Advanced Complexity Theory

Markus Bläser
Universität des Saarlandes

Draft—June 25, 2014 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 $O(n)$ 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 $\frac{31}{32}$, 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 $\{(1-y)x + yt \mid y \in F\}$. By abuse of notation, we will also write $\ell_{x,t}$ for the function $y \mapsto (1-y)x + yt$.*

2. *The* restriction *of f to $\ell_{x,t}$ is the function $f|_{\ell_{x,t}} : F \to F$ defined by $y \mapsto f(\ell_{x,t}(y))$.*

First assume that we know that $f|_{\ell_{x,t}}$ and $p|_{\ell_{x,t}}$ differ in $\leq k$ positions. Note that $\hat{p} := p|_{\ell_{x,t}}$ is a univariate polynomial of degree at most d. Let $\hat{f} := f_{\ell_{x,t}}$. We want to prove that if $|F| \geq 2k+d+1$, 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 $n = 2k + d + 1$ distinct points a_i in F.

Lemma 5.2 *There is a nonzero 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 an* 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* $i = 1, \ldots, n$,
4. $\hat{p} = E_0/N_0$.

Proof. $N_0(a_i) = E_0(a_i)$ are $2k + d + 1$ homogeneous linear equations in $(d + k + 1) + (k + 1) = 2k + d + 2$ unknowns. It has a nontrivial solution. We can find a nontrivial solution in time polynomial in d, k, and $\log |F|$.

Let E_1 and N_1 be another solution. We have that

$$(N_0(a_i)E_1(a_i) - N_1(a_i)E_0(a_i))\hat{f}_i = 0, \quad i = 1, \ldots, n$$

If $\hat{f}_i \not\equiv 0$, then a_i is a zero of $N_0E_1 - N_1E_0$. If $\hat{f}_i = 0$, then $N_0(a_i) = 0 = N_1(a_i)$. Hence a_i is a zero of $N_0E_1 - N_1E_0$ in this case, too. Since the degree of this polynomial is $d + 2k$ and is has $d + 2k + 1$ zeros, $N_0E_1 = N_1E_0$ and $N_0/E_0 = N_1/E_1$. Since $N_1 = N$ and $E_1 = E$ is a solution and $E|N$ by construction, we see that $E_0|N_0$. ∎

Remark 5.5 *We assume that $\hat{p}$ is close to $\hat{f}$. It is quite easy to show that if $\hat{p}$ differs from $\hat{f}$ in more than k places, then E_0 does not divide N_0.*

Theorem 5.6 *Let $f : F^m \to F$ be a function that has agreement $\frac{31}{32}$ 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 $|F| > 2d + 2$.*

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 less thann $3/4$ of the lines have agreement at least $3/4$, we can only have a total agreement of $\frac{3}{4} + \frac{1}{4} \cdot 3/4 < \frac{31}{32} - \frac{1}{|F|}$) 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 $2|F|/4 + d + 1 \leq |F|$ which is equivalent to $d + 1 \leq |F|/2$. 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 5.7 *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 nonzero 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 5.8 *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 5.9 *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 5.8, 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, there 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 5.10 *Let p be an m-variate polynomial of degree d that has agreement $\geq 5\sqrt{d/|F|}$ with f. Then with probability $\geq \frac{31}{32}$, $p|_\ell$ 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 $\mathrm{E}(S_z) \geq 5\sqrt{d/|F|}$ and $\mathrm{E}(S) \geq 5\sqrt{d \cdot |F|}$. 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, $\frac{1}{32}$, provided that $d/|F|$ is small enough. Hence with probability $\frac{31}{32}$, p_ℓ and f_ℓ have agreement $\geq 4\sqrt{d/|F|}$. ∎

Lemma 5.11 *Let $q_1, \ldots, q_s$ be the polynomials output in step 2. With probability $\geq \frac{31}{32}$, $q_i(a) \neq q_\sigma(a)$ for all $\sigma \neq i$, where the probability is taken over a.*

Proof. Two different univariate polynomials of degree $3d$ can agree in at most $3d$ points. By Corollary 5.9, $s \leq \sqrt{|F|/d}$. So the probability that $q_i(a) = q_\sigma(a)$ for some σ is bounded by $\frac{3d\sqrt{|F|/d}}{|F|} \leq 3 \cdot \sqrt{d/|F|} \leq 1/31$ 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 5.10, 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 5.11, 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 $\geq 31/32$) 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 $\geq 31/32$, 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 $\geq 31/32$, 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 modify 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 Hardness based on derandomization

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

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

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

6.1 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.

6.1.1 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

$$p_1(X^{(1)}) = X_{n,n} \tag{6.1}$$

$$p_i(X^{(i)}) = \sum_{j=1}^{i} X_{n-i+1,n-i+j} p_{i-1}(X_j^{(i)}) \tag{6.2}$$

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 (6.1) and (6.2) necessarily computes per.

Exercise 6.1 *Prove this last claim.*

Lemma 6.1 *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 (6.1) and (6.2). In other words, we have to check whether

$$h_1(X) = p_1(X^{(1)}) - X_{n,n} \tag{6.3}$$

$$h_i(X) = p_i(X^{(i)}) - \sum_{j=1}^{i} X_{n-i+1,n-i+j} p_{i-1}(X_j^{(i)}), \qquad 2 \le i \le n, \tag{6.4}$$

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 6.2 *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 6.1. 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. ∎

6.2 Lower bound

We now come to the proof of the lower bound based on derandomization of ACIT.

Definition 6.3 *We say that* NEXP ∩ co-NEXP *is computable by polynomial-size circuit if the following two conditions hold:*

1. NEXP ∩ co-NEXP ⊆ P/poly *and*
2. per *over* $\mathbb{Z}$ *is computable by polynomial-size arithmetic circuits (without divisions).*

Exercise 6.2 *Computing* per *of a* $\{0,1\}$*-matrix is clearly possible in deterministic exponential time. Why does the first condition of Definition 6.3 not imply the second one?*

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

Theorem 6.4 (Toda) $\mathsf{PH} \subseteq \mathsf{P}^{\mathrm{per}}$.

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

Exercise 6.3 *If* per ∈ NP*, then* $\mathsf{P}^{\mathrm{per}} \subseteq \mathsf{NP}$.

Theorem 6.5 (Meyer) *If* EXP ⊆ 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 6.4 *Conclude that* EXP ⊆ P/poly *implies* P ≠ NP

Theorem 6.6 (Impagliazzo, Kabanets & Wigderson) *If* NEXP ⊆ P/poly *then* NEXP = EXP.

We do not give a proof of this result here.

Corollary 6.7 *If* NEXP ⊆ P/poly*, then* per *over* $\mathbb{Z}$ *is* NEXP*-hard.*

Proof. If NEXP ⊆ P/poly, then NEXP = EXP = PH by Theorems 6.5 and 6.6. Since per is PH-hard by Theorem 6.4, it is also NEXP-hard. ∎

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

Theorem 6.8 (Kabanets & Impagliazzo) *If* ACIT *over* $\mathbb{Z}$ *is in* NP*, then* NEXP ∩ co-NEXP *is not computable by polynomial-size circuits (in the sense of Definition 6.3).*

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 6.4. Since ACIT $\in$ NP, we also know that per $\in$ NP by Corollary 6.2. By Exercise 6.3, we know that the polynomial hierachy collapses to NP, i.e., PH = NP = co-NP. A simply padding argument (see Exercise 6.5) implies that NEXP = co-NEXP.

If NEXP $\not\subseteq$ P/poly, then we are done, because NEXP = co-NEXP = NEXP$\cap$co-NEXP. Therefore, assume that NEXP $\subseteq$ P/poly. By Corollary 6.7, per over $\mathbb{Z}$ is NEXP-hard. On the other hand, per $\in$ NP. Thus co-NEXP = NEXP = NP.

co-NEXP = NP is easily disproved by the following diagonalization argument. We can even refute the weaker statement co-NEXP $\subseteq$ 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 6.5 *Show that* NP = co-NP *implies that* NEXP = co-NEXP.

7 Pseudorandom generators for ACIT

Throughout this chapter, k denotes a field of characteristic zero. Think of $k = \mathbb{Q}$. Similar results hold for fields of finite characteristic, but some complications arise when the characteristic divides the degree or the field is too small.

7.1 Roots of polynomials

Let $g \in k[X_1, \ldots, X_n, Y]$ be a polynomial. We call a polynomial $p = k[X_1, \ldots, X_n]$ a root of g if

$$g(X_1, \ldots, X_n, p(X_1, \ldots, X_n)) \equiv 0.$$

You can think of g being a polynomial in Y with coefficients in $k[X_1, \ldots, X_n]$. Then we are just looking for all roots of the univariate polynomial in the coefficient ring.

We are interested in the case when g is given implicitely: Given an arithmetic circuit of size s computing a polynomial g of degree $\leq d$, output a list of arithmetic circuits such that for every polynomial $p \in k[X_1, \ldots, X_n]$ that is a root of g, there is a circuit in the list that computes p.

Theorem 7.1 (Kaltofen [?]) *There is a probabilistic Turing machine that given an arithmetic circuit of size s computing a polynomial $f \in k[X_1, \ldots, X_n]$ of total degree d and maximum coefficient size α, outputs numbers $e_i \geq 1$ and arithmetic circuits computing irreducible polynomials h_i, $1 \leq i \leq r$, such that*

$$f = h_1^{e_1} \ldots h_r^{e_r}$$

with probability $\geq 3/4$. The running time of the machine is $\mathrm{poly}(s, d, \alpha)$.

The theorem above states that polynomial factorization is feasible even when the polynomial is given by a circuit. The maximum coefficient size is the number of bits needed to represent any of the coeffcients of g. We assume that encoding is chosen in such a way that we can perform arithmetic operations on numbers of size α in time $\mathrm{poly}(\alpha)$.

Lemma 7.2 (Gauss) *Let $g \in k[X_1, \ldots, X_n, Y]$ and $p \in k[X_1, \ldots, X_n]$ such that*

$$g(X_1, \ldots, X_n, p(X_1, \ldots, X_n)) \equiv 0.$$

Then $(Y - p(X_1, \ldots, X_N)$ divides g in $k[X_1, \ldots, X_n, Y]$ and is irreducible in $k[X_1, \ldots, X_n, Y]$..

Theorem 7.3 *There is a probabilistic Turing machine that given an arithmetic circuit of size s computing a polynomial $g \in k[X_1, \ldots, X_n, Y]$ of total degree d with maximum coefficient size bounded by α, computes a list of circuits in time $\mathrm{poly}(s, d, \alpha)$ such that with probability $\geq 3/4$, for every p that is a root of g, there is a circuit in the list computing p.*

Proof. This follows immediately from Theorem 7.1 together with lemma 7.2. Simply take the list of circuits, subtract Y, and multiply by -1. ∎

7.2 Algebraic Nisan-Wigderson generator

We have a polynomial f in n variables. We want to test whether it is identically zero. To this aim, we design an algebraic version of the Nisan-Wigderson generator. It gets a seed $a \in k^\ell$ and has oracle access to a "hard" polynomial p in m variables. The parameters ℓ, m, and n are connected by Lemma 3.7. The generator will use a design $\mathcal{S} = (S_1, \ldots, S_n)$ with $|S_i| = m$ and $S_i \subseteq \{1, \ldots, \ell\}$ with $\ell = O(m^2/\log n)$ such that the size of the pairwise intersections $S_i \cap S_j$ is bounded by $\log n$. Such a design can be computed in time $\mathrm{poly}(n, 2^\ell)$. (To get this, set $\ell = m$ and $\gamma = \log n/m$ in Lemma 3.7. I am sorry for the clash of parameters, I will fix this in a future version of these notes.Maybe not.) The algebraic Nisan-Wigderson generator is the function

$$\begin{aligned} NW_{p,\mathcal{S}} : k^\ell &\to k^n \\ a &\mapsto (p(a|_{S_1}), \ldots, p(a|_{S_n}) \end{aligned}$$

where $a|_{S_i} \in k^m$ is the vector obtained by selecting the entries of a according to the indices in S_i. Recall that $|S_i| = m$.

Lemma 7.4 *Let $f \in k[Y_1, \ldots, Y_n]$ be a nonzero polynomial of total degree d_f that is computable by an arithmetic cirucit of size s. Let $p \in k[X_1, \ldots, X_m]$ be a polynomial of total degree d_p. Let $S \subseteq k$ be a set of size $> d_f d_p$. If $f(NW_{p,\mathcal{S}}(a)) = 0$ for all $a \in S^\ell$, then there is a circuit of size $\mathrm{poly}(m, n, d_f, d_p, s, \alpha, M)$ where α is an upper bound on the maximum coefficient size of f and p and M is an upper bound on the number of monomials of any restriction of p to $\log n$ variables.*

In general, we can bound M by $(d_p + 1)^{\log n}$. If p is multilinear, then we even have $M \leq 2^{\log n} = n$.

Proof. Like in the Boolean case, we will use a "hybrid" argument to obtain a small circuit of the polynomial p. Let

$$\begin{aligned} g_0(X_1, \ldots, X_\ell, Y_1, \ldots, Y_n) &= f(Y_1, \ldots, Y_n) \\ g_i(X_1, \ldots, X_\ell, Y_{i+1}, \ldots, Y_n) &= f(p(X_1, \ldots, X_\ell)|_{S_1}, \ldots, p(X_1, \ldots, X_\ell)|_{S_j}, Y_{i+1}, \ldots, Y_n) \qquad \text{for } 1 \leq j \leq i. \end{aligned}$$

We have $g_n(X_1, \ldots, X_n) = f(NW_{p,\mathcal{S}}(X_1, \ldots, X_\ell))$. The total degree of g_n is bounded by $d_f d_p$. Since g_n vanishes on S^ℓ by assumption and $|S| > d_f d_p$, $g_n \equiv 0$ by the Schwartz-Zippel lemma. Since $g_0 \not\equiv 0$ by assumption, there must by an i such that $g_i \not\equiv 0$ but $g_{i+1} \equiv 0$. Since $g_i(X_1, \ldots, X_\ell, Y_{i+1}, \ldots, Y_n) \not\equiv 0$, we can substitute values $\beta_{i+2}, \ldots, \beta_n$ for $Y_{i+2}, \ldots, Y_n$ such that the polynomial obtained is not identically zero (again, use the Schwartz-Zippel lemma). We denote this polynomial $g(X_1, \ldots, X_n, Y_{i+1}, \beta_{i+2}, \ldots, \beta_n)$ by $g(X_1, \ldots, X_n, Y)$.

By the choice of i, $g \not\equiv 0$, but $g(X_1, \ldots, X_m, p(X_1, \ldots, X_m)) \equiv 0$. By Theorem 7.3, there is an arithmetic circuit computing p, the size of which is polynomial in the degree of g, the circuit size of g, and the maximum coefficient size of g. Note that g is the composition of f with some instances of p. The degree of g is $\leq d_f d_p$, and the maximum coefficient size of g is bounded by $\text{poly}(\alpha, d_f)$. We do not have an upper bound on the arithmetic circuit size of p. But this is not necessary to get an upper bound on the circuit size of g. The instances of p only depend on X-variables the index of which is in $S_j \cap S_i$, that is, on at most $\log n$ variables. Therefore, we can compute each instance trivially by a circuit of size $\text{poly}(M)$ where M is the number of monomials of such a polynomial. ∎

Theorem 7.5 *Let $p = (p_m)$ be a familiy of exponential-time computable multilinear m-variate polynomials over $\mathbb{Z}$. Let $s_p(m)$ be the size of a minimum arithmetic circuit computing p_m over k. Assume that the maximum coefficient size of p_m is at most* $\text{poly}(m)$. *Let C be a circuit of size* $\text{poly}(n)$ *computing an n-variate polynomial f of degree $d_f = \text{poly}(n)$ and maximum coefficient size* $\text{poly}(n)$. *Then testing whether f is the zero polynomial can be done deterministically in time*

- 2^{n^ϵ} *for any $\epsilon > 0$ if $s_p(m) = m^{\omega(1)}$ and*
- $2^{\text{poly}\log n}$ *if $s_p(m) \in 2^{m^{\Omega(1)}}$.*

Proof. Our algorithm does the following: Let $S = \{1, \ldots, nd_f\}$. We evaluate f at all points $NW_{p,\mathcal{S}}(a)$ for $a \in S^\ell$. We claim that f is identically zero if f is zero at all points. Note that by the assumptions on the degree and coefficient sizes of f, $f(NW_{p,\mathcal{S}}(a))$ will have size $\text{poly}(n)$, hence we can evaluate C modulo 2^{n^t} for some constant t.

When $m = n^\epsilon$, we have $\ell = n^{2\epsilon}$. Note that the size of S is $\text{poly}(n)$ and that the degree of p_m is at most n for $\epsilon < 1$. The size of S^ℓ is at most $(nd_f)^\ell < 2^{n^{3\epsilon}}$. Assume that we can evaluate p on inputs of total size w in time 2^{w^c} for some constant c. Therefore we can evaluate $NW_{p,\mathcal{S}}$ at one point in time $< 2^{n^{3c\epsilon}}$ and at all points in time $< |S^\ell| \cdot 2^{n^{3c\epsilon}} < 2^{n^{4c\epsilon}}$. Since ϵ can we arbitrarily small, we get a subexponential-time deterministic identity test. The correctness of the test follows from Lemma 7.4, since all parameters occurring there are polynomial in m, hence we would get a polynomial upper bound for the circuit complexity of p_m.

In the second case, we set $m = \log^d n$ for some constant d to be chosen later. Now $\ell \leq \log^{2d} n$. The running time of the generator is $< |S^\ell| \cdot 2^{\log^{cd} n} < 2^{\log^{c'd} n}$ for some constant c'. The correctness follows again from Lemma 7.4. If the test fails, we would get a circuit of size n^t for some constant t which is independent of d. Since we can d make as large as we want, we get for any $\epsilon > 0$, a circuit of size 2^{m^ϵ} for p, a contradiction. ■

Unfortunately, we do not get anything, if we assume that p_m is computable in linear exponential-time. In particular, it is an open problem to get a deterministic polynomial time algorithm from an exponential lower bound. The problem is that the size of the set S^ℓ is superpolynomial.

8 A first lower bound for $\Sigma\Pi\Sigma$-circuits

We do not know any superpolynomial bounds for general arithmetic circuits, not even superlinear ones. Therefore, we will look at restricted circuits classes. The simplest circuit classes that we considered in the Boolean case were constant depth circuits. The algebraic analogon are constant depth arithmetic circuits with unbounded fanin sum and product gates. Since we do not want to count multiplications with constants, the edges in the circuit will be labelled with constants. If a gate g is connected to a gate g' by an edge e labeled with α, then the output of g is multiplied by α before it is fed into g'.

Circuits of depth one only compute linear forms or products of variables. Circuits of depth two consist of a product gate at the top and sum gates at the bottom or a sum gate at the top or product gates at the bottom. In the first case, we compute a product of linear forms. In this model, we can only compute polynomials that factor in to linear forms. The second case is the sparse representation of polynomials. The size of such circuits it polynomial in the number of monomials and the degree. Depth three circuits have a product gate at the top, a layer of sum gates, followed by a layer of product gates. Such circuits compute products of sparse polynomials.

The first nontrivial case are depth three circuits with a sum gate at the top, a layer of products gates, and a final layer of sum gates at the bottom. Such circuits compute sums of products of linear forms. This is the first nontrivial case in in this model, and we do not know superpolynomial lower bounds over fields of characteristic zero. We will even see that exponential lower bounds will yield superpolynomial bounds for general arithmetic circuits.

Definition 8.1 *A $\Sigma\Pi\Sigma$-circuit is an arithmetic circuit with an unbounded fanin sum gate at the top, followed by a layer of unbounded fanin product gates, and a layer of unbounded fanin sum gates at the bottom-*

Definition 8.2 *An arithmetic circuit is called homogeneous, if at each gate, it computes a homogeneous polynomial.*

Definition 8.3 *Let $f \in k[X_1, \ldots, X_n]$ be a polynomial of degree d. Let $\ell \leq d$ and $i_1, \ldots, i_\ell \in \{1, \ldots, n\}$. The polynomial $\frac{\partial^\ell}{\partial X_{i_1} \ldots \partial X_{i_\ell}} f$ is called a partial derivative of f. We denote the vector space spanned by the set of all partial derivates of a polynomial f by $\partial(f)$.*

Lemma 8.4 *Let $f, g \in k[X_1, \ldots, X_n]$ be polynomials and $\alpha \in k \setminus \{0\}$.*

1. $\dim \partial(\alpha \cdot f) = \dim \partial(f)$,
2. $\dim \partial(f+g) \leq \dim \partial(f) + \dim \partial(g)$,
3. $\dim \partial(fg) \leq \dim \partial(f) \cdot \dim \partial(g)$.

Proof. The first two items follow immediately from the fact that taking any partial derivative is a linear operator. If $u_1, \ldots, u_M$ is a basis of $\partial(f)$ and $v_1, \ldots, v_N$ is a basis of $\partial(g)$, then $u_1, \ldots, u_M$ is a basis of $\partial(\alpha \cdot f)$, too, and $u_1, \ldots, u_M, v_1, \ldots, v_N$ spans $\partial(f+g)$. Finally, by the product rule, $u_i v_j$, $1 \leq i \leq M$, $1 \leq j \leq N$, spans $\partial(fg)$. ∎

Theorem 8.5 *Let C be a $\Sigma\Pi\Sigma$-circuit computing a polynomial f such that the degree of the output of every multiplication is bounded by d and the fanin of the sum gate at the top is s. Then $\dim \partial(f) \leq s2^d$.*

Proof. A sum gate at the bottom of C computes a linear function ℓ. We have $\partial(\ell) = \text{lin}\{1, \ell\}$ and $\dim \partial(\ell) = 2$. By Lemma 8.4, if p is the polynomial computed at some multiplication gate, then $\dim \partial(p) \leq 2^d$. Once more, by Lemma 8.4, $\dim \partial(f) \leq s2^d$. ∎

Let

$$\text{Sym}_n^d(X_1, \ldots, X_n) = \sum_{I \subseteq \{1,\ldots,n\},\ |I|=d} \prod_{i \in I} X_i$$

be the ith symmetric polynomial.

Theorem 8.6 *Every homogenous $\Sigma\Pi\Sigma$ circuit computing Sym_n^d has size at least $\Omega((\frac{n}{4d})^d)$.*

Proof. We can assume that in a homogenous circuit, every product gate computes a polynomial of degree exactly $2d$. Otherwise, it can be removed since the results of all product gates computing a polynomial of degree different from $2d$ cancel. Therefore, the result follows from Theorem 8.5 and the lemma below. ∎

If we set $d = \sqrt{n}$, we get an exponential lower bound.

Lemma 8.7 $\dim \partial(\text{Sym}_n^{2d}) \geq \binom{n}{d}$.

Proof. Let $I = \{i_1, \ldots, i_d\}$ be a set of indices. In

$$\frac{\partial^d}{\partial X_{i_1} \ldots X_{i_d}} \text{Sym}_n^{2d},$$

a monomial $\prod_{j\in J} X_j$ survives only if $I \subseteq J$. If $|I| = d$, then

$$\frac{\partial^d}{\partial X_{i_1} \dots X_{i_d}} \mathrm{Sym}_n^{2d} = \sum_{J\subseteq\{1,\dots,n\}\setminus I, |J|=d} \prod_{j\in J} X_j.$$

Every partial derivative of order d of Sym_n^{2d} corresponds to a set I of size d. Write the coefficients of such a derivative as a row vector and label its entries by sets $J \subseteq \{1,\dots,n\}$ of size d. There is a one in position J if $I \cap J \neq 0$. If we list all the derivatives of order d row by row, we get a $\binom{n}{d} \times \binom{n}{d}$-matrix $(a_{I,J})$ with $a_{I,J} = 1$ if $I \cap J = \emptyset$ and 0 otherwise. This matrix is called the disjointness matrix. It has full rank (Exercise). ∎

The restriction to homogeneous circuits is necessary. There are nonhomogeneous ΣΠΣ-circuits of polynomial size computing Sym_n^d. Note that

$$(Y - X_1) \cdots (Y - X_n) = \sum_{i=0}^{d} \mathrm{Sym}_n^{n-i} \cdot Y^i$$

We evaluate the polynomial at $n+1$ values $v_0, \dots, v_n \in k$. Note that

$$s_j := (v_j - X_1) \cdots (v_j - X_n)$$

can be computed by a product gate at the top and sum gates at the bottom. We have

$$\begin{pmatrix} s_0 \\ \vdots \\ s_n \end{pmatrix} = \begin{pmatrix} 1 & \dots & v_0^n \\ \vdots & \ddots & \vdots \\ 1 & \dots & v_n^n \end{pmatrix} \begin{pmatrix} \mathrm{Sym}_n^n \\ \vdots \\ \mathrm{Sym}_n^0 \end{pmatrix}$$

We get the symmetric polynomials by multiplying the lefthand side with the inverse of the matrix. Note that this inverse is just a matrix filled with constants. The matrix-vector product can be realized by an additional sum gate at the top.

9 Valiant's classes

Our objects that we study will be families of polynomials. Each polynomial will depend on a polynomial number of variables and its degree will be bounded by a polynomial.

Definition 9.1 *Let $X_1, X_2, \ldots$ be a sequence of variables over a field k. A sequence of polynomials (f_n) is called a p-family, if there are polynomials p and q such that $f_n \in k[X_1, \ldots, X_{p(n)}]$ and $\deg f_n \leq q(n)$.*

Definition 9.2 *A p-family (f_n) is in VP, if there is a sequence of arithmetic circuits (C_n) such that C_n has size $\mathrm{poly}(n)$ and computes f_n.*

VP is an obvious algebraic analogue of P, more precisely, of $\mathsf{P/poly}$, since it is a nonuniform class.

Definition 9.3 *A p-family (f_n) is in VNP, if there are polynomials p and q and a sequence $(g_n) \in \mathsf{VP}$ of polynomials $g_n \in k[X_1, \ldots, X_{p(n)}, Y_1, \ldots, Y_{q(n)}]$ such that*

$$f_n = \sum_{e \in \{0,1\}^{q(n)}} g_n(X_1, \ldots, X_{p(n)}, e_1, \ldots, e_{q(n)})$$

You can think of the X-variables representing the input and the Y-variables the witness. With this interpretation, VNP is more like $\#\mathsf{P}$. In particular, we will see that the permanent polynomial

$$\mathrm{per}_n = \sum_{\sigma \in S_n} X_{1,\sigma(1)} \cdots X_{n,\sigma(n)}$$

is complete for VNP. On the other hand, the determinant family $(\det_n)$ is contained in VP. We do not know whether it is complete for VP, we will discuss this issue later.

Definition 9.4 *A family $f_n \in \mathsf{VP}$ is contained in the class VP_e if there is a family of formulas (F_n) such that F_n has polynomials size and computes f_n. A family of polynomials f_n is in VNP_e if in the definition of VNP, the family (g_n) is in VP_e.*

The "e" in the subscript stands for *expression*, another word for formula.

9.1 Reduction and completeness

Definition 9.5 *1. A polynomial $p \in k[X_1, \ldots, X_m]$ is a* projection *of a polynomial $q \in k[Y_1, \ldots, Y_n]$ if there is a mapping $s : \{Y_1, \ldots, Y_n\} \to \{X_1, \ldots, X_m\} \cup k$ such that $p(X_1, \ldots, X_m) = q(s(Y_1), \ldots, s(Y_n))$. We write $p \leq q$ in this case.*

2. A p-family $p = (p_n)$ is a projection of a p-family $q = q(n)$ if there is a function $r : \mathbb{N} \to \mathbb{N}$ such that $r(n) = \text{poly}(n)$ and $p_n \leq q_{r(n)}$. In this case, we write $p \leq_{\text{P}} q$.

p-projections play the role of many one reductions in Valiant's world. They are, however, somewhat weaker, since we are only allowed to replace input variables by other input variables or constants.

It is quite easy to verifiy that p-projections have all the properties that a reduction should have.

Lemma 9.6 *1. $\leq_{\text{P}}$ is transitive.*

2. The classes VP_e, VP, VNP_e, and VP are closed under $\leq_{\text{P}}$, that is, if a p-family q is in one of these classes and $p \leq_{\text{P}} q$ then p is in the same class.

Definition 9.7 *A p-family q is called* VNP*-hard, if for all $q \in \mathsf{VNP}$, $q \leq_{\text{P}} p$. (In the same way, we can define* VP*-hardness, etc.) If in addition q is in* VNP*, then we call q* VNP*-complete.*

Note that since p-projections are weak reductions, they are also suited to define VP-completeness. Polynomial-time many-one reductions are not useful to define P-completeness, since the whole computation can be carried out in the reduction.

Lemma 9.8 *If a p-family p is* VNP*-hard and $p \in \mathsf{VP}$, then $\mathsf{VNP} = \mathsf{VP}$.*

Proof. Let $q \in \mathsf{VNP}$ be arbitrary. Since p is VNP-hard, then $q \leq_{\text{P}} p$. Since $p \in \mathsf{VP}$, $q \in \mathsf{VP}$, too, since VP is closed under p-projections. ■

Valiant's hypothesis states that $\mathsf{VP} \neq \mathsf{VNP}$. Once we have shown that per is VNP-complete, Valiant's hypothesis is equivalent to proving per $\notin$ VP-

9.2 A characterisation of VP

Definition 9.9 *An arithmetic circuit is* multiplicatively disjoint *if for all multiplication gates, the subcircuits induced by its two children are disjoint.*

Multiplicatively disjoint circuits are between circuits and formulas. In a formula, also the subcircuits of addition gates are disjoint.

Definition 9.10 *Let C be an arithmetic circuit. The formal degree of a gate g is defined inductively: A leaf has formal degree 1. If g is a multiplication gate, then its formal degree is the sum of the formal degrees of its two children. If g is an addition gate, then the formal degree of g is the maximum of the formal degrees of its children. The formal degree of C is the formal degree of its output gate.*

The formal degree of a circuit disregards that the degree at gate might drop when there are cancellations. Multiplications with constants might also increase the formal degree. This will be important when we consider so-called constant-free variants of Valiant's classes.

Lemma 9.11 *If a circuit has size s and formal degree d, then there is a circuit C' of size $\leq sd$ computing the same polynomial.*

Proof. Each gate g of formal degree $e \leq d$ will be replaced by $d+1-e$ copies $g_1, \ldots, g_e$. Let g_i be one of these copies. We call i the index of the copy. We will make sure that all gates of the subcircuit with output g_i are copies with an index lying between i and $i+e-1$. In this way we ensure that we will get multiplicatively disjoint circuits.

Inductively, we construct a circuit C_e with the following property: For each gate g for formal degree $f \leq e$ in C, there are copies of the gates $g_1, \ldots, g_{d+1-f}$ in C_e computing the same function as g and all the gates of the subcircuit with root g_i have indices lying between i and $i+f-1$.

C_1 just consists of d copies of the input nodes and the nodes carrying constants. Assume that we constructed C_{e-1}. To obtain C_e, we now add copies of all gates g of formal degree e in C. Let g' and g'' be the children of g of formal degrees e' and e'', respectively.

We start with the multiplication gates. In this case $e = e' + e''$ with $e', e'' < e$. This means that the copies $g'_1, \ldots, g'_{d+1-e'}$ and $g''_1, \ldots, g''_{d+1-e''}$ were constructed in a previous step. We add the copies $g_1, \ldots, g_{d+1-e}$ and connect g_i with g'_i and $g''_{i+e'}$. These copies exist, since $i \leq d+1-e \leq d+1-e'$ and $i+e' \leq d+1-e+e' = d+1-e''$. The indices of the copies of the subcircuit with root g'_i lie between i and $i+e'-1$, the indices of the copies in the subcircuit with root $g''_{i+e'}$ lie between $i+e'$ and $i+e'+e''-1 = i+e-1$. Furthermore, $i \leq i+e' \leq i+e-1$. Therefore the condition on the indices of the subcircuits is fulfilled.

Next come the addition gates of formal degree e. Note that an addition gate of formal degree e might have a predecessor of formal degree e. Since C is acyclic, we can order the addition gates in such a way, that whenever we deal with a gate g, all its predecessors have been processed. For each addition gate g of formal degree e, we add copies $g_1, \ldots, g_{d+1-e}$. Let g' and g'' be the children of g in C with formal degrees $e' \leq e$ and $e'' \leq e$, respectively. We connect g_i with the copy g'_i and g''_i. The indices of the copies in these subcircuits lie in the range from i to $i+e'$ and $i+e''$, respectively.

The circuit C_d is the circuit we are looking for. It contains a copy of the output gate of C. The circuit is multiplicatively disjoint by the way we chose the indices when connecting the copies of the children to the multiplication gate. ■

Theorem 9.12 (Malod & Portier) *A p-family (g_n) is in* VP *if and only if there is a family of polynomial size multiplicative disjoint circuits (C_n) computing (g_n).*

Proof. One direction immediately follows from Lemma 9.11. It can be easily proven by induction that the degree of a multiplicatively disjoint circuit of size s is bounded by s. ■

Exercise 9.1 *Prove the last statement.*

10 Formulas

Lemma 10.1 *Let T be a binary tree with s nodes. Then there is an edge e in T such that removing e separates T into two trees both having between $n/3$ and $2n/3$ nodes.*

Proof. We construct a path $u_1, \ldots, u_m$ starting from the root as follows: We set u_1 to be the root of the path. Let u_i be the current end node of the path and let w and w' be its children. If the subtree with root w is larger than then subtree with root w', then $u_{i+1} := w$, otherwise $u_{i+1} := w'$. We stop when the size of the subtree with root u_i is $< 2/3n$. The edge e is the edge (u_{m-1}, u_m). The subtree with root u_m has size $< 2/3n$ by construction. The subtree with root u_{m-1} has size $\geq 2/3n$. Since u_m is the root of the larger subtree, its size is as least $n/3$. The size of the remaining tree is between $n - 2/3n = n/3$ and $n - n/3 = 2n/3$. ∎

Theorem 10.2 *Let F be a formula of size s. Then there is a formula F' of size* $\mathrm{poly}(s)$ *and depth $O(\log s)$ computing the same polynomial as F.*

Proof. By Lemma 10.1, there is an edge in F such that when removing e, we get two parts, each of size between $s/3$ and $2s/3$. The part not containing the output gate of F is again a formula, which we call H. The part containing the output gate is not a formula, since one of the gates has fanin one after removal of e. We add a new child to this gate, which is labeled with a new input variable Y. Call the resulting formula G. G computes a linear form $aY + b$, since Y appears only once in G. (a and b are polynomials in the original input variables.) If we substitute the polynomial h computed by H for Y, then we get the polynomial f computed by F. If we substitute 1 for Y, then we get $a + b$, and if we substitute 0 for Y, then we get b. Therefore, there are formulas of size $\leq 2n/3$ computing $a+b$, b and h. With these formulas, we can proceed recursively. We get formulas G'_1, G'_0, and H' computing $a + b$, b, and h, respectively. We can combine them to a formula computing $ah + b = f$ as depicted in Figure 10.1: For the size $s(n)$, we get the recursion

$$s(n) = 4 \cdot s(2n/3) + 3$$

and for the depth $d(n)$, we get the recursion

$$d(n) = d(2n/3) + 3.$$

It is a routine check that $s(n) = \mathrm{poly}(n)$ and $d(n) = O(\log n)$. ∎

Figure 10.1: The new formula computing f

Let $X_{i,j}^{(\ell)}$, $1 \leq i, j, \ell \leq n$, be indeterminates and let $M_\ell = (X_{i,j}^{(\ell)})_{1 \leq i,j \leq n}$ for $1 \leq \ell \leq n$. The polynomial IMM_n is a polynomial in n^3 variables and is the $(1,1)$ entry of the matrix product $M_1 \cdots M_n$. The p-family $\mathrm{IMM} = (\mathrm{IMM}_n)$ is the called *iterated matrix multiplication*. By using the trivial algorithm for matrix multiplication, it is easy to see that $\mathrm{IMM} \in \mathsf{VP}$.

If the matrices M_ℓ are of constant size $c \times c$, then we get the family $\mathrm{CIMM}^{(c)} = (\mathrm{CIMM}_n^{(c)})$. $\mathrm{CIMM}_n^{(c)}$ is a polynomial in $c^2 n$ variables.

Theorem 10.3 (Ben-Or & Cleve) *Let F be a formula of depth d computing a polynomial f, then f is a projection of* $\mathrm{CIMM}_{4^d}^{(3)}$.

Proof. We will prove by induction on d, that we can find 4^d 3×3-matrices the entries of which are either indeterminates or constants such that the product of these matrices is

$$\begin{pmatrix} 1 & f & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This is obviously true for depth zero formulas, since these formulas compute constants or single variables.

If the depth d is larger than zero, we either have $f = g + h$ or $f = gh$ and g and h are both computed by formulas of depths $\leq d-1$. By the induction hypothesis, there are two sets of 4^{d-1} 3×3-matrices each such that their products are

$$\begin{pmatrix} 1 & g & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

respectively. In the case of an addition gate we have

$$\begin{pmatrix} 1 & g & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & g+h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Therefore we can write f as a projection of a 3×3-iterated matrix multiplication of length $2 \cdot 4^{d-1} \leq 4^d$.

In the case of a multiplication gate, we have

$$\begin{pmatrix} 1 & g & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & h \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & g & gh \\ 0 & 1 & h \\ 0 & 0 & 1 \end{pmatrix}.$$

Note that h is standing in the "wrong" position. But we can easily fix this by applying permutation matrices from the left and the right. This just corresponds to exchanging the rows or columns of the first and last matrix of the corresponding matrix product, respectively. We proceed with

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -h \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & g & gh \\ 0 & 1 & h \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & -g & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & gh \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Note that we now have a $-g$ and $-h$ instead of a g and h. But this is easily by multiplying the second row and column by -1. This can again be achieved by doing this with the first and last matrix of the 4^{d-1} matrices. Altogether, we get that f is a projection of a product of $4 \cdot 4^{d-1} = 4^d$ matrices. ∎

Corollary 10.4 $\mathrm{CIMM}^{(3)}$ *is* VP_e *complete.*

Proof. Let $f = (f_n) \in \mathsf{VP}_e$. Let F_n be a formula of polynomial size computing f_n. By Theorem 10.2, there is an equivalent formula of polynomial size and depth $O(\log n)$. By Theorem 10.3, f_n is a projection of $\mathrm{CIMM}^{(3)}_{\mathrm{poly}(n)}$. This proves the hardness.

To construct a formula of polynomial size for $\mathrm{CIMM}^{(3)}_n$, we divide the product into two products of size $n/2$ each. We can assume that n is a power of 2, since $\mathrm{CIMM}^{(3)}_{n'} \leq \mathrm{CIMM}^{(3)}_n$ if $n' \leq n$. The entries of the result of the two products can we computed by 18 instances of $\mathrm{CIMM}^{(3)}_{n/2}$. From these two results, we can compute $\mathrm{CIMM}^{(3)}_n$ by a constant size formula. Therefore, we get the following recursion for the size $s(n)$ of the formula:

$$s(n) = 18s(n/2) + O(1).$$

Therefore, $s(n) = \mathrm{poly}(n)$. ∎

Todo: Note on $\mathrm{CIMM}^{(2)}$**.**

11 The determinant

Let $M = (m_{i,j})$ be an $n \times n$ matrix. We can interpret M as the weighted adjacency matrix of some graph over the node set $\{1, \ldots, n\}$. For every (i, j), there is an edge (i, j) of weight $m_{i,j}$. A *cycle cover* in a directed graph is a collection of node-disjoint directed cycles such that every node is contained in exactly one cycle. Permutations in S_n stand in a one-to-one correspondence with cycle covers. Every permutation σ yields a cycle cover consisting of the edges $(i, \sigma(i))$. On the other hand, the edges of a cycle cover encode a permutation of the nodes with the intepretation that an edge (i, j) means that i is mapped to j. Note that this is nothing but the cycle decomposition of a permutation. The sign of the permutation is -1 if the number of cycles is even, and 1 if it is odd. The weight $w(C)$ of a cycle cover C is the product of the weights of the edges in it. Therefore,

$$\det M = \sum_{\text{cycle covers } C} (-1)^{1+\text{number of cycles in } C} w(C)$$

Conceptually, it is often easier to think of an edge of weight zero as not being present in the graph. Since the weight of a cycle cover is the product of its edge weights, this does not make any difference in the above equation for $\det M$.

Definition 11.1 *A circuit is called* weakly skew *if every multiplication gate g has at least one child g' such that after removing the edge (g', g), the graph consists of two weakly connected components.*

In a formula, this is true for every child of a gate, i.e., no intermediate result is reusable. In a weakly skew circuit, one child of every gate can be reused, but not both. Weakly skew is however stronger than multiplicatively disjoint, since in the later case, while the subcircuits need to be disjoint, they can be connected to the rest of the circuit.

We formalize the notion of begin *reusable*.

Definition 11.2 *Let C be an arithmetic circuit. The set of* reuseable *gates in C is inductively defined as follows: Every gate of outdegree zero is reusable. (We consider circuits with multiple output gates to simplify some proofs in the following.) We remove every gate g of outdegree zero from C and for each such multiplication gate, we also remove the subcircuit of that child g' that is only connected to the rest of the circuit via the edge (g', g). Let C' be the resulting circuit. Every gate that is reusable in C' is reusable in C, too.*

Definition 11.3 *Let k be a field and $X_1, \ldots, X_n$ be indeterminates.*

1. *An* algebraic branching program *A is an acyclic graph with two distinguished nodes s and t and an edge labeling with labels from $k \cup \{X_1, \ldots, X_n\}$.*
2. *The weight $w(P)$ of a path P from s to t is the product of the labels of the edges in the path.*
3. *The polynomial computed by A is*
$$\sum_{s\text{-}t \text{ path } P} w(P).$$
4. *The size of an arithmetic branching program is the number of edges in it.*
5. *A is called* layered *if for every node v in A, all s-v paths have the same length.*

If A is layered, then we can think of the nodes of A being grouped into layers: two nodes are in the same layer if the length of any path from s to them is the same. In a layered branching program, edges only go from one layer to the next.

Lemma 11.4 *Let A be a branching program of size s. Then there is a layered branching program of size $O(s^2)$ computing the same function.*

Proof. For a node v in the branching program, let $d(v)$ be the length of a longest path from s to v. Scan through the nodes by increasing value of $d(v)$ (breaking ties arbitrarily) until you find the first node v_0 such that there is a path from s to v_0 that has length $\ell < d(v_0)$. Take the last edge e of such a path and subdivide it $d(v_0) - \ell$ many times. If e has label w, then one of the new edges gets label w and all other ones will get label 1. We do this with all shorter paths from s to v_0 and then go on with the next node at which the layering property is violated. ∎

Theorem 11.5 *Let $f \in k[X_1, \ldots, X_n]$ with $\deg f = \mathrm{poly}(n)$. The following statements are equivalent:*

1. *f is computed by a weakly skew circuit of size $\mathrm{poly}(n)$.*
2. *f is computed by an algebraic branching program of size $\mathrm{poly}(n)$.*
3. *f is a projection of $\det_{p(n)}$ for some polynomially bounded function p.*
4. *f is a projection of $\mathrm{IMM}_{p(n)}$ for some polynomially bounded function p.*

Proof. (1) $\Rightarrow$ (2): Assume that f is computed by a weakly skew circuit C of size m. We now prove by induction on m that there is a arithmetic branching program computing A of size $\leq 2m$ such that for every reusable gate g in C there is a node v_g such that the sum of the weights of all paths from s to v_g is the same polynomial as computed at g. If C is an input node, then A consists of a single edge.

If g is an addition gate, then we remove g from C, let C' be the resulting circuit. By the induction hypothesis, there is an algebraic branching program such that for every gate g' that is reusable in C', there is a node $v_{g'}$ in C' such that the sum of the weights of all path from s to $v_{g'}$ equals the polynomial computed at g'. Let h and h' be the children of g'. We add a new node v_g and connect the nodes v_h and $v_{h'}$ to it. Both edges get weight one. If $h = h'$, then we add only one edge with weight two. By construction, the sum of the weights of all paths from s to v_g is the sum of the polynomials computed at h and h'. The resulting arithmetic branching program has two more edges than A'.

If g is a multiplication gate, then after removal of g, we get two separate circuits C_1 and C_2. Let g_1 and g_2 be the children of g. Only the gates of one of them, say C_2, can be reusable in C. Let m_1 and m_2 be the sizes of C_1 and C_2. From the induction hypotheses, we get corresponding algebraic branching programs A_1 and A_2 with sources s_1 and s_2. In A_1, there are vertices s_1 and v_{g_1} such that the sum of the weights of all path from s_1 to v_{g_1} equals the polynomial computed at g_1. We identify the node s_1 of A_1 with the node v_{g_2} in A_2. Then the sum of the weights of all path from s_2 to v_{g_1} is the product computed at g. For all gates h in C_2, the sum of the weights of all paths from s_2 to v_h paths equals the polynomial computed at h. The size of the new branching program is $2m_1 + 2m_2 \leq 2m$.

(2) $\Rightarrow$ (3): Let A be an algebraic branching program computing f. By Lemma 11.4 we can assume that A is layered. Let ℓ be the maximum size of a layer and let m be the number of layers. We will inductively construct $\ell \times \ell$-matrices $M_1, \ldots, M_m$ with entries from $k \cup \{X_1, \ldots, X_n\}$ such that the first row of $M_1 \cdots M_i$ are the polynomials computed at the nodes in the ith layer, that is, the sum of the weights of all path from s to each node in this layer. M_1 has a one in position $(1,1)$ and zeroes elsewhere. This one corresponds the the source node s. Assume we constructed $M_1, \ldots, M_i$. Let $(a_1, \ldots, a_\ell)$ be the first row of $M_1 \cdots M_i$. A node v in the $(i+1)$th layer receives edges from the nodes of the ith layer. Let $(b_1, \ldots, b_\ell)$ be the labels of these edges (if an edge is not present, the corresponding $b_j = 0$.) The polynomial computed at v is given by

$$(a_1, \ldots, a_\ell) \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_\ell \end{pmatrix}.$$

Figure 11.1: Transforming weakly skew circuits into algebraic branching programs.

The matrix M_{i+1} simply consists of the corresponding columns $(b_1, \ldots, b_\ell)^T$. Since we can embed a product of m $\ell \times \ell$-matrices into a product of d $d \times d$-matrices with $d = \max\{m, \ell\}$, we get that f is a projection of $\mathrm{IMM}_{\mathrm{poly}(n)}$.

(3) $\Rightarrow$ (4): Note that an iterated matrix product can be easily computed by a layered algebraic branching program, you just have to "reverse" the construction of the previous step. Therefore it suffices to prove that every polynomial that is computed by a layered algebraic branching program A is a projection of a determinant of polynomial size. We modiy A as follows: add an edge of weight one from t to s and add a self loop of weight one to every node except s and t. Let M be the weighted adjacency matrix of this modified program A'. $\det M$ is the sum of the weights of all cycle covers in A'. All cycle covers in A' consist of one big cycle through s and t and the remaining nodes are covered by self-loops. Since the program is layered, all cycle covers have the same number of cycles and therefore the same sign. The weight of a cycle cover equals the weight of the corresponding path from s to t, potentially with an opposite sign. Therefore, f is a projection of a polynomially large matrix.

(4) $\Rightarrow$ (1): One way to evaluate the determinant by a weakly skew circuit is known as Csanky's algorithm [**?**], see the next exercise. ■

Exercise 11.1 *The characteristic polynomial of a matrix A is defined as $c_A(X) = \det(A - X \cdot I)$ where I is the identity matrix. Let $c_A(X) = s_{A,0}X^n + s_{A,1}X^{n-1} + \cdots + s_{A,n}$.*

1. *Show that*

$$s_{A,0} = (-1)^n$$
$$s_{A,k} = \frac{1}{k}\sum_{\kappa=1}^{k}(-1)^{\kappa-1}s_{k-\kappa}\,\mathrm{trace}(A^\kappa), \qquad 1 \le k \le n.$$

2. *Show that $s_{A,n} = \det A$.*

3. *Show that there is a logarithmic space uniform family of Boolean circuits of polynomial size and polylogarithmic depth that computes the determinant of a matrix A. (Assume that A has dimension $n \times n$ and entries with $p(n)$ bits for some polynomial p.)*

Definition 11.6 *A p-family (f_n) is in* VDET *if it is a p-projection of* $(\det_n)$.

Theorem 11.5 gives us further, equivalent definitions of VDET.

12 The permanent

12.1 VNP and formulas

Definition 12.1 *Let C be an arithmetic circuit.*

1. *A* parse tree *of C is defined recursively as follows: Every circuit consisting of one node is a parse tree. If the size of C is larger than one, let g be the output gate and g_1 and g_2 be its children. Let C_1 and C_2 be the subcircuits with output gates g_1 and g_2. If g is an addition gate, then we get the set of all parse trees by either taking a parse tree of C_1 or a path tree of C_2 and connecting it to g. If g is a multiplication gate, then we get the set of all parse trees by taking a parse of C_1 and a parse tree of C_2 and connecting both to g.*

2. *The set of all parse trees of C is denoted by* $\mathrm{PT}(C)$.

3. *The weight $w(T)$ of a parse tree T is the product of the labels of its leaves.*

For every multiplication gate, we have to include both children in the parse tree, for every addition gate we have to choose one of them. Note that a gate may occur several times in a parse tree, since it is reused in the circuit several times. For each occurrence in the parse tree, we introduce a new copy. (Otherwise, it would not be a tree.)

Let p be the polynomial computed by C. It is quite easy to prove by structural induction that

$$p = \sum_{T \in \mathrm{PT}(C)} w(T).$$

Lemma 12.2 *A circuit C is multiplicatively disjoint if every parse tree of C is a subcircuit of C.*

Proof. Assume that C is not multiplicatively disjoint. Then there is a node v in C such that there a two node disjoint paths to some multiplication gate g. Since g is a multiplication gate, these two paths can be extended to a parse tree.

Conversely, if there is a parse tree T that is not a subcircuit of C, then there are gates g and h in T such that there a two nodes disjoint path from g to h. Since T is a parse tree, h is a multiplication gate. Thus, C is not multiplicatively disjoint. ∎

Lemma 12.3 *Let C be a multiplicatively disjoint circuit with edge set E. For each edge $e \in E$, let X_e be an indeterminate. There is a formula F in the X_e's of size polynomial in the size of C such that for every $a \in \{0,1\}^{|E|}$, $F(a)$ is the weight of the parse tree, if the edges "selected" by the vector a form a parse tree in C, and zero otherwise.*

Proof. For every node v in C, we introduce an additional variable Y_v. Note that for $\{0,1\}$ valued variables X and Y, we can simulate Boolean AND by XY and Boolean NOT by $1 - X$. We can write the fact that a given vector encodes a parse tree by the following Boolean expressions:

$$\bigwedge_{(i,j)\in E} X_{(i,j)} \Rightarrow Y_i \wedge Y_j$$

ensures that whenever an edge is selected, its end points are selected, too. Let g be the output gate of C. Then

$$Y_g$$

ensures that the output gate is selected. For a gate g, let $\ell(g)$ and $r(g)$ be its children. The following expression ensures that for every multiplication gate g that is selected, both incoming edges are selected, too.

$$\bigwedge_{\text{multiplication gate } g} Y_g \Rightarrow X_{(\ell(g),g)} \wedge X_{(r(g),g)}.$$

If we replace the Boolean AND on the righthand side by a Boolean XOR, we get an expression that checks for every selected addition gate whether exactly one of the incoming edges is chosen. Finally, we have to check that every selected gate has at least one outgoing edge. This is done by the following expression:

$$\bigwedge_{v\in V} \left(Y_v \Rightarrow \bigvee_{(v,u)\in E} X_{(v,u)} \right).$$

We can eliminate all occurences of the newly introduced variables by replacing Y_v by the expression

$$\bigvee_{(v,u)\in E} X_{(v,u)}$$

and Y_g by 1. The Boolean AND of these expressions is a Boolean formula that is true iff the vector a encodes a parse tree. By the considerations above, it can be replaced by an arithmetic formula.

If a encodes a parsetree, we can get the corresponding weight by the following expression:

$$\prod_{v\in V} (Y_v \cdot w_v + 1 - Y_v).$$

Here w_v is the label of v if it is an input gate and 1 otherwise. Again, we can eliminate the Y_v's as above. The product of the two expressions, one for checking whether a is a parse tree and one for computing its weight, is the formula F. ■

Corollary 12.4 *Let f be a polynomial computed by an arithmetic circuit of size s. Then there is an arithmetic formula F of size polynomial in s and a polynomial p such that*

$$f(X) = \sum_{a \in \{0,1\}^{p(s)}} F(X, a).$$

Theorem 12.5 $\mathsf{VNP} = \mathsf{VNP}_e$.

Proof. Let (f_n) be in VNP and $(g_n) \in \mathsf{VP}$ such that

$$f(X) = \sum_{e \in \{0,1\}^{q(n)}} g_n(X, e).$$

As seen above, there is a formula F_n of polynomial size such that

$$g_n(X, Y) = \sum_{a \in \{0,1\}^{p(n)}} F_n(X, Y, a).$$

Therefore,

$$f(X) = \sum_{e \in \{0,1\}^{p(n)}, a \in \{0,1\}^{q(n)}} F_n(X, e, a).$$ ■

While the statement of the theorem sounds astonishing at a first glance, it just uses the fact that we can write the result of a polynomially large circuit by an exponential sum over a polynomially large formula and then combines the two exponential sums into one.

12.2 Hardness of the permanent

Let $G = (V, E)$ be an edge weighted graph. A cycle cover C of G is a selection of node disjoint directed cycles such that every node is contained in exactly one cycle. The weight $w(C)$ of C is the product of the weight of the edges in C. Cycle covers can be viewed as the graph of a permutation. The cycles in the cycle cover correspond to the cycles in the cycle decomposition of a permutation. If we also write G for the weighted adjacency matrix of G (by abuse of notation), then

$$\mathrm{per}(G) = \sum_{\text{cycle cover } C \text{ of } G} w(C).$$

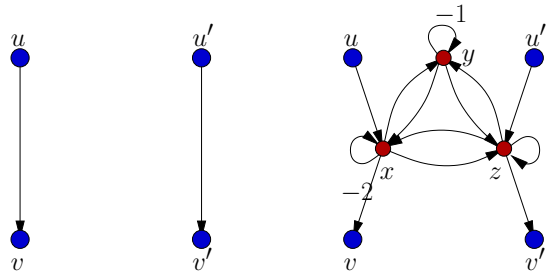

Figure 12.1: The equality gadget. The pair of edges (u, v) and (u', v') of the left-hand side is connected as shown on the right-hand side.

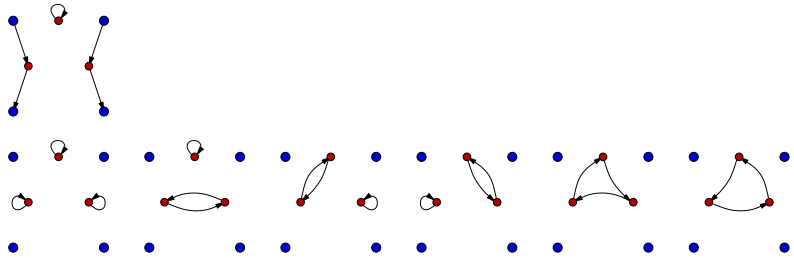

Figure 12.2: First row: The one possible configuration if both edges are taken. Second row: The six possible configurations if none of the edges is taken.

Let G be a graph and $e = (u, v)$ and $e' = (u', v')$ be two edges in G. As a first step, we want to replace G by a graph $\hat{G}$ such that $\mathrm{per}(\hat{G})$ is the sum over all $w(C)$ such that C is a cycle cover of G that either contains both e and e' or none of them. This is achieved by subdividing the edges an connecting them by an equality gadget as depicted in Figure 12.1.

Let C be a cycle cover of G that takes both edges. Then there is one way to extend this to a cycle cover of $\hat{G}$. The weight of this new cycle cover is $-2 \cdot w(C)$, see Figure 12.2. When C does not take any of the two edges, then there are six ways to extend C. This six ways some up to weight $-2 \cdot w(C)$.

Finally, there are inconsistent ways to cover the equality gadget in G, see Figure 12.3. We can pairs these covers in such a way the two covers have the same weight, but differ in sign. Therefore their contribution cancels.

Lemma 12.6 *Let k be a field of characteristic distinct from 2. Let G be a graph and e and e' be edges in G. Then there is a graph $\hat{G}$ such that*

$$-\frac{1}{2}\,\mathrm{per}(\hat{G}) = \sum_C w(C),$$

where the sum is taken over all cycle covers C of G that either use both of e and e' or none of them.

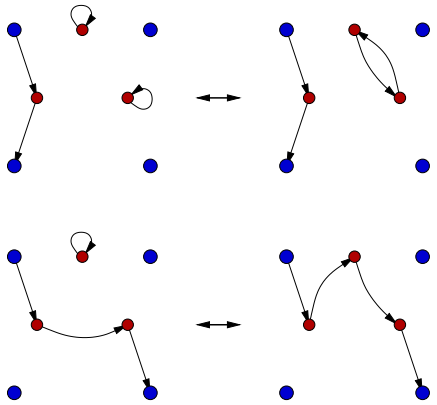

Figure 12.3: Inconsistent covers of the equality gadget. There are four symmetric cases.

Figure 12.4: The rosette graph of size four.

Let $(f_n) \in \mathsf{VNP}$ and let $(g_n) \in \mathsf{VP}$ such that

$$f_n(X_1, \ldots, X_n) = \sum_{e \in \{0,1\}^{q(n)}} g_n(X_1, \ldots, X_n, e_1, \ldots, e_{q(n)}.$$

We may assume that $(g_n) \in \mathsf{VP}_e$. We proved that every polynomial that is computed by a formula of size s is a projection of a determinant of polynomial size. The same proof yields that it is also a projection of a polynomially large permanent, since the cycle covers of the arithmetic branchning program occuring in the proof all had the same sign. It follows that we can write f_n as an exponential sums of permanents. The permanent itself is an exponential sum. So we are done if we can "squeeze" the outer exponential sum into the inner one.

The *rosette graph* of size t consists of a directed cycle of size t. The edges $c_1, \ldots, c_t$ of this cycle are called connector edges. The head and the tail of each connector edge are connected by a path of length two. Every node has a self-loop. All edges have weight one in the rosette graph. The following fact is easily verified:

Lemma 12.7 *Let S be a subset of the connector edges.*

1. *If S is nonempty, then there is exactly one cycle cover of the rosette graph containing the edges in S and no other connector edges.*

2. *There are two cycle covers containing no connector edges.*

g_n is a projection of a polynomially large permanent. This means that there is an edge weighted graph G (with the weights being field elements and

variables) such that

$$g_n(X_1, \ldots, X_n, Y_1, \ldots, Y_{q(n)}) = \sum_{\text{cycle cover } C} w(C).$$

Assume that the variable Y_i occurrs ℓ_i times in G. We add a rosette graph of size ℓ_i and connect every edge labeled with Y_i with one of the connector edges of the rosette. All edges inherit their weights from the corresponding subgraphs except that the edges carrying a weight Y_i get the weight 1 instead. We do this for each i. Assume, we introduced t equality gadgets altogether. We will add one isolated self loop with weight $(-1/2)^t$ to compensate for the -2 that is introduced by every equality gadget. (The characteristic of k should be distinct from 2 for this!) Let H be the resulting graph.

Let C be a cycle cover of G. $w(C)$ is a monomial $m(X_1, \ldots, X_n, Y_1, \ldots, Y_{q(n)})$. Let I be the set of indices such that Y_i appears in $w(C)$. What is the contribution of C in

$$\sum_e g_n(X_1, \ldots, X_n, e_1, \ldots, e_{q(n)})?$$

If Y_i appears in $w(C)$, then we have to set $e_i = 1$, otherwise, the constribution to the exponential sum will be zero. If Y_i dies not appear in $w(C)$, then we can set e_i to 0 or 1. Therefore, the contribution of C is

$$2^{q(n)-|I|} m(X_1, \ldots, X_n, 1, \ldots, 1).$$

We call a cycle cover D of H consistent if for every equality gadget, either both edges it connects are chosen or none of them is chosen. A cycle cover C of G can be extended to a consistent cycle cover of H. If an edge with label Y_i appears in C, then we can extend it in one possible way in the corresponding rosette. If no such edge appears in C then there are two ways. In total, there are $2^{q(n)-|I|}$ extensions. By Lemma 12.6, we know that

$$\operatorname{per} H = \sum_{\text{consistent D}} w(D).$$

Therefore,

$$\operatorname{per} H = \sum_e g_n(X_1, \ldots, X_n, e_1, \ldots, e_{q(n)}).$$

Theorem 12.8 *Over fields of characteristic distinct from* 2, per *is* VNP*-complete.*

Proof. It remains to show that per $\in$ VNP. It is quite easy to write a Boolean expression $E(Y)$ of polynomial size which checks whether a given matrix $Y \in \{0,1\}^{n \times n}$ is a permutations matrix. As done before, we can write this as an equivalent arithmetic formula $\hat{E}(Y)$. Now it is easy to check that

$$\operatorname{per} X = \sum_{Y \in \{0,1\}^{n \times n}} \hat{E}(Y) \prod_{i,j} X_{i,j} Y_{i,j}. \quad \blacksquare$$

Over fields of characteristic 2, the permanent can only be VNP-hard, if VNP = VP, since it conincides with the determinant in this case. But there are other VNP-complete polynomials that are also hard over fields of characteristic two.

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.

[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.