Derandomization and Circuit Lower Bounds

Markus Bläser
Universität des Saarlandes

Draft—August 25, 2008 and forever

1 Introduction

Primality testing is the following problem: Given a number n in binary, decide whether n is prime. In 1977, Solovay and Strassen [SS77] proposed a new type of algorithm for testing whether a given number is a prime, the celebrated randomized Solovay-Strassen primality test. This test and similar ones proved to be very useful. This fact changed the common notion of "feasible computations" to probabilistic polynomial time algorithms with bounded error. While randomization seems to be helpful, it is an interesting question whether it is really necessary, a question which initiated the studies on derandomization. Since then, the field of randomized algorithms and of derandomization flourished, with primality being one of its key problems. "Unfortunately", Agrawal, Kayal, and Saxena [AKS] recently proved that primality can be decided in deterministic polynomial time, taking away one of the main arguments for derandomization. This raises some interesting philosophical questions. If Argawal, Kayal, and Saxena had proven their result 20 years earlier, would we ever have thought about derandomization? Would you be reading this script? Can we derandomize any given probabilistic polynomial time algorithm with bounded error probability? While I do not have an answer to the first two questions, I will try to answer the third one. The answer that we will learn in the course of this lecture is something like "Maybe, but it will be quite hard to do so". (This sounds really encouraging, doesn't it?)

1.1 Randomized Algorithms

A randomized algorithm has the ability to flip a fair coin in every step. With probability 1/2, the outcome is 1 and otherwise it is 0. Randomized algorithm are used widely; the question that we want to answer is whether randomization *really* helps. Is there a problem that can be solved by a randomized algorithm with polynomial running time but not by a deterministic one with polynomial time?

We model randomized algorithms by probabilistic Turing machines. *Probabilistic Turing machines* have an additional *random tape*. On this tape, the Turing machine gets a one-sided infinite bit string y. The random tape is read-only and one-way.

Right at the moment, we are considering the random string y as an additional input. The name random string is justified by the following definition: A probabilistic Turing machine accepts an input x with *acceptance probabil-*

ity at least p if $\Pr[M(x,y)=1] \geq p$. Here the probability is taken over all choices of y. We define the *rejection probability* in the same way. The running time $t(n)$ of a probabilistic Turing machine M is the maximum number of steps, M performs on any input of length n and and any random string y. Note that if $t(n)$ is bounded, then we can consider y to be a finite string of length $t(n)$. The maximum number of random bits a Turing maching reads on any input x and random string y is called the amount of randomness used by the machine.

We define $\mathsf{RTime}(t(n))$ to be the class of all languages L such that there is a Turing machine M with running time $O(t(n))$ and for all $x \in L$, M accepts x with probability at least $1/2$ and for all $x \notin L$, M rejects L with probability 1. Such an M is said to have a *one-sided error*. If M in fact accepts each $x \in L$ with probability $\geq 1-\epsilon \geq 1/2$, then we say that the error probability of M is bounded by ϵ.

The class $\mathsf{BPTime}(t(n))$ is defined in the same manner, but we allow the Turing machine M to err in two ways. We require that for all $x \in L$, M accepts x with probability at least $2/3$ and for all $x \notin L$, M rejects with probability at least $2/3$ (that is, accepts with probability at most $1/3$). Such an error is called a *two-sided error*. If M actually accepts each $x \in L$ with probability $\geq 1-\epsilon$ and rejects each $x \notin L$ with probability $\geq 1-\epsilon$, then we say that the error probability is bounded by ϵ.

Definition 1.1 *1.* $\mathsf{RP} = \bigcup_{i\in\mathbb{N}} \mathsf{RTime}(n^i)$,

2. $\mathsf{BPP} = \bigcup_{i\in\mathbb{N}} \mathsf{BPTime}(n^i)$.

So the question whether every randomized algorithm can be simulated by a deterministic one can be rephrased as "Does BPP equal P?".

Why do we need bounded error? We could just say that a probabilistic Turing machine accepts an x if the acceptance probability is $> 1/2$ and rejects x if the acceptance probability is $\leq 1/2$. This leads to the class PP, a truly powerful class.[1] But if the acceptance probability of an input x is close to $1/2$, then the event that the machine accepts x is not a strong indication that x is in the language. But if we have a gap between the acceptance probabilities of inputs in the language and of inputs not in the language, then we can do something: For instance, let M be an RP machine[2] for some $L \in \mathsf{RP}$. We construct a new machine M' as follows: On a given input x, we run M k times on x, each time using new random bits. If M accepts x at least once, then M' accepts x, otherwise, M' rejects. If $x \in L$, then M accepts x with probability $\geq 1/2$. Since the runs of M are independent, the probability that M accepts x at least once is $1-(1/2)^k$. If $x \notin L$, then M

[1] For instance, $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$.

[2] For a complexity class C that is defined in terms of Turing machines, an C machine is a Turing machine that obeys the definitions of this class.

will always reject, and so does M'. This means we can make the acceptance probability larger than any given constant < 1.

1.2 Polynomial identity testing

If we want to show $\mathsf{NP} \neq \mathsf{P}$, then "it is sufficient" to show that the satisfibility problem is not in P. Unfortunately, we do not know any complete problems for BPP.[3] Even no "generic" problems are known. The language

$$\{\langle M, x, 1^t\rangle \mid M \text{ is a nondeterministic Turing machine that accepts } x \text{ within } t \text{ steps}\}$$

is a generic NP-complete language. The corresponding language

$$\{\langle M, x, 1^t\rangle \mid M \text{ is a } \mathsf{BPP} \text{ machine that accepts } x \text{ within } t \text{ steps}\}$$

is certainly BPP-hard, but it is not clear whether it is in BPP. The reason is that being a BPP machine is a *semantic* property, we cannot decide this by just inspecting the encoding of M.

Exercise 1.1 *Show that it is undecidable whether a given Turing machine has error probability bounded by* $1/3$.

What we can hope for are problems that are in BPP (or RP) but no proof is known that they are in P. Until several years ago, the candidate that was presented in such a lecture was `PRIMES`. Here is a problem that today has the same role as primality had before: Given a polynomial p of degree d in n variables $X_1, \ldots, X_n$ over $\mathbb{Z}$, decide whether p is identically zero. If the coefficients of p are given, then this task is of course easy. Representing a polynomial in such a way might not be that clever, since it has $\binom{d+n+1}{n+1}$ coefficients. Representing polynomials by arithmetic formulas or circuits often is a much better alternative. An *arithmetic circuit* is an acyclic directed graph with exactly one node of outdegree zero, the *output gate*. Each node has either indegree zero or two. A node with degree zero is either labeled with a constant from $\mathbb{Z}$ or with a variable X_i. A gate of indegree two is either labeled with "$+$" (addition gate), "$\times$" (multiplication gate), or "$/$" (division gate). In the later case, we have to ensure that there is no division by zero (as a rational function). For simplicity, we will solely deal with division-free circuits. This is justified by Strassen's result [Str73]. An *arithmetic formula* is an arithmetic circuit where all gates except the output gate have outdegree one, i.e., the underlying graph is a tree. The *size* of a circuit or formula is the number of nodes.

Definition 1.2 *1. Arithmetic circuit identity testing problem* (`ACIT`)*: Given an (encoding of an) arithmetic circuit C computing a polynomial p in $X_1, \ldots, X_n$, the task is decide whether p is identically zero.*

[3] Many researchers believe that $\mathsf{BPP} = \mathsf{P}$, in this case BPP has complete problems.

2. *Arithmetic formula identity testing problem* (`AFIT`)*: Given an (encoding of an) arithmetic formula computing a polynomial p, decide whether p is identically zero.*

For $a_1, \ldots, a_n \in \mathbb{Z}$, $C(a_1, \ldots, a_n)$ denotes the value that we obtain by substituting X_i by a_i, $1 \leq i \leq n$, and evaluating the circuit C. Occasionally, we write $C(X_1, \ldots, X_n)$ for the polynomial computed by C.

How do we check whether a polynomial p given by a circuit or formula is identically zero? We can of course compute the coefficients of p from the circuit. The output may be exponential in the size of the circuit, e.g., $(1 + X_1) \cdots (1 + X_n)$ has size $\mathrm{O}(n)$ but 2^n monomials, so this is highly inefficient. A better way to solve this problem is provided by randomization. We simply assign random values to the variables and evaluate the circuit. If p is nonzero, then it is very unlikely that p will evaluate to zero at a random point. This intuition is formalized in the following lemma.

Lemma 1.3 (Schwartz–Zippel [Sch80, Zip79]) *Let $p(X_1, \ldots, X_n)$ be a nonzero polynomial of degree d over a field k. Let $S \subseteq k$ be finite. Then*

$$\Pr_{r_1,\ldots,r_n \in S}[p(r_1, \ldots, r_n) = 0] \leq d/|S|.$$

Proof. The proof is by induction in n. The case $n = 1$ is easy: A univariate polynomial $p \neq 0$ of degree d has at most d roots. The probability of picking such a root from S is at most $d/|S|$. For the induction step $n \to n + 1$, we write p as an element of $k[X_1, \ldots, X_n][X_{n+1}]$. Let d' be the degree of X_{n+1} in p. We have

$$p(X_1, \ldots, X_{n+1}) = \sum_{\delta=0}^{d'} p_\delta(X_1, \ldots, X_n) X_{n+1}^\delta \qquad \text{with } p_\delta \in k[X_1, \ldots, X_n].$$

Obviously, $d' \leq d$ and $\deg p_{d'} \leq d - d'$. By the induction hypothesis,

$$\Pr_{r_1,\ldots,r_n}[p_{d'}(r_1, \ldots, r_n) = 0] \leq (d - d')/|S|.$$

If we know that $p_{d'}(r_1, \ldots, r_n) \neq 0$, then

$$\Pr_{r_{n+1}}[p(r_1, \ldots, r_n, r_{n+1}) = 0] \leq d'/|S|,$$

since once we fix $r_1, \ldots, r_n$, p is a nonzero univariate polynomial of degree d'. Altogether, our chosen point $(r_1, \ldots, r_{n+1})$ will fulfill $p(r_1, \ldots, r_{n+1}) = 0$ if either $p'_d(r_1, \ldots, r_n) = 0$ or $p'_d(r_1, \ldots, r_n) \neq 0$ but $p(r_1, \ldots, r_n, r_{n+1}) = 0$. This happens with probability $(d - d')/|S| + d'/|S| = d/|S|$. ∎

Let p be a nonzero polynomial of degree d. If we choose values $x_1, \ldots, x_n \in \{0, \ldots, 2d - 1\}$ at random, then the probability that $p(x_1, \ldots, x_n) = 0$ is at

most $\leq 1/2$. So a first algorithm looks as follows: Choose a point $(x_1, \ldots, x_n)$ as above. If $p(x_1, \ldots, x_n) = 0$, then claim that p is zero, otherwise that p is nonzero. If p is zero, then the algorithm never errs. If p is nonzero, then its error probability is at most $1/2$.

However, there is a catch. We have to estimate the cost of evaluating $p(x_1, \ldots, x_n)$. We first treat the case that p is given by an arithmetic formula of size s given by an encoding of length ℓ.[4]

Exercise 1.2 *Show the following:*

1. *Every arithmetic formula of size s computes a polynomial of degree at most s.*

2. *Consider an arithmetic formula of size s and let c be an upper bound for the absolute values of the constants in C. Assume we evaluate the formula at a point $(a_1, \ldots, a_n)$ with $|a_\nu| \leq b$, $1 \leq \nu \leq n$. Then $|C(a_1, \ldots, a_n)| \leq \max\{c, b\}^s$.*

If the largest absolute value of the constants in the formula is c, then the absolute value of the output is at most $(\max\{c, 2s\})^s$. (This follows from the last exercise, since the degree of the polynomial is bounded by s, hence we plug in values from $\{1, \ldots, 2s\}$.) Its bit representation has at most $s \cdot \log \max\{c, 2s\}$ many bits. Since $\log c \leq \ell$ (the bit representation of c is somewhere in the encoding), this is polynomial in the length of the input. This shows the following result.

Theorem 1.4 $\texttt{AFIT} \in$ co-RP. ■

The case where p is given by an arithmetic circuit is somewhat trickier, since we cannot evaluate the circuit in polynomial time. Modular arithmetic saves the day. We start with an analogue of Exercise 1.2.

Exercise 1.3 *Show the following:*

1. *Every arithmetic circuit of size s computes a polynomial of degree at most 2^{s-1}.*

2. *Consider an arithmetic circuit of size s and let c be an upper bound for the absolute values of the constants in C. Assume we evaluate the circuit at a point $(a_1, \ldots, a_n)$ with $|a_\nu| \leq d$, $1 \leq \nu \leq n$. Then $|C(a_1, \ldots, a_n)| \leq \max\{c, d\}^{2^s}$.*

Theorem 1.5 $\texttt{ACIT} \in$ co-RP.

[4] The size of a formula or circuit bounds the number of gates; but it does not bound the size of the constants in the formula or circuit. But the length ℓ of the encoding bounds both. For some proofs, it is however more convenient to work with the size s, therefore we use both quantities.

Proof. The following Turing machine is a probabilistic Turing machine for ACIT.

Input: a description of length ℓ of a circuit C of size s.

1. Choose random values $a_1, \ldots, a_n \in \{1, \ldots, 8 \cdot 2^s\}$.
2. Let $m = 2^s \cdot \max\{\log c, s+3\}$.
3. Choose a random prime number $q \leq m^2$ (Exercise 1.4).
4. Evaluate the circuit at $a_1, \ldots, a_n$ modulo q
5. Accept if the result is 0, otherwise reject.

Assume that in step 3, q is a prime number with probability $\geq 7/8$. q has $O(s \cdot \max\{\log\log c, \log s\})$ many bits. By Exercise 1.3, this is bounded by $O(s \log \ell)$. Thus we can evaluate the circuit modulo q in polynomial time; we can perform the operation at every gate modulo q, since the mapping $\mathbb{Z} \to \mathbb{Z}/q\mathbb{Z}$ is a ring homomorphism.

If C is zero, then the Turing machine will always accept C. (If we do not find a prime q in step 3, we will simply accept.)

Now assume that C is nonzero. By the Schwartz-Zippel lemma, the probability that C evaluates to zero is $\leq 2^s/(8 \cdot 2^s) = 1/8$. The probability that we do not find a prime in step 3 is 1/8, too. We have $|C(a_1, \ldots, a_n)| \leq \max\{c, 2^{s+3}\}^{2^s}$. Thus there are at most $2^s \cdot \max\{\log c, s+3\} = m$ different primes that divide $C(a_1, \ldots, a_n)$. The prime number theorem tells us that there are at least $m^2/(2\log m)$ many primes smaller than m^2. The probability that we hit a prime that divides $C(a_1, \ldots, a_n)$ is $(2\log m)/m \leq 1/8$ for s and henceforth m large enough. Thus the probability that the Turing machines accepts C is $\leq 3/8$. ∎

Exercise 1.4 *By the prime number theorem, a random m bit number is a prime with probability $\geq 1/m$.*

1. *Conclude that among m random m bit numbers, there is at least one prime with constant probability.*
2. *Give an efficient randomized algorithm that given m, returns a random m bit prime number with high probability (hint: Chernoff bound).*

The Schwartz-Zippel lemma even works, when the polynomial is not given by a circuit but as an *oracle* or *blackbox*, i.e., the Turing machine can write some (encoding of) an $x \in \mathbb{Z}$ on a special oracle tape and then immediately

gets back the value $p(x)$. This evaluation is counted as one step. In this situation, we do not need to compute modulo a random prime, since the blackbox does the evaluation for us.

If the polynomial is given by a circuit C, then we have an upper bound for the size of the coefficients of p, the polynomial computed by C. This upper bound is 2^{2^ℓ}, where ℓ is the size of the encoding of C, this is shown as in exercise 1.3. The interesting point is that we can find deterministically a point at which p does not vanish provided that p is not the zero polynomial. So in this case, evaluating C is the problem!

Exercise 1.5 (Kronecker substitution) *Let $p \in k[X_1, \ldots, X_n]$ be a polynomial with maximum degree $d_1, \ldots, d_n$ in $X_1, \ldots, X_n$. Let $D_i = (d_1 + 1) \cdots (d_{i-1} + 1)$. Let $\hat{p}(Y) = p(X_1^{D_1}, \ldots, X_n^{D_n})$. Then $\hat{p}$ is nonzero iff p. Moreover, there is a bijection between the terms of p and terms of q.*

Given C, we can compute a circuit $\hat{C}$ in polynomial time that computes the univariate polynomial $\hat{p}$ from the exercise above. Note that we do not need to take the exact value d_i to build $\hat{p}$, any upper bound is sufficient. So we can just take 2^s, where s is the size of C. So in $\hat{C}$, we first compute the powers $Y^{2^s}, Y^{2 \cdot 2^s}, \ldots, Y^{n \cdot 2^s}$. This can be done by a circuit of size polynomial in s and n by the well known *square and multiply method*.

Exercise 1.6 *Let $p = \sum_{i=0}^{d} a_i X^i \in \mathbb{R}[X]$ be a univariate polynomial. Let $c > \max_i |a_i|$. Then $p(c) \neq 0$.*

So in $\hat{C}$, we just have to replace Y by the constant $2^{2^\ell} + 1$. Let C_0 be this new circuit, which just computes a constant, i.e., polynomial of degree 0. This constant can be computed by the square and multiply method, thus C_0 can be computed from $\hat{C}$ in polynomial time. Now we just have to evaluate C_0 modulo a random prime like we did in Theorem 1.5.

1.3 Boolean circuits

The question whether we can derandomize BPP seems to be intimately related to lower bounds for circuit size, as we will see soon. A (Boolean) circuit C is an acyclic directed graph with exactly one node of outdegree zero. This node is called the output gate. Nodes with indegree zero are called input gates. The number n of input gates is the length of the input. Each other node has either indegree one or two. If it has indegree one, it is labeled with $\neg$ and called a NOT gate. Otherwise, it is labeled with $\vee$ or $\wedge$ and called an OR or AND gate, respectively. Any circuit C accepts a language $L \subseteq \{0,1\}^n$ where n is the number of input gates of C. For a given $x \in \{0,1\}^n$, we assign each gate a Boolean value inductively. The ith input gate gets the value x_i. (Order the input nodes arbitarily.) If all direct predecessors of a gate v

have already a value, then the value of v is the Boolean negation, Boolean disjunction or conjunction of the values of its direct predecessors, depending on the type of the gate. The string x is in L, if the output gate evaluates to 1, otherwise x is not in L. The *size* of a circuit C is the number of NOT, OR, and AND gates of C. A family of circuits C_n, $n \in \mathbb{N}$, accepts a language $L \subseteq \{0,1\}^*$, if C_n accepts $L \cap \{0,1\}^n$ for all n. In this case, we also write $L = L(C_n)$ for short.

Definition 1.6 *The class* P/poly *is the class of all languages $L \subseteq \{0,1\}^*$ such that there is a family of circuits C_n, $n \in \mathbb{N}$, and a polynomial p with $L = L(C_n)$ and* $\text{size}(C_n) = O(p(n))$.

Exercise 1.7 *Show that there is a nonrecursive language in* P/poly.

We call a family C_n of size $s(n)$ *uniform*, if there is a $O(\log s(n))$-space bounded Turing machine M that, on input n written in unary form on the input tape, outputs a description of C_n on the output tape. Circuits and deterministic Turing machines are polynomially related.

Exercise 1.8 *Prove the following theorem: For any deterministic Turing machine M with running time bounded by $t(n)$, there is a family of circuits C_n of size $t(n)^{O(1)}$ with $L(C_n) = L(M)$. This family is even uniform.*

Remark 1.7 *We can also define probabilistic or nondeterministic circuits. The input bits of the circuit is divided into two groups, the input x and a second string y. y can serve for instance as a random string. A circuit C accepts an input x, if for a fraction of at leasts $2/3$ of all random string y, $C(x,y) = 1$. C rejects x if $C(x,y) = 1$ for a fraction of at most $1/3$ of all random strings y. If y is polynomially long in $|x|$, then this precisely models* BPP. *In the same way, we can model* RP.

Or y can be seen as a witness. Then C accepts x if there is a y such that $C(x,y) = 1$. Otherwise, C rejects x. If y is polynomially long in $|x|$, then we get NP.

1.4 Derandomization versus circuit lower bounds

For a language $L \subseteq \{0,1\}^n$, let $L_n = L \cap \{0,1\}^n$ be the set of all words of length n in L. For a language $S \subseteq \{0,1\}^n$, $\text{Size}(S)$ denotes the size of a smallest circuit accepting S. We also need a somewhat strange concept: Let χ_S be the characteristic function of S. Let $h \in \mathbb{N}$ be minimal such that there is a circuit C of size h with $\Pr_{x \in \{0,1\}^n}[C(x) = \chi_S(x)] \geq 1/2 + 1/h$. We denote this number h by $\text{H}(S)$. The quantity $\text{H}(S)$ is called the *average-case hardness* of S. It essentially measures how large a circuit has to be in order to have a significant advantage of computing χ_S correctly over a

circuit that simply guesses randomly. In the same way we have defined Size and H for languages, we also define Size and H for Boolean functions $\{0,1\}^n \to \{0,1\}$ (which stand in one-to-one correspondence to subsets of $\{0,1\}^n$ via the characteristic function).

Theorem 1.8 (Impagliazzo & Wigderson [IW97]) *If there is an $L \in \mathsf{E}$ and a $\delta > 0$ such that $\mathrm{Size}(L_n) \geq 2^{\delta n}$ for almost all n, then $\mathsf{P} = \mathsf{BPP}$.*

The theorem above derandomizes BPP completely under the rather believable assumption that there is an language $L \in \mathsf{E}$ that requires circuits of size $\geq 2^{\delta n}$. For brevity, we call this assumption the IW assumption in the following. (Note that this is a nonuniform assumption, $\mathrm{Size}(L_n)$ is a nonuniform measure.)

If the IW assumption is true, then we can choose

$$L = \{\langle M, x, 1^t\rangle \mid M \text{ is a DTM and accepts } x \text{ within } 2^t \text{ steps}\},$$

since any other language in E is reducible to L by a linear time reduction, i.e., L is complete for E. (Note that a reduction may only blow up the size of the input by a constant factor, if E should be closed under this type of reduction.) By the time hierachy theorem, $L \notin \mathsf{SUBEXP}$. It would be surprising if L was solvable by circuits of subexponential size, i.e., of size $2^{o(n)}$. This would mean that nonuniformity gives more than a polynomial speed-up.

We will prove Theorem 1.8 in the first part of this lecture. It is also possible to trade hardness for running time in Theorem 1.8: If there is a language in E that requires circuits of size $s(n)$ then $\mathsf{BPP} \subseteq \mathsf{DTime}(2^{s^{-1}(n^{O(1)})})$. Thus if E requires superpolynomial circuits, then $\mathsf{BPP} \subseteq \mathsf{SUBEXP}$. We also get Theorem 1.8 out of this: If E requires circuits of size $2^{\Omega(n)}$, then $\mathsf{P} = \mathsf{BPP}$, and so forth. We will not prove this more general result here.

There is also a second derandomization result that uses a uniform hardness assumption.

Theorem 1.9 (Impagliazzo & Wigderson [IW98]) *If $\mathsf{BPP} \neq \mathsf{EXP}$, then for every $L \in \mathsf{BPP}$ and all $\epsilon > 0$ there is a deterministic Turing machine M with running time $2^{O(n^\epsilon)}$ such that for infinitely many n, L_n and $L(M)_n$ agree on a fraction of $1 - 1/n$ of $\{0,1\}^n$.*

Loosely speaking, if $\mathsf{BPP} \neq \mathsf{EXP}$, then BPP has subexponential deterministic algorithms that work for infinitely many input lengths on a large fraction of the inputs of this length. We will not present a proof of this theorem in this lecture. The result of Theorem 1.9 is not known to scale up. In particular, to my knowledge, the following problem is still open.

Open Problem 1.10 *Prove the following: If* $\mathsf{E} \not\subseteq \bigcap_{\delta>0} \mathsf{BPTime}(2^{\delta n})$*, then there is a deterministic Turing machine* M *with polynomial running time such that for infinitely many* n*,* L_n *and* $L(M)_n$ *agree on a fraction of* $1-1/n$ *of* $\{0,1\}^n$*.*

1.5 Further exercises

The fact that BPP might not have complete problems changes if we look at so called promise problems. A *partial language* is a pair of languages $L \subseteq U$. A Turing machine M accepts such a partial language, if it accepts all input in L and rejects all inputs in $U \setminus L$. We do not care about the behaviour of M on inputs in $\Sigma^* \setminus U$. Informally, we give M the promise that it will only get inputs from U. Therefore, partial languages are often called *promise problems*. The trick is that we can choose any language, even a nonrecursive one for U. For instance, in the above generic Turing machine simulation problem, we would choose U to be the set of all $\langle M, x, 1^t\rangle$ such that M is a probabilistic Turing machine that has error probability $\leq 1/3$. In this way, we overcome the problem that it is not decidable whether a Turing machine has error probability $\leq 1/3$. We simply do not care what the simulating machine does if M does not have error probability bounded by $1/3$.

For any complexity class C that is defined in terms of Turing machines, we can define a corresponding promise class prC in the obvious way: prC is the set of all partial languages (L, U) such that there is a Turing machine M that "has the properties" of the class C on all inputs from U and for all $x \in L$, M accepts x and for all $x \in U \setminus L$, M rejects x.

For classes that are defined in terms of syntactic properties, like P or NP, it does not make a real difference whether we consider promise problems or not. For instance, the statements $\mathsf{P} = \mathsf{NP}$ and $\mathsf{prP} = \mathsf{prNP}$ are equivalent. For classes defined by semantic properties, like RP and BPP, promise version are much easier to treat than the original classes. The additional set U gives the promise classes complete problems.

Exercise 1.9 *Show that* $\mathsf{prBPP} = \mathsf{prP}$ *implies* $\mathsf{BPP} = \mathsf{P}$*. What about the converse?*

Exercise 1.10 *Show the following:* $\mathsf{prP} = \mathsf{prNP}$ *if and only if* $\mathsf{P} = \mathsf{NP}$*.*

The following two problems are complete for BPP and RP, respectively.

Definition 1.11 *1. Circuit acceptance probability estimation* CAPE*: Given a circuit* C *with the promise that either* $\Pr_{x\in\{0,1\}^n}[C(x) = 1] \leq 1/3$ *or* $\Pr_{x\in\{0,1\}^n}[C(x) = 1] \geq 2/3$*, decide which of the two properties is fulfilled by* C*. (Here* n *is the length of the input.)*

2. *One-sided acceptance probability estimation* $\texttt{CAPE}_1$*: Given a circuit* C *with the promise that either* $\Pr_{x\in\{0,1\}^n}[C(x)=1]=0$ *or* $\Pr_{x\in\{0,1\}^n}[C(x)=1]\geq 1/2$*, decide which of the two properties is fulfilled by* C*.*

Exercise 1.11 *Show the following:*

1. $\texttt{CAPE}$ *is* prBPP*-complete (under logarithmic space many-one reductions).*
2. $\texttt{CAPE}_1$ *is* prRP*-complete (under logarithmic space many-one reductions).*

2 Some easy derandomization results

We start with comparing RP and BPP with some standard complexity classes. Since these are non-randomized complexity classes, one can view these results as a kind of derandomization.

2.1 Probability amplification

Before, we study *probability amplification*. Here we do not want to derandomize but just to reduce the error probability. In particular, it turns out that the choice of the constants 1/2 and 2/3 in the definitions of RP and BPP is fairly arbitrary.

Lemma 2.1 *Let M be a Turing machine for some language $L \in \mathsf{RP}$ that runs in time $t(n)$, uses $r(n)$ random bits, and has error probability $\epsilon < 1$. For any $k \in \mathbb{N}$, there is a Turing machine M' for L that runs in time $O(kt(n))$, uses $kr(n)$ random bits, and has error probability ϵ^k.*

Proof. M' works as follows:

Input: $x \in \{0,1\}^*$

1. M' simulates M k times, each time using new random bits.
2. M' accepts, if in at least one of the simulations, M accepts. Otherwise, M' rejects.

The bounds on the time and randomness are obvious. If $x \notin L$, then M' also rejects, since M does not err on x. If $x \in L$, then with probability at most ϵ, M rejects x. Since M' performs k independent trials, the probability that M' rejects x is at most ϵ^k. ∎

Lemma 2.2 *Let M be a Turing machine for some language $L \in \mathsf{BPP}$ that runs in time $t(n)$, uses $r(n)$ random bits, and has error probability $\epsilon < 1/2$. For any $k \in \mathbb{N}$, there is a Turing machine M' for L that runs in time $O(kt(n))$, uses $kr(n)$ random bits, and has error probability $2^{-c_\epsilon k}$ for some constant c_ϵ that solely depends on ϵ.*

Proof. M' works as follows:

Input: $x \in \{0,1\}^*$

1. M'simulates M k times, each time with fresh random bits.
2. M' accepts, if in at least half of the simulations (rounded up), M accepts. Otherwise, M' rejects.

Let μ be the expected number of times that a simulated run of M accepts. If $x \in L$, then $\mu \geq (1-\epsilon)k$. The probability that less than half of the simulated runs of M accept is $< e^{-\frac{(1-\epsilon)\delta^2}{2}k}$ with $\delta = 1 - \frac{1}{2(1-\epsilon)}$ by the Chernoff bound (see below). The case $x \notin L$ is treated similarly. In both cases, the error probability is bounded by $2^{-c_\epsilon k}$ for some constant c only depending on ϵ. ■

Remark 2.3 *In both lemmas, k can also be a function in n, as long as k is computable in time $O(k(n)t(n))$. (All reasonable functions k are.) In particular, if k is a polynomial then we still stay in* BPP *and can reduce the error probability to $2^{-\mathrm{poly}(n)}$.*

In the proof above, we used the so-called *Chernoff bound*. A proof of it can be found in most books on probability theory.

Lemma 2.4 (Chernoff bound) *Let $X_1, \ldots, X_m$ be independent 0–1 valued random variables and let $X = X_1 + \cdots + X_m$. Let $\mu = \mathrm{E}(x)$. Then for any $\delta > 0$,*

$$\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \quad \textit{and} \quad \Pr[X < (1-\delta)\mu] < e^{-\frac{\mu\delta^2}{2}}.$$

A "softer" task compared to derandomization is *randomness efficient probability amplification*: Here we try to reduce the error probability of an RP or BPP machine with as few as possible random bits. In this section, we just performed independent runs, which is not very efficient, since every time, we need fresh random bits. We will see more efficient constructions later on.

2.2 BPP versus time and space

Theorem 2.5 BPP $\subseteq$ PSPACE.

Proof. Let M be a BPP-machine for some $L \in \mathsf{BPP}$. Assume that M reads at most $r(n)$ random bits on inputs of length n. Turing machine M' simulates M as follows: M' systematically lists all bit strings of length $r(n)$ and simulates M with the current string as random string. M' counts how often M accepts and rejects. If the number of accepting computations exceeds the number of rejecting computations, M' accepts. Otherwise, M' rejects. Since M is polynomial time, $r(n)$ is bounded by a polynomial. Hence M' uses only polynomial space. ■

Corollary 2.6 $\mathsf{BPP} \subseteq \mathsf{EXP}$.

Theorem 2.7 $\mathsf{RP} \subseteq \mathsf{NP}$.

Proof. Let M be an RP-machine for some $L \in \mathsf{RP}$. We convert M into an NP-machine M' as follows. Whenever M would read a bit from the random tape, M' nondeterministically branches to the two states that M would enter after reading zero or one, respectively. If M does not accept x, then there is no random string such that M on input x reaches an accepting configuration. Thus there is no accepting path in the computation tree of M' either.

On the other hand, if M accepts x, then M reaches an accepting configuration on at least half of the random strings. Thus at least half of the paths in the computation tree of M' are accepting ones. In particular, there is at least one accepting path. Hence M' accepts x. ■

2.3 BPP and circuits

Next we turn to the relation between BPP and circuits. The class P/poly can be viewed as the languages accepted by polynomial time deterministic Turing machines with polynomial *advice*. Such a Turing machine has an additional read-only advice tape.

Definition 2.8 *Let t and a be two functions $\mathbb{N} \to \mathbb{N}$. A language L is in the class $\mathsf{DTime}(t)/a$ if there is a deterministic Turing machine M with running time $O(t)$ and with an additional advice tape and a sequence of strings $\alpha(n) \in \{0,1\}^{a(n)}$ such that the following holds: For all $x \in L$, M accepts x with $\alpha(|x|)$ written on the advice tape. For all $x \notin L$, M rejects x with $\alpha(|x|)$ written on the advice tape.*

For each input length n, we give the Turing machine an advice $\alpha(n)$. Note that we do no restrict this sequence, except for the length. In particular, the sequence need not be to computable at all.

This definition extends to nondeterministic classes and space classes in the obvious way. We can also extend the definition to sets of functions T and A. We define $\mathsf{DTime}(T)/A = \bigcup_{t \in T, a \in A} \mathsf{DTime}(t)/a$. If we choose T and A both to be the class of all polynomials, then we get exactly P/poly.

Lemma 2.9 *For all languages L, if there is a polynomial time Turing machine M with polynomial advice accepting L, then $L \in \mathsf{P/poly}$.*

Proof. For the moment, let us view the advice $\alpha(n)$ as a part of the input, i.e., M gets $\alpha(|x|)$ concatenated with its regular input x. By Exercise 1.8, for each n, there is a circuit C_n such that $C_n(x, \alpha(n)) = M(x, \alpha(n))$ for all x of length n. Let C'_n be the sequence of circuits obtained from C_n by fixing the second part of the input to $\alpha(n)$. This gives a sequence of polynomial size circuits such that $C'_n(x) = C_n(x, \alpha(n)) = M(x, \alpha(n))$ for all x of length n. Thus $L \in \mathsf{P/poly}$. ■

Exercise 2.1 *Show that the converse of Lemma 2.9 holds, too.*

Theorem 2.10 (Adleman [Adl78]) $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

Proof. Let $L \in \mathsf{BPP}$. By Lemma 2.2, there is a BPP-Machine with error probability $< 2^{-n}$ that accepts L. There are 2^n possible input strings of length n. Since for each string x of length n, the error probability of M is $< 2^{-n}$, M can err on x only for a fraction of all possible random strings that is smaller than 2^{-n}. Thus there must be one random string that is good for all inputs of length n. We take this string as an advice string for the inputs of length n. By Lemma 2.9, $L \in \mathsf{P/poly}$. ■

How do we find this good random string? If we amplify the error probability even further, say to 2^{-2n}, then almost all, namely a fraction of $1 - 2^{-n}$ random strings are good. Thus picking the advice at random is a good strategy. (This, however, requires randomness!)

2.4 BPP and the polynomial hierachy

For two strings $u, v \in \{0,1\}^n$, $u \oplus v$ denotes the string that is obtained by taking the bitwise XOR.

Lemma 2.11 (Lautemann) *Let $n > \log m$. Let $S \subseteq \{0,1\}^m$ with $|S| \geq (1 - 2^{-n})2^m$.*

1. *There are $u_1, \ldots, u_m$ such that for all v, $u_i \oplus v \in S$ for some $1 \leq i \leq m$.*
2. *For all $u_1, \ldots, u_m$ there is a v such that $u_i \oplus v \in S$ for all $1 \leq i \leq m$.*

Proof.

1. We have

$$\begin{aligned}
&\Pr_{u_1,\ldots,u_m\in\{0,1\}^m}[\exists v : u_1 \oplus v, \ldots, u_m \oplus v \notin S] \\
&\leq \sum_{v\in\{0,1\}^m} \Pr_{u_1,\ldots,u_m\in\{0,1\}^m}[u_1 \oplus v, \ldots, u_m \oplus v \notin S] \\
&= \sum_{v\in\{0,1\}^m} \prod_{i=1}^{m} \Pr_{u_i\in\{0,1\}^m}[u_i \oplus v \notin S] \\
&\leq 2^m \cdot (2^{-n})^m \\
&< 1,
\end{aligned}$$

   since $u_i \oplus v$ distributed uniformly in $\{0,1\}^m$ and all the u_i's are drawn indepently. Since the probability that a v with the desired properties does not exist is < 1, there must be a v that fulfills the assertions of the first claim.

2. Fix $u_1, \ldots, u_m$. We have

$$\begin{aligned}
\Pr_{v\in\{0,1\}^m}[\exists i : u_i \oplus v \notin S] &\leq \sum_{i=0}^{m} \Pr_{v\in\{0,1\}^m}[u_i \oplus v \notin S] \\
&\leq m \cdot 2^{-n} \\
&< 1.
\end{aligned}$$

   Thus a v exists such that for all i, $u_i \oplus v \in S$. Since $u_1, \ldots, u_m$ were arbitrary, we are done. ■

Exploiting the previous lemma, it is very easy to show that BPP is contained in the second level of the polynomial hierachy.

Theorem 2.12 (Sipser) $\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{P}} \cap \Pi_2^{\mathsf{P}}$.

Proof. Let A be a language in BPP and M be a BPP machine for A. Using probability amplification, we can assume that the error probability of M is $\leq 2^{-n}$. Assume that M uses $p(n)$ random bits.

Let $T_x = \{y \mid M \text{ accepts } x \text{ with random string } y\}$. If $x \in A$, then $|T_x| \geq (1 - 2^{-n})2^{p(|x|)}$. In this case, the first statement of Lemma 2.11 is true with $m = p(|x|)$. If $x \notin A$, then $|\bar{T}_x| \geq (1-2^{-n})2^{p(|x|)}$. Thus the second statement of Lemma 2.11 is true for $\bar{T}_x$. But this is the negation of the first statement for T_x. Hence

$$x \in A \iff \exists^p u_1, \ldots, u_{p(|x|)} \forall^p v : u_1 \oplus v \in T_x \vee \cdots \vee u_{p(|x|)} \oplus v \in T_x.$$

The relation on the righthand side is cleary verifiable in polynomial time. Hence, $A \in \Sigma_2^{\mathsf{P}}$.

This also shows that $A \in \Pi_2^{\mathsf{P}}$, because BPP is closed under complementation (see below) and $\Pi_2^{\mathsf{P}} = \mathsf{co}\text{-}\Sigma_2^{\mathsf{P}}$. ■

Exercise 2.2 *Prove that* $\mathsf{BPP} = \mathsf{co\text{-}BPP}$.

2.5 Further exercises

The answer to the following question is not known.

Open Problem 2.13 *Prove or disprove:* $\mathsf{RP} = \mathsf{P}$ *implies* $\mathsf{BPP} = \mathsf{P}$.

3 The Nisan–Wigderson generator

3.1 Pseudorandom generators

The tool for proving Theorem 1.8 will be pseudorandom generators.

Definition 3.1 *A function $G : \{0,1\}^t \to \{0,1\}^m$ is a (s,ϵ)-*pseudorandom generator *if for every Boolean circuit D of size $\leq s$ we have*

$$\left| \Pr_{r \in \{0,1\}^m}[D(r) = 1] - \Pr_{z \in \{0,1\}^t}[D(G(z)) = 1] \right| \leq \epsilon.$$

The input of a pseudorandom generator is also called the *seed*. Its length t is called the *seed length*.

Any circuit of size s cannot distinguish the distribution generated by G from a uniform distribution, up to ϵ. If we have a (s,ϵ)-pseudorandom generator, $G : \{0,1\}^t \to \{0,1\}^m$, then we also have a pseudorandom generator $G' : \{0,1\}^t \to \{0,1\}^{m'}$ for any $m' \leq m$, just by computing $G(x)$ and omitting the last $m - m'$ bits. Therefore, we will set $s = 2m$ in the following. This is simply done to reduce the number of parameters.

The following theorem shows the importance of pseudorandom generators. We say that a family of pseudorandom generators (G_m) is uniformly computable in time $t(m)$ if there is a deterministic Turing machine with running time $t(m)$ that computes the mapping $\langle m, x\rangle \mapsto G_m(x)$. Note that for a pseudorandom generator $G_m : \{0,1\}^{t(m)} \to \{0,1\}^m$ we have $t(m) < m$, therefore it is useful to take the size of the output as the index.

Theorem 3.2 *If there is a family of $(2m, 1/8)$-pseudorandom generators $G_m : \{0,1\}^{O(\log m)} \to \{0,1\}^m$ for $m \in \mathbb{N}$ such that G_m is uniformly computable in time polynomial in m, then* $\mathsf{P} = \mathsf{BPP}$.

Proof. Let M be a BPP machine with running time $p(n)$ for some polynomial p. There is a circuit C_n of size $s(n) = \mathrm{poly}(p(n))$ that simulates M on every input x of length n and random string y of length $p(n)$. We choose $m = s(n)$

G_m stretches $O(\log m)$ bits into m bits and is $(\frac{1}{8}, 2m)$-pseudorandom. We build a new BPP machine $\hat{M}$ for the same language that takes $O(\log m)$ bits, stretches them to m bits, and then simulates M using the pseudorandom string. We derandomize $\hat{M}$ by enumerating all $2^{O(\log m)} = \mathrm{poly}(n)$ seeds and taking a majority vote, see Algorithm 1.

Algorithm 1 Deterministic Simulation

Input: w

1: **for all** $r \in \{0,1\}^{O(\log m)}$ **do**
2: Stretch r to $y \in \{0,1\}^m$ using G_m.
3: Simulate M on input w with random string y.
4: **od**
5: **if** in the majority of all simulations, M accepted **then**
6: return 1
7: **else**
8: return 0
9: **fi**

The running time of the deterministic simulation is $O(2^{O(\log m)} \cdot p(n)) = \text{poly}(n)$. The deterministic simulation of $\hat{M}$ is obviously correct.

It remains to show that M and $\hat{M}$ accept the same language. Let w be an arbitrary input. By the definition of pseudorandom generator,

$$\left| \Pr_{y \in \{0,1\}^m}[M(w,y) = 1] - \Pr_{r \in \{0,1\}^{O(\log m)}}[M(w, G_m(r)) = 1] \right| \leq 1/8,$$

since $M(w,\cdot)$ can by simulated by a circuit of size $m = s(n)$.[1] If M accepts x with probability $\geq 2/3$, then $\hat{M}$ accepts x with probability $\geq 2/3 - 1/8 = 13/24 > 1/2$. If M rejects x with probability $\geq 2/3$, then $\hat{M}$ rejects x with probability $\geq 2/3 - 1/8 = 13/24 > 1/2$. ■

Thus the existence of a family of efficiently computable random generators allows us to derandomize BPP. This is of course only one possible approach for derandomizing BPP. Except the concept of a hitting set generator, which could be weaker than that of a pseudorandom generator, we currently do not know any other approach for derandomizing BPP. For a discussion of hitting set generators, the interested reader is refered to Miltersen's bookchapter [Mil01].

3.2 Outline of the proof of Theorem 1.8

The proof of Theorem 1.8 contains two main ingredients. The first one is the *Nisan–Wigderson generator*.

Theorem 3.3 (Nisan & Wigderson [NW94]) *If there is an $L \in \mathsf{E}$ and a $\delta > 0$ such that $\mathrm{H}(L_n) \geq 2^{\delta n}$ for almost all n, then there is a family of $(2m, 1/8)$-pseudorandom generators $G_m : \{0,1\}^{O(\log m)} \to \{0,1\}^m$ that is computable in time polynomial in m.*

[1] Here is the place where we need that pseudorandom generators fool circuits and not Turing machines. Since the input w is fixed, $M(w,\cdot)$ is a nonuniform computation device.

Together with Theorem 3.2, this immediately yields the following corollary.

Corollary 3.4 *If there is an* $L \in \mathsf{E}$ *and a* $\delta > 0$ *such that* $\mathrm{H}(L_n) \geq 2^{\delta n}$ *for almost all* n*, then* $\mathsf{P} = \mathsf{BPP}$

The assumption of Theorem 3.3 seems to be really strong: For almost all n, even circuits of size $2^{\delta n}$ are unable to decide L on more than a fraction of $1/2 + 2^{-\delta n}$ of all inputs of length n.

There is a circuit of size 2^n that decides L on every input. (It is easy to see that there is a circuit of size $O(n \cdot 2^n)$ by simply implementing the CNF or DNF. But the Shannon-Lupanov bounds (see below) says that there is even a circuit of size $(1+o(1)) \cdot 2^n/n$.) On the other hand, there is a circuit of size 1 that solves the problem on a fraction of $1/2$ of all inputs. (The circuit always outputs zero or one, whichever is better.) We can also design a circuit as follows: On the inputs that have zeros in the first $n/2$ positions, the circuit computes the result correctly. On the other inputs, it either always outputs zero or one, whichever is better. This circuits has size $\leq 2^{n/2}$ and solves the problem correctly on a fraction of $2^{-n/2} + \frac{1}{2}(1 - 2^{n/2}) = 1/2 + 2^{-n/2-1}$.

Exercise 3.1 *Prove the* Shannon–Lupanov bound*: For any function* $f : \{0,1\}^n \to \{0,1\}$*, there is a circuit of size* $(1 + o(1)) \cdot 2^n/n$ *computing* f*. (The function hidden in the* $o(1)$ *is independent of* f*.)*

This result holds only, if we allow gates that compute arbitrary 2-ary Boolean functions. (There are 16.) By switching to the basis AND, OR, NOT, we only loose a constant factor.

The above arguments are not very encouraging that the Nisan–Wigderson assumption is true. The next results however shows that it is implied by the believable IW assumption.

Theorem 3.5 ([BFNW93, Imp95, IW97]) *If there is an* $L \in \mathsf{E}$ *and a* $\delta > 0$ *such that* $\mathrm{Size}(L_n) \geq 2^{\delta n}$ *for almost all* n*, then there is an* $L' \in \mathsf{E}$ *and a* $\delta' > 0$ *such that* $\mathrm{H}(L'_n) \geq 2^{\delta' n}$ *for almost all* n*.*

Loosely speaking, the theorem says that if there is a language in E that is hard in the worst case, then there is also one that is hard in the average case. Therefore, this theorem is also called *worst case to average case reduction.*

In this chapter, we will prove Theorem 3.3: Based on the language $L \in \mathsf{E}$ in the NW assumption, we try to construct a pseudorandom generator. Its seed length will be $O(\ell)$, its output has length $2^{\Theta(\ell)}$, it is computable in time $2^{O(\ell)}$ and there is no circuit of size $2^{O(\ell)}$ that distinguishes the distribution of its output from the uniform distribution by more than $1/8$.

In the next chapter, we will prove Theorem 3.5.

3.3 Combinatorial Designs

Combinatorial designs are an important tool in the Nisan–Wigderson construction.

Definition 3.6 *A family $(S_1, \ldots, S_m)$ of subsets of the universe $U = \{1, 2, \ldots, t\}$ is an (m, t, ℓ, α)*-design *if $|S_\mu| = \ell$ for all $1 \leq \mu \leq m$ and $|S_\mu \cap S_\nu| \leq \alpha$ for all $\mu \neq \nu$.*

The question is of course for which parameters designs exist and how to construct them. The following lemma shows that designs exist for a certain choice of parameters that will be sufficient for our needs. The result is somewhat surprising since it basically states that in a universe that is larger than the sets only by a constant factor, we can pack exponentially many sets whose pairwise intersection is by a constant factor smaller than the sets.

Lemma 3.7 *For all integers ℓ and all $\gamma > 0$ there is an $(m, t, \ell, \log m)$-design where $t = O(\ell/\gamma)$ and $m = 2^{\gamma\ell}$. This design is computable in time $O(2^t t m^2)$.*

Proof. We construct the family of sets inductively. For the first set, we can choose any subset of $\{1, \ldots, t\}$ of size ℓ.

Suppose we have already constructed the sets $S_1, \ldots, S_i$. We next prove that there exists a set S_{i+1} such that $S_1, \ldots, S_{i+1}$ is a $(i+1, t, \ell, \log m)$-design. Then we are done, since we can enumerate all subsets of size ℓ of $\{1, \ldots, t\}$ and can check whether the pairwise intersections with $S_1, \ldots, S_i$ all have size at most $\log m$. One such check takes time $O(t)$, there are $\leq m$ sets we have to intersect the candidate with, and there are at most 2^t candidates, since there are 2^t subsets of $\{1, \ldots, t\}$. Thus the whole procedure needs $O(2^t t m^2)$ time, as we have to construct m sets.

To show that S_{i+1} exists, we pick randomly 2ℓ many elements from $\{1, \ldots, t\}$ with replacement. Let S be resulting set. Let $t \leq c\ell/\gamma$. If we choose c large enough, then we can show that with high probabilily, S has size at least ℓ. Furthermore, the pairwise intersections of S with $S_1, \ldots, S_i$ have size $\leq \log m = \gamma\ell$, again with high probability. Hence such a set S exists. We obtain S_{i+1} by taking a subset of appropriate size of S. ∎

Exercise 3.2 *Fill in the missing details of the proof of Lemma 3.7. (Hint: Chernoff bound)*

3.4 Hybrid argument

The next lemma is crucial for the analysis of the Nisan-Wigderson generator, but it is also useful for analysing different notions of pseudorandomness (cf.

the exercises at the end of this chapter). A mapping $T : \{0,1\}^m \to \{0,1\}$ is called a *statistical test* on $\{0,1\}^m$. (One could call it a characteristic function, too, but statistical test is more appropriate here.) For a string x and $i \leq |x|$, $x_{\leq i}$ denotes its prefix of length i.

Lemma 3.8 (Hybrid argument) *Let $g : \{0,1\}^n \to \{0,1\}^m$. Let T be some statistical test on $\{0,1\}^m$. If*

$$\left| \Pr_{y \in \{0,1\}^m}[T(y) = 1] - \Pr_{x \in \{0,1\}^n}[T(g(x)) = 1] \right| > \delta,$$

then there is an index i_0 such that

$$\Pr[S(g(x)_{\leq i_0} u_{i_0+1} \dots u_m) = g(x)_{i_0+1}] > \frac{1}{2} + \frac{\delta}{m}$$

where the probability is taken over x, $u_{i_0+1}, \dots, u_m$, and S is the test defined by

$$S(g(x)_{\leq i_0} u_{i_0+1} \dots u_m) = \begin{cases} u_{i_0+1} & \text{if } T(g(x)_{\leq i_0} u_{i_0+1} \dots u_m) = 1, \\ \overline{u_{i_0+1}} & \text{otherwise.} \end{cases}$$

Proof. For $0 \leq i \leq m$, let D_i be the distribution on $\{0,1\}^m$ generated by picking $x \in \{0,1\}^n$ and $u \in \{0,1\}^m$ uniformly at random and then outputting $g(x)_{\leq i} u_{i+1} \dots u_m$. Let $t(D_i) = \Pr_{z \sim D_i}[T(z) = 1]$. In particular, $t(D_0) = \Pr_{y \in \{0,1\}^m}[T(y) = 1]$ and $t(D_m) = \Pr_{x \in \{0,1\}^n}[T(g(x)) = 1]$, thus

$$t(D_m) - t(D_0) \geq \delta. \tag{3.1}$$

In the last equation, we do not need to take the absolute value of the lefthand side, since we can replace T by the test that flips each answer. (3.1) implies

$$\sum_{i=0}^{m-1} (t(D_{i+1}) - t(D_i)) \geq \delta.$$

In particular, there is an i_0 such that

$$t(D_{i_0+1}) - t(D_{i_0}) \geq \frac{\delta}{m}. \tag{3.2}$$

Let $\overline{D}_j$ be the distribution that is obtained from D_j by flipping the bit in position i. In D_i, the $(i+1)$th bit is chosen uniformly at random. Therefore, it is $g(x)_{i+1}$ or $\overline{g(x)_{i+1}}$ with the same probability. Thus,

$$t(D_i) = \frac{1}{2}(t(D_{i+1}) + t(\overline{D}_{i+1})). \tag{3.3}$$

Now,

$$\begin{aligned}
&\Pr[S(g(x)_{\le i_0} u_{i_0+1} \dots u_m) = g(x)_{i_0+1}] \\
&\quad = \frac{1}{2} \Pr[T(g(x)_{\le i_0} u_{i_0+1} \dots u_m) = 1 \mid u_{i_0+1} = g(x)_{i_0+1}] \\
&\qquad + \frac{1}{2} \Pr[T(g(x)_{\le i_0} u_{i_0+1} \dots u_m) = 0 \mid u_{i_0+1} = \overline{g(x)_{i_0+1}}] \\
&\quad = \frac{1}{2}(t(D_{i_0+1}) + 1 - t(\overline{D}_{i_0+1})) \\
&\quad = \frac{1}{2} + t(D_{i_0+1}) - t(D_{i_0}) \\
&\quad > \frac{1}{2} + \frac{\delta}{m}
\end{aligned}$$

where $t(D_{i_0+1}) - t(\overline{D}_{i_0+1}) = 2(t(D_{i_0+1}) - t(D_{i_0}))$ follows from (3.3). ■

3.5 Nisan–Wigderson generator

We start by introducing some notation: For $z \in \{0,1\}^t$ and $S \subseteq \{1, \dots, t\}$, $z|_S$ denotes the string of length $|S|$ that is obtained from z by ommitting all bits whose index is not in S.

For a Boolean function $f : \{0,1\}^\ell \to \{0,1\}$ and an $(m, t, \ell, \log m)$-design $\mathcal{S} = (S_1, \dots, S_m)$, the *Nisan–Wigderson generator* $NW_{f,\mathcal{S}} : \{0,1\}^t \to \{0,1\}^m$ is defined as follows:

$$NW_{f,\mathcal{S}}(z) = f(z|_{S_1}) f(z|_{S_2}) \cdots f(z|_{S_m}).$$

The following lemma analyses the Nisan–Wigderson generator.

Lemma 3.9 *Let $f : \{0,1\}^\ell \to \{0,1\}$ be a Boolean function and $\mathcal{S} = (S_1, \dots, S_m)$ be an $(m, t, \ell, \log m)$-design. For every $D : \{0,1\}^m \to \{0,1\}$ and ϵ such that*

$$\left| \Pr_{r \in \{0,1\}^m}[D(r) = 1] - \Pr_{z \in \{0,1\}^t}[D(NW_{f,\mathcal{S}}(z)) = 1] \right| > \epsilon,$$

there exists a circuit C of size $\mathrm{size}(D) + \mathrm{O}(m^2)$ such that

$$\Pr_{x \in \{0,1\}^\ell}[C(x) = f(x)] \ge \frac{1}{2} + \frac{\epsilon}{m}.$$

Proof. The high level idea is that if D is able to distinguish the distribution generated by $NW_{f,\mathcal{S}}$ from the uniform distribution, then it is possible to find a single bit in the output where the difference is noticeable. On such a bit, D distinguishes the bit $f(x)$ from a random one and we can use this fact to construct a circuit that predicts $f(x)$.

By Lemma 3.8, there is an index i_0 and a statistical test S that given $f(z|_{S_1}) \dots f(z|_{S_{i_0}}) u_{i_0+1} \dots u_m$ can predict $f(z|_{S_{i_0+1}})$ with probability $1/2 + \epsilon/m$. This is exactly what we want; it remains to show that we can get an efficient circuit out of the algorithm for S.

Rename the indices $\{1, \dots, t\}$ such that $S_{i_0+1} = \{1, \dots, \ell\}$. Then $z|_{S_{i_0+1}}$ are the first ℓ bits of z. Write $z = xy$. Let $f_\mu(x, y) = f(z|_{S_\mu})$ for $1 \leq \mu \leq m$ and let $P(x, y, u) = S(f_1(x, y) \dots f_{i_0}(x, y) u_{i_0+1} \dots u_m)$. We have $\Pr_{x,y,u}[P(x, y, u) = f(x)] \geq \frac{1}{2} + \frac{\epsilon}{m}$, in particular, there exist constants $c_i, \dots, c_m \in \{0, 1\}$ and $w \in \{0, 1\}^{t-\ell}$ such that

$$\Pr_x[P(x, w, c_{i+1} \cdots c_m) = f(x)] \geq \frac{1}{2} + \frac{\epsilon}{m}.$$

Since w is fixed, we only have to compute the functions $x \mapsto f_\mu(x, w)$ But for each μ, $f_\mu(x, w)$ depends only on $\leq \log m$ bits of x by the properties of the combinatorial design. Thus, we can compute $x \mapsto f_\mu(x, w)$ by a circuit of size $O(m)$. Thus $f_1(x, w), \dots, f_{i-1}(x, w)$ can be computed by a circuit of size $O(m^2)$. Since S only envokes the circuit D once, the total construction has size $\mathsf{Size}(D) + \mathrm{O}(m^2)$. ∎

3.6 Proof of Theorem 3.3

We finally prove Theorem 3.3: Let f be the restriction to $\{0, 1\}^n$ of the characteristic function of the hard language L whose existence is assured by the NW assumption. Let $\mathcal{S}$ be an $(m, t, \ell, \log m)$-design as constructed in Lemma 3.7. We set $\ell = n$ and $m = 2^{\delta/3 \cdot n}$. This also defines the parameter t.

We claim that $NW_{f,\mathcal{S}}$ is $(1/8, 2m)$-pseudorandom. Suppose that this is not the case. Then there is a circuit D of size $\leq 2m$ such that

$$\left| \Pr_r[D(r) = 1] - \Pr_z[D(NW_{f,\mathcal{S}}(z)) = 1] \right| > \frac{1}{8}.$$

By Lemma 3.9, there is now a circuit C of size $\mathsf{Size}(D) + O(m^2) = O(2^{2\delta/3 \cdot n})$ such that

$$\Pr_x[C(x) = f(x)] \geq \frac{1}{2} + \frac{1}{8m} = \frac{1}{2} + \frac{1}{8 \cdot 2^{\delta/3 \cdot n}}$$

In particular, $\mathrm{H}(f) \leq 2^{2\delta/3 \cdot n}$, a contradiction.

It remains to show that $NW_{f,\mathcal{S}}$ is computable in time polynomial in m. By Lemma 3.7, the design $\mathcal{S}$ is computable in time $O(2^t t m^2)$. But $t = O(\log m)$, which is fine. Next we have to evaluate f at m inputs of the length $\ell = n = O(\log m)$. This can be done in time $m 2^{O(n)}$, which is again polynomial in m. This completes the proof of Theorem 3.3.

3.7 Further Exercises

Here is another definition of pseudorandom strings by Blum and Micali [BM84]. Blum and Micali observed that if we draw a string $y \in \{0,1\}^n$ uniformly at random, then the probability that y_{i+1} equals 1 given $y_{\leq i}$ is exactly 1/2. Each single bit is unpredictable.

Definition 3.10 (Blum & Micali) *A function $G : \{0,1\}^t \to \{0,1\}^m$ is* (s,ε)-unpredictable *if for every s size bounded circuit B,*

$$\Pr[B(G(x)_{\leq i}) = G(x)_{i+1}] \leq \frac{1}{2} + \varepsilon.$$

Above, the probability is taken over the choice of $x \in \{0,1\}^n$ and the index $0 \leq i \leq n-1$.

In the definition above, we relax the condition that each bit is truly unpredictable in two ways: first, it is possible to predict a bit with a slighlty larger probability than 1/2 and second, the computational power of the predictor, i.e, the circuit B, is limited.

Yao [Yao82] showed that the two definitions of pseudorandomness are essentially equivalent. If G is bitwise unpredictable then it is a pseudorandom generator and vice versa.

Theorem 3.11 (Yao) *Let $G : \{0,1\}^t \to \{0,1\}^m$.*

1. *If G is $(O(s), \varepsilon/m)$-unpredictable, then it is (s,ε)-pseudorandom.*
2. *If G is (s,ε)-pseudorandom, then it is (s,ε)-unpredictable.*

Remark 3.12 *We here consider unpredictability and pseudorandomness against circuits. In cryptography, one also considers unpredictability and pseudorandomness against probabilistic Turing machines. If we just look at a generator for one seed length, this does not make any difference. If we consider families of generators, then this new concept is weaker.*

Let $B_{n,m}$ denote the set of all functions $\{0,1\}^n \to \{0,1\}^m$. Let $f : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^m$. For each $z \in \{0,1\}^m$, let $f|_z$ denote the function $\{0,1\}^n \to \{0,1\}^m$ defined by $x \mapsto f(z,x)$. In other words, f generates a family of 2^m functions.

We now can generate a function $f|_z$ by picking z at random. When does $f|_z$ look pseudorandom?

Definition 3.13 *Let $f : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^m$. f is* (s,δ)-pseudorandom *if for every oracle circuit C of size s,*

$$\left| \Pr_{h \in B_{n,m}}[C^h(1^n) = 1] - \Pr_{z \in \{0,1\}^m}[C^{f_z}(1^n) = 1] \right| \leq \delta.$$

Algorithm 2 Pseudorandom function generator

Input: $z \in \{0,1\}^m$, $x \in \{0,1\}^n$

```
1: Let y = z
2: for i = 1, ..., n do
3:    Let y_1 y_2 = g(y) with y_1, y_2 ∈ {0,1}^m.
4:    if x_i = 1 then
5:       y = y_1
6:    else
7:       y = y_2
8:    fi
9: od
10: return y
```

In the definition above, C gets a function as an oracle. That means, it can evaluate such a function by using oracle gates. This oracle is here considered to be the input of C. C does not get a "real" input, just a string of 1s. We cannot give C the function as an input, since then the input size would be too large.

Assume we have a function $g : \{0,1\}^m \to \{0,1\}^{2m}$ that is (s,δ)-pseudorandom. With g, we build a pseudorandom function generator as follows: Given $z \in \{0,1\}^m$, construct a complete binary tree T_z of height n as follows: The root is labeled with z. If any node is labeled with $y \in \{0,1\}^m$, we compute $g(y)$ and label the left child with the first half of $g(y)$ and the right child with the second half of $g_m(y)$. Any $x \in \{0,1\}^n$ defines a path from the root to some leaf in T_z by identifying "1" with "left" and "0" with "right". The label at the leaf at the end of this path is the value $f|_z(x)$. Algorithm 2 shows how to compute $f|_z(x)$.

Exercise 3.3

1. *Analyse the time needed to evaluate f at $(z,x) \in \{0,1\}^m \times \{0,1\}^n$.*
2. *Show that f is indeed pseudorandom by using a hybrid argument. For this argument, assign the first i levels of the tree random labels. What parameters do you get for f?*

4 Worst case to average case reduction

In this section, we prove Theorem 3.5. We present a newer proof due to Sudan, Trevisan, and Vadhan [?]. Their proof exploits error-correcting codes. The idea is the following: Assume that $f : \{0,1\}^n \to \{0,1\}$ is a function that is hard in the worst case. We can think of f as a string s of length 2^n that we get by considering the table of values as a string. Now we take a suitable error correcting code C and apply it to s. We consider this new string $C(s)$ as the table of values of some function f'. We claim that this function is hard on the average. If this were not the case, then we could compute a string that is close enought to $C(s)$ such that we can correct the errors and recover the string s. But this would be an efficient procedure to compute f in the worst case, a contradiction.

4.1 Locally decodable codes

We consider error-correcting codes over arbitrary alphabets. In the following, $[q]$ shorthands the set $\{1, 2, \ldots, q\}$. A word w of length n over a q-ary alphabet is simply an element of $[q]^n$. Instead of thinking of a word, one can view w as a vector. Or as a function $[n] \to [q]$. It will be convenient to switch between these representations. In particular, if we say that we compute a message or code word x as a function, then we mean that we compute the function $i \mapsto x(i)$, that is, on input i, we ouput the ith symbol of x (now viewed as a string). Instead of outputting the whole string, we compute x locally. This idea will be crucial.

Definition 4.1 *An $(n,k)_q$-code C is an injective map $[q]^k \to [q]^n$ where n, k, and q are positive integers with $n \geq k$ and $q \geq 2$.*

The domain $\operatorname{dom} C$ of C is called the set of *messages*, the image $\operatorname{im} C$ of C is called the set of *code words*. Given $x, y \in [q]^n$, the *Hamming distance* $\delta(x,y)$ is the number of positions in which x and y differ, i.e., $\delta(x,y) = |\{i \mid x(i) \neq y(i)\}|$. Their *relative Hamming distance* is $\Delta(x,y) = \Pr_{i \in \{1,\ldots,n\}}[x(i) \neq y(i)]$. Note that $\delta(x,y)/n = \Delta(x,y)$. The *(minimum) distance* $\delta(C)$ and *relative (minimum) distance* $\Delta(C)$ of a code C is the minimum of $\delta(x,y)$ and $\Delta(x,y)$, respectively, taken over all pairs of code words $x \neq y$. If $q = 2$, then the code is called *binary*.

Definition 4.2 *An $(n,k)_q$-code C is called* (ϵ,ℓ)-list decodable *if for every $r \in [q]^n$, there are $\leq \ell$ code words $c \in \operatorname{im} C$ such that $\Delta(r,c) \leq 1 - 1/q - \epsilon$.*

The definition above essentially states that if C is (ϵ,ℓ)-list decodable, then at most ℓ codewords agree with any word r in a fraction of at least $\epsilon + 1/q$ of the coordinates. In other words, if there is a word r (the "received word") that is disturbed in at most $(1 - 1/q - \epsilon) \cdot n$ places, then there are at most ℓ candidates that can be the original message. Classically, one considers the case $\ell = 1$. List decoding allows to correct more errors than in the classical setting. However, list decoding is only meaningful if ℓ is nontrivially bounded, for instance by a polynomial.

The parameter ϵ should be between zero and $1 - 1/q$. The smaller ϵ the better the code. One cannot correct more than $(1 - 1/q)n$ errors (for any meaningful notion of correction). This is due to the fact that a random code word agrees with any code word in roughly $1/q \cdot n$ letters.

We will use codes to transform a language L that cannot be computed by subexponential circuits into a language L' that cannot be approximated by subexponential circuits. To do so, we want to encode the truth table of the characteristic function of L_n. Note that the size of the messages and the code words are exponential, thus we cannot compute with them directly, at least not efficiently. However, it is sufficient to compute only one entry of the truth table at a time. This is achieved by locally decodable codes.

We are looking for an infinite family of codes $[q]^k \to [q]^n$, one for every message length k. It should be uniformly constructible, efficiently encodable, and efficiently list decodable. The decoding procedure will be randomized.

Definition 4.3 *A probabilistic Turing machine M computes a function f at some x if $M(x,y) = f(x)$ for at least a fraction of $3/4$ of all random strings y. We say that M has* agreement α *with f if $\Pr_y[M(x,y) = f(x)] \geq \alpha$ for all x.*

Remark 4.4 *We can also do probability amplification when computing functions. We do k iterations and return the function value that appeared most often. The error probability drops down exponentially with k.*

Sudan, Trevisan, and Vadhan now define a class of codes which they call *nice*. Nice codes are sufficient to perform the worst case to average case reduction.

Definition 4.5 *A family of codes $C_{k,\epsilon}$ is* nice *if there exist functions $n, q, \ell : \mathbb{N} \times \mathbb{Q} \to \mathbb{N}$ and Turing machines* Enc *and* Dec *such that the following holds:*

1. *For all k and ϵ, $C_{k,\epsilon} : [q]^k \to [q]^n$ is (ϵ,ℓ)-list decodable, where $n = n(k,\epsilon) = \operatorname{poly}(k, 1/\epsilon)$, $q = q(k,\epsilon) = \operatorname{poly}(k, 1/\epsilon)$, and $\ell = \ell(k,\epsilon) = \operatorname{poly}(\log k, 1/\epsilon)$.*

2. Enc *is a deterministic Turing machine that on input* x, k, ϵ, *computes* $C_{k,\epsilon}(x)$ *and runs in time* $\mathrm{poly}(n) = \mathrm{poly}(k, 1/\epsilon)$.

3. Dec *is a probabilistic oracle Turing machine such that* Dec *with oracle* r *(note that be can interpret binary strings a characteristic functions) computes a list of encodings of probabilistic oracle Turing machines* $M_1, \ldots, M_\ell$ *in time* $\mathrm{poly}(\log k, 1/\epsilon)$. *The running time of each* M_λ *is bounded by* $\mathrm{poly}(\log k, 1/\epsilon)$. *For every message* $x \in [q]^k$ *with* $\Delta(r, C_{k,\epsilon}(x)) \leq 1 - 1/q - \epsilon$, *there exists an index* λ *such that* M_λ^r *computes* x *(viewed as a function, i.e.,* $M_\lambda^r(i) = x(i)$ *for all* $1 \leq i \leq n$*).*

Note that the upper bound on $\ell(k, \epsilon)$ also follows from the running time bound of Dec, since the length of its output is at least $\ell(k, \epsilon)$.

The uniformity condition and the encoding condition are standard. More interesting is the decoding procedure. It has a lot of non-standard features. First, Dec is expected to give the whole list of up to ℓ code words instead of returning just one. Second, the decoding procedure has running time polynomial in $\log k$ and $1/\epsilon$. This would be impossible, if the input r would be written on the input tape of D, since we need $n \geq k$ steps to read the received word. Therefore, the code word is given to Dec as an oracle and Dec may query single bits of r via the oracle tape. Since we cannot even write down the whole code word in time $\mathrm{poly}(\log k)$, we output a short description in form of ℓ encodings of Turing machines, each computing a message whose image is close to the received word. Such a behaviour is also called *locally decodable codes*: We can compute a bit of the original message by looking at only $\mathrm{poly}(\log k)$ many bits from the received word. Third, we allow that the decoding procedure itself as well as the machines $M_1, \ldots, M_\ell$ describing the output of Dec are probabilistic Turing machines.

4.2 Nice codes allow worst case to average case reduction

Nice codes are nice because nice binary codes solve the worst case to average case problem.

Theorem 4.6 *Let* $C_{k,\epsilon}$ *be a nice family of binary codes. There exists a constant* c *such that the following holds: For every function* $f : \{0,1\}^m \to \{0,1\}$ *and for every* $\epsilon > 0$, *there is no circuit of size* $s' = (\frac{\epsilon}{m})^c \cdot \mathrm{size}(f)$ *that computes the function* $f' : \{0,1\}^{m'} \to \{0,1\}$ *defined by* $f' = C_{2^m,\epsilon}(f)$ *correctly on more than a fraction of* $1/2 + \epsilon$ *of the inputs.*

Before we prove the result above, one should first explain the notation $C_{2^m,\epsilon}(f)$. By interpreting f as a bit string of length 2^m, we can apply $C_{2^m,\epsilon}$ to this string. We get a new string which we then can again interpret as a

Boolean function $\{0,1\}^{\log n(2^m,\epsilon)} \to \{0,1\}$. ($\ell$, k, and n are the parameters in the Definition above.)

Proof. The proof is by contradiction. Let $k = 2^m$ and $s = \text{size}(f)$. Assume that there is a circuit B of size $s' = (\frac{\epsilon}{m})^c \cdot s$ that computes f' correctly on a fraction of more than $\frac{1}{2} + \epsilon$ of the inputs. By definition, $\text{Dec}^B(k,\epsilon)$ outputs ℓ encodings of Turing machines $M_1, \ldots, M_\ell$ such that for some λ, M_λ^B computes f correctly (in the sense of Definition 4.3). Dec^B and M_λ^B here means that the Turing machines have oracle access to the function computed by B (abuse of notation).

The running times of $M_1, \ldots, M_\ell$ are $\text{poly}(\log k, 1/\epsilon) \leq (\frac{m}{\epsilon})^{c'}$ for some constant c'. As in the proof that $\mathsf{BPP} \subseteq \mathsf{P/poly}$, we can simulate M_λ by a deterministic circuit C by amplifying the acceptance probability and choosing a random string that is good for all inputs. The size of C is bounded by $(\frac{m}{\epsilon})^{c''}$ for some constant c''. This circuit will be an oracle circuit, i.e, it has so-called oracle gates that correspond to oracle calls of M_λ. The output of the oracle gate is one if B accepts the input and zero otherwise.

Finally, we replace every oracle query by a copy of B. This yields a circuit of size $(\frac{m}{\epsilon})^{c''} \cdot s'$. If we set $c'' = c + 1$, we end up with a circuit for f whose size is smaller than $\text{size}(f)$, a contradiction. (Note that the constants c' and c'' only depend on the family of codes.) ∎

Exercise 4.1 *Give a formal proof that the probabilistic circuit in the proof of Theorem 4.6 can be turned into a deterministic one.*

By setting the parameters in Theorem 4.6 in the right way, we get a proof of Theorem 3.5: Let $L \in \mathsf{E}$ be a language sucht that $\text{size}(L_m) \geq 2^{\delta m}$ for almost all m. Let f be the characteristic function of L_m. We have $s = 2^{\delta m}$. We set $\gamma = \frac{1}{3c}$ and $\epsilon = 1/s^\gamma$. The resulting function f' has the following parameters: The input length is

$$m' = \log n(2^m, \epsilon) = \log \text{poly}(2^m, 2^{\gamma m}) = \Theta(m).$$

No circuit of size at most

$$s' = \left(\frac{1}{m \cdot 2^{\delta m/(3c)}}\right)^c \cdot 2^{\delta m} \geq 2^{\frac{1}{2}\delta m}$$

can compute f' correctly on a fraction more than $1/2 + 1/s^\gamma = 1/2 + 2^{-\gamma\delta m}$. In particular, for any circuit D of size $\leq 2^{\gamma\delta m}$ (note that $\gamma < 1/2$), we have

$$\Pr_{x \in \{0,1\}^{m'}}[D(x) = f'(x)] \leq 1/2 + 2^{-\gamma\delta m}$$

Thus $\text{H}(f') \geq 2^{\gamma\delta m}$. Since $m' = \Theta(m)$, $\text{H}(f')$ is also linear exponential in m'.

For every m, this process defines a Boolean function f' and henceforth a subset of $\{0,1\}^{m'}$. We define the language L' to be the union of these subsets. There might be some input lengths for which there are no words in L. We fill these lengths by padding. By the considerations above,

$$\mathrm{H}(L'_{m'}) \geq \Omega(2^{\gamma\delta m}) \geq 2^{\delta' m}$$

for some δ' and almost all m. (The Ω is introduced by the padding.)

To complete the proof of Theorem 3.5, it remains to show that $L' \in \mathsf{E}$. Let M be an E-machine for L. Given $x \in \{0,1\}^{m'}$, we can decide whether x in L' as follows. First we compute the characteristic function of L_m (where m corresponds to m') and write it down as a bit string. This takes time $2^{O(m)} \cdot 2^{O(m)} = 2^{O(m)}$. Then we encode x. The encoding can be computed in time $\mathrm{poly}(2^{O(m)}) = 2^{O(m)}$. To see whether x in L', we simply have to check the appropriate position of $C_{2^m,\epsilon}(x)$. This completes the proof of Theorem 3.5 (provided that nice binary codes exist).

4.3 Nice codes exist

We prove the existence of nice binary codes in two steps: We first show that there are nice codes over an alphabet whose size grows with the size of the code. These codes are based on multivariate polynomials. Then we make the code binary by concatenating it with a Hadamard code.

4.3.1 Outer code—polynomial code

Lemma 4.7 *There exists a nice family of codes with* $q(k,\epsilon) = \mathrm{poly}(\log k, 1/\epsilon)$, $n(k,\epsilon) = \mathrm{poly}(k)$, *and* $\ell(k,\epsilon) = O(1/\epsilon)$.

Proof. The encoding works as follows: We interpret a message as the values of a multivariate polynomial over a finite field F, evaluated at a specified set of points (to be determined later). The encoding of the message is the polynomial evaluated at *all* possible points. This creates the redundancy necessary for error correction.

We have to choose the following parameters: the number of variables m, the field F, and a subset $H \subseteq F$. H^m is the set of points one which the m-variate polynomial is specified.

Given k and ϵ, we choose parameters as follows. Our field F will have size roughly $(c \log k)^2/\epsilon^3$ for some constant c that will be determined later. By "roughly" we mean that we choose the next prime greater than the stated bound. We can find such a prime trivially in time $\mathrm{poly}(\log k, 1/\epsilon)$. (Note that between x and $2x$, there is always a prime number by Bertrand's postulate.) $H \subseteq F$ is a subset of cardinality $(\log k)/\epsilon$. We set $m = (\log k)/(\log |H|)$. Then $|H|^m \geq k$. The size of the alphabet will be $q = |F|$ and we identify $[q]$ with F (in the canonical way).

Let $b : [k] \to H^m$ be any injective map such that b and its inverse is computable in time $\text{poly}(k, 1/\epsilon)$. To encode a string $x \in F^k$, we first compute an m-variate polynomial p over F of degree $\leq |H| - 1$ in each variable such that $p(b(i)) = x(i)$ for all $i \in [k]$. Such a polynomial p exists, it has total degree $\leq |H|^m = k$, and can be easily computed in time $\text{poly}(k, 1/\epsilon)$. It can be made unique by requiring $p(z) = 0$ for all $z \notin H^m \setminus \text{im}\, b$.

Next we set $n = |F|^m$ and identify $[n]$ with F^m. The code word corresponing to the message x is now the function $p : [n] \to F$. Note that this is merely a table of the values of p at all points in F^m. We have

$$\log n = m \log |F| = \frac{\log k \cdot \log |F|}{\log |H|} = O(\log k),$$

because $\log |F| = \Theta(\log |H|)$. Thus, $n = \text{poly}(k)$. Furthermore, $q = |F| = \text{poly}(\log k, \epsilon)$.

The codes constructed this way are uniformly constructible and there is an efficient encoding procedure as outlined above (see also Exercise 4.2). To obtain an efficient decoding procedure, it is sufficient to solve the following *polynomial reconstruction problem*: Given oracle access to a function $r : F^m \to F$, find a list of (short descriptions of) all polynomials of total degree $d = m|H|$ that agree with r on a fraction of at least $\epsilon + 1/|F|$ of the inputs. Theorem 4.8 gives a solution to this problem provided that $\epsilon + 1/|F| \geq c\sqrt{d/|F|}$ for some constant c. We have

$$\frac{d}{|F|} \leq \frac{m|H|}{|F|} \leq \frac{(\log k)^2/\epsilon}{(c \log k)^2/\epsilon^3} = \frac{\epsilon^2}{c^2}.$$

Thus the requirement of Theorem 4.8 is met. The algorithm given there runs in time $\text{poly}(m, d, \log |F|, 1/\epsilon) = \text{poly}(\log k, 1/\epsilon)$. It produces a list of at most $\ell = O(1/\epsilon)$ code words. This list is given as a list of oracle Turing machines whose running times are $\text{poly}(m, d, \log |F|, 1/\epsilon) = \text{poly}(\log k, 1/\epsilon)$. ■

Exercise 4.2 *Consider the proof of Lemma 4.7.*

1. *Show that for every function $a : H^m \to F$, there is a polynomial L of degree $\leq |H|$ in every variable such that $L(x) = a(x)$ for all $x \in H^m$. This polynomial is unique.*

2. *Describe an efficient implementation of the encoding procedure.*

The following theorem solves the polynomial reconstruction problem. We will prove it at the end of this section.

Theorem 4.8 *Let F be a finite field. There exists a constant c such that given a function $r : F^m \to F$, there is a PTM that computes in time*

$\mathrm{poly}(m, d, \log|F|, 1/\epsilon)$ *a list of* $\ell = O(1/\epsilon)$ *oracle TM* $M_1, \ldots, M_\ell$ *with running times* $\mathrm{poly}(m, d, \log|F|, 1/\epsilon)$ *such that for each polynomial* $p : F^m \to F$ *that has degree* d *and has agreement* ϵ *with* r*, there exists a* j *such that* M_j^r *computes* p*, provided that* $\epsilon > c\sqrt{d/|F|}$.

4.3.2 Inner code—Hadamard code

We convert the code of Lemma 4.7 by concatenating it with the *Hadamard code*. Given a string $z \in \{0,1\}^k$, the Hadamard code $\mathrm{Had}_k : \{0,1\}^k \to \{0,1\}^{2^k}$ is defined as follows: Think of the coordinates of $\{0,1\}^{2^k}$ indexed by the elements from $\{0,1\}^k$. Then for $y \in \{0,1\}^k$, the yth coordinate is defined as the scalar product $\langle y, z\rangle = \sum_{i=1}^k y_i z_i \mod 2$. Here y_i and z_i denote the entries of y and z, respectively. The Hadamard code has an exponential blowup in code word length. This is in general bad, but does not matter in the following. On the other hand, the Hadamard code has nice list-decoding properties.

Lemma 4.9 *For any* $(n,k)_q$*-code with minimum distance* $d = (1-1/q)(1-\tau)n$ *and for any received word* $r \in [q]^n$*, there are at most* $(1-\tau)/(\gamma^2-\tau)$ *code words* c *with* $\delta(c,r) \le \alpha$ *where* γ *is defined by* $q\alpha/(q-1) = (1-\gamma)n$*, provided* $\gamma > \sqrt{\tau}$.

Proof. Let $r \in [q]^n$ be the received word and let $c_1, \ldots, c_m$ be the set of all code words with $\delta(c_\mu, r) \le \alpha$. W.l.o.g. we may assume that $r = qq\cdots q$. Let $\alpha_\mu = \delta(c_\mu, r)$ and $\bar\alpha = \frac{1}{m}\sum_{\mu=1}^m \alpha_\mu$ be their average distance from r. Obviously, $\bar\alpha \le \alpha$.

Let $e_1, \ldots, e_q$ be the unit vectors of $\mathbb{R}^q$. We associate $i \in [q]$ with e_i. Given a code word $c \in [q]^n$, we associate with it the vector in $\mathbb{R}^{qn}$ that is obtained by sending every letter j of c to the vector e_j and concatenating these n vectors in $\mathbb{R}^q$ to get one big vector of $\mathbb{R}^{qn}$. The vectors in $\mathbb{R}^{qn}$ get a block structure via this construction: There are n blocks of size q, each refering to a position in the code words. By abuse of notation, we also call the vectors in $\mathbb{R}^{qn}$ associated with r and $c_1, \ldots, c_m$ again r and $c_1, \ldots, c_m$.

We will now estimate

$$S = \sum_{1 \le j,k \le m} \langle c_j - r, c_k - r\rangle. \tag{4.1}$$

We start with a lower bound. Each scalar product in (4.1) can be written as the sum of n scalar products, one for each block. For the pth such block, let N_p denote the number of vectors $c_j - r$ that are nonzero on the pth block. For $1 \le \beta < q-1$, let $N_{p,\beta}$ denote the number of those vectors $c_j - r$ whose pth block is of the form $(\underbrace{0,\ldots,0}_{\beta-1}, 1, \underbrace{0,\ldots,0}_{\beta-q-1}, -1)$. Obviously,

$N_p = N_{p,1} + \cdots + N_{p,q-1}$. Since the contributions of two vectors $c_j - r$ and $c_k - r$ to the pth block is

$$\begin{cases} 0 & \text{if } c_j \text{ or } c_k \text{ has the same symbol as } r \text{ in position } p \\ 1 & \text{if } c_j \text{ and } c_k \text{ and } r \text{ have pairwise different symbol in position } p \\ 2 & \text{if the symbols of } c_j \text{ and } c_r \text{ are the same} \\ & \text{but different from the symbol of } r, \end{cases}$$

the contribution to S from the q positions of the pth block is

$$N_p^2 + \sum_{\beta=1}^{q-1} N_{p,\beta}^2 \geq \frac{q}{q-1} N_p^2.$$

The last inequality follows from the fact that the sum of the squares of t numbers is at least $1/t$ times the square of their sum.

We have $\sum_{p=1}^{n} N_p = \sum_{j=1}^{m} \alpha_j = m\bar{\alpha}$. Hence $\sum_{p=1}^{n} N_p^2 \geq (m\bar{\alpha})^2/n$. Therefore

$$S \geq \sum_{p=1}^{n} \left(N_p^2 + \sum_{\beta=1}^{q-1} N_{p,\beta}^2 \right) \geq \frac{q}{q-1} \sum_{p=1}^{n} N_p^2 \geq \frac{q}{q-1} \cdot \frac{m^2 \bar{\alpha}^2}{n}. \tag{4.2}$$

For the upper bound on S, fix a pair of vectors $c_j - r$ and $c_k - r$. If $j = k$, then

$$\langle c_j - r, c_k - r \rangle = 2\alpha_j, \tag{4.3}$$

since each block where c_j and r differ contributes 2 to the scalar product. Let $d_{j,k} = \delta(c_j, c_k)$. If $j \neq k$, then

$$\begin{aligned} \langle c_j - r, c_k - r \rangle &= \langle c_j, c_k \rangle + \langle r, r \rangle - \langle c_j, r \rangle - \langle v_k, r \rangle \\ &= n - d_{j,k} + n - (n - \alpha_j) - (n - \alpha_k) \\ &= \alpha_j + \alpha_k - d_{j,k} \\ &\leq \alpha_j + \alpha_k - d \end{aligned} \tag{4.4}$$

(4.3) and (4.4) imply

$$S \leq 2m^2\bar{\alpha} - m(m-1)d. \tag{4.5}$$

From (4.2) and (4.5), we obtain

$$m^2 \left(\frac{q}{q-1} \cdot \frac{\bar{\alpha}^2}{n} - 2\bar{\alpha} - d \right) \leq md$$

thus

$$m \leq d \left(\frac{q}{q-1} \cdot \frac{\bar{\alpha}^2}{n} - 2\bar{\alpha} + d \right)^{-1}, \tag{4.6}$$

provided that the expression in the parentheses is positive. Consider this expression in (4.6) as a function in $\bar{\alpha}$. This function is decreasing in $\bar{\alpha}$ (as long as $\bar{\alpha} \leq n$). Since $\bar{\alpha} \leq \alpha$, we may replace $\bar{\alpha}$ by α. We can get an upper bound on m via (4.6) as long as

$$\begin{aligned}
&\frac{q}{q-1} \cdot \frac{\alpha^2}{n} - 2\alpha + d > 0 \\
&\iff \left(\frac{q}{q-1}\alpha\right)^2 - 2 \cdot \frac{q}{q-1}\alpha \cdot n > -dn\frac{q}{q-1} \\
&\iff \left(n - \frac{q}{q-1}\alpha\right)^2 > n^2 - dn\frac{q}{q-1} \\
&\iff (\gamma n)^2 > n(n - \frac{q}{q-1}d) \\
&\iff \gamma^2 > 1 - \frac{q}{q-1} \cdot \frac{d}{n} \\
&\iff \gamma^2 > \tau.
\end{aligned}$$

That means, if $\gamma > \sqrt{\tau}$, we get

$$m \leq \frac{1-\tau}{\gamma^2 - \tau}$$

by (4.6) after replacing $\bar{\alpha}$ by α, since $d = \frac{q-1}{q}(1-\tau)n$ and $\alpha = \frac{q-1}{q}(1-\gamma)n$. ■

This lemma is a very useful tool to prove that a certain code is list decodable. Consider the Hadamard code. The size of the alphabet q is two. The minimum distance of the code is $\frac{1}{2}n$. This is seen as follows: Given two messages x and y, the probability that $\langle x - y, z\rangle = 0$ is exactly $1/2$, if z is choosen at random.

We get the following corollary to Lemma 4.9. Note that for the Hadamard code, $\tau = 0$ and $\gamma n = n - 2\alpha$. If we set $\alpha = (1 - (\epsilon + \frac{1}{2}))n$, then $\gamma = 2\epsilon$.

Corollary 4.10 *For every k and every $\epsilon > 0$, Had_k is $(\epsilon, \frac{1}{4\epsilon^2})$-list decodable.*

There exist efficient list-decoding algorithms for the Hadamard code [**?**]. Here it is enough to uses the trivial algorithm that checks all possible 2^k code words and runs in time $\mathrm{poly}(2^k)$.

4.3.3 Concatenation

Theorem 4.11 *There exists a nice family of binary codes with $\ell = \mathrm{poly}(1/\epsilon)$.*

Proof. Let C be the family of codes constructed in Lemma 4.7. We obtain a nice family of binary codes C' by *concatenating* C with the Hadamard code.

Given k and ϵ, we set $\epsilon' = \epsilon^3/4$. Let n, k, and ℓ be the parameters of $C_{k,\epsilon'}$ Let $t = \log q$ and let $b : [q] \to \{0,1\}^t$ be an injective map. Given $z \in [q]$, $\mathrm{Had}(z)$ shorthands $\mathrm{Had}_t(b(z))$.

Our encoding scheme works as follows: Given $x \in \{0,1\}^k$, we first encode x using $C_{k,\epsilon'}$. Let $y \in [q]^n$ be the resulting code word. Then we interpret each of the n entries of y as an element of $\{0,1\}^t$ and encode each of the n entries of y using the Hadamard code, i.e.,

$$C'_{k,\epsilon}(x) = \mathrm{Had}(y(1))\,\mathrm{Had}(y(2))\cdots\mathrm{Had}(y(n)) \qquad \text{where } y = C_{k,\epsilon'}(x).$$

(This is a so-called *concatenated code*. The Hadamard code plays the role of the *inner code*, C is called the *outer code*.) Algorithm 3 describes the encoding procedure. The length of the code is $n' = n \cdot 2^t = \mathrm{poly}(k, 1/\epsilon)$. The encoding procedure is also polynomial time, since $\mathrm{Had}(y(\nu))$ can be computed in time $\mathrm{poly}(2^t) = \mathrm{poly}(k, 1/\epsilon)$.

There is a natural way to decode a concatenated code. First decode each "symbol" of the inner code using the corresponding decoding procedure. Then decode the outer code with its decoding procedure. In a classical setting, that is, we have unique decoding and want to decode the whole received word explicitly, this works well. In our case, we have to overcome some problems:

- Right now, we do not have specified a decoding procedure for the Hadamard code.
- We have to deal with the fact that input and output are implicit.
- When decoding the inner code, we do not get a one code word but a list of code words.

Algorithm 4 describes the decoding process. Given k and ϵ and an oracle for the received word $r : [n] \times [2^t] \to \{0,1\}$, we get oracles $r'_1, \ldots, r'_{1/\epsilon^2} : [n] \to [q]$ as follows: For a given input $i \in [n]$, we define an oracle $r|_i : [2^t] \to \{0,1\}$ by $r|_i(j) = r(i,j)$. We compute a list of all $z \in [q]$ such that $\mathrm{Had}(z)$ has agreement of at least $1/2 + \epsilon/2$ with $r|_i$. By Corollary 4.10, this list consists of at most $1/\epsilon^2$ elements. The oracle r'_μ on query i outputs the μth element of this list. (To do so, we order the list arbitrarily.) Then we call the decoding procedure for $C_{k,\epsilon'}$, once for each oracle of $r'_1, \ldots, r'_{1/\epsilon^2}$. Then we output the union of all $1/\epsilon^2$ lists. Its length is $\ell/\epsilon^2 = \mathrm{poly}(1/\epsilon)$.

The running times of the computed TMs is $\mathrm{poly}(\log k, 1/\epsilon)$. To compute the oracles r'_μ from r, one has to try all code words of the Hadamard code. Since there are only q, everything is fine. For the same reason, the time used by the decoding procedure for C' is also $\mathrm{poly}(\log k, 1/\epsilon)$.

It remains to show the correctness of the algorithm: Let x be a message such that $C'_{k,\epsilon}(x)$ has agreement $1/2 + \epsilon$ with the received word r. Let

Algorithm 3 Encoding procedure

Input: Message $x \in [q]^k$, $\epsilon > 0$

Output: Code word $\in \{0,1\}^{n2^t}$ where $t = \log q$.

1: Let C be the family of codes of Lemma 4.7.
2: Choose a injective map $b : [q] \to \{0,1\}^t$.
3: Let $y = C_{k,\epsilon'}(x)$ where $\epsilon' = \epsilon^3/4$.
4: Return $\mathrm{Had}_t(b(y(1)))\,\mathrm{Had}_t(b(y(2)))\cdots\mathrm{Had}_t(b(y(n)))$.

Algorithm 4 Decoding procedure

Input: Received word $r : [n] \times [2^t] \to \{0,1\}$ as oracle, $\epsilon > 0$

Output: TMs $M_1, \ldots, M_{\ell/\epsilon^2}$ such that for all code words x with $\Delta(C'_{k,\epsilon}(x), r) \leq 1 - (1/2 + \epsilon)$, there is a j such that M_j^r computes x.

1: For $i \in [n]$, define $r|_i$ by $r|_i(j) = r(i,j)$.
2: We define "oracles" $r'_1, \ldots, r'_{1/\epsilon^2} : [n] \to [q]$ as follows: On input i, r'_μ computes the μth $z \in [q]$ with $\Delta(\mathrm{Had}_t(b(z)), r|_i) \leq 1 - (1/2 + \epsilon/2)$. This is done by simply checking all possible code words z.
3: The list of Turing machines is obtained as follows: For $1 \leq \mu \leq 1/\epsilon^2$, compute $\mathrm{Dec}^{r'_\mu}(k, \epsilon')$ where Dec is the decoding procedure of C. {Formally the list of Turing machines require the oracle r'_μ to run properly. However given r as an oracle and i as an input, it is easy to simulate r'_μ. Furthermore, the Turing machines output elements from $[q]$ and not bits from $\{0,1\}$. This can be achieved by selecting the approriate bit given by the second parameter of the input after applying b^{-1}.}

$y = C_{k,\epsilon'}(x)$. By Markov's inequality, for a fraction of at least $\epsilon/2$ of the $i \in [n]$, $r|_i$ has at least agreement $1/2 + \epsilon/2$ with $\mathrm{Had}(y(i))$. (Let's quickly do this argument: If this were not true, then the agreement with r would be at most $(1-\epsilon/2)\cdot(1/2+\epsilon/2)+\epsilon/2\cdot 1 < 1/2+\epsilon$, a contradiction.) Therefore, $r|_i = r'_m(i)$ for some m and i. Since there are only $1/\epsilon^2$ choices for m, there exists an m_0 such that $r'_{m_0}(i) = r|_i$ for at least a fraction of $\epsilon/2 \cdot \epsilon^2 = \epsilon^3/2$ of the indices $i \in [n]$. Since $1/q + \epsilon' \leq 2\epsilon' = \epsilon^3/2$, the agreement of r'_{m_0} with y is large enough to decode the outer code. The decoding algorithm for $C_{k,\epsilon'}$ will produce a list of $\ell = \mathrm{poly}(1/\epsilon)$ oracles, one of which is x. ■

Bibliographic notes

The proof presented in this chapter is due to Madhu Sudan, Luca Trevisan, and Salil Vadhan [**?**]. Theorem 4.8 was first proven by Sanjeev Arora and Madhu Sudan [**?**], however with worse parameters. The version given here is shown by Madhu Sudan, Luca Trevisan, and Salil Vadhan [**?**]. The proof of the Lemma 4.9 is due to Venkatesan Guruswami and Madhu Sudan [**?**].

5 The polynomial reconstruction problem

When we constructed nice codes, we postponed the solution of one problem, the polynomial reconstruction problem.

Input: function $f : F^m \to F$ (as an oracle)
$d \in \mathbb{N}$ and $\epsilon \in \mathbb{R}$
Goal: PTMs $M_1, \ldots, M_\ell$ such that for every polynomial $p \in F[X_1, \ldots, X_m]$ of degree d that has agreement ϵ with f
there is an index j such that M_j^f computes p.

In this chapter, we will prove Theorem 4.8. As a first step, we show that we do not need to compute the polynomial p but it is sufficient to approximate it, because we can "correct" multivariate polynomials.

5.1 A correction procedure

If we have high agreement, say $\frac{9}{10}$, then there is only one polynomial p close to f. In this case, we can "correct" f to get p. We first reduce the multivariate case to the univariate case.

Definition 5.1 *Let $f : F^m \to F$ and $x, t \in F^m$.*

1. *The* line *$\ell_{x,t}$ is the set of all points $\{(1-y)x + yt \mid y \in F\}$.*
2. *The* restriction *of f to $\ell_{x,t}$ is the function $f|_{\ell_{x,t}} : F \to F$ defined by $x \mapsto f(\ell_{x,t}(x))$.*

First assume that we know that $f|_{\ell_{x,t}}$ and $p|_{\ell_{x,t}}$ differ in $\leq k$ positions. Note that $\hat{p} := p|_{\ell x,z}$ is a univariate polynomial of degree at most d. Let $\hat{f} := f_{\ell_{x,t}}$. We want to prove that if $|F| \geq 2k+d+2$, then we can reconstruct $\hat{p}$ from the values $\hat{f}_1, \ldots, \hat{f}_n$, where $\hat{f}_i = \hat{f}(a_i)$ for $n = 2k + d + 2$ distinct points a_i in F.

Lemma 5.2 *There is a polynomial E that has degree $\leq k$ and is zero on the set $S = \{b \mid \hat{f}(b) \neq \hat{p}(b)\}$*

Proof. $E(X) = \prod_{b \in S}(X - b)$ is zero exactly on S. By assumption, $|S| \leq k$, hence $\deg E \leq k$. ∎

Definition 5.3 *A polynomial E as in the lemma above is called a* error locator polynomial.

Exercise 5.1 *Let $N(X) = E(X) \cdot \hat{p}(X)$. Verify the following claims:*

1. $E \neq 0$.
2. $\deg N \leq d + k$.
3. $N(a) = E(a)\hat{f}(a)$ *for all* $a \in F$.
4. $\hat{p} = N/E$.

Theorem 5.4 *Given $\hat{f}_1, \ldots, \hat{f}_n$, we can find in time* $\mathrm{poly}(d, k, \log |F|)$ *two polynomials E_0 and N_0 such that*

1. $\deg E_0 \leq k$,
2. $\deg N_0 \leq d + k$,
3. $N_0(a_i) = E_0(a_i) \cdot \hat{f}_i$ *for* $i = 1, \ldots, n$.

Proof. $N_0(a_i) = E_0(a_i)$ are $2d + k + 2$ homogeneous linear equations in $(d + k + 1) + (k + 1) = 2k + d + 2$ unknowns. We can find a solution in time polynomial in d, k, and $\log |F|$. Now it is easy to verify that for any solution N_0 and E_0, we have $N_0/E_0 = P$. ∎

Exercise 5.2 *Show this last claim in the proof above.*

Theorem 5.5 *Let $f : F^m \to F$ be a function that has agreement $\frac{9}{10}$ with some polynomial $p \in F[X_1, \ldots, X_m]$ of degree d. There is a PTM C that given d, m and f (as oracle) such that C^f computes p in time* $\mathrm{poly}(d, m, \log |F|)$ *provided that $|F| > 2d + 3$.*

Proof. Assume we want to compute $p(x)$. C randomly chooses $r \in F$. For how many choices of r does some $u \in F^m$ lie on $\ell_{x,r}$? $u \in \ell_{x,r}$ means $yr = u - (1 - y)x$. This gives $|F| - 1$ solutions for y. Thus every $u \neq x$ appears on exactly $|F| - 1$ many lines. The average agreement on a line is therefore $\frac{9}{10} - \frac{1}{|F|}$. Now with probability $3/4$, the agreement on a random line through x is at least $3/4$. (If $3/4$ of the lines have aggreement less than $3/4$, we can only have a total agreement of $\frac{9}{16} + \frac{1}{4} < \frac{9}{10} - \frac{1}{|F|}$) for sufficiently large F. Thus f and p differ in at most $|F|/4$ places on such a line. Now we can use Theorem 5.4 to reconstruct p on this line and in particular $p(x)$, if $2|F|/4 + d + 2 \leq |F|$ which is equivalent to $d + 2 \leq |F|/2$. This is exactly our assumption. ∎

5.2 Low agreement

In this section, we treat the case when we have only small agreement, that is, $c \cdot \sqrt{d/|F|}$. Now we can have several polynomials that have this agreement with the given f. Again we first treat the univariate case and then reduce the multivariate case to it.

5.2.1 Univariate polynomial

Lemma 5.6 *For every set of pairs $(x_1, y_1), \ldots, (x_s, y_s) \in F^2$ and degrees d_X, d_Y with $d_X \cdot d_Y \geq s$ there is a bivariate polynomial $Q(X, Y)$ with degree in X bounded by d_X and in Y bounded by d_Y such that $Q(x_\sigma, y_\sigma) = 0$ for all $1 \leq \sigma \leq s$. Q can be constructed in polynomial time.*

Proof. Let $Q = \sum_{i=0}^{d_X} \sum_{j=0}^{d_Y} a_{i,j} X^i Y^i$ and consider the coefficients as unknowns. These coefficients have to fulfill s homogeneous linear equations $Q(x_\sigma, y_\sigma) = 0$. We have $(d_X + 1)(d_Y + 1) > s$ many unknowns, thus, there is always a nontrivial solution. One such solution can be easily found by solving the linear equations. ■

Lemma 5.7 *Let $t > d_X + d \cdot d_Y$. If a degree d polynomial p fulfills $p(x_\sigma) = y_\sigma$ for at least t pairs, then $(Y - p(X))|Q$.*

Proof. $Q(x_\sigma, y_\sigma) = 0$ holds for all σ by construction. Thus $Q(x_\sigma, p(x_\sigma)) = 0$ for at least t pairs. Thus, the univariate polynomial $Q(X, p(X))$ has at least t roots. The degree of $Q(X, p(X))$ is $\leq d_X + d \cdot d_Y$. Thus $Q(X, p(X))$ is identically zero. By Gauß' Lemma, $Y - p(X)$ divides Q. ■

Corollary 5.8 *There are at most d_Y polynomials $p(X)$ of degree d, that describe more than $d_X + d \cdot d_Y$ pairs.*

This yields the following algorithm for the univariate polynomial reconstruction problem.

Input: $f : F \to F$ given as an oracle.
Output: all univariate polynomials of degree d with agreement $2\sqrt{d/|F|}$.
(Since p is univariate, we can output an explicit list of coefficients.)

1. Compute Q with $d_X = \sqrt{|F| \cdot d}$ and $d_Y = \sqrt{|F|/d}$ using the pairs $(x, f(x))$, $x \in F$.
   $(d_X \cdot d_Y \geq |F|)$
2. Find all factors of the form $Y - p(X)$.
   (There are efficient randomized algorithms for bivariate polynomial factorization.)
3. For each such p check whether $f(x) = p(x)$ for at least $2\sqrt{d/|F|} \cdot |F| = 2\sqrt{d \cdot |F|}$ many x. If yes, output p.

For the correctness, note that that $d_X + d \cdot d_Y = 2\sqrt{d \cdot |F|} = 2\sqrt{d/|F|} \cdot |F|$. Thus, by Lemma 5.7, the algorithm will find all polynomials with agreement at least $2\sqrt{d/|F|}$.

5.2.2 Multivariate polynomials

We assume that $d/|F|$ is smaller than some small constant to be chosen later. This is no restriction for our applications. Again we want to reduce the multivariate case to the univariate one. But now, we just have low agreement. In the case of high agreement, there was only one polynomial close to f restricted to a line. In the case of low agreement, there are more, roughly $\sqrt{|F|/d}$. When we now want to compute $p(x)$ from f and x, we have to make sure that we always take the right univariate polynomial from the list $q_1, \ldots, q_s$ obtained via the algorithm above. Therefore, the will be another point a and value b and we require that $p(a) = b$. We will see that with high probability, only one polynomial q_i will fulfill $q_i(a) = b$, namely the restriction of p.

However, since we already fixed two points, x and a, we cannot use lines anymore, since there is only one line through x and a which is not very random. Therefore, we use curves of degree 3. We choose two other points r and s in F^m uniformaly at random. Next we choose three distinct nonzero values t_a, t_r, t_s in F. Let $\ell : F \to F^m$ be the unique curve of degree three that fulfills $\ell(0) = x$, $\ell(t_a) = a$, $\ell(t_r) = r$, and $\ell(t_s) = s$. (To do this, we just have to do univariate interpolation in each coordinate.)

Input: $f : F^m \to F$ given as an oracle, a point $x \in F^m$, a point $a \in F^m$, and a value b

Output: $p(x)$, if there is a unique polynomial p such that p has agreement $\geq 5\sqrt{d/|F|}$ and $p(a) = b$.

1. Construct a random degree 3 curve ℓ as described above.
2. Construct all univariate polynomials $q_1, \ldots, q_s$ of degree $\leq 3d$ that have agreement $\geq 4\sqrt{d/|F|} \geq 2\sqrt{3d/|F|}$ with $f|_\ell$
3. If there is a unique σ with $q_\sigma(a) = b$ then output $q(x)$.

Lemma 5.9 *Let p be an m-variate polynomial of degree d that has agreement $\geq 5\sqrt{d/|F|}$ with f. Then with probability $\geq \frac{9}{10}$, $p|_\ell$ has agreement $\geq 4\sqrt{d/|F|}$ with $f|_\ell$ for a random degree 3 curve ℓ chosen as above.*

Proof. For $z \in F$, let S_z be the random variable that is 1 if $p|_\ell(z) = f|_\ell(z)$ and 0 otherwise. Let $S = \sum_{z \in F} S_z$. We have $\mathrm{E}(S_z) = 5\sqrt{d/|F|}$ and $\mathrm{E}(S) = 5\sqrt{d \cdot |F|}$. The variance of S_z is $\mathrm{E}[S_z] - \mathrm{E}[S_z]^2 \leq E[S_z]$, since S_z is $\{0,1\}$-valued. By construction of ℓ, the S_z are pairwise independent. Thus $\mathrm{Var}[S] \leq 5\sqrt{d \cdot |F|}$. Now by Chebycheff's inequality, $\Pr[|S - \mathrm{E}[X]| \geq \sqrt{d/|F|}]$ is much smaller than, say, $\frac{1}{10}$, provided that $d/|F|$ is small enough. Hence with probability $\frac{9}{10}$, p_ℓ and f_ℓ have agreement $\geq 4\sqrt{d/|F|}$. ∎

Lemma 5.10 *Let $q_1, \ldots, q_s$ be the polynomials output in step 2. With probability $\geq \frac{9}{10}$, $q_i(a) \neq q_\sigma(a)$ for all $\sigma \neq i$, where the probability is taken over a.*

Proof. Two different univariate polynomials of degree 3 can agree in at most $3d + 1$. By Corollary 5.8, $s \leq \sqrt{|F|/d}$. So the probability that $q_i(a) = q_\sigma(a)$ for some σ is bounded by $\frac{(3d+1)\sqrt{|F|/d}}{|F|} \leq 3 \cdot \sqrt{d/|F|} \leq 1/10$ for $d/|F|$ small enough. ∎

Now it is easy to see that the algorithm above gives the correct answer with probability 4/5: By Lemma 5.9, if a polynomial p has agreement $\geq 5\sqrt{d/|F|}$ with f, then its restriction $p|_\ell$ has agreement $\geq 4\sqrt{d/|F|}$ with $f|_\ell$ with probability $\geq 9/10$. By Lemma 5.10, with probability $\geq 9/10$, there is only one polynomial that has value b at the point a.

Now the complete decoding procedure goes as follows: Now consider the following Turing machine.

Input: f (as an oracle)
Output: A list of Turing machines $M_1, \ldots, M_s$

1. Choose a at random and a random degree 3 curve ℓ with $\ell(0) = a$.
2. Construct all univariate polynomials $q_1, \ldots, q_s$ of degree $\leq 3d$ that have agreement $\geq 4\sqrt{d/|F|} \geq 2\sqrt{3d/|F|}$ with $f|_\ell$.
3. Let $b_\sigma = q_\sigma(0)$, $1 \leq \sigma \leq s$.
4. For each σ, compute a description of the Turing machine above where a is initialized by the value chosen in step 1 and b with b_σ.
5. Take a description of the Turing machine for correction (i.e, agreement $\geq 9/10$) and replace all oracle calls by the description from step 4. Output all such descriptions.

We claim that with probability $\geq$??, for all polynomials p that have agreement $\geq 5\sqrt{d/|F|}$, there is an index λ such that M_λ^f computes p.

- With probability $\geq 9/10$, a is chosen in such a way that $b_1, \ldots, b_s$ are pairwise distinct.
- In this case, the Turing machine in step 4, outputs $p(x)$ with probability $\geq 4/5$. This is however not true, since the a is not random any more once we fixed it.
- Instead, we consider the x to be random. Then with probability $\geq 9/10$, M_λ^f computes a function that has agreement $4/5$ with p.
- If we use this machine as an input for the correction machine, then we get a machine computing p. (Currently, this requires to modidy the probabilities in such a way that the correction procedure also works with agreement $\geq 4/5$ which is easy.

Note that $s \leq \sqrt{|F|/d}$.

6 Extractors

Extractors are a useful tool for randomness efficient error probability amplification. To define extractors, we first have to be able to measure the closeness of probability distributions.

6.1 Preliminaries

Definition 6.1 *Let X and Y two random variables with range S. The* statistical difference *of X and Y is* $\mathrm{Diff}(X,Y) = \max_{T\subseteq S} |\Pr[X \in T] - \Pr[Y \in T]|$. *$X$ and Y are called* ϵ-close *if* $\mathrm{Diff}(X,Y) \leq \epsilon$.

In the same way, we can define the statistical difference of two probability distributions.

We can think of T as a statistical test which tries to distinguish the distributions of X and Y. The L_1-distance of X and Y is defined as

$$|X - Y|_1 = \sum_{s\in S} |\Pr[X = s] - \Pr[Y = s]|$$

L_1-distance and statistical difference are related as stated below.

Exercise 6.1 *Prove the following: Two random variables X and Y are ϵ-close if and only if $|X - Y|_1 \leq 2\epsilon$.*

Statistical closeness is preserved under application of functions.

Exercise 6.2 *Prove the following statements:*

1. *Let X and Y be random variables with range S that are ϵ-close. Let f be a function with domain S. Then $f(X)$ and $f(Y)$ are ϵ-close.*
2. *If Z is a random variable independent of X and Y, then the random variables (X,Z) and (Y,Z) are ϵ-close.*

A classical measure for the amount of randomness contained in a random source X is the *Shannon entropy* $H(X) = -\sum_{s\in S} \Pr[X = s] \log \Pr[X = s]$. This is however not a suitable measure in our context. Consider for instance the following source: With probability 0.99 it returns the all-zero string. With probability 0.01 is returns a string in $\{0,1\}^N$ chosen uniformly at random. The Shannon entropy of this source is $\geq 0.01N$ which is quite large,

in particular unbounded. If we want to use this source for simulating ramdomized algorithms, we will take one sample from this source. But with probability 0.99, we see a string that contains no randomness at all which is not very useful for derandomization. The Shannon entropy measures "randomness on the average" and particularly does not talk about variance. It is useful when one draws many samples from a source. For our purposes, the following definition is more useful.

Definition 6.2 *Let X be a random variable with support S.*

1. *The* min-entropy *of X is* $\min_{s \in S} -\log \Pr[X = s]$.
2. *If X has min-entropy at least k, then X will be called a k-source. If in addition its range is $\{0,1\}^N$, then X is an (N,k)-source.*

Note that the min-entropy of the source above is only $\log 1/0.99$ which is constant. In some sense, the min-entropy measures "randomness in the worst-case".

Definition 6.3 *Let U_d be the uniform distribution on $\{0,1\}^d$. A function* $\mathrm{Ext} : \{0,1\}^N \times \{0,1\}^d \to \{0,1\}^m$ *is called a* (k,ϵ)-extractor *if for any (N,k)-source X,* $\mathrm{Ext}(X, U_d)$ *is ϵ-close to uniform.*

Above, we call a source ϵ-close to uniform, if it and U_m are ϵ-close. Our aim is to construct extractors with small d and large m. An extractor extracts the randomness of the weak source in the sense that given a sample of the weak random source and a short truly random string, it produces a string that is nearly uniformly distributed.

Sometimes it is convenient to view an extractor Ext as a bipartite multigraph. The nodes are $\{0,1\}^N$ on the one and $\{0,1\}^m$ on the other side. Each node $v \in \{0,1\}^N$ has degree 2^d. It is incident with the edges $(v, \mathrm{Ext}(v,i))$ for all $i \in \{0,1\}^d$.

A family of extractors $\mathrm{Ext}_m : \{0,1\}^{N(m)} \times \{0,1\}^{d(m)} \to \{0,1\}^m$ is called *explicit*, if the mapping $(m,v,e) \to \mathrm{Ext}_m(v,e)$ is computable in time $\mathrm{poly}(N(m), d(m), m)$. (Usually, $N \geq m$ for an extractor. Therefore, we parameterize the family by the size of the image.)

Lemma 6.4 *If there is an explicit family of $(k(r), 1/8)$-extractors* $\mathrm{Ext}_r : \{0,1\}^{N(r)} \times \{0,1\}^{d(r)} \to \{0,1\}^r$, *then for any* BPP*-Turing machine M that runs in time t, uses r random bits, and has error probability $1/3$, there is a* BPP*-machine M' with $L(M) = L(M')$ that runs in time* $\mathrm{poly}(N(r), 2^{d(r)}, t)$, *uses $N(r)$ random bits, and has error probability bounded by $2^{k(r)-N(r)}$.*

Proof. M' uses its $N(r)$ random bits and interprets it as a string $x \in \{0,1\}^{N(r)}$. Let $y_i = \mathrm{Ext}(x,i)$ for all $i \in \{0,1\}^{d(r)}$. M' now simulates $2^{d(r)}$

runs of M, each one with a different string y_i as random string. M' accepts if the majority of these runs lead to an accepting configuration and rejects otherwise.

The bound on the running time is clear from the construction. We have to estimate the error probability. Assume that a given input u is in $L(M)$, i.e., M accepts u with probability at least $2/3$. The case $u \notin L(m)$ is symmetric. To show the bound on the error probability, it is sufficient to show that less than $2^{k(r)}$ of the random strings x lead to a rejecting configuration. Suppose on the contrary that this is not the case. Let S be the set of all such x. Then the uniform distribution X on S has min-entropy at least $k(r)$. Thus $\mathrm{Ext}(X, U_{d(r)})$ is $1/6$-close to uniform. Let $T \subseteq \{0,1\}^r$ be the statistical test that consists of all random strings that make M accept. The probability that a string drawn uniformly at random from $\{0,1\}^r$ is in T is at least $2/3$. By definition, the probability that the y_i are in T is $\geq 2/3 - 1/8 > 1/2$.

This is a contradiction, since for each choice of x that makes M' reject, more than half of the string $\mathrm{Ext}(x,i)$ lead to a rejecting configuration, i.e., are not in T. ■

Extractors can also be used to run PTMs with a weak random source instead of a prefect random string. The proof of the following lemma is similar to the proof of the previous one and is left as an exercise.

Lemma 6.5 *If there is an explicit family of $(k(r), 1/6)$-extractors $\mathrm{Ext}_r : \{0,1\}^{N(r)} \times \{0,1\}^{d(r)} \to \{0,1\}^r$ then for any* BPP*-machine M that runs in time t, uses r random bits, and has error probability $1/3$, there is a Turing machine M' with $L(M) = L(M')$ that runs in time $\mathrm{poly}(N(r), 2^{d(r)}, t)$, uses one sample of an $(N(r), k(r)+\ell(r))$-source, and has error probability bounded by $2^{-\ell(r)}$.*

Exercise 6.3 *Prove Lemma 6.5*

By a probabilistic argument, we will show that optimal extractors exits. It is however an open problem whether these extractors are also explicit. It turns out that a random extractor is optimal with constant probability. But choosing an extractor at random is not a good idea when preforming randomness efficient probability amplification.

Theorem 6.6 ((without proof)) *For every N, $k \leq N$, and $\epsilon > 0$, there exists a (k,ϵ)-extractor $\mathrm{Ext} : \{0,1\}^N \to \{0,1\}^d \to \{0,1\}^m$ with $d = \log(N-k) + 2\log(1/\epsilon) + O(1)$ and $m = k + d - 2\log(1/\epsilon) - O(1)$.*

6.2 Trevisan's extractor

We now present an extractor that in some sense solves the randomness efficient probability amplifictation problem and the computing with weak random sources problem. The construction is due to Trevisan [**?**]. Basically,

it is the Nisan–Wigderson construction of a pseudorandom generator where everthing is thrown away that is not needed for an extractor.

Let $C : \{0,1\}^N \to \{0,1\}^{\bar{N}}$ be a polynomial-time computable code that is $(\delta, O(1/\delta^2))$ list-decodable. Such code exists with $\bar{N} = \text{poly}(N, 1/\delta)$.

Exercise 6.4 *Construct such a code C.*

Let $\ell = \log \bar{N}$ and let $S = (S_1, \ldots, S_m)$ be a $(m, d, \ell, \log m)$-design over $[d]$. Let $\text{ExtNW} : \{0,1\}^N \times \{0,1\}^d \to \{0,1\}^m$ be defined by

$$\text{ExtNW}_{C,S}(x, z) = NW_{C(x),S}(z).$$

Here $C(x)$ is a bit-string of length $\bar{N}$ which we again interpret as a Boolean function $\{0,1\}^{\log \bar{N}} \to \{0,1\}$. In other words, we encode the first part of the input using the code C, interpret $C(x)$ as a function that is used in the Nisan–Wigderson construction and take the second part of the input as the seed for the Nisan–Wigderson construction.

Lemma 6.7 $\text{ExtNW}_{C,S}$ *is a $(m^3, 2\epsilon)$-extractor for m large enough and all $\epsilon > 2^{-m^2}$, if we choose the parameter δ of C as $\delta = \epsilon/m$.*

Proof. Let X be a (N, m^3)-source. Let $D : \{0,1\}^m \to \{0,1\}$. We will show that

$$\left| \Pr_{r \in \{0,1\}^m}[D(r) = 1] - \Pr_{b \sim X,\ z \in \{0,1\}^d}[D(\text{ExtNW}(b, z)) = 1] \right| \leq 2\epsilon.$$

A particular value b is called *bad* if

$$\left| \Pr_{r \in \{0,1\}^m}[D(r) = 1] - \Pr_{z \in \{0,1\}^d}[D(\text{ExtNW}(b, z)) = 1] \right| > \epsilon.$$

Let B be the set of all bad b. By Lemma **??**, if b is bad, then there is a circuit K of size $\mathsf{Size}(D) + O(m^2)$ such that D agrees with $C(b)$ on a fraction of $1/2 + \epsilon/m$ of its inputs.

Thus b is completely determined by K and $2\log(m/\epsilon) + O(1)$ additional bits. This is due to the list-decodability of C: In the ball of radius $1/2 - \epsilon/m$ around the function computed by K (interpreted as a bit string), there are at most $O((m/\epsilon)^2)$ code words, one of them being b. For a given circuit D, the number of bad code words $|B|$ is bounded by

$$|B| \leq 2^{O(m^2 \log m)} \cdot O(\Big(\frac{m}{\epsilon}\Big)^2) = 2^{O(m^2) \log m}.$$

($1/\epsilon^2 \leq 2^{2m^2}$.) Note that if D is given, then K consists of one copy of D and $\mathrm{O}(m^2)$ additional gates. The number of circuits with $O(m^2)$ gates is $2^{\mathrm{O}(m^2 \log m)}$ (cf. Exercise 6.5).

The probability that an element b drawn according to the distribution of X is bad is therefore bounded by $2^{-m^3} \cdot 2^{O(m^2 \log m)} < \epsilon$ for m large enough. Therefore,

$$\begin{aligned}
&\left| \Pr_{r\in\{0,1\}^m}[D(r)=1] - \Pr_{b\sim X,\ z\in\{0,1\}^d}[D(\mathrm{ExtNW}(b,z))=1] \right| \\
&\leq \sum_b \Pr[X=b] \left| \Pr_{r\in\{0,1\}^m}[D(r)=1] - \Pr_{z\in\{0,1\}^d}[D(\mathrm{ExtNW}(b,z))=1] \right| \\
&\leq \Pr[X\in B] + \sum_{b\notin B} \Pr[X=b] \left| \Pr_{r\in\{0,1\}^m}[D(r)=1] - \Pr_{z\in\{0,1\}^d}[D(\mathrm{ExtNW}(b,z))=1] \right| \\
&\leq 2\epsilon.
\end{aligned}$$

The completes the proof, since we can interpret D as a characteristic function of a statistical test. ■

Theorem 6.8 *For every constant $\epsilon > 0$ and every function $N(m) \geq k(m)$ there is an explicit family of $(k(m), \epsilon)$-extractor $\mathrm{Ext}_m : \{0,1\}^{N(m)} \times \{0,1\}^{d(m)} \to \{0,1\}^m$ with $k(m) = m^3$ and $d(m) = O(\log m + \log 1/\epsilon)$.*

Proof. Let C_N be a family of codes that is a polynomial-time computable and $(\epsilon/m, O(m^2/\epsilon^2))$ list-decodable. Consider $\mathrm{ExtNW}_{C_N,S} : \{0,1\}^N \times \{0,1\}^d \to \{0,1\}^m$. In the design of the Nisan–Wigderson construction, we have $\ell = \log N = O(\log m + \log 1/\epsilon)$. Then $m = 2^{\gamma\ell}$ for some $\gamma = \Omega(\log m + \log 1/\epsilon)$. By Lemma **??**, $d = c\ell/\gamma = O(\log m + \log 1/\epsilon)$.

The code C_N is polynomial time computable. It has already been demonstrated that the Nisan-Widgerson generator is polynomial time computable (even if the function given to it is not). Thus $\mathrm{ExtNW}_{C,S}$ is explicit, too. Now the theorem follows from Lemma 6.7. ■

Exercise 6.5 *Show that there are $2^{O(n \log n)}$ circuits with $\leq n$ gates.*

Theorem 6.8 can now be plugged into Lemmas 6.4 and 6.5.

Corollary 6.9 *For any* BPP*-Turing machine M that runs in time t, uses r random bits, and has error probability $1/3$, there is a* BPP*-machine M' that runs in time $\mathrm{poly}(t)$, uses $N(r) \geq r^3$ random bits, and has error probability bounded by $2^{r^3 - N(r)}$.*

If choose $N(r) = r^4$ for instance, this corollary is almost optimal in the sense that for each of the $N(r)$ random bits reduces the error probability by almost $1/2$. This is the best we can expect.

Corollary 6.10 *For any* BPP*-Turing machine M that runs in time t, uses r random bits, and has error probability $1/3$, there is a Turing machine M' that runs in time $\mathrm{poly}(t)$, uses one sample of an $(N(r), r^3 + \ell(r))$-source for a polynomial $N(r)$ and has error probability bounded by $2^{-\ell(r)}$.*

7 Hardness based on derandomization

In some sense, this section is the converse of Section ??. While in that section, we tried to derandomize BPP under the assumption that E requires exponentially large circuits, we here try to show the opposite: Suppose we can derandomize BPP. Can we get circuit lower bounds out of it? We will assume something perhaps weaker, namely, that ACIT can be derandomized, the problem of testing whether a polynomial given by an arithmetic circuit is identically zero.

Before we start, we introduce the following notion: We say that the permanent is in NP, if the following language

$$\{(A, v) \mid A \text{ is a } \{0,1\}\text{-matrix and } \mathrm{per}(A) = v\}.$$

We abuse of notation, we call this language per again.

7.1 Testing arithmetic circuits for the permanent

The aim of this section is given some polynomial f, to construct a circuit C such that C computes the zero polynomial if and only if f is the permananent.

7.1.1 Division-free circuits over $\mathbb{Z}$

Let p_n be a polynomial in n^2 indeterminates $X_{i,j}$, $1 \leq i, j \leq n$. Assume that p_n computes the permanent of $n \times n$-matrices, i.e., $p_n(X) = \mathrm{per}(X)$, where X denotes the matrix with entries $X_{i,j}$.

We can use p_n to compute the permanent of any size $\leq n$: If A is an $i \times i$-matrix, then we place A into the lower right corner, place ones to the remaining entries on the diagonal, and fill the rest of the entries with zeros. Let p_i be the restriction obtained from p_n. By the definition of the permanent, we then have

$$p_1(X^{(1)}) = X_{n,n} \tag{7.1}$$

$$p_i(X^{(i)}) = \sum_{j=1}^{i} X_{n-i+1,n-i+j} p_{i-1}(X_j^{(i)}) \tag{7.2}$$

where $X^{(i)}$ is the $i \times i$-matrix in the lower right corner of X and $X_j^{(i)}$ is the jth minor of $X^{(i)}$ along the first row, i.e., the matrix obtained from $X^{(i)}$ by deleting the first row and the jth column.

On the other hand, any sequence of polynomials $p_1, \ldots, p_n$ fulfilling (7.1) and (7.2) necessarily computes per.

Exercise 7.1 *Prove this last claim.*

Lemma 7.1 *The language*

$$\texttt{ACP} := \{\langle C, n\rangle \mid C \text{ is an arithmetic circuit for per of } n \times n\text{-matrices over } \mathbb{Z}\}$$

is polynomial-time many-one reducible to ACIT.

Proof. Assume that C computes a polynomial p_n. Let p_i be the restriction of p_n such that p_i computes the permanent of $i \times i$-matrices provided that $p_n(X) = \mathrm{per}(X)$.

To check whether p_n computes indeed the permanent, it suffices to check whether $p_1, \ldots, p_n$ fulfill (7.1) and (7.2). In other words, we have to check whether

$$h_1(X) = p_1(X^{(1)}) - X_{n,n} \tag{7.3}$$

$$h_i(X) = p_i(X^{(i)}) - \sum_{j=1}^{i} X_{n-i+1,n-i+j} p_{i-1}(X_j^{(i)}), \qquad 2 \le i \le n, \tag{7.4}$$

are identically zero. To test whether $h_1, \ldots, h_n$ are identically zero, we can equivalently test whether

$$h(X, Y) = h_1(X) + h_2(X)Y + \cdots + h_n(X)Y^{n-1}$$

is identically zero, where Y is a new variable.

By construction, C computes $\mathrm{per}(X)$ iff $h(X, Y) = 0$. Since every h_i is computable by a circuit of size polynomial in the size of C, h is also computable by such a circuit. This circuit can be constructed from C in polynomial time. ∎

Corollary 7.2 *Suppose that* ACIT *over* $\mathbb{Z}$ *is in* NP. *If* per *over* $\mathbb{Z}$ *is computable by division-free arithmetic circuits of polynomial size over* $\mathbb{Z}$, *then* per $\in$ NP.

Proof. If per is computable by arithmetic circuits of polynomial size, then we can nondeterministically guess such a circuit C that computes the permanent of $n \times n$-matrices in time polynomial in n. Since ACIT is in

NP, so is ACP by Lemma 7.1. Therefore, we can verify our guess for C nondeterministically in polynomial time. Once we have found C, we evaluate it deterministically at the given $\{0,1\}$-matrix A in polynomial time, by doing all operations modulo $2^{n \log n} + 1$. Note that $2^{n \log n}$ has only polynomially many bits and that the permanent of a $\{0,1\}$-matrix cannot exceed $2^{n \log n}$, since it has at most $n!$ terms. Finally, we simply check whether the computed result equals v, the second part of the input. ■

7.1.2 Circuits over $\mathbb{Q}$ with divisions

Next we generalize the results of the preceding subsection to the case where we only assume that per has polynomial size arithmetic circuits over $\mathbb{Q}$ (possibly using divisions). An arithmetic circuit C over $\mathbb{Q}$ with divisions computes a rational function p/q where p and q are polynomials over $\mathbb{Z}$. C computes per, if $p = q \cdot \text{per}$. If no division by zero occurs when evaluating C at a particular matrix A whose permanent we want to compute, then $\text{per}(A) = p(A)/q(A)$.

Since we only want to compute a polynomial, we can however use the following lemma due to Strassen, which basically states that we do not need divisions (as long as we do not care about polynomial factors).

Lemma 7.3 (Strassen [Str73]) *Let C be an arithmetic circuit over $\mathbb{Q}$ of size s that computes a polynomial $p \in \mathbb{Q}[X_1, \ldots, X_n]$ of degree d. Let $\xi = (\xi_1, \ldots, \xi_n)$ be a point such that when we evaluate C at ξ, no division by zero occur. Then there is a circuit C' of size $\text{poly}(s, d, \log \max_i \xi_i)$ that computes p and uses only divisions by constants from $\mathbb{Q}$. Given C, C' can be computed in time $\text{poly}(s, d, \log \max_i \xi_i)$.*

Before we proof this lemma, we first introduce some useful terminology. A polynomial is called *homogeneous*, if all its monomials have the same degree. Let $f \in \mathbb{Q}[X_1, \ldots, X_n]$ be a polynomial of degree d. We can write $f = f_0 + f_1 + \cdots + f_d$ such that f_δ is a homogeneous polynomial of degree δ.

The main idea of the proof of the lemma is to consider the computation not as a computation in $\mathbb{Q}(X_1, \ldots, X_n)$ of rational functions, but in the ring of *formal power series* $\mathbb{Q}[[X_1, \ldots, X_n]]$. Let f and g be polynomials that we view now as formal power series. Let $g = g_0 - \hat{g}$ where g_0 has degree zero and $\hat{g}$ has no constant term. If $g_0 \neq 0$, then we can express $1/g$ as a formal power series as follows:

$$\frac{1}{g} = \frac{1}{g_0} \cdot \frac{1}{1 - \hat{g}/g_0} = \frac{1}{g_0} \sum_{i=0}^{\infty} \left(\frac{\hat{g}}{g_0} \right)^i . \tag{7.5}$$

Thus we can replace each division f/g in C by a multiplication $f \cdot \frac{1}{g}$ with a power series. But since we are computing only polynomials of degree d, we

can truncate the power series at degree $d+1$ without changing the result. Simce $\hat{g}$ has no constant term, $\hat{g}^{d+1}$ has degree at least $d+1$ and cannot contribute to the result. Therefore, we may truncate everything with degree $d+1$ or higher.

Lemma 7.3. Our aim is to inductively transfer the circuit C into C'. Each gate that computes a rational function f/g will be replaced by a bunch of gates that compute the homogeneous parts of $f \cdot \frac{1}{g}$ up to degree d. By the arguments given above, it follows that the circuit C' will produce the desired result.

One problem when we want to replace the division by g with the multiplication of the geometric series is that we have to ensure that the degree zero monomial g_0 of g is nonzero. We can achieve this by first applying a *Taylor Shift*: We replace each variable X_ν at the input gates of C by $X_\nu + \xi_\nu$. To do so, we replace the input gate by one addition gate whose inputs are X_ν and ξ_ν. Let $\tilde{g}$ be the resulting polynomial by which we want to divide now. We claim that after this shift, $\tilde{g}_0 \neq 0$. This is easily seen to be true since $\tilde{g}_0 = \tilde{g}(0,\ldots,0) = g(\xi_1,\ldots,\xi_n) \neq 0$ by the choice of ξ. After we have removed all the divisions, we are then able to reverse the Taylor shift by replacing the input variables by $X_\nu - \xi_\nu$

We now come to the inductive argument: We want to replace each gate of C computing a rational function f/g by a bunch of gates that compute the homogeneous parts up to degree d of $f \cdot \frac{1}{g}$ (as a formal power series).

The induction start is easy: Each input gate is either X_ν or a constant. Both are homogeneous polynomials.

For the inductions step consider a gate G of C and assume that each direct predecessor of C has be replaced by a bunch of gates computing the homogeneous parts, i.e., if the two predecessors computed f_1/g_1 and f_2/g_2 in C, we now compute homogeneous polynomials $h_{i,0},\ldots,h_{i,d}$ of degrees $0,\ldots,d$, respectively, such that

$$f_i/g_i = h_{i,0} + \cdots + h_{i,d} \qquad \text{up to degree } d \tag{7.6}$$

in the ring of formal power series. If G is an addition gate, then we replace it by $d+1$ addition gates computing $h_{1,i} + h_{2,i}$ for $0 \leq i \leq d$. If G is a multiplication gate, then we replace it by a bunch of gates computing

$$\sum_{j=0}^{i} h_{1,j} h_{2,i-j}, \qquad 0 \leq i \leq d, \tag{7.7}$$

which are the homogeneous parts of the product (up to degree d). Finally, if G is a division gate, we first compute the homogeneous parts of g_2/f_2 up to degree d. By (7.5), they are

$$\frac{1}{h_{2,0}} \sum_{jk=i} \left(-\frac{h_{2,k}}{h_{2,0}}\right)^j, \qquad 0 \leq i \leq d. \tag{7.8}$$

Then we just have to multiply like in (7.7). Note that $h_{2,0}$ is a constant. Thus division by it is allowed.

In each of (7.6), (7.7), and (7.8), we only add a polynomial number of gates. Thus the size of the new circuit C' is polynomial in the size of C, the degree d, and the number of bits needed to write down the numbers ξ_i. Furthermore, it is easy to see that C' can be computed in polynomial time. ■

Corollary 7.4 *If there is a family of polynomial size arithmetic circuits over $\mathbb{Q}$ with divisions computing* per*, then there are two families $C_{1,n}$ and $C_{2,n}$ over $\mathbb{Z}$ without divisions, such that $C_{2,n}$ computes a nonzero constant $c_n \in \mathbb{Z}$ and $C_{1,n}$ computes $c_n \cdot \mathrm{per}(X)$ where X is an $n \times n$-matrix with indeterminates as entries.*

Proof. We will modify the construction of the previous lemma. If there is a rational constant used in C, we split it into its numerator and denominator. The only other step that we will have to modify is (7.8). Here $C_{2,n}$ computes the constant $q_{2,0}^{d+1}$ and $C_{1,n}$ computes

$$\sum_{jk=i} q_{2,0}^{d+1-j} h_{2,k}^{j}, \qquad 0 \leq i \leq d.$$

where $q_{2,0}$ is the denominator of $h_{2,0}$ instead of (7.8).

It remains to show how to find ξ and to prove that the size of the entries is not too large, i.e., the number of bits needed to represent them should be polynomial in n. The degree of the denominator of the rational function computed at each gate is bounded by $2^{\mathrm{poly}(n)}$. Thus by the Schwartz–Zippel Lemma applied to the product of all denominators, there is a ξ whose entries have size $2^{\mathrm{poly}(n)}$ such that none of the denominators vanishes at ξ. ■

Modifiying the proof in the preceding subsection, it is now rather easy to strengthen Corollary 7.2 to arithmetic circuits with divisions.

Theorem 7.5 *Suppose that* ACIT *over $\mathbb{Z}$ is in* NP*. If* per *over $\mathbb{Q}$ is computable by polynomial size arithmetic circuits with divisions over $\mathbb{Q}$, then* per $\in$ NP.

Proof. Given n, we nondeterministically guess two polynomial size division-free arithmetic circuits C_1 and C_2 over $\mathbb{Z}$. C_1 depends on n^2 variables and C_2 depends on no variables, that is, it computes a constant c.

We now check whether C_1 computes $c \cdot$per. This can be done by reduction to ACIT over $\mathbb{Z}$ as in the proof of Lemma 7.1. The only thing we have to change is (7.3) to $h_1(X) = p_1(X^{(1)}) - c \cdot X_{n,n}$. The constant c can be computed via C_2.

Since per is computable by polynomial size circuits over $\mathbb{Q}$ with divisions, we know by Corollary 7.4 that such circuits C_1 and C_2 exists.

Assume that we guessed C_1 and C_2 correctly. We now have to compute per of $\{0,1\}$-matrices. The problem is that intermediate results may get very large when evaluating C_1 and C_2. The output still is bounded by $n!$. This time it does not however not suffice to compute mode $n!+1$ since we have to ensure that c is nonzero mod $n!+1$, since we afterwards have to divide by c. We have $c \leq (s_2)^{2^{s_2}}$ where s_2 is the size of C_2. Therefore, c has at most $m := 2^{s_2} \log s_2$ many prime divisors. (Each of them is at least two.) We guess a number p between $2^{n \log n}$ and $2^{n \log n + m^2}$ and deterministically check whether it is prime. Then we check whether $c \mod p \neq 0$. By the prime number theorem, the number of primes in the given interval exceeds the number of possible prime divisors of c. Thus, such a number p exists and we will surely find it.

Finally, we evaluate C_1 and C_2 modulo p and then divide the results of C_1 by the constant computed by C_2 modulo p. The results is the permanent of the given input matrix. ■

7.2 Hardness result

We now come to the proof of the hardness result based on derandomization of ACIT. The hardness result will not be as strong as the assumption in Theorem 1.8. On the other hand, we will only assume that $\texttt{ACIT} \in \mathsf{NP}$ (instead of $\mathsf{BPP} = \mathsf{P}$).

Definition 7.6 *We say that* $\mathsf{NEXP} \cap \mathsf{co\text{-}NEXP}$ *is computable by polynomial-size circuit if the following two conditions hold:*

1. $\mathsf{NEXP} \cap \mathsf{co\text{-}NEXP} \subseteq \mathsf{P/poly}$ *and*
2. per *over* $\mathbb{Q}$ *is computable by polynomial-size arithmetic circuits (with divisions).*

Exercise 7.2 *Computing* per *of a* $\{0,1\}$*-matrix is clearly possible in deterministic exponential time. Why does the first condition of Definition 7.6 not imply the second one?*

To prove our main theorem, we need the following results from complexity theory.

Theorem 7.7 (Toda) $\mathsf{PH} \subseteq \mathsf{P}^{\mathrm{per}}$.

You saw a proof of this theorem in the complexity theory lecture.

Exercise 7.3 *If* $\mathrm{per} \in \mathsf{NP}$*, then* $\mathsf{P}^{\mathrm{per}} \subseteq \mathsf{NP}$.

Theorem 7.8 (Meyer, see [?]) *If* $\mathsf{EXP} \subseteq \mathsf{P/poly}$*, then* $\mathsf{EXP} = \Sigma_2^{\mathsf{P}} \cap \Pi_2^{\mathsf{P}}$.

The proof of this theorem was an exercise in the complexity theory lecture.

Exercise 7.4 *Conclude that* $\mathsf{EXP} \subseteq \mathsf{P/poly}$ *implies* $\mathsf{P} \neq \mathsf{NP}$

Theorem 7.9 (Impagliazzo, Kabanets & Wigderson [?]) *If* $\mathsf{NEXP} \subseteq \mathsf{P/poly}$ *then* $\mathsf{NEXP} = \mathsf{EXP}$.

We do not give a proof of this result here.

Corollary 7.10 *If* $\mathsf{NEXP} \subseteq \mathsf{P/poly}$*, then* per *over* $\mathbb{Z}$ *is* NEXP*-hard.*

Proof. If $\mathsf{NEXP} \subseteq \mathsf{P/poly}$, then $\mathsf{NEXP} = \mathsf{EXP} = \mathsf{PH}$ by Theorems 7.8 and 7.9. Since per is PH-hard by Theorem 7.7, it is also NEXP-hard. ∎

Finally, we come to the main result of this chapter.

Theorem 7.11 *If* `ACIT` *over* $\mathbb{Z}$ *is in* NP*, then* $\mathsf{NEXP} \cap \mathsf{co\text{-}NEXP}$ *is not computable by polynomial-size circuits (in the sense of Definition 7.6).*

Proof. If per is not computable by polynomial-size arithmetic circuits, then the proof is finished by definition. It remains the case that per is computable by polynomial-size arithmetic circuits.

per is PH-hard by Theorem 7.7. Since $\texttt{ACIT} \in \mathsf{NP}$, we also know that per $\in \mathsf{NP}$ by Theorem 7.5. By Exercise 7.3, we know that the polynomial hierachy collapses to NP, i.e., $\mathsf{PH} = \mathsf{NP} = \mathsf{co\text{-}NP}$. A simply padding argument (see Exercise 7.5) implies that $\mathsf{NEXP} = \mathsf{co\text{-}NEXP}$.

If $\mathsf{NEXP} \not\subseteq \mathsf{P/poly}$, then we are done, because $\mathsf{NEXP} = \mathsf{co\text{-}NEXP} = \mathsf{NEXP} \cap \mathsf{co\text{-}NEXP}$. Therefore, assume that $\mathsf{NEXP} \subseteq \mathsf{P/poly}$. By Corollary 7.10, per over $\mathbb{Z}$ is NEXP-hard. On the other hand, per $\in \mathsf{NP}$. Thus $\mathsf{co\text{-}NEXP} = \mathsf{NEXP} = \mathsf{NP}$.

$\mathsf{co\text{-}NEXP} = \mathsf{NP}$ is easily disproved by the following diagonalization argument. We can even refute the weaker statement $\mathsf{co\text{-}NEXP} \subseteq \mathsf{NTime}(2^n)$: We define a co-NEXP-machine M as follows: On input x of length n, M simulates the xth nondeterministic Turing machine M_x for 2^n steps. If M_x accepts, then M rejects. Otherwise M accepts. Since M is a co-NEXP-machine, it is easy for M to flip the outcome of the computation of M_x. ∎

Exercise 7.5 *Show that* $\mathsf{NP} = \mathsf{co\text{-}NP}$ *implies that* $\mathsf{NEXP} = \mathsf{co\text{-}NEXP}$.

Exercise 7.6 *Give a more detailed description of the diagonalization argument in the proof of Theorem 7.11.*

Bibliographic notes

The results in this section are taken from the recent work by Kabanets and Impagliazzo [KI03], though Kabanets and Impagliazzo prove many more results than presented in this section.

Bibliography

[Adl78] L. Adleman. Two theorems on random polynomial time. In *Proc. 19th Ann. IEEE Symp. on Foundations of Comput. Sci. (FOCS)*, pages 75–83, 1978.

[AKS] M. Argawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of Mathematics*, ???:???, ???

[BFNW93] L. Babai, L. Fortnow, N. Nisan, and A. Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Comput. Complexity*, 3(4):307–318, 1993.

[BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput*, 3(4):850–864, 1984.

[Imp95] R. Impagliazzo. Hard-core distributions for somewhat hard problems. In *Proc. 36th Ann. IEEE Symp. on Foudations of Comput. Sci.*, pages 538–545, 1995.

[IW97] R. Impagliazzo and A. Wigderson. P = BPP unless E has subexponential circuits. In *Proc. 29th Ann. ACM Symp. on Theory of Comput. (STOC)*, pages 220–229, 1997.

[IW98] R. Impagliazzo and A. Wigderson. Randomness versus time: Derandomization under a uniform assumption. In *Proc. of the 39th IEEE Symp. on Foundations of Comput. Sci. (FOCS)*, pages 734–743, 1998.

[KI03] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identites tests means proving circuit lower bounds. In *Proc. 35th Ann. ACM Symp. on Theory of Comput.*, pages 355–364, 2003.

[Mil01] P. B. Miltersen. *Handbook of Randomized Computing*, chapter Derandomizing complexity classes. Kluwer, 2001.

[NW94] N. Nisan and A. Wigderson. Hardness vs randomness. *J. Comput. System Sci.*, 49:149–167, 1994.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verfication of polynomial identities. *J. ACM*, pages 710–717, 1980.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6:84–85, 1977.

[Str73] Volker Strassen. Vermeidung von Divisionen. *J. Reine Angew. Math.*, 264:184–202, 1973.

[Yao82] A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd Ann. IEEE Symp. on Foundations of Comput. Sci. (FOCS)*, pages 80–91, 1982.

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. Int. Symp. on symbolic and algebraic computation (EUROSAM)*, volume 72 of *Lecture Notes in Comput. Sci.*, pages 216–226. Springer, 1979.