1 Zooming out to try other algorithmic approaches

As we mentioned in the beginning of the course, the problem of proving lower bounds is *inherently* an algorithmic problem, since we want to prove that *explicit* functions are hard. That is, recall that we already know that a random function is hard; the main challenge is in finding an *upper bound*, i.e. designing an algorithm.

Definition 1. *For a set $S \subseteq \{0,1\}^*$, the* search problem of *S is the problem in which the input is 1^N and the output may be any $s \in S_N = S \cap \{0,1\}^N$, or $\perp$ if $S_N = \emptyset$. We will mostly be concerned with S's that are dense, e.g. $|S_N| \geq 1/2 \cdot 2^N$ for all $N \in \mathbb{N}$.*

Indeed, this definition focuses on explicit construction problems, in which the input is 1^N. A more general definition allows for input-output pairs, a-la (x, s) where $s \in S_x$ is a solution for x, but we will not need this more general form in this class.

Definition 2. *For $s\colon \mathbb{N} \to \mathbb{N}$ and any $N = 2^n$, we define $TT^{(s)}_N$ as the set of N-bit truth-tables with circuit complexity at least $s(n)$. (On other N's the problem is defined trivially.)*

How is the search problem of TT^s related to proving lower bounds?

Theorem 3. *The problem $TT^{(s)}$ can be solved in deterministic polynomial time if and only if $\mathcal{E} \not\subset \mathtt{i.o.}\mathcal{SIZE}[s]$ for some $\epsilon > 0$.*

Proof. If $TT^{(s)} \in \mathcal{FP}$, via some polynomial-time algorithm A, define

$$x \in L_n \iff A(1^N)_x = 1 \,.$$

Since A runs in time $\mathrm{poly}(N) = 2^{O(n)}$, we have $L \in \mathcal{E}$. Also, the truth-table of L_n is $A(1^N)$, so the circuit complexity of L_n is at least $s(n)$.

For the other direction, given 1^N where $N = 2^n$, run the time-$2^{O(n)}$ algorithm A for L on all n-bit inputs to produce the truth-table of L_n, which has circuit complexity $s(n)$. This runs in $\mathrm{poly}(N)$ time, since $2^n \cdot 2^{O(n)} = \mathrm{poly}(N)$. ∎

So far we proved lower bounds via algorithms using indirect ways, e.g. an algorithm for CAPP implies circuit lower bounds using a complicated argument. From now on we will be more direct, and try to directly solve the search problem $TT^{(s)}$.

2 A derandomization problem

We can think of $TT^{(s)}$ as a derandomization problem, for the following reason:

Observation 4. *For any $s(n) \ll 2^n/n$, the problem $TT^{(s)}$ can be solved in probabilistic polynomial time.*

The reason is that most strings are hard, so choosing a random string solves the problem. This should not be enough to convince you that the problem is also solvable deterministically. For example, the same argument applies to finding a string with high Kolmogorov complexity, but this task can't be solved deterministically (in any runtime). More generally, to have a meaningful notion of search problems that are solvable probabilistically and apply derandomization approaches to this notion, we must consider the complexity of *deciding* the set S (see [Tel24; Gol25])

When this happens, we can rely on an efficient deterministic search-to-decision reduction for search problems solvable in probabilistic polytime. In particular, when we can deterministically estimate the density of S in certain nice sets (e.g., subcubes), we can deterministically solve the search problem of S.

Definition 5. *Let* $Density_T(1^p, x_1, ..., x_\ell)$ *be the problem of estimating* $\Pr_{x_{\ell+1},...,x_n}[x_1, ..., x_p \in T_p]$ *up to an additive error of* $1/p^2$.

Theorem 6 (search-to-decision)**.** *Any search problem S solvable by a ppt algorithm A can be solved in deterministic polynomial time with oracle access to (a decision problem in)* $pr\mathcal{BPP}^{Density_T}$, *where* $T(r) = S(A(1^n, r))$.

Proof. The idea is to construct random coins for A that cause it to output $s \in S$, and we do so bit-by-bit. In each iteration, we approximately preserve the density of coins that cause A to output a string in S.

In more detail, denote the runtime of A by $p = p(n)$. We construct $r \in T$ bit-by-bit. In each iteration $i \in [p]$ we start with a prefix $\sigma_{i-1} \in \{0,1\}^{i-1}$ such that $\Pr_{z \in \{0,1\}^{p-(i-1)}}[\sigma_{i-1}z \in T] \geq 1/2 - (i-1)/p$. (Note that this is true at $i = 1$, since T is dense.) By this assumption, there is $b \in \{0,1\}$ such that $\Pr_{z \in \{0,1\}^{p-i}}[\sigma_{i-1}bz \in T] \geq 1/2 - (i-1)/p$.

We will find b that meets the more relaxed requirement $\Pr_{z \in \{0,1\}^{p-i}}[\sigma_{i-1}bz \in T] \geq 1/2 - i/p$. Specifically, we use the oracle to estimate $v = \Pr_{z \in \{0,1\}^{p-i}}[\sigma_{i,...,i-1}0z \in T]$ up to accuracy $1/p^2$, and if our estimate for v is more than $1/2 - (i-1)/p - 1/2p$, we extend the prefix by zero; otherwise we extend it by one. Assuming that the oracle returns an estimate that is correct up to $1/p^2$, we will extend the prefix such that the new probability is at least $1/2 - (i-1)/p - 1/2p - 1/p^2 > 1/2 - i/p$. ■

What does this tell us about $TT^{(s)}$? We can decide $TT^{(s)}$ in $co\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$, and under derandomization assumptions estimating the density of S can also be done in $\mathcal{P}^{\mathcal{NP}}$. Hence, under these assumptions, we can solve $Density_T$ for this problem in $\mathcal{P}^{\mathcal{NP}}$, and $TT^{(s)}$ is solvable in $\mathcal{FP}^{\mathcal{NP}}$. This gives circuit lower bounds in $\mathcal{E}^{\mathcal{NP}}$.

(Indeed, we hope to solve $TT^{(s)}$ in polynomial time and get lower bounds in $\mathcal{E}$, but the above gives theoretical support for, at least, $\mathcal{E}^{\mathcal{NP}}$.)

3 Pseudodeterministic explicit constructions

We will solve $TT^{(s)}$ in a slightly larger class, which we will define now, and deduce lower bounds not in $\mathcal{E}^{\mathcal{NP}}$ but in $\mathcal{ZPE}^{\mathcal{NP}} \subseteq \mathcal{BPE}^{\mathcal{NP}}$.

Let us first see an algorithm for a class of explicit construction problems that does not include $TT^{(s)}$, but it will guide us next for working with $TT^{(s)}$. The algorithm is not deterministic, and will not yield an $\mathcal{FP}^{\mathcal{NP}}$ upper-bound. Instead, it satisfied a relaxed notion, defined by Gat and Goldwasser.

Definition 7 ([GG11]). *A probabilistic algorithm A is pseudodeterministic if for every x there is v such that $\Pr_r[A(x,r) = v] \geq 2/3$.*

Note that the probability bound is arbitrary, since we can reduce the error. The point of the definition is that for an external user or set of users, this algorithm is as good as a deterministic algorithm, since with all but tiny probability, the output is always the same. The internal mechanisms used to produce this output are randomized, but from outside, the output looks deterministic.

As proved by Chen, Lu, Oliveira, Ren, and Santhanam, any explicit construction problem $S \in \mathcal{P}$ can be solved (i.o.) in pseudodeterministic polynomial time.

Theorem 8 ([CLO+23]). *For any dense $S \subseteq \{0,1\}^*$ such that $S \in \mathcal{P}$, there is a probabilistic polynomial-time algorithm that, infinitely often, is pseudodeterministic and solves the search problem of S.*

The hallmark example of S to which Theorem 8 applies is the set of prime numbers, and indeed this is arguably the best algorithm that we know for finding a prime (other algorithms either run in exponential time, or do not have a pseudodeterministic guarantee).[1] The result does not apply to $TT^{(s)}$ because it needs $S \in \mathcal{P}$.

3.1 Tools from hardness vs randomness

We will need several tools from hardness-vs-randomness. These are often thought of as translating hardness to randomness (i.e., building PRGs from hard functions), but we will think of them in a different way, which will be explained below.

Theorem 9 ([CT21; CLO+23]). *There is a pair of algorithms G, R such that for every function $f\colon \{0,1\}^* \to \{0,1\}^*$ computable by logspace-uniform circuits of size T and depth d, and every nice $m(n) \leq n$, and every fixed $x \in \{0,1\}^n$:*

- **Generator:** *$G = G_f(x)$ outputs $\mathrm{poly}(T)$ strings of length m.*

[1] The set of primes has density $1/n$ rather than $1/2$. The result of [CLO+23] holds also with this density, and also more generally, we can error-reduce in this setting (i.e., by repeating a search algorithm with success ϵ for $O(1/\epsilon)$ times and checking each outcome, to obtain an algorithm with success $1/2$; and then defining the search problem as the problem of finding good coins for the latter algorithm).

- **Reconstruction:** *$R = R_f$ gets input $x \in \{0,1\}^n$ and oracle access to $D\colon \{0,1\}^m \to \{0,1\}$, and runs in time $\mathrm{poly}(n, m, \log T)$. If $\Pr_{r\in\{0,1\}^m}[D(r) = 1] \geq 1/2$ but D rejects all strings in $G(x)$, then $\Pr_r[R^D(x,r) = f(x)] \geq 2/3$.*

In hardness-vs-randomness, the generator is often thought of as the "real" algorithm, and the reduction is part of the analysis (i.e., a security reduction). But it's also useful to think of them as two "real" algorithms of equal status, where for every x at least one of them works; that is, for every x, either $f(x)$ is easy to compute, or $G_f(x)$ is pseudorandom (or both).

In the current lecture, we'll only think of the special case $x = 1^n$, and of D that is an efficient deterministic algorithm, in fact $D = S$ (the search problem). In this case the reconstruction R is a pseudodeterministic algorithm, i.e. $R(x) = f(x)$ whp.

Proof Sketch for Theorem 9. We'll need the Nisan-Wigderson generator.

Definition 10 (distinguisher)**.** *An ϵ-distinguisher for a distribution $\mathbf{w}$ on $\{0,1\}^n$ is a function $D\colon \{0,1\}^n \to \{0,1\}$ such that $\left|\Pr_{r\in\{0,1\}^n}[D(r) = 1] - \Pr_{w\sim\mathbf{w}}[D(w) = 1]\right| \geq \epsilon$.*

Theorem 11 ([NW94; STV01])**.** *There is a pair of algorithms $G_{\mathsf{NW}}, R_{\mathsf{NW}}$ such that for any nice $m(n) \leq n$ and any $f \in \{0,1\}^n$ and any good enough locally list-decodable code ECC:*

- **Generator:** *$G_{\mathsf{NW}}(f)$ runs in time $\mathrm{poly}(n)$ and outputs a list of m-bit strings.*
- **Reconstruction:** *R_{NW} gets input $1^{\log n}$ and oracle access to $D\colon \{0,1\}^m \to \{0,1\}$ and to $\mathsf{ECC}(f)$ and runs in time $\mathrm{poly}(\log n, m)$. If D is a $(1/m)$-distinguisher for $G_{\mathsf{NW}}(f)$, then wp at least $2/3$, $R_{\mathsf{NW}}^{D,ECC(f)}(1^{\log n})$ outputs a circuit $C\colon \{0,1\}^{\log n} \to \{0,1\}$ such that the truth-table of C^D is f.*

As observed in [IW98], the reconstruction is similar to an algorithm that learns f from membership queries (whenever it has access to a distinguisher), where the main difference is that queries are to $\mathsf{ECC}(f)$ rather than to f.

For simplicity, let's first ignore the difference between querying $\mathsf{ECC}(f)$ and querying f, and let's also assume that the circuit satisfies a strong uniformity condition (given the index of a gate, we can quickly compute the indices of its children). Consider the computational history of $f(x)$ as a circuit of depth d, and use feed each of d layers $f_1, ..., f_d \in \{0,1\}^T$ into the NW generator; define $G(x) = \cup_{i\in[d]} G_{\mathsf{NW}}(f_i)$. If D is a distinguisher for $G(x)$, then the reconstruction R acts as follows: For $i = 1, ..., d$, it starts with a circuit C_{i-1} of size $\mathrm{poly}(n, m, \log T)$ whose truth-table is f_{i-1} (the base case is trivial, because R can construct such a circuit from x), and it uses R_{NW} to construct a circuit C_i of size $\mathrm{poly}(n, m, \log T)$ whose truth-table is f_i. The circuit requires queries to f_i, but we can provide them using the fact that f_i reduces to f_{i-1} (i.e., each gate in layer i depends on two gates in layer $i-1$, whose indices are easily computable) and our circuit C_{i-1}.[2] At the end we have C_d whose truth-table is the output layer, so we can compute $f(x)$. Our total runtime is proportional to $d \cdot \mathrm{poly}(n, m, \log T) \ll T$.

[2]Note that the circuit C_i does not contain C_{i-1} as a sub-component (i.e., C_{i-1} is only used to answer the queries of R_{NW} when constructing C_i), and hence there is no blow-up in circuit size across iteration.

The main complication is that R_{NW} requires queries to $\mathsf{ECC}(f)$ rather than to f. If we just naively encode each layer via some good code, we lose the "easy downward self-reducibility" structure.[3] Getting around this can be done by encoding the circuit's gate values in a careful way, following ideas from [GKR15]. ■

3.2 Proof of Theorem 8

The main idea is the following. Fix f such that $f(1^n)$ outputs a prime number. Consider the distinguisher D "is this a prime number?", which is in a polynomial-time algorithm. So either G_f fools D, i.e. hits the set of prime numbers, or the reconstruction $R_f(1^n)$ outputs a prime number. In both cases, we get a prime number! Either we find it deterministically (if the generator works) or we find it pseudodeterministically (if the reconstruction works). The point is that both G_f and f work towards the same goal – finding a prime number – and at least one of them is guaranteed to work.

Details. The parameters are subtle, though. Let's work out a parameter setting that gives something non-trivial, and continue from there.

Iteration 0. Consider $f(1^n)$ that brute-forces over all n-bit numbers to find a prime, in time $2^n \cdot \mathrm{poly}(n)$. Note that we can compute f by circuits of depth $\mathrm{poly}(n)$, since we check all possibilities in parallel. For simplicity, we assume the circuits are of size exactly 2^n and depth exactly n.[4]

Instantiate Theorem 9 with f and output length $m = n^C$ for a big constant $C > 1$.

- If $G(1^n)$ outputs a list that contains a prime number, then we found a prime of length n^C in time $\mathrm{poly}(2^n)$. This is an improvement over the brute-force algorithm that takes time $\tilde{O}(2^{n^C})$. For simplicity again, let's assume that the runtime is exactly 2^n (i.e., let's ignore the poly time overhead of the PRG).[5]
- Otherwise, D has high acceptance probability but rejects all strings in $G_f(1^n)$. Hence, R^D finds an n-bit prime in pseudodeterministic time $\mathrm{poly}(m) = \mathrm{poly}(n)$.

The second case is clearly an immediate win. The first case is partial progress, but still far from polynomial-time. How shall we exploit this?

Iteration 1. Denote the previous input length by $n_0 = n$, and let us now consider input length $n_1 = m = n_0^C$. Define f on input length n_1 such that:

> Given 1^{n_1}, compute $G(1^{n_0})$, and output the first prime number in the list that G prints.

[3] Recall that in any code with constant relative distance, most output coordinates depend on at least $\Omega(1)$ coordinates of the message.

[4] We also ignore the "logspace-uniform" requirement throughout the presentation, and pretend that any uniform circuit will do.

[5] All the overheads that we ignore are taken care of by choosing C to be large enough in each iteration.

With some work, $f(1^{n_1})$ can be computed by circuits of size $\text{poly}(2^{n_0})$ and depth $\text{poly}(n_0)$. (This is because the PRG in Theorem 9 is depth-efficient.) For simplicity, we will again pretend that the size is 2^{n_0} and the depth is n_0.

We instantiate Theorem 9 on input 1^{n_1} with f and output length n_1^C.

- If $G(1^{n_1})$ outputs a list that contains a prime number, then we found a prime of length $n_2 = n_1^C$ in time $2^{n_0} = 2^{n_1^{1/C}} = 2^{n_2^{1/C^2}}$, improving on the brute-force $\tilde{O}(2^{n_2})$ by a bigger factor than in the first iteration.
- Otherwise, D is a distinguisher, and we found an n_1-bit prime in pseudodeterministic time $\text{poly}(n_1)$.

Iteration i. Let $n_i = n_0^{C^i}$. Either we find a prime of length $n_{i+1} = n_0^{C^{i+1}}$ in deterministic time $2^{n_0} = 2^{n_{i+1}^{1/C^{i+1}}}$, or we find an n_i-bit prime in pseudodeterministic time $\text{poly}(n_i)$.

For $i = t \stackrel{\text{def}}{=\!=} \Theta(\log(n_0)/\text{loglog}(n))$, we have $n_{i+1} = n_0^{C^i} < 2^{n_0}$, so either we find an n_{i+1}-bit prime in time $\text{poly}(N)$,[6] or we find an n_i-bit prime in time $\text{poly}(n_i)$.

Finalizing the argument. We define infinitely many disjoint intervals of the form $n_0, n_0^C, n_0^{C^2}, ..., n_0^{C^t}$. There are two possibilities:

- For infinitely many intervals, the deterministic algorithm at the last input length n_t outputs an n_t-bit prime.
- For infinitely many intervals there is some intermediate n_i such that the reconstruction algorithm pseudodeterministically outputs an n_i-bit prime.

In both cases we have a ppt algorithm that, infinitely often, is pseudodeterministic (or even deterministic) and outputs a prime in polynomial time.

Remark 12. *The description above sweeps some things under the rug:*

1. *There are polynomial overheads all around (by the PRG, by the f that search among the previous PRG's outputs, by the AKS primality testing [AKS04]); all of these are taken care of by choosing C to be big enough.*
2. *We need to ensure that f is still computable by logspace-uniform circuits after $t \approx \log(n)$ compositions (e.g., at the last iteration). This requires cumbersome low-level technical work, see [CLO+23] for details.*

[6] The hidden $\text{poly}(n_{i+1})$ factor, which we suppressed for simplicity when 2^{n_0} seemed exponential in comparison to it, now comes into play.

3.3 Possible extensions

Extending Theorem 8 by designing an algorithm that works on *all input lengths* and with $S \in \mathcal{BPP}$ rather than $S \in \mathcal{P}$ would imply a $\mathcal{BPP}$ time hierarchy, i.e. $\mathcal{BPTIME}[t^{O(1)}] \not\subset \mathcal{BPTIME}[t]$ for every polynomial t. In fact, even extending this to an algorithm that works on all input lengths with $S \in \mathcal{P}$ would imply $\mathcal{BPTIME}[t^{O(1)}] \not\subset \mathcal{RTIME}[t]$ for every polynomial t. Such a hierarchy-type result is also not currently known.

In fact, even extending Theorem 8 to an algorithm that is always pseudodeterministic and still only finds a solution infinitely often (i.e., on other input lengths it pseudodeterministically outputs junk) yields the same conclusion, as long as we can easily recognize intervals of input lengths on which the algorithm works (as in [CLO+23]).

Thus, the main current bottleneck for extension seems to be getting an *algorithm that is pseudodeterministic on all input lengths*. See [Tel24; Gol25] for details.

References

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES is in P". In: *Annals of Mathematics. Second Series* 160.2 (2004), pp. 781–793.

[CLO+23] Lijie Chen, Zhenjian Lu, Igor Carboni Oliveira, Hanlin Ren, and Rahul Santhanam. "Polynomial-Time Pseudodeterministic Construction of Primes". In: *arXiv preprint arXiv:2305.15140* (2023).

[CT21] Lijie Chen and Roei Tell. "Hardness vs Randomness, Revised: Uniform, Non-Black-Box, and Instance-Wise". In: *Proc. 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2021, pp. 125–136.

[GG11] Eran Gat and Shafi Goldwasser. "Probabilistic search algorithms with unique answers and their cryptographic applications". In: *Electronic Colloquium on Computational Complexity: ECCC* 18 (2011), p. 136.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. "Delegating computation: interactive proofs for muggles". In: *Journal of the ACM* 62.4 (2015), 27:1–27:64.

[Gol25] Oded Goldreich. "On defining PPT-search problems". In: *Computational complexity and local algorithms*. Vol. 15700. Lecture Notes in Comput. Sci. Springer, Cham, 2025, pp. 1–21.

[IW98] Russell Impagliazzo and Avi Wigderson. "Randomness vs. Time: De-Randomization under a Uniform Assumption". In: *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 1998, pp. 734–743.

[NW94] Noam Nisan and Avi Wigderson. "Hardness vs. randomness". In: *Journal of Computer and System Sciences* 49.2 (1994), pp. 149–167.

[STV01] Madhu Sudan, Luca Trevisan, and Salil Vadhan. "Pseudorandom generators without the XOR lemma". In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 236–266.

[Tel24] Roei Tell. "On defining PPT-search problems and PPT-optimization problems". In: *Electronic Colloquium on Computational Complexity: ECCC* (2024).