "reverse engineering" to figure it out in the same way that [Hua19] might have done it. First, we can recall some facts

[24] We can prove that $\lambda_1(A(G)) \geq \sqrt{\Delta(G)}$, where $\lambda_1(A(G))$ is again the largest eigenvalue of $A(G)$; this implies that proving polynomial bounds on $\lambda_1(A(G))$ is not only sufficient to bound $\Delta(G)$, it is actually necessary — so we might be on the right track!

about eigenvalues. First off, the sum of the eigenvalues of a matrix is the trace of the matrix; since $A(Q_n)$ has trace 0 (since all of its diagonal entries are 0), the eigenvalues of $A'(Q_n)$ must also sum to 0. Furthermore, the *Frobenius norm* of a matrix (the square root of the sum of its entries squared) is equal to the square root of the sum of its eigenvalues squared; thus, since $A(Q_n)$ and $A'(Q_n)$ will have the same squared norm, the sum of the squared eigenvalues of $A'(Q_n)$ must agree with the sum of the squared eigenvalues of $A(Q_n)$. Finally, it is a known fact in spectral graph theory that for any G, $A(G)$ has a spectrum (set of eigenvalues) that is symmetric around 0 iff G is bipartite; furthermore, any signed adjacency matrix of a bipartite graph must still have a symmetric spectrum.

Intuitively, we can think about the spectra of $A(Q_n)$ and $A'(Q_n)$ as distributions; by the above discussion, they must have mean 0; they must have the same variance; and they must be symmetric about 0. So to make $\lambda_{2^{n-1}}$ as large as $\sqrt{n}$ in $A'(Q_n)$, the only thing we can do is make our spectrum very "bimodal". In particular, we can show that if we could get the lower bound we wanted from $A'(Q_n)$, we must have half of the eigenvalues be $\sqrt{n}$ and the other half be $-\sqrt{n}$. And if $(A'(Q_n))^2 v = nv$ for any eigenvector, since eigenvectors form a basis, we actually have $(A'(Q_n))^2 v = nv$ for all v, so $(A'(Q_n))^2 = nI$. So if the signed submatrix approach will work, we must construct a signing $A'(Q_n)$ such that $(A'(Q_n))^2 = nI$. And the converse is true: if $(A'(Q_n))^2 = nI$, then the eigenvalues of $A'(Q_n)$ are only $\sqrt{n}$ and $-\sqrt{n}$. All that remains is to find a signing $A'(Q_n)$ such that $(A'(Q_n))^2 = nI$.

First, let's think about the structure of $A(Q_n)$. The first two examples are

$$A(Q_1) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \text{ and } A(Q_2) = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

In particular, in block matrix form, we have

$$A(Q_2) = \left[\begin{array}{c|c} A(Q_1) & I_2 \\ \hline I_2 & A(Q_1) \end{array}\right].$$

This generalizes to

$$A(Q_n) = \left[\begin{array}{c|c} A(Q_{n-1}) & I_n \\ \hline I_n & A(Q_{n-1}) \end{array}\right];$$

this makes sense because the graph Q_n is constructed by joining two "parallel" copies of Q_{n-1} and attaching only "parallel" vertices (e.g. think about going from the square Q_2 to the cube Q_3 — the cube is two squares, along with edges between matching vertices of the squares).

And now all we have to do is define $A'(Q_1) = A(Q_1)$ and

$$A'(Q_n) = \left[\begin{array}{c|c} A'(Q_{n-1}) & I_n \\ \hline I_n & -A'(Q_{n-1}) \end{array}\right].$$

It follows by a simple induction that $(A'(Q_n))^2 = nI$, as desired.

8 Tselil Schramm: *Approximation Algorithms* (11/01/2019)

Biography

Tselil Schramm is a postdoc cohosted by Boaz Barak here at Harvard and by Jon Kelner, Ankur Moitra, and Pablo Parrilo at MIT. She was an undergraduate at Harvey Mudd and then got her Ph.D. in theoretical computer science at Berkeley, where she studied approximation algorithms and hardness of approximation. Through her work in approximation algorithms, she became really interested in *semidefinite programming*, which is a useful tool in thereotical computer science that will play an important role in today's talk. These days, she's also interested in applications of semidefinite programming outside of approximations (and in particular in statistics), as well as spectral algorithms.

8.1 Optimization problems and approximations

In class, we've seen a number of **NP**-hard problems, which we believe to lack efficient algorithms (assuming $\mathbf{P} \neq \mathbf{NP}$). However, a natural question to ask about such questions is whether there are efficient algorithms that can solve them *approximately* (i.e., within some acceptable margin of error).

First off, note that the particular functions we have considered in class have been *decision problems*, i.e., functions whose codomain is $\{0,1\}$. Examples of decision problems include "Is the 3CNF formula ϕ satisfiable?" (3SAT) and "Does the graph G have a cut of size at least k?" (MaxCut). However, the clearest definition of approximation algorithms is for *optimization problems*. The optimization form of 3SAT is "What is the maximum number of clauses that can be simultaneously satisfied in the 3CNF formula ϕ?", and the optimization form of MaxCut is "What is the largest number of edges that can be cut by any partition of the graph G?" In general, optimization problems will be functions of the form $F : \{0,1\}^* \to \mathbb{N}$. However, note that given an efficient algorithm to solve the decision form of a problem, we can solve the optimization form via binary search. (Conversely, it is easy to solve the decision problem once the actual optimal value is known.)

Now, we will define approximation algorithms:

Definition 35 (α-approximation algorithm)**.** *Let $F : \{0,1\}^* \to \mathbb{N}$ be an optimization problem, and let $\alpha < 1$. An algorithm $\mathcal{A}$ α-approximates F iff for all $x \in \{0,1\}^*$,*

$$\alpha F(x) \leq \mathcal{A}(x) \leq F(x).$$

The closer α is to 1, the stronger of a guarantee we have that the output of $\mathcal{A}$ is close to the actual value of F for all x. Studying which **NP**-complete optimization problems have efficient approximations (and for which values of α) is the principal goal of the field of *hardness of approximation*. Note that this entire subject is predicated upon the assumption that $\mathbf{P} \neq \mathbf{NP}$, since if $\mathbf{P} = \mathbf{NP}$ then all **NP**-complete problems can be solved exactly (i.e., non-approximately) in polynomial time, and approximations become unnecessary. But if $\mathbf{P} \neq \mathbf{NP}$, it turns out that there is a lot of variation among **NP**-complete problems in terms of approximability (some can be approximated "well", some can be approximated "okay", and some can't be approximated "at all"). Some examples:

- Hard problems to approximate: Feige and Haståd showed that it is impossible to get even a $(1/n^{1-\epsilon})$-approximation to the MaxClique problem for any $\epsilon > 0$ (if $\mathbf{P} \neq \mathbf{NP}$). (In other words, no approximation can do better than checking some fixed, constant number of vertices).
- Easy problems to approximate: Some **NP**-hard problems have *polynomial-time approximation schemes (PTAS)*, which means that for all $\epsilon > 0$, they can be $(1-\epsilon)$-approximated by polynomial-time algorithms. An example of a PTAS problem is the *Euclidean traveling salesman problem* EuclideanTSP, which is the problem of finding optimal Hamiltonian cycles in *Euclidean graphs* (graphs where the vertices are embedded in $\mathbb{R}^d$ and edge lengths are distances between the vertices). Even stronger than a PTAS is a *fully polynomial-time approximation scheme* (FPTAS), which is a single algorithm that gives a $(1-\epsilon)$-approximation and has runtime that is polynomial in both n and $\frac{1}{\epsilon}$. Examples of problems with an FPTAS include the SubsetSum and Knapsack problems.

Typically, theoretical computer scientists think of all of **NP**-complete problems as "equivalent", since they can all be reduced to each other. Thus, an added benefit of considering hardness of approximation is that it some additional interesting "structure" among the **NP**-complete problems; we have a well-defined way of saying that some are harder than others.[25]

[25] To make this a bit more concrete, we can introduce the notion of a *promise problem*, which are "decision problems that are only defined for a subset of inputs." In general, if we have a promise problem $F : S \to \{0,1\}$ for some set $S \subset \{0,1\}^*$, an algorithm $\mathcal{A}$ computes F iff for all $x \in S$, $F(x) = \mathcal{A}(s)$. However, if $x \notin S$, $F(x)$ can be anything — F can return 0 or 1, or not halt. In other words, since F is "promised" that its input is in x, it doesn't even have to be able to tell if $x \in S$ or $x \notin S$. For any $c, s \in \mathbb{N}$, we can convert an optimization problem $F : \{0,1\}^* \to \mathbb{N}$ into a (c,s)-*promise problem*, which is the problem of telling whether $F(x) \geq c$ or $F(x) < s$, given the promise that one of these is true. Then an α-approximation for F naturally

8.2 Naïve approximation for MaxCut

Today, we will almost exclusively focus on the optimization form of the MaxCut problem, which we define as follows:

Definition 36 (MaxCut problem)**.** *Given a graph G, $MAXCUT(G)$ is the highest number of edges that can be cut by some partition of G.*

To reiterate, the decision version of this problem is **NP**-hard; thus, if we believe that $\mathbf{P} \neq \mathbf{NP}$, we believe that there is no efficient algorithm to solve this problem.

There is a classic, well-known $\frac{1}{2}$-approximation for MaxCut (it was known as far back as [SG76]):

Algorithm 3 MaxCut $\frac{1}{2}$-approximation

Input: A graph $G = ([n], E)$

```
L, R := ∅
for i := 1 to n do
    Flip a coin c_i
    if c_i = "heads" then
        L := L ∪ {i}
    else
        R := R ∪ {i}
    end if
end for
return (L, R)
```

In other words, this algorithm randomly and independently assigns each vertex to one side of the partition. Thus, every edge in the graph is cut with probability $\frac{1}{2}$ (since a single edge being cut is equivalent to its two endpoints being in different sets). Hence, by linearity of expectation, the expected number of edges cut by the resulting partition is $\frac{1}{2}$ of the total edges. And since the size of the maximum cut is at most the number of edges, we have that the expected size of the cut is at least $\frac{1}{2}MAXCUT(G)$. This algorithm is linear-time in the size of G, and standard derandomization arguments (that are out of scope for this lecture) can be used to turn this algorithm into a *deterministic*, polynomial-time $\frac{1}{2}$-approximation algorithm.

So the "best possible α" for which MaxCut has a polynomial-time α-approximation algorithm is at least $\frac{1}{2}$. It was conjectured until the 90s that there was no better approximation.[26]

8.3 Convex relaxations and semidefinite programming

It turns out that we can do much better than the $\frac{1}{2}$-approximation, but we need to use some more sophisticated technique. In particular, we will derive a polynomial time 0.878-approximation for MaxCut using a form of optimization called *semidefinite programming*, the application of which is still a state-of-the-art research area in approximation algorithms. This algorithm was first developed in [GW95].

extends to solving (c, s)-promise problems for F where $\frac{c}{s} \geq \alpha$. For instance, a $\frac{7}{8}$-approximation for 3SAT yields an algorithm for the decision problem "Is ϕ satisfiable, or are fewer than $\frac{7}{8}$ of its clauses simultaneously satisfiable, given that one of these is true?" The results of this section imply that some of these promise problem versions of **NP**-complete problems are strictly harder than others.

[26] Why was this a reasonable conjecture? There is a similar algorithm to the above for the 3SAT optimization problem ("Find the maximum number of simultaneously satisfiable clauses in the formula ϕ") that just picks a random assignment; each clause is satisfied with probability $\frac{7}{8}$ (since there is only one of eight possible settings of the three variables such that all three literals in the clause are false), and so a random assignment is expected to satisfy $\frac{7}{8}m$ clauses if ϕ has m clauses. Thus, this algorithm is a $\frac{7}{8}$ approximation for 3SAT. And [Hås01] showed that it is a consequence of the celebrated *PCP theorem* (from [Aro+98]) that there is no polynomial-time $(\frac{7}{8}+\epsilon)$-approximation algorithm for $3SAT$ for any $\epsilon > 0$ (as long as $\mathbf{P} \neq \mathbf{NP}$). The conjecture that the random-partition $\frac{1}{2}$-approximation is the best approximation for MaxCut likely seemed plausible because it is very similar to the statement for 3SAT that ended up being true.

First, let's rewrite the MaxCut problem as an integer programming problem. We will look at solutions coming from the Boolean hypercube $S = \{-1, 1\}^n$. We claim that the MaxCut problem can be formulated as follows:

$$MAXCUT(G) = \max_{\mathbf{x} \in S} \sum_{(i,j) \in E} \frac{1}{2}(1 - x_i x_j).$$

Here's why this is equivalent: To any partition (U_1, U_2) of $V = \{v_1, \ldots, v_n\}$, we associate a vector $\mathbf{x} \in S$ where we set $x_i = -1$ if $v_i \in U_1$ and $x_i = 1$ if $v_i \in U_2$. Then the term inside the sum is 0 if $x_i = x_j$ and 1 if $x_i \neq x_j$, and so the sum counts the total number of edges cut by the partition. (Note that this is a *quadratic* function we are trying to optimize, not a linear function, but we won't require linearity.) $\frac{1}{m}$?

Now, we will proceed by doing a *relaxation* of the MaxCut integer program. This means that we will loosen the constraints that we are optimizing over, and then prove that the new "relaxed" solution we get is within a factor of α of the actual optimum for MaxCut under the stricter constraints. Formally, we will have two steps:

1. We will perform a *convex relaxation*: Instead of solving

   $$\max_{x \in S} f(x)$$

   directly, we will pick a convex, "nice" set $C \subset \mathbb{R}^n$ such that $S \subset C$.

2. We will employ a *rounding algorithm* $\mathcal{A} : C \to S$ with the property that if

   $$y^* = \arg\max_{y \in C} f(y),$$

   then

   $$f(\mathcal{A}(y^*)) \geq \alpha f(y^*)$$

   for some α.

These two ingredients together give an α-approximation to the original approximation together as follows: First, solve the relaxed optimization problem (this is possible to do efficiently due to certain technical conditions on the "niceness" of the convex set C), and then round the result back to a point in S. The resulting point in S, by the properties of the rounding algorithm, cannot be "too much" worse than the actual optimal point in S; this is where we will prove a bound of a specific α.[27]

The particular type of problem we will relax to is called a *semidefinite program*. First, we need the following definition:

Definition 37 (Positive semidefinite matrices)**.** *A symmetric matrix $X \in \mathbb{R}^{n \times n}$ is positive semidefinite (PSD) if all n eigenvalues of X are at least 0; this is denoted*

$$X \succeq 0.$$

(Recall that if $X \in \mathbb{R}^{n \times n}$ is symmetric then it must have n eigenvalues by the spectral theorem.)

Definition 38. *A semidefinite program is an optimization problem of the form*

$$\max_{X \in C} \langle X, O \rangle$$

where

$$C = \{X \in \mathbb{R}^{n \times n} : X \succeq 0, \forall i \in \{1, \ldots, k\} \ \langle X, A_i \rangle \geq c_i\},$$

[27] A standard example of this technique that you might have seen in an algorithms textbook (and is also mentioned in the textbook) is actually the MinCut problem. It turns out that it is possible to prove that when we relax the integer linear program for MinCut into a linear program over the reals, we are guaranteed that the optimum we find over the reals will actually still be at an integer point. This means that MinCut can be solved efficiently using linear programming solvers, and we don't even need to employ a rounding algorithm.

where O and $A_1, \ldots, A_k$ are matrices in $\mathbb{R}^{n \times n}$ and $\langle X, Y \rangle$ is the element-wise dot product

$$\langle X, Y \rangle = \sum_{k,\ell=1}^{n} x_{k\ell} y_{k\ell}.$$

For comparison, we can think of linear programming using the same type of setup except that X must be a *diagonal* matrix as opposed to any positive semidefinite matrix (since then each inner product $\langle X, Y \rangle$ is simply an inner product of the diagonals of X and Y, as vectors).

We will employ the following lemma:

Lemma 39. *A matrix $X \in \mathbb{R}^{n \times n}$ is positive semidefinite iff $X = VV^T$ for some matrix V.*

Proof. ($\Longrightarrow$) Suppose that $X \succeq 0$. Then we can diagonalize X as $U\Sigma U^T$. Then consider the entrywise square-root matrix $\Sigma^{\frac{1}{2}}$: We have $X = (U\Sigma^{\frac{1}{2}})(U\Sigma^{\frac{1}{2}})^T$.

($\Longleftarrow$) Conversely, suppose that $X = VV^T$ and X has a negative eigenvalue; then pick an eigenvector $\mathbf{u}$ for that eigenvalue, and so $\mathbf{u}^T X \mathbf{u} < 0$. Then we have $(\mathbf{u}^T V)(V^T \mathbf{u}) = ||V^T \mathbf{u}||^2 < 0$, which is a contradiction since vector norms are always nonnegative. □

First, we will write the following relaxation for MaxCut:

$$\max_{\substack{\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^n, \\ ||\mathbf{v}_1||^2 = \cdots = ||\mathbf{v}_n||^2 = 1}} \sum_{(i,j) \in E} \frac{1}{2}(1 - \langle \mathbf{v}_i, \mathbf{v}_j \rangle).$$

Each vertex is now associated to an n-length vector (as opposed to a scalar ± 1 as in the original problem, although this is equivalent when $n = 1$). And in this relaxed version, the inner product $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ now represents "how much" the edge between vertex i and vertex j is cut. If $\langle \mathbf{v}_i, \mathbf{v}_j \rangle \approx 1$, then $\mathbf{v}_i$ and $\mathbf{v}_j$ are close together and the corresponding edge is not cut; if $\langle \mathbf{v}_i, \mathbf{v}_j \rangle \approx -1$, then $\mathbf{v}_i$ and $\mathbf{v}_j$ are close to being opposites and the corresponding edge is cut. We are embedding the graph onto a hypersphere and then making vertices that share edges be as far from each other as possible. The unrelaxed optimization problem will have a maximum corresponding to placing all vertices on one side of the partition on the point $\mathbf{u}$ and the vertices on the other side to its antipode $-\mathbf{u}$.[28] To successfully pull off this relaxation, we will have to give a rounding algorithm to construct a partition from any embedding (not just one where all vertices are either at $\mathbf{u}$ or $-\mathbf{u}$ for some $\mathbf{u}$), and show that this partition cannot be too much worse than the actual optimal partition.

Now note that by the above lemma, we can re-express this new optimization problem directly as a semidefinite program:

$$\max_{\substack{X \in \mathbb{R}^{n \times n}, X \succeq 0, \\ x_{11} = \cdots = x_{nn} = 1}} \sum_{(i,j) \in E} \frac{1}{2}(1 - x_{ij})$$

since $X = VV^T$ and then $x_{ij} = \langle \mathbf{v}_i, \mathbf{v}_j \rangle$. It turns out that the niceness condition we need for convex relaxations to be efficiently solvable is that we can efficiently test whether a given point is in the set or not — and we can efficiently check whether a matrix is positive semidefinite by diagonalizing! Thus, we can solve this optimization problem efficiently. It remains for us to specify the rounding algorithm.

We will employ Algorithm 4 as our rounding algorithm; note that it is randomized (but it can be derandomized, although this is out of scope). Intuitively, we are randomly sampling a hyperplane to "cut" the sphere, and partition the vertices based on whether the corresponding vectors are on one side or the other of the hyperplane. The entire process is depicted in Figure 11.

Now it remains to prove a good bound for the result of this approximation:

[28] In particular, an equivalent form of the unrelaxed optimization problem is to optimize over only *rank-1* matrices $X = vv^T$; then each entry of v is $\pm c$ for some fixed c.

Algorithm 4 MaxCut semidefinite program rounding algorithm

Input: A list of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$

$L, R := \emptyset$
Randomly sample a unit vector $\mathbf{g}$
for $i := 1$ **to** n **do**
  if $\langle g, v_i \rangle > 0$ **then**
    $L := L \cup \{i\}$
  else
    $R := R \cup \{i\}$
  end if
end for
return (L, R)

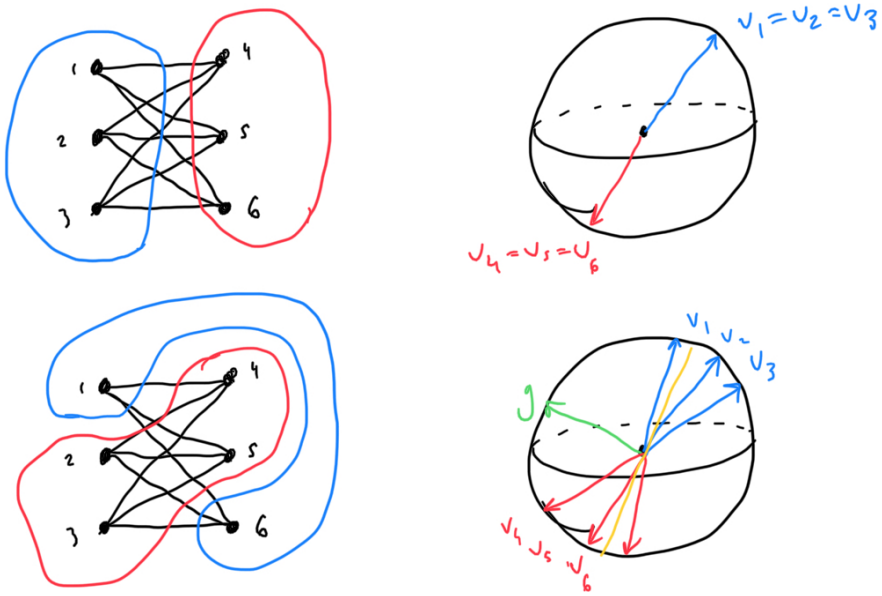

Figure 11: Clockwise from top left: (1) A graph G with $n = 6$ edges and $m = 9$ edges. This graph is bipartite, and so all 9 edges can be cut (the partition is depicted in the red and blue bubbles). (2) A pictorial representation of the answer to the optimization problem with the original constraints (although the actual optimization is over a 6-dimensional sphere): The vectors are constrained to be antipodal. (3) The problem is "relaxed", so that each vertex has its own vector and they are not necessarily antipodes. Note that this looks "close" to the solution to the unrelaxed problem. Furthermore, a particular (and unfortunate) choice of the separating vector g (in green) and a projection of the corresponding hyperplane (in yellow) is indicated. (4) The outcome of this unfortunate choice: Vertices 1 and 6 are placed on one side of the partition, and vertices 2, 3, 4, and 5 are on the other side. Only 6 edges are cut by this partition. However, we will show that the expected number of edges cut by the partition resulting from a random hyperplane is at least 0.878 times the best possible maximum cut.

Theorem 40. *Consider any unit vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$; let $x_i(g)$ be the particular side of the partition (-1 or 1) that vertex i is placed on by Algorithm 4 if the vector $\mathbf{g}$ is sampled (i.e., $x_i(\mathbf{g}) = 1$ if $\langle \mathbf{v}_i, \mathbf{g} \rangle \geq 0$ and $x_i(\mathbf{g}) = -1$ otherwise). Then*

$$\mathop{\mathbb{E}}_{\mathbf{g}} \sum_{(i,j) \in E} \frac{1}{2}(1 - x_i(\mathbf{g})x_j(\mathbf{g})) \geq 0.878 \sum_{(i,j) \in E} \frac{1}{2}(1 - \langle \mathbf{v}_i, \mathbf{v}_j \rangle).$$

Note in particular that this implies that the expected number of edges cut by the partition is at least $MAXCUT(G)$, since the optimal solution for the relaxed optimization problem is lower bounded by the solution for the unrelaxed problem.

Proof. We can rewrite the left side as a sum over edges of $\mathbb{P}[(i,j) \text{ is cut}]$ by linearity of expectation; thus, to prove the inequality, it suffices to prove that for each edge (i,j),

$$\mathbb{P}[(i,j) \text{ is cut}] \geq (1 - \langle \mathbf{v}_i, \mathbf{v}_j \rangle).$$

But this probability on the left-hand side is equal to the probability that if we randomly pick a hyperplane, it will separate $\mathbf{v}_i$ and $\mathbf{v}_j$; this is equal to the angle $\cos^{-1}(\langle \mathbf{v}_i, \mathbf{v}_j \rangle)$ between them over π.

Now using calculus, we can show that for all $x \in [-1, 1]$, $\cos^{-1}(x)\frac{1}{\pi} \geq 0.878\frac{1}{2}(1-x)$ (you can see this for yourself by minimizing the ratio using Mathematica). Thus,

$$\mathbb{P}[(i,j) \text{ is cut}] = \frac{\cos^{-1}(\langle \mathbf{v}_i, \mathbf{v}_j \rangle)}{\pi} \geq 0.878\frac{1}{2}(1 - \langle \mathbf{v}_i, \mathbf{v}_j \rangle),$$

as desired. □

Finally, we should note that it is conjectured that this constant $\alpha \approx 0.878$ is the best possible constant. In particular, the *unique games conjecture*, a major unsolved conjecture in complexity theory, was shown in to imply that many approximations for different problems are optimal, including the algorithm we've discussed for MaxCut [Kho+07].

9 Josh Alman: *Fine-Grained Complexity* (11/08/2019)

Biography

Josh Alman is a new postdoc in the theory group here at Harvard; he graduated just this past summer with his Ph.D. from MIT, where he was advised by Ryan Williams and Virginia Vassilevska Williams. (Originally, he was a grad student at Stanford, but he moved along with his advisers to MIT.) He also was an undergraduate at MIT. He works on topics in algorithm design and complexity theory, and especially problems on that have to do with algebra. His two favorite algorithms problems are:

1. The *matrix multiplication problem*: Given two $n \times n$ matrices A and B, compute their product $A \times B$. The naïve algorithm takes $O(n^3)$ time (to calculate each of the n^2 entries of AB, sum n pairwise products of entries in A and B). But we can do much better: The best known current algorithm is $O(n^{\approx 2.37298})$ (which is the fruit of years of effort in lowering the exponent).[29] There is a trivial $\Omega(n^2)$ lower-bound on the running time of any matrix multiplication algorithm, since such an algorithm must always look at all entries of at least one matrix. Many theorists conjecture that there is an $O(n^{2+\epsilon})$ algorithm for all $\epsilon > 0$. Matrix multiplication is an exceptionally important problem because matrix multiplication is very often used as a subroutine in the best-known algorithms for other problems; thus, improving the runtime of matrix multiplication also improves the runtime of many other best-known algorithms.

[29] The first-discovered better-than-$O(n^3)$ algorithm for matrix multiplication was *Strassen's algorithm*. This algorithm is very similar to Karatsuba's algorithm, the better-than-$O(n^2)$ algorithm we saw for integer multiplication. The idea for Strassen's algorithm is to express matrix multiplication recursively by breaking each matrix into four "quarter" blocks; the naïve algorithm then requires eight quarter-block multiplications, but clever algebra can reduce it to seven multiplications, resulting in an exponent of $\log_2 7 \approx 2.807$.

2. The *nearest neighbor problem*: Given a "universe" U, a distance metric d, and two sets $A, B \subset U$, find the points $a \in A$ and $b \in B$ that minimize $d(a, b)$. There are two main techniques for solving this problem efficiently:

   - *Locality-sensitive hashing (LSH)*: Use a hash-table-esque data structure with a hashing algorithm that is modified so that it is very likely to put nearby points in the same hash bucket, but other collisions are unlikely, and then, say, only compute the distance d when you get collisions.
   - *Fast matrix multiplication.* Embed the points into a vector space with the property that the inner product between two embedded vectors will be small iff the original two points of U were close, and then use a fast matrix multiplication algorithm to find the smallest inner product between any two vectors.

   Again, this comes up often as a subroutine in other currently best-known algorithms.

His third favorite problem is the *orthogonal vectors problem*, and this is the one we'll be talking about today!

9.1 Orthogonal vectors problem

Orthogonal vectors problem (OrthVec).
Given: Two sets $X, Y \subset \{0,1\}^d$, with $|X| = |Y| = N$.
Goal: Decide if there is some $x \in X$ and $y \in Y$ such that $\langle x, y \rangle = 0$.

Note that this inner product is over $\mathbb{R}$, not, say, $\mathbb{F}_2$; thus, it will only be zero if $x_i y_i = 0$ for all $i \in \{1, \ldots, d\}$. Thus, an equivalent formula of this problem is to decide whether there is some $x \in X$ and $y \in Y$ such that for all $i \in \{1, \ldots, d\}$, $x_i = 0$ or $y_i = 0$ (or both).

Again, OrthVec comes up as a subroutine in many different areas, one of which we will see later today. And OrthVec has a straightforward, polynomial-time algorithm, which consists of iterating through all possible pairs (x, y) and computes each dot product; it runs in time $O(N^2 d)$.

Since solving OrthVec is used so widely as a subroutine for other algorithms, it'd be really nice if we could prove some lower bounds on the runtime of the fastest OrthVec algorithm.[30] However, OrthVec $\in \mathbb{P}$, and it's not clear what tools we have to distinguish between polynomial complexities! The typical notions of polynomial-time reductions and **NP**-hardness that we've developed doesn't suffice.

One tool we might try to apply is the *time hierarchy theorem.* Recall, this theorem implies that, for instance, there is a problem that can be solved in $O(n^2)$ time but not in $O(n^{1.99})$ time. And this proof is actually a constructive proof (i.e., it gives an explicit function); however, this explicit function is very unnatural and it is not clear how it could be reduced to orthogonal vectors.

9.2 The strong exponential time hypothesis

We will actually prove that OrthVec is hard by reducing from k-sat to OrthVec. However, there are two important qualifications we must make:

- This won't be a standard polynomial-time reduction (in fact it won't even run in polynomial time), and we will have to be more careful than usual with the exact runtime of the algorithm and the size of the transformed instances.

[30] We have to fix a particular model of computation, since converting between two different models (e.g. Turing machines to λ-calculus) might incur a polynomial overhead. However, it turns out that converting between all commonly used models aside from the single-tape Turing machine only incurs polylogarithmic overheads (i.e., some polynomial in $\log n$); thus, when we give bounds like $O(f(n))$ throughout this talk, we really mean $O(f(n)p(\log n))$ for some polynomial p. This is often denoted $\tilde{O}(f(n))$. We will omit this technicality of notion for the purposes of simplification, and simply write $O(f(n))$.

- In order to prove a lower bound for OrthVec, we must assume a lower bound for k-SAT that is stronger than $\mathbf{P} \neq \mathbf{NP}$.

What particular complexity assumption will we make about k-SAT? Well, let's first think about k-SAT in general. Note that the number of unique clauses in a k-SAT formula is at most $(2n)^k = O(n^k)$, and so there is a brute-force algorithm (checking all possible assignments) in $O(2^n n^k)$ time. An improved algorithm was given in [Pat+05], with runtime $O(2^{n(1-c/k)})$ for some constant $1 < c < 2$, based on dynamic programming techniques. Note that as $k \to \infty$, the algorithm's runtime approaches 2^n. The particular, fine-grained complexity assumption we will make about k-SAT is that we can't do much better than this:

Conjecture 41 (Strong exponential time hypothesis (SETH))**.** *For all $\epsilon > 0$, there is some $k \geq 3$ such that k-SAT cannot be solved in $O(2^{n(1-\epsilon)})$ time.*

The conjecture that we want to prove about OrthVec is that the brute-force algorithm is optimal.

Conjecture 42 (Orthogonal vectors conjecture (OVC))**.** *There is no algorithm for* OrthVec *running in time $O(n^{2-\epsilon}d^c)$, for any $\epsilon > 0$ and $c \geq 0$.*

Note that these conjectures are very similar in nature! To prove that SETH implies OVC, we will perform a very careful reduction from k-SAT to OrthVec; we will have to carefully control the running time of the reduction, as well as the size of the instances we get. In particular, an instance of k-SAT on n variables will become an instance of OrthVec with $N = 2^{n/2}$ vectors in each set, each of length $d = O(n^k)$. This reduction is Algorithm 5.

Algorithm 5 k-SAT-to-OrthVec reduction

Input: A k-SAT formula $\varphi(z_1, \ldots, z_n) = c_1 \wedge \cdots \wedge c_m$

$X, Y := \emptyset$
for all $s \in \{0,1\}^{n/2}$ **do**
  for $i = 1$ **to** m **do**
    $x_i, y_i := 0$
    if the assignment $s0^{n/2}$ satisfies clause i **then**
      $x_i := 1$
    end if
    if the assignment $0^{n/2}s$ satisfies clause i **then**
      $y_i := 1$
    end if
  end for
  $X := X \cup \{x_1 \cdots x_m\}$
  $Y := Y \cup \{y_1 \cdots y_m\}$
end for
return (X, Y)

In this reduction, X represents all the possible subsets of clauses that can be satisfied by just setting the variables $z_1, \ldots, z_{n/2}$, and Y represents all the possible subsets of clauses satisfiable by setting $z_{n/2+1}, \ldots, z_n$. Thus, the reduction is correct, since φ will be satisfiable iff every clause is satisfiable (and each clause will be satisfied by a variable either coming from $z_1, \ldots, z_{n/2}$ or $z_{n/2+1}, \ldots, z_n$).

Theorem 43. *The strong exponential time hypothesis implies the orthogonal vectors conjecture.*

Proof. We will prove the contrapositive. Suppose that OrthVec has a $O(n^{2-\epsilon}d^c)$ algorithm for some $\epsilon > 0$ and $c \geq 0$. Then combine this algorithm with the k-SAT-to-OrthVec reduction (Algorithm 5) to get an algorithm for k-SAT. The runtime of this algorithm is the sum of the runtime of the reduction and the

runtime of the OrthVec algorithm. For k-SAT, the reduction takes $O(2^{n/2}n^k)$ time; and then the OrthVec algorithm takes time $O((2^{n/2})^{2-\epsilon}(n^k)^c)$. Thus, the total runtime is

$$O(2^{n/2}n^k + 2^{(n/2)(2-\epsilon)}n^{kc}) = O(2^{n(1-\epsilon/2)}n^{kc}) < O(2^{n(1-\epsilon/3)}).$$

Critically, this result holds *regardless of* k, violating SETH. □

So we have our first fine-grained complexity result. As an aside, a good question to ask at this point is whether SETH is a good assumption. It is strictly stronger than $\mathbf{P} \neq \mathbf{NP}$, so to the best of our knowledge, it could be possible that $\mathbf{P} \neq \mathbf{NP}$ but SETH is false. There also is, as you might expect, an *exponential time hypothesis (ETH)*, which posits that 3SAT requires time $2^{\epsilon n}$ for some $\epsilon > 0$; this is weaker than SETH but stronger that $\mathbf{P} \neq \mathbf{NP}$.

Experts have differing opinions. Our very own Prof. Madhu Sudan says:

> "The ETH looks very solid. The SETH looks increasingly solid."

But Ryan Williams, Josh's adviser, is a prominent SETH pessimist, citing a number of results showing that if SETH is false, there are nontrivial lower bounds. (See his survey [Wil19a], where he attempts to assign "likelihoods" to a number of open conjectures in complexity theory.) But this basically means that thinking about SETH is a win-win: Either it's true, and we get a rich theory of lower bounds in $\mathbf{P}$, or it's false, and we get hitherto unknown lower bounds!

9.3 The fine-grained hierarchy

We previously saw that SETH implies OVC. It turns out that OVC has many implications for the complexities of other problems in $\mathbf{P}$. We will prove such things by constructing new reductions. For instance, consider the following problem:

Regular expression sub-matching (SubRexp).
Given: A string s and a regular expression r.
Goal: Decide if there is a contiguous substring s' of s that matches r.

Letting $n = \max\{|r|, |s|\}$, there is a known dynamic programming algorithm that solves this problem in $O(n^2)$ time [Tho68]. Furthermore, we claim that this is actually optimal, assuming OVC:

Theorem 44. *Assuming the orthogonal vectors conjecture, there is no $O(2^{n-\epsilon})$-time algorithm for* SubRexp *for any constant $\epsilon > 0$.*

The major ingredient in the proof is the following lemma:

Lemma 45. *Given any $u \in \{0,1\}^d$, there is a regular expression r_u that exclusively matches strings $v \in \{0,1\}^d$ with $\langle u, v\rangle = 0$; furthermore, $|r_u| = O(d)$.*

Proof. Simply let $s_v = v$, and build r_u by replacing each 0 with (0|1) and leaving the 1s. □

This reflects the fact that if $u_i = 0$, then v_i can be either 0 or 1, whereas if $u_i = 1$, v_i must be 0 in order for u and v to be orthogonal. For instance, if $v = 1101$ and $u = 0110$, we will just set $s_v = 1101$ and $r_u = (0|1)00(0|1)$. Now, we describe a reduction:

Proof of Theorem 44. In the reduction (Algorithm 6), each r_{y_i} only matches strings of length d orthogonal to y_i. Thus, r_y will match a substring of s_x iff there is some $x_j \in X$ and $y_i \in Y$ such that $\langle x_j, y_i\rangle = 0$. Also, $|s_x|$ and $|r_y|$ are $O(Nd)$.

Now, suppose that for the sake of contradiction that there is an algorithm to solve SubRexp in time $O(n^{2-\epsilon})$. We can compose this algorithm with the above reduction to get an algorithm for OrthVec. The running time of this algorithm is $O(Nd + (N(d+1))^{2-\epsilon}) < O(N^{2-\epsilon/2}d^{2-\epsilon/2})$, contradicting OVC. □

Algorithm 6 OrthVec-to-SubRexp reduction

Input: Two sets $X = \{x_1, \ldots, x_N\}$, and $Y = \{y_1, \ldots, y_N\}$

1: $s_x := x_1 \# x_2 \# \cdots \# x_N$
2: $r_y := r_{y_1} | r_{y_2} | \cdots | r_{y_N}$ (as in the lemma)
3: **return** (s_x, r_y)

Other well-known results of this form (reducing from OrthVec) include one for the *graph diameter problem* of finding the longest path between any two vertices in a graph. And there are also a number of different conjectures aside from SETH that give rise to interesting fine-grained complexity results. These include:

- One example is the problem 3sum: Given a list of n numbers, does any subset of 3 sum to 0? There is a naïve $O(n^3)$-time brute force algorithm, and a classic $O(n^2)$ dynamic programming algorithm. If we assume that $O(n^2)$ is the running time of the fastest 3sum algorithm, we get interesting hardness results for a lot of problems in computational geometry (e.g. deciding whether, given n points, three of the lie on the same line).
- Another example is the *all-pairs shortest paths problem* apsp. The classic *Floyd-Warshall* algorithm gives an $O(n^3)$-time algorithm for this problem. Assuming that this is optimal yields many interesting lower bounds for other graph problems.

An excellent survey on fine-grained complexity is [Wil19b].

10 Raghu Meka: *Communication Complexity* (11/15/2019)

Biography

Prof. Raghu Meka is a professor at UCLA, but he's visiting MIT for the year. He mainly works in complexity theory; in particular, he is interested on pseudorandomness and communication complexity. He really likes communication complexity because it's a very "elegant" field — there are a lot of natural problems that have deep connections to other areas such as combinatorics, algebra, analysis, algorithms, etc.

10.1 Introduction

The basic setup of a *communication problem* was introduced in [Yao81]. We may phrase it as follows: Two parties, Alice and Bob, want to compute some two-input function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$; however, Alice only has access to the first half the input (x) and Bob only has access to the second half of the input (y). Given that Alice and Bob individually have unlimited resources (they can even, say, solve the halting problem), how can they together compute $f(x, y)$ while exchanging the smallest possible amount of information? For instance, say Alice and Bob have two calendars (represented as strings of bits indicating availability in a series of timeslots), and they together want to compute whether there is a time that both of them are available.

More specifically, Alice and Bob will send messages back and forth, and we will keep track of the total length of all sent messages. We assume that Alice and Bob are cooperating fully and have agreed on a *protocol* π describing their interaction; formally, this is a series of functions, each of which takes as input a party's input (x for Alice and y for Bob) and the previous messages and outputs the next message to send. A valid protocol π, on a particular problem instance (x, y), outputs $\pi(x, y) = f(x, y)$, while generating a *transcript* $\pi_{tr}(x, y)$ (the string of all sent messages). Then our goal, for a given communication problem f, is to find protocols with short worst-case transcript lengths.

The calendar problem we mentioned above is known in the literature as the Disjointness problem:

Problem DISJOINTNESS**.**
Given: Two strings $x, y \in \{0,1\}^n$.
Goal: Decide if there is no common element, i.e., some i such that $x_i = y_i = 1$.

It turns out that the DISJOINTNESS problem is a very fundamental problem in communication complexity theory; it might be considered the analogue of 3SAT for communication.

Another canonical problem is the EQUALITY problem:

Problem EQUALITY**.**
Given: Two strings $x, y \in \{0,1\}^n$.
Goal: Decide if $x = y$.

We will abbreviate DISJOINTNESS to DISJ and EQUALITY to EQ.

Given a communication problem f, we will let $\mathrm{CC}(f)$ denote its communication complexity (the lowest worst-case transcript length). We assume, for technical reasons, that the last transmitted message is a single-bit message containing the final answer.

We also want to extend this notion of communication complexity to the randomized setting; we should have some notion of correctness

$$\mathbb{P}[f(x,y) = \pi(x,y)] \geq \frac{2}{3}.$$

But the way that randomness works is really subtle here; in particular, there are two distinct parties, and if they share any information between them, they might be penalized! We can actually define two notions of randomized communication complexity:

- With *public randomness*, there is a single random string r that both parties can read; oftentimes, communication theorists say that r is "in the sky". In effect, Alice and Bob share r without having to communicate it at all. As usual, they must correctly compute $f(x,y)$ with probability $\frac{2}{3}$ over the choice of r. We will denote the complexity in this case as $\mathrm{CC}^{pub}(f)$.
- With *private randomness*, Alice gets a random string r_x and Bob gets a random string r_y; r_x and r_y are independent. Again, Alice and Bob must compute $f(x,y)$ correctly with probability $\frac{2}{3}$ over the joint choice of r_x and r_y, but now, they must pay a price to establish a shared source of randomness by communicating r_x and r_y. We will denote this complexity as $\mathrm{CC}^{priv}(f)$.

See Figure 12 for a illustration of these three different models.

Observe that for all communication problems f, $\mathrm{CC}^{pub}(f) \leq \mathrm{CC}^{priv}(f) \leq \mathrm{CC}(f)$, since any private-randomness protocol for f can be transformed into a public-randomness protocol for f that just uses, say, the first half of r as r_x and the second half of r as r_y, and any deterministic protocol for f can be transformed into a private-randomness protocol for f that simply ignores r_x and r_y. Finally, observe that in general, $\mathrm{CC}(f) \leq n+1$ for any f, since a trivial protocol is for Alice to just send x to Bob, and then Bob can compute $f(x,y)$ and send the result back to Alice.

10.2 The complexity of EQUALITY

Let's do a simple example: We will find the public-randomness, private-randomness, and deterministic communication complexities for EQUALITY.

A reasonable guess for the deterministic complexity of EQUALITY is $n+1$, since it seems like in the worst case, we should have to communicate a bit in order to check whether $x_i = y_i$ for each i. But how would we formalize this intuition?

Lemma 46 (Cut-paste lemma)**.** *Say Alice and Bob follow a protocol π. If $\pi_{tr}(x,y) = \pi_{tr}(x',y')$, then these must also both be equal to $\pi_{tr}(x,y') = \pi_{tr}(x',y)$.*

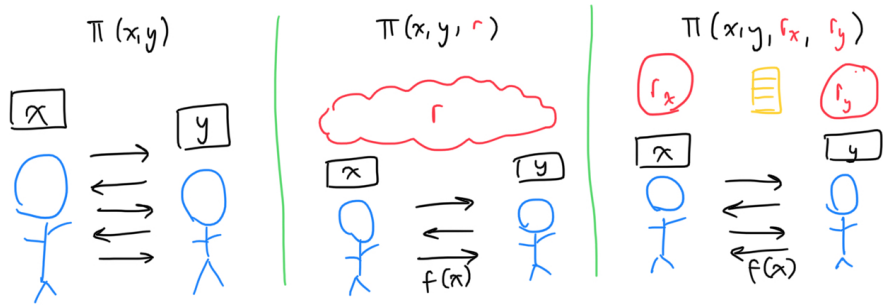

Figure 12: Three different models for communication. On the left, Alice and Bob engage in *deterministic communication*; they each get an input x and y, respectively, and cooperate to execute a protocol $\pi(x, y)$ to compute $f(x, y)$. In the middle, Alice and Bob share access to *public randomness* "in the sky"; both can make decisions based on the same sequence r of random bits, and their protocol $\pi(x, y, r)$ must correctly compute $f(x, y)$ with high probability over choice of r. On the right, Alice and Bob have their own independent sources of *private randomness*, r_x and r_y; unlike in the public model, they must pay a communication penalty to established a shared source of randomness. Again, their protocol $\pi(x, y, r_x, r_y)$ must correctly compute $f(x, y)$ with high probability over independent choice of r_x and r_y.

Proof. Suppose that the given condition holds, and that Alice has input x' and Bob has input y. When Bob receives Alice's first message, since Alice sends the same first message for x and x', he will behave the same as he would have if Alice instead had input x. Similarly, since Alice receives the same message from Bob and cannot tell if Bob instead has y', she will behave the same as Bob *did* have y'. This continues on similarly until the protocol ends. All these messages match the original, identical, pair of transcripts, and the case where Alice has x and Bob has y' is similar.[31] □

So we can apply the pigeonhole principle on the transcripts to get a bound for the communication complexity of EQUALITY!

Now suppose that there exists a protocol π with requiring less than $n+1$ bits. Consider the set $\{(x, x) : x \in \{0,1\}^n\}$ of satisfying inputs for EQUALITY. Then there are $\leq 2^{n-1}$ possible transcripts for these input pairs, since all the transcripts must end with a 1. So by PHP, there is some $x \neq x'$ such that $\pi_{tr}(x, x) = \pi_{tr}(x', x')$. But then by the cut-paste lemma, $\pi_{tr}(x, x) = \pi_{tr}(x, x')$. But this is impossible since $\pi(x, x) = 1$, while $\pi(x, x') = 0$!

A more systematic way to think about this proof is to consider the $2^n \times 2^n$ matrix with, say, rows indexed by x and columns by y, where the entry in position (x, y) is $f(x, y)$; this matrix is called the *communication matrix* of f. Each time there is a message, the set of possible values of (x, y) decreases; in particular, when Alice sends a message, the matrix is split horizontally (since now the actual input (x, y) can only lie in a subset of the rows), and when Bob sends a message, the matrix is split vertically. At any given point in the protocol, the set of all possible pairs (x, y) forms a *combinatorial rectangle* (a subset $A \times B \in \{0,1\}^n \times \{0,1\}^n$) called a *cell*. When the protocol ends, for it to be correct, f must have the same value on each cell; we say that each cell must be *monochromatic*. Furthermore, if the protocol's transcript is c bits, then the matrix ends up in $\leq 2^c$ distinct cells (not including the final message, which actually represents the protocol's computed value — all cells must be monochromatic *before* the final message is sent). A visualization of a toy communication matrix is in Figure 13.

So a new proof of the lower bound for EQUALITY is as follows: If $\mathrm{CC}(f) = c$, then communication matrix of f can be split into 2^c monochromatic cells. However, the communication matrix for EQUALITY is the identity matrix. The identity matrix has 2^n entries, and no monochromatic cell can contain more than a

[31] All of this can be made completely rigorous with a more formal definition of a protocol as a *sequence of functions*: The first message (from Alice) is a function of her input, the second message (from Bob) is a function of his input and Alice's message, and the in general, each message is a function of the party's input and all previously sent messages (as well as the randomness in the random case).

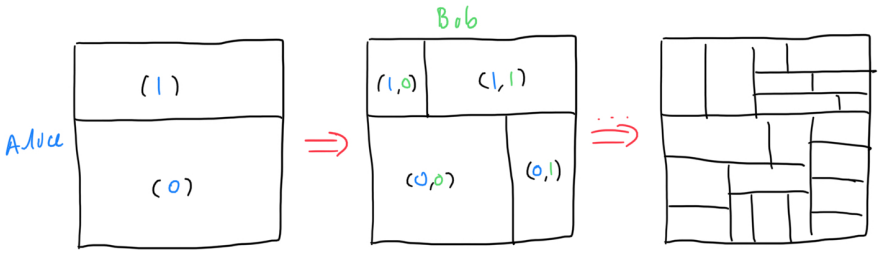

Figure 13: A visualization of the evolution of the communication matrix of a particular protocol. The first matrix represents the state of the protocol after Alice has sent her first message (a 1-bit message, for simplicity). There are only two possible transcripts, a 0 and a 1; thus, the matrix of all possible transcripts has been divided in half. (Note that, in general, this division doesn't have to be into sequential indices, but again, it makes it simple to visualize.) Next, Bob sends a message; the matrix is now divided into four rectangles, each corresponding to one of the possible transcripts. At each stage, a given rectangle represents a single transcript, as well as all possible input pairs that could have led to that transcript.

single diagonal entry (since then it must contain both 1's and 0's). Thus, there must be at least 2^n cells, and so the shortest possible transcript for EQUALITY has length $n+1$.

So deterministically, EQUALITY is hard. What if we incorporate randomness? It turns out that the communication complexity of EQUALITY with public randomness is *constant*!

The key idea is to use some sort of *hash function*, which is a randomized function $h : \{0,1\}^n \to \{0,1\}^m$, with the property that if $x \neq x'$, then

$$\mathbb{P}[h(x) = h(x')] \leq \epsilon$$

for some small ϵ; a pair of distinct strings $x, x' \in \{0,1\}^n$ with $h(x) = h(x')$ is termed a *hash collision*, so this property captures a notion of *collision resistance*.[32] For instance, in this particular case, we can use a hash function with $m = 2$ and $\epsilon = \frac{1}{4}$ as follows:

1. We use a public random string r of length 2^{n+1}, which we think of as divided into 2^n 2-bit strings $r_1 \cdots r_{2^n}$.
2. Alice, on input x, uses x to index into the random string r and get the 2-bit string r_x. She sends r_x to Bob.
3. Bob checks if $r_x = r_y$, and uses this as his answer. (He sends the answer to Alice so that they both know.)

Note that only 3 bits of communication are required; furthermore, if $x = y$, then we will always have $r_x = r_y$, so the protocol will output 1, whereas if $x \neq y$, then since r_x and r_y are independently drawn from the uniform distribution on $\{0,1\}^2$, $\mathbb{P}[r_x = r_y] \leq \frac{1}{4}$.

(An alternative protocol is for Alice to multiply x with a sufficiently large r in a finite field that is large enough so that the probability that $rx = ry$ is low if $x \neq y$ is low; then Alice reduces this into a smaller finite field and sends the result to Bob.)

What about private randomness? It turns out that there is an $O(\log n)$ algorithm. First, think of the algorithm that picks a random index i, and then sends (i, x_i) to Bob; Bob responds with the answer 1 iff $x_i = y_i$. If $x = y$, clearly this will always return 1; however, if x and y differ by one bit, it will only return 0 with probability $\frac{1}{n}$. Repeating this a constant number of times makes the success probability $1 - (1 - \frac{1}{n})^k$; k must actually be a constant fraction of n in order to bring the probability above $\frac{2}{3}$, which gives us an $O(n)$ protocol. How can we do better?

[32] Formally, we actually require a *hash family*, which is a set of function h_r indexed by a random string r, with low collision probability over the choice of r. A hash family is called *universal* if the collision probability is at most $\frac{1}{2^m}$; the construction that we provide is a universal hash family for $m = 2$.

It turns out that we can, using a good *error-correcting code (ECC)*, which we saw in Section 1. In particular, Theorem 7 implies the existence of a "good" ECCs, which is a function $E : \{0,1\}^n \to \{0,1\}^m$ with $m/n \leq 10$ such that if $x, y \in \{0,1\}^n$ with $x \neq y$, $E(x)$ and $E(y)$ differ in at least $n/2$ positions. (These particular constants are picked arbitrarily.) Thus, Alice can just compute $E(x)$ and then choose a random index i and send $(i, (E(x))_i)$ (and we are guaranteed that $|E(x)| \leq 10n$, so this message has length $O(\log n)$); Bob compares $(E(x))_i$ to $(E(y))_i$ and outputs 1 iff they match. Then the definition of ECCs gives us exactly the correctness guarantee condition we want: If $x = y$, then $E(x) = E(y)$, while if $x \neq y$, then $E(x)$ and $E(y)$ are guaranteed to differ in more than $\frac{1}{2}$ of all bits, so if we randomly choose a position and compare $E(x)$ and $E(y)$ in that position, Bob will report that $x \neq y$ with probability at least $\frac{1}{2}$. (We can simply repeat this twice if we want to have a probability above $\frac{2}{3}$.)

So, to summarize, $\mathrm{CC}(\textsc{Eq}) = n + 1$; $\mathrm{CC}^{pub}(\textsc{Eq}) = 3$; and $\mathrm{CC}^{priv}(\textsc{Eq}) = O(\log n)$. This final bound also turns out to be tight. So there is a huge, and very provable, separation in complexity between the three cases!

Can we generalize these at all? It turns out that in general, for *any* function f, $\mathrm{CC}^{pub}(f) \leq \mathrm{CC}^{priv}(f) + O(\log n)$ [New91]. However, this doesn't follow from the type of argument we used for Equality (i.e., the idea to use ECCs is very specialized for computing equality). Madhu says that the argument is similar to the proof that $\mathbf{BPP} \subset \mathbf{P/poly}$ proof (but it came well after).

Also, in general, $\mathrm{CC}(f) \leq 2^{\mathrm{CC}^{priv}(f)}$. An intuitive argument for this is that we can convert a private-randomized protocol $\pi(x, y, r_x, r_y)$ to a deterministic protocol $\pi'(x, y)$ as follows: First, Alice looks at all possible settings of r_x, and sends the *distribution* over messages she would have sent (i.e., the frequencies of each of the messages with respect to choices of r_x); Bob responds by looking at possibilities for r_y and sending the distribution over replies he would have sent; and they continue in this fashion until they have a distribution over the final answer, in which case they can simply take the majority (since the final answer is guaranteed to occur for a majority of settings of the randomness).

Now, let's briefly discuss applications of these bounds. One consequence is that a single-tape Turing machine needs $\Omega(n^2)$ time to compute Palindrome. Intuitively, the argument is that if we divide the tape into thirds, to compute Palindrome, we must compute whether the first third is equal to the last third (reversed), but information can only pass from the first third to the last third when the machine traverses the middle third. We know that since the first third and last third have $n/3$ bits, at least $n/3$ bits must be exchanged, but the Turing machine must pass over the middle third (which has length $n/3$) to exchange each bit, resulting in a total of at least $\Omega(n^2)$ steps. This can be made rigorous, but the basic concept is that we can break down a computational process into a communication process between two subparts, and then apply a communication lower bound. This allows us to prove lower bounds on things that have nothing to do with communication *a priori*. For example, there are many well-known applications in chip layout problems (proving that a chip, which is a 2D embedding of a circuit, computing a given function must have at least some area) and in proving circuit lower bounds (for restricted circuit models).

Finally, we can briefly discuss the Disjointness problem. First off, $\mathrm{CC}(DISJ) = n + 1$; it is easy to show a cn lower bound for some constant c, but to prove the $n + 1$ lower bound, we must use something called the *fooling set method*.

What might be considering the Cook-Levin theorem for communication complexity, in terms of its fundamental importance in understanding lower bounds, is that $\mathrm{CC}^{pub}(DISJ) = \Omega(n)$. This has been proven via several different techniques (c.f. [KS92], [Raz92], and [Bar+04]), and each one of these proofs has given rise to a whole new set of methods for other problems.

10.3 Communication with more than two parties

What if we had more than two parties? There are several different communication models we could consider. Perhaps the most straightforward generalization of the communication complexity defined above is that there are k parties; party i has an input $x^{(i)}$, and the parties jointly want to compute $f(x^{(1)}, \ldots, x^{(k)})$. They

proceed in some organized sequence, and each party can select a message to broadcast to all other parties.[33]

Another very interesting model is the *number-on-forehead (NOF) model*, which is somewhat different in flavor. Instead of each party knowing their own input, each player knows all inputs *except* their own [CFL83]. (The other model introduced above is often correspondingly called the *number-in-hand (NIH)* model.) Switching to NOF gives more power to the players, since they essentially have strictly more information. (In particular, an NIH protocol can be transformed into an NOF protocol, as follows: Each NOF player is assigned bijectively to a different player, and then they simulate the NIH protocol for that player, since they know that player's input.) For instance, the NOF complexity for three players — Alice, Bob, and Carol — computing EQUALITY is 3, since Alice can broadcast whether Bob's and Carol's inputs agree and Bob can broadcast whether Alice's and Carol's inputs agree (then any of them can send the final message that is the AND of Alice's and Bob's messages). On the other hand, it can be shown that the NIH complexity is $\Omega(n)$.

Here are two interesting puzzles involving multiparty communication complexity:

1. In the two-party case, checking the equality of x and y is the same as checking whether $x \oplus y = 0$. What if we consider the same problem for three parties in the NOF model — that is, Alice, Bob, and Carol want to decide whether $x \oplus y \oplus z = 0$. It turns out that the NOF complexity of this problem is at most $O(\sqrt{n})$ — can you see why?[34]

2. This problem isn't quite a communication complexity problem, but it has an NOF "flavor". There are n parties, and party i gets a uniform bit X_i on their forehead; $X_1, \ldots, X_n$ are independent. Each party can read all other parties' bits except their own, and there is no communication between the players, but each party does get some private randomness. Each party must write down a single bit. The parties win if they all write down the same bit. What is the highest possible probability of winning?

Some final thoughts: [BNS92] gave the first nontrivial lower bound for multi-party communication complexity; they showed that the public-randomness NOF complexity for the *generalized inner product function* (there are k parties, and we want to compute the XOR of the bitwise products of their inputs, i.e., $\bigoplus_{m=1}^{n} \prod_{j=1}^{k} x_m^{(j)}$ where the j-th party has input $x^{(j)}$) is $n/2^k$. For DISJOINTNESS, the best known lower bound is $\Omega(\sqrt{n}/2^k k)$, which is quite recent and due to [She14]. Overall, we want to know lower bounds for explicit functions where the number of parties, k, exceeds even, say, $1.1 \log n$. This would have important applications in circuit complexity.

An excellent, modern reference for all of this is [RY19]. In particular, the first problem above is solved in Chapter 4 (under the name "Exactly n"), and the second is an excercise at the end of Chapter 4.

11 Yael Kalai: *The Evolution of Proofs in Computer Science* (11/22/2019)

Biography

Yael Kalai is a researcher at Microsoft Research and an adjunct professor at MIT. She started studying in Israel and got her Ph.D. at MIT, where Shafi Goldwasser was her adviser (and she was Madhu's visiting student). She worked at Georgia Tech and then at Microsoft Research. Her research is mainly on cryptography and its relation with complexity theory, and today she will talk about proofs, which is one of her favorite topics: It is "mathematically beautiful, impactful, and mind-blowing"! We will see several of her favorite open problems.

[33] One interesting generalization of this model is to consider, instead of every party's messages being sent to every other party, a particular graph where parties are vertices and allowed message-passing channels are edges. For instance, the full broadcasting model we will focus on is given by the *complete graph* (i.e.,, the graph will all possible edges) on the parties; we could instead consider the model given by the *circular graph* (where each party can only send messages to its two "neighbors"), where it might be harder for one party to disseminate information to a faraway party.

[34] *Hint:* The protocol with $O(\sqrt{n})$ complexity is based off of a combinatorial argument called *Behrend's construction* [Beh46]. The full argument is in [RY19] in Chapter 4.

11.1 Classical and zero-knowledge proofs

What is a proof? The Ancient Greeks already had the idea of using axiomatic systems, and in the early 20th century pioneers like David Hilbert developed *proof theory* — the mathematical study of proofs — which led to results like Gödel's incompleteness theorems, which we've seen in class. A *classical proof*, roughly, is a series of mathematical statements that can be verified line by line, convincing some *verifier* that a statement is true. From a computational standpoint, problems for which (short) classical proofs exist that can be (efficiently) verified form the class **NP**. We could formally define a classical proof system as follows:

Definition 47 (Classical proof system)**.** *A classical proof system is a deterministic verifier algorithm V that satisfies the following two properties:*

- *Completeness: If x is a true statement,*[35] *there exists a proof w such that $V(x, w) = 1$, and*
- *Soundness: If x is a false statement, then there is no string w such that $V(x, w) = 1$.*

We typically will also include restrictions on the runtime of V and the length of the proof x.

In the 80s, the way that computer scientists think about proofs started evolving in very interesting ways. [GMR88] introduced the notion of a *zero-knowledge proofs*, which are *proofs that give no information about a theorem aside from that it is true.* For instance, if Bob sees a correct zero-knowledge proof that a theorem x is true, he should be convinced that x is true, but he should not be able to convince Carol of this fact.

From a classical standpoint, this might seem paradoxical: If Alice gives Bob a classical proof for x, of course he can just repeat the proof to Carol. So perhaps the easiest way to understand a zero-knowledge proof is by way of a physical example. Let's say that Alice wants to prove to Bob that she can distinguish two objects, a and b, without revealing to him any details about why they're different. Bob can flip a coin; if it's heads, he gives Alice (a, b), while if it's tails, he gives her (b, a). Then he asks her which one he gave her first. If Alice can actually tell the difference, then she'll always be able to know which one he gave her; otherwise, she can be right as most half of the time. So if Bob repeats the experiment many times, and Alice successfully distinguishes them each time, he can become convinced that Alice knows how to distinguish them, without learning anything about how Alice can distinguish them!

There's a common theme to what we'll do today: When something is impossible, *change the model.* We no longer just consider classical proofs in the sense of **NP**. In particular, there are two critical changes:

1. We no longer stipulate that the verifying algorithm is deterministic.
2. The proof is *interactive*, in the sense that the verifier gets to ask the prover questions.

More formally, we have the following definition:

Definition 48 (Interactive proofs)**.** *An interactive proof system involves a randomized verifier V, along with a prover P. With n rounds, the system proceeds as follows on input x:*

1. *In each round i:*
   (a) *V generates a query q_i based on x, r, and $w_1, \ldots, w_{i-1}$ (i.e., just x and r if $i = 1$).*
   (b) *P generates a response w_i based on q_i.*
2. *At the end, V decides whether to accept the responses $w_1, \ldots, w_n$ as a proof for x.*

The following correctness properties must be satisfied:

1. *Completeness: If x is a true statement, then V accepts P's responses with probability 1 over r,*[36] *and*

[35] Formally, we think of "true statements" as a *language* $\mathcal{L} \subset \{0,1\}^*$. To be more concrete, we can replace conditions such as "if x is a true statement" with "if $x \in \mathcal{L}$". Equivalently, true statements are statements that $f(x) = 1$ for some function $f : \{0,1\} \to \{0,1\}$.

[36] It is a nontrivial result that this is equivalent to requiring correctness probability only $\frac{2}{3}$.

2. Soundness: If x is a false statement, then V accepts P's responses with probability at most $\frac{1}{3}$ over r.

In other words, P should be always able to convince V of true statements, but rarely be able to convince V of false statements. Having more rounds (i.e., higher n) is reasonably conjectured to allow a larger class of statements to be proven; if V can ask more questions, then it might be harder for P to fool V. In particular, it is a classic result of [Lun+90; Sha90] that **IP**, the class of problems with interactive proofs with a polynomial number of rounds,[37] is equal to **PSPACE**; while the class of with interactive proofs with a constant number of rounds is equivalent to a class called **AM** which is conjectured to be equal to **NP**.

Again, typically we might want to make restrictions based on the runtimes of V (and sometimes P as well). We can interpret this definition as follows: The prover P is an untrustworthy adversary that has more resources than the verifier. If the verifier V follows a valid interactive proof system, then the prover can lie to the verifier that a statement is false, and the verifier is likely to catch the prover in a lie that a statement is true. Observe that standard techniques we've seen in class can be used to amplify this soundness probability to be exponentially small.[38]

11.2 Multi-party interactive proofs and probabalistically checkable proofs

[Sha90] showed that any statement that has a classical proof can also be converted into a zero-knowledge, interactive proof, assuming that *one-way functions* exist.[39] On the other hand, there are statements that have efficient interactive proofs while no efficient classical proofs are known! A classic example is whether the white or black player, in a given chess situation, has a *winning strategy* within t moves; that is, are they guaranteed that, no matter what the opponent does, they can win? No classical proofs for this problem are known (i.e., this problem is not known to be in **NP**); however, it has a succinct interactive proof, since **IP** = **PSPACE** and it is possible to enumerate over t moves of chess in poly(t) time. More broadly, **IP** = **PSPACE** is one piece of a link between the *space* required for a computation and the *communication complexity* required to prove it.

Roughly, there are two types of zero-knowledge proofs, *statistical* zero-knowledge proofs and *computational* zero-knowledge proofs. This corresponds to whether it is *statistically impossible* to learn anything based on the proof, or just difficult for any computationally bounded adversary to learn anything, respectively (this is discussed in a bit more detail in the footnote above). These two distinct notions of security give rise to two classes, **SZK** and **CZK**. Note that **SZK** $\subset$ **CZK**, since it is easier to fool computationally-bounded adversaries than arbitrarily powerful adversaries. An implication of the [Sha90] result shows that **NP** $\subset$ **CZK**, or in other words, that problems with classical proofs also have zero-knowledge interactive proofs. However, proving this inclusion requires *conditioning* on a hardness conjecture (in particular, the existence of one-way functions). What about **SZK** (i.e., "unconditional zero proofs"). Unfortunately, [For89] shows that **NP** $\not\subset$ **SZK** unless the polynomial hierarchy collapses.

So what do we do if we want unconditional results? Change the model again! We introduce the notion of *multi-prover interactive proofs*. In a typical multi-prover interactive proof system, V gets to ask questions to two non-communicating provers P_1 and P_2 (in particular, each prover doesn't know the question that V asked the other prover), and based on their responses, V gets to decide whether the statement is true or

[37] These are proofs of statements of the form $f(x) = 1$, and they require $p(n)$ rounds of interaction, where p is a polynomial and $n = |x|$ is the length of a particular instance of the problem f.

[38] If we want our interactive proof system to be zero-knowledge, we must add a third condition, the *zero-knowledge property*. This is difficult to define formally, but intuitively, it just says that V should be able to "simulate the conversation with P". In other words, V should be able to generate a query and know, *before sending the query to* P, the answer that P could give that would convince V that the statement is true. Thus, the proof is solely on the basis of P's ability to answer V's query without the additional knowledge of what answer V expects. There are actually two distinct types of zero-knowledge proof guarantees, depending on whether we want V's simulation of a conversation with P to be *statistically indistinguishable* from a real conversation with P, or whether we just need the simulation to be *computationally indistinguishable* from reality (i.e., no polynomial-time algorithm can distinguish with nonnegligible success probability).

[39] These are essentially functions that can be efficiently computed, but are impossible to invert efficiently. Their existence is equivalent to the existence of good pseudorandom generators, or the existence of a secure private key encryption scheme with message lengths longer than key lengths.

not. The completeness and soundness conditions are similar to the above: P_1 and P_2 must always be able to convince V of true statements, but it should be difficult for them to convince V of a false statement.

Why is this condition of "non-communication" so important? If we allowed them to interact, they might be able to "collude" to fool V. Erecting this barrier between them means that soundness is *easier* to achieve, since it is *harder* for the provers to fool V. Think about interrogating suspects as you might see on a police procedural. You would rather interrogate two suspects independently then just have a single suspect, because you can ask them questions independently and see if their answers line up! If their stories agree, you can be more convinced then you'd allow yourself to be if there was only a single suspect. This intuition explains why every problem that has a single-prover interactive proof also has a multi-prover interactive proof. And furthermore, it turns out that all statements with classical proofs also have zero-knowledge multi-prover proofs, unconditioned on a hardness result [Ben+88].

One fascinating property of the multi-prover case is that having more provers and more questions does *not* let you prove more types of statements. More specifically, multi-prover systems with an arbitrary (polynomial) number of provers, each an arbitrary (polynomial) number of questions that can be asked sequentially are equivalent to systems with two provers, each of which is asked one question in parallel (i.e., not depending on the other answer)! Here's a rough sketch: Replace asking the same prover many questions with asking more provers each one question. Then this can be replaced by two provers; ask one prover "What were all of the questions I would have asked an their answers?", and ask the other, "What is the answer to this specific question?". The original paper that made this argument [FRS94] claimed that this protocol could be repeated many times *in parallel*[40] to make the soundness error at most $\frac{1}{3}$ using a standard amplification argument, but this ended up being incorrect! The error was eventually bounded correctly by Ran Raz using his famous *parallel repetition theorem* [Raz95].

A seemingly unrelated notion is that of the *probabalistically checkable proof (PCP)* [FRS94], which is, roughly, a (non-interactive) proof that can convince a verifier that only sees a small subset of its bits. We can sketch the existance of such proofs: [BFL91] showed that any classical proof can be made into an *exponentially shorter* two-prover interactive proof. In other words, a verifier can be convinced by only asking two polylog-length questions with polylog-length answers. This protocol can be made into a non-interactive protocol if the two provers just write down all their answers in a long string ahead of time; then the verifier can also be convinced by randomly picking two polylog-sized substrings! This result implies that any classical proof of length t can be replaced by a probabalistically checkable proof of length $\text{poly}(t)$, in which the verifier needs to only see a $\text{polylog}(t)$-sized substring to be convinced of the proof. And observe, crucially, that PCPs are non-interactive (i.e., they don't involve a prover at all).

A series of works culminated in [Aro+98], which showed the famous *PCP theorem* — all classical proofs can be converted to a probabalistically checkable proof of almost the same length that can be verified by reading a *constant*-sized substring. For more information, see Madhu's notes on PCPs in [Sud00] (as well as the preceding chapters on interactive and zero-knowledge proofs).

11.3 Delegating computation

These notions are of central importance in secure cloud computing and in blockchain techniques. The general goal is to be able to achieve some form of *delegated computation*: A weak device, such as a smartphone, wants to do a computation (say, $f(x)$) but doesn't have the computational power. Thus, the weak device requests that a more powerful cloud server performs the computation. However, the cloud isn't trustworthy, so it must prove to the client that its computation is correct, in addition to carrying out the computation. In such a system, then, there are two things we want:

1. The verifier's job should be easy (less work than computing $f(x)$ directly), and
2. The prover shouldn't have to do too much extra work to prove the correctness of its response.

For instance, in blockchains, we want *succinct proofs* that transactions are valid; this is a form of delegated computation, because a verifier should be convinced that all transactions are valid without having to simulate

[40] This means that only one question is asked to each prover, instead of a sequence of questions being asked to each prover.

all of the transactions themselves. Furthermore, these proofs should be non-interactive; they should just be text strings that sit on the chain.

Are any of the types of proofs we've seen so far sufficient for these purposes? Classical proofs, by themselves, are *too long* (the classical proof that a computation is correct is just a transcript of the algorithm). Interactive proofs don't require as much space, but can require high runtime for the prover. It's not clear how multi-prover interactive proofs would be applicable — would you use two non-communicating cloud servers? And, while PCPs are good in the sense that the verifier doesn't actually have to check many bits, they are, by themselves, too long — they're longer than the corresponding classical proof, so it's not reasonable for the prover to communicate them. The critical idea that we will see later is for a cloud server to be able to *commit* to a PCP without actually having to send the whole PCP; the prover sends the commitment to the verifier, and then the verifier just asks for certain desired bits and uses the commitment to ensure that they indeed belong to a single PCP committed to by the verifier.

So we arrive at a central question: Is *proving correctness of a computation* harder than *performing the computation*? In particular, let's say we had a program that runs in time t and space s. The verifier can't run the program, but wants to delegate it to the cloud server; the cloud server must only have to do $\text{poly}(t)$ additional work to prove correctness to the verifier, and the verifier should only require $\text{poly}(s)$ work to check that the proof is correct.

Such proofs are called *doubly efficient interactive proofs*. [GKR08] introduced this notion and showed the existence of doubly efficient interactive proofs for *depth-bounded computations* (more specifically, the communication complexity grows with depth, so such proofs are only useful for computations with small-depth circuits). The depth of a computation can intuitively be thought of as a measure of the difficulty of parallelizing a computation (the number of sequential operations that are inherently necessary to perform the computation).

[RRR16] showed the existence of doubly efficient interactive proofs for space-bounded computations, where the communication complexity grows polynomially with space s; the prover's time grows with t^ϵ for some small ϵ, and the verifier's runtime is polynomial in s and t^ϵ. This is often fine for problems that are in $\mathbf{P}$ if ϵ is sufficiently small, since it is only a small polynomial factor that might be dominated by a power of s; however, if the problem is more difficult, then the verifier's runtime might become prohibitively large.

Nobody knows how to get rid of this dependence on t in the verifier's runtime. So, what do we do? Again, we change the model! [Kil92] and [Mic94] showed a succinct delegation scheme for all functions, with one caveat: They relax soundness to hold against only *polynomial-time adversarial provers*, as opposed to arbitrarily powerful provers. Such proofs are called *computationally-sound interactive proofs*, as well as *arguments* (in the sense of "quasi-proofs").

In particular, their insight was to use *succinct commitments* to a PCP. That is, the cloud server computes the function, and then builds a PCP that their proof is correct (which has length $\text{poly}(t)$). However, they don't send the entire PCP, which would be too long. Instead, they only send a *commitment* to the PCP; then the verifier can query particular bits of the PCP, and use the commitment to check that the bits that they receieve were actually the bits of a single PCP (i.e., the server isn't just adaptively coming up with new bits to fool the verifier).

What does such a commitment scheme look like? A first guess might be to send a hash of the PCP; while this does commit to the PCP (the server cannot change its mind about the PCP), it is not a succinct commitment, since in order to actually check the hash, the verifier must see the entire PCP! What we want is a hash that can be checked *locally*, in the sense that we can check whether a "piece of the commitment" matches a "piece of the input" without seeing the entire input. One common implementation for this is a *hash tree*, which is depicted in Figure 14. Finally, note that the security of such a scheme relies on the computational boundedness assumption about the server; that is, the server isn't powerful enough to cheat by actually finding collisions for the hash. Unfortunately, the assumption that hash functions with such guarantees (i.e., inability to find collisions in polynomial time) exist is a stronger assumption than the existence of one-way functions, which itself is stronger than $\mathbf{P} \neq \mathbf{NP}$, so we are a long way off from unconditional proofs of security!

However, all of the above schemes still require interactivity; recall, we said that for applications such

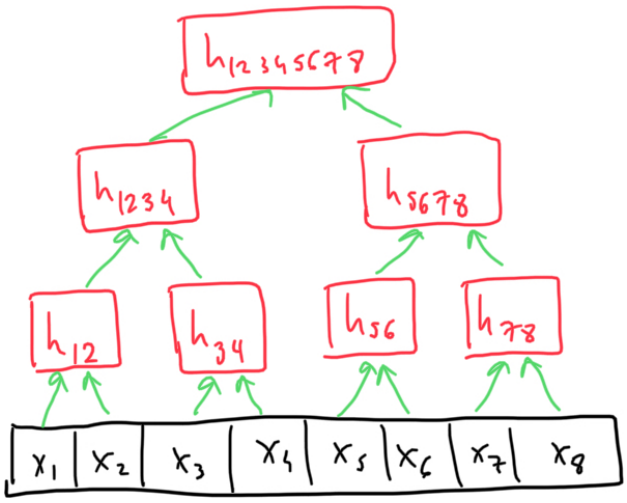

Figure 14: A hash tree for a message x divided into eight blocks $x_1, \ldots, x_8$. Instead of simply hashing the entire string x, the prover recursively calculates hashes. First, it calculates the four hashes $h_{12} = h(x_1, x_2)$, $h_{34} = h(x_3, x_4)$, etc. Then, it combines these hashes four into a pair of hashes: $h_{1234} = h(h_{12}, h_{34})$ and $h_{5678} = h(h_{56}, h_{78})$. Finally, it computes $h_{12345678} = h(h_{1234}, h_{5678})$. This final hash is its commitment, which it sends to the verifier. Then when the verifier wants to check that the prover has committed to, say, x_2, the prover needs only send (x_1, h_{34}, h_{5678}), as opposed to the entire message x. Using these values, the verifier can compute h_{12}, h_{1234}, and $h_{12345678}$; then, the verifier can check this final value against the commitment that was sent by the prover to verify that the prover did indeed commit to x_2. If the hashes do match but the prover didn't commit to x_2, then we have a hash collision; however, we assumed it is hard to find such collisions, so this should happen rarely. In the general case (where there are 2^k blocks in the original string), the verifier can check the commitment by seeing and calculating $\Theta(k)$ hashes, as opposed to having to see the entire string (of length 2^k) that the prover committed to.

as the blockchain, we would like the proof system to be non-interactive. Such systems are called *succinct non-interactive arguments (SNARGs)*. But under standard complexity assumptions (i.e., that $\mathbf{P} \not\subset \mathbf{NTIME}(\alpha(n))$ for $\alpha(n) < n$), this is naïvely impossible! We again must change the model slightly: We introduce a *common random string (CRS)*, which is a random string that is generated by a trusted third party; assuming that neither the prover nor the verifier could influence the choice of the string, soundness and completeness should hold.

How can we construct SNARGs? There is a well-known heuristic called the *Fiat-Shamir paradigm* that eliminates interaction from interactive schemes. While it is used in practice, there are many ways to construct counterexamples (i.e., sound interactive proof systems become non-sound non-interactive systems), the first of which was noticed by our very own Boaz [Bar01]!

An alternative strategy is to try and convert a multi-prover proof into a non-interactive delegation scheme. First off, [BMW99] gave a general technique to accomplish this; it relied on the notion of *fully homomorphic encryption* (FHE), which allows for a party to efficiently execute arbitrary nand circuits on encrypted data without revealing any information about the data; FHE was first constructed in [Gen09]. Surprisingly, proving soundness for this type of scheme turns out to require ideas from quantum physics [KRR13]! Such schemes can also furthermore be made publicly verifiable (so that anyone can be convinced of the correctness of the delegated computation, not just the original requester) [KPY19].

In conclusion, we have developed succinct delegations schemes for **P** problems (where the prover can just do the computation) — so "proving" is not harder than "solving" for **P** problems But what about succinct proofs for **NP** problems? Is giving proofs of correct computations for **NP** problems harder than solving them? This is an open question!

A variation on this talk was presented in Yael's blog post [Kal12].

References

[AB12] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge, United Kingdom: Cambridge University Press, 2012.

[Aro+98] Sanjeev Arora et al. "Proof verification and the hardness of approximation problems". In: *Journal of the ACM* 45.3 (May 1998), pp. 501–555.

[Bar+04] Ziv Bar-Yossef et al. "Information statistics approach to data stream and communication complexity". In: *Journal of Computer and System Sciences* 68 (2004), pp. 702–732.

[Bar01] Boaz Barak. "How to go beyond the black-box simulation barrier". In: *Proceedings of the IEEE Symposium on Foundations of Computer Science* 42 (Oct. 2001), pp. 106–115.

[Bar19] Boaz Barak. *Introduction to Theoretical Computer Science*. 2019. URL: https://introtcs.org/public/index.html.

[Beh46] Felix Behrend. "On the sets of integers which contain no three in arithmetic progression". In: *Proceedings of the NAS* 23 (1946), pp. 331–332.

[Bel+19] Mikhail Belkin et al. "Reconciling modern machine-learning practice and the classical bias–variance trade-off". In: *Proceedings of the NAS* 116.32 (July 2019), pp. 15849–15854.

[Ben+88] Michael Ben-Or et al. "Multi-Prover Interactive Proofs: How to Remove Intractability Assumptions". In: *Proceedings of the ACM Symposium on Theory of Computing* 20 (May 1988), pp. 113–131.

[BFL91] Lázló Babai, Lance Fortnow, and Carsten Lund. "Non-deterministic exponential time has two-prover interactive protocols". In: *Computational Complexity* 1 (1 1991), pp. 3–40.

[BMW99] Ingrid Biehl, Bernd Meyer, and Susanne Wetzel. "Ensuring the integrity of agent-based computations by short proofs". In: *Proceedings of the International Workshop on Mobile Agents* 2 (1999), pp. 183–194.

[BNS92] László Babai, Noam Nisant, and Márió Szegedy. "Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs". In: *Journal of Computer and System Sciences* 45 (2 Oct. 1992), pp. 204–232.

[Cab04] Lawrence Cabusora. "Diophantine Sets, Primes, and the Resolution of Hilbert's 10th Problem". Bachelor's Thesis. Harvard University, 2004.

[CFL83] Ashok Chandra, Merrick Furst, and Richard Lipton. "Multi-party protocols". In: *Proceedings of the ACM Symposium on Theory of Computing* 15 (1983), pp. 94–99.

[Dav53] Martin Davis. "Arithmetical Problems and Recursively Enumerable Predicates". In: *The Journal of Symbolic Logic* 18.1 (Mar. 1953), pp. 33–41.

[Dem+10] E. Demenkov et al. "New upper bounds on the Boolean circuit complexity of symmetric functions". In: *Information Processing Letters* 110.7 (2010), pp. 264–267.

[DPR61] Martin Davis, Hilary Putnam, and Julia Robinson. "The Decision Problem for Exponential Diophantine Equations". In: *Annals of Mathematics* 74.3 (Nov. 1961), pp. 425–436.

[For89] Lance Fortnow. "The complexity of perfect zero-knowledge". In: *Advances in Computing Research*. Vol. 5. JAC Press, Inc., 1989, pp. 327–343.

[FRS94] Lance Fortnow, John Rompel, and Michael Sipser. "On the power of multiprover interactive protocols". In: *Theoretical Computer Science* 134 (2 Nov. 1194), pp. 545–557.

[Gen09] Craig Gentry. "A fully homomorphic encryption scheme". crypto.stanford.edu/craig. PhD thesis. Stanford University, 2009.

[GKR08] Shafi Goldwasser, Yael Kalai, and Guy Rothblum. "Delegating computation: interactive proofs for muggles". In: *Proceedings of the ACM Symposium on Theory of Computing* 40 (2008), pp. 113–122.

[GL92] Craig Gotsman and Nathan Linial. "The equivalence of two problems on the cube". In: *Journal of Combinatorial Theory, Series A* 61.1 (Sept. 1992), pp. 142–146.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ron Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks". In: *SIAM Journal of Computation* 17 (2 Oct. 1988), pp. 281–308.

[GW95] Michael Goemans and David Williamson. "Improved approximation algorithms for MaxCut and Satisfiability problems using semidefinite programming". In: *Journal of the ACM* 42.6 (Nov. 1995), pp. 1115–1145.

[Ham50] Richard Hamming. "Error Detecting and Error Correcting Codes". In: *Bell System Technical Journal* 29.2 (Apr. 1950), pp. 146–160.

[Hås01] Johan Håstad. "Some optimal inapproximability results". In: *Journal of the ACM* 48.4 (July 2001), pp. 798–859.

[Hil02] David Hilbert. "Mathematical Problems". In: *Bulletin of the American Mathematical Society* 8 (1902), pp. 437–479.

[HKP10] Pooya Hatami, Raghav Kulkarni, and Denis Pankratov. *Variations on the Sensitivity Conjecture*. arXiv:1011.0354 [cs.CC]. 2010.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and Combinators*. Cambridge, United Kingdom: Cambridge University Press, 2008.

[Hua19] Hao Huang. *Induced subgraphs of hypercubes and a proof of the Sensitivity Conjecture*. arXiv:1907.00847 [math.CO]. 2019.

[Juk12] Stasys Jukna. *Boolean function complexity: Advances and Frontiers*. Vol. 27. New York, New York: Springer Science & Business Media, 2012.

[Kal12] Yael Kalai. *The Evolution of Proofs*. Sept. 2012. URL: https://windowsontheory.org/2012/09/18/the-evolution-of-proofs/.

[Kho+07] Subhash Khot et al. "Optimal inapproximability results for MaxCut and other two-variable CSPs?" In: *SIAM Journal on Computing* 37 (1 2007), pp. 319–357.

[Kil92] Joe Kilian. "A note on efficient zero-knowledge proofs and arguments". In: *Proceedings of the ACM Symposium on Theory of Computing* 24 (1992), pp. 723–732.

[KPY19] Yael Tauman Kalai, Omer Paneth, and Lisa Yang. "How to delegate computations publicly". In: *Proceedings of the ACM SIGACT Symposium on Theory of Computing* 51 (2019), pp. 1115–1124.

[KRR13] Yael Tauman Kalai, Ran Raz, and Ron Rothblum. "How to Delegate Computations: The Power of No-Signaling Proofs". In: *Proceedings of the ACM Symposium on Theory of Computing* 46 (2013), pp. 485–494.

[KS92] Bala Kalyanasundaram and Georg Schnitger. "The probabilistic communication complexity of set intersection". In: *SIAM Journal on Discrete Mathematics* 5 (1992), pp. 545–557.

[Lun+90] Carsten Lund et al. "Algebraic methods for interactive proof systems". In: *Proceedings of the Symposium on Foundations of Computer Science* 31 (Oct. 1990), pp. 2–10.

[Lup58] Oleg B. Lupanov. "A Method of Circuit Synthesis". Russian. In: *Izvesitya VUZ, Radiofiz* 1 (1958), pp. 120–140.

[Mat70a] Yuri Matiyasevich. "Enumerable Sets are Diophantine". In: *Soviet Mathematics Doklady* 11 (1970), pp. 354–358.

[Mat70b] Yuri Matiyasevich. "My Collaboration with Julia Robinson". In: *Soviet Mathematics Doklady* 11 (1970), pp. 354–358.

[Mic94] Silvio Micali. "Computationally-Sound Proofs". In: *Proceedings of the IEEE Symposium on the Foundations of Computer Science* 35 (1994), pp. 436–453.

[Nak+19] Preetum Nakkiran et al. *Deep double descent: where more models and bigger data hurt*. Oct. 2019. URL: https://mltheory.org/deep.pdf.

[New91] Ivan Newman. "Private vs. common random bits in communication complexity". In: *Information Processing Letters* 39.2 (July 1991), pp. 67–71.

[NS94] Noam Nisan and Mario Szegedy. "On the degree of boolean functions as real polynomials". In: *Computational Complexity* 4.4 (Dec. 1994), pp. 301–313.

[Pas07] Grant Olney Passmore. "Diophantine Sets and their Decision Problems". Bachelor's Thesis. University of Texas, Austin, 2007.

[Pat+05] Ramamohan Paturi et al. "An Improved Exponential-Time Algorithm for k-SAT". In: *Journal of the ACM* 52.3 (May 2005), pp. 337–364.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Massachusetts: MIT Press, 2002.

[Raz92] Alexander Razborov. "The distributional complexity of disjointness". In: *Theoretical Computer Science* 106 (1992), pp. 385–390.

[Raz95] Ran Raz. "A parallel repetition theorem". In: *Proceedings of the ACM Symposium on Theory of Computing* 27 (May 1995), pp. 447–456.

[Rob52] Julia Robinson. "Existential Definability in Arithmetic". In: *Transactions of the American Mathematical Society* 72.3 (May 1952), pp. 437–449.

[RRR16] Omer Reingold, Guy Rothblum, and Ron Rothblum. "Constant-round interactive proofs for delegating computation". In: *Proceedings of the ACM Symposium on Theory of Computing* 48 (2016), pp. 49–62.

[RY19] Anup Rao and Amir Yehudayoff. *Communication Complexity*. 2019.

[SB14] Shai Shalev-Schwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, United Kingdom: Cambridge University Press, 2014.

[Sch74] Claus-Peter Schnorr. "Zwei lineare untere schranken für die komplexität boolescher funktionen (*Two linear lower bounds for the complexity of Boolean functions*)". German. In: *Computing* 13.2 (1974), pp. 155–171.

[SG76] Sartaj Sahni and Teofilo Gonzalez. "–-complete approximation problems". In: *Journal of the ACM* 23.3 (July 1976), pp. 555–565.

[Sha48] Claude Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.10 (July 1948), pp. 379–423, 623–656.

[Sha49] Claude Shannon. "The Synthesis of Two-Terminal Switching Circuits". In: *Bell System Technical Journal* 28.1 (Jan. 1949), pp. 59–98.

[Sha90] Adi Shamir. "**IP** = **PSPACE**". In: *Proceedings of the IEEE Symposium on Foundations of Computer Science* 31 (Oct. 1990), pp. 11–15.

[She14] Alexander Sherstov. "Communication Lower Bounds Using Directional Derivatives". In: *Journal of the ACM* 61 (6 2014).

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[Sud00] Madhu Sudan. "Probabalistically Checkable Proofs". In: *Computational Complexity Theory*. IAS/Park City Mathematical Series, 2000. Chap. 3.III, pp. 349–384.

[Sud08] Madhu Sudan. "Reliable Transmission of Information". Princeton Companion to Mathematics. 2008. URL: http://madhu.seas.harvard.edu/papers/2008/pcm.pdf.

[Tho68] Ken Thompson. "Regular Expression Search Algorithm". In: *Communications of the ACM* 11.6 (June 1968), pp. 419–422.

[Val84] Leslie Valiant. "A Theory of the Learnable". In: *Communications of the ACM* 27.11 (Nov. 1984), pp. 1134–1142.

[Wad15] Philip Wadler. "Proposition as Types". In: *Communications of the ACM* 58.12 (Dec. 2015), pp. 75–84.

[Wil19a] Ryan Williams. *Some Estimated Likelihoods For Computational Complexity*. 2019. URL: http://people.csail.mit.edu/rrw/likelihoods.pdf.

[Wil19b] Virginia Vassilevska Williams. *On some fine-grained questions in algorithms and complexity*. 2019. URL: https://people.csail.mit.edu/virgi/eccentri.pdf.

[Yao81] Andrew Yao. "The entropic limitations on VLSI computations". In: *Proceedings of the ACM Symposium on Theory of Computing* 13 (May 1981), pp. 308–311.